INVESTIGATING ESCAPE VULNERABILITIES IN CONTAINER RUNTIMES

by

Michael Reeves

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer Science

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Z. Berkay Celik

School of Computer Science

Dr. Antonio Bianchi

School of Computer Science

Dr. Dave (Jing) Tian

School of Computer Science

Approved by:

Dr. Kihong Park

Dedicated to all the friends, family, teachers, and colleagues who supported me throughout my Purdue education.

ACKNOWLEDGMENTS

First, I want to thank my advisor Prof. Berkay Celik. His persistent dedication to my education, future career, and life goals has played a pivotal role in my development as a student, researcher, and human being. Without his support and teaching, I would never have pursued a thesis nor gained such invaluable research skills.

Second, I want to thank my other committee members Prof. Antonio Bianchi and Prof. Dave Tian, who not only provided priceless advice in completing my thesis, but also taught a majority of my master's education. Under your guidance, I learned to think like a security researcher.

Third, I want to thank all my Purdue colleagues I've collaborated with throughout my Purdue education. I hope the bonds we've developed can continue long past our time here. I would especially like to thank the staff at CERIAS who have supported my security-related activities and programs in my six years at Purdue.

Fourth, I want to thank my colleagues at Sandia National Labs for funding my master's education and providing amazing mentorship, especially my everyone in RecoilLab, Han Lin, and Ken Patel.

Fifth, I want to thank my friends for helping me stay relaxed, have fun, and appreciate the moments that make life worth living.

Finally, I want to thank my family, especially my mom and dad, who have supported me through everything. Your guidance, love, and support gave me the energy to follow through on my thesis and master's studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Container adoption has exploded in recent years with over 92% of companies using containers as part of their cloud infrastructure. This explosion is partly due to the easy orchestration and lightweight operations of containers compared to traditional virtual machines. As container adoption increases, servers hosting containers become more attractive targets for adversaries looking to gain control of a container to steal trade secrets, exfiltrate customer data, or hijack hardware for cryptocurrency mining. To control a container host, an adversary can exploit a vulnerability that enables them to escape from the container onto the host. This kind of attack is termed a "container escape" because the adversary is able to execute code on the host from within the isolated container. The vulnerabilities which allow container escape exploits originate from three main sources: (1) container profile misconfiguration, (2) the host's Linux kernel, and (3) the container runtime. While the first two cases have been studied in the literature, to the best of the author's knowledge, there is, at present, no work that investigates the impact of container runtime vulnerabilities. To fill this gap, a survey over container runtime vulnerabilities was conducted investigating 59 CVEs for 11 different container runtimes. As CVE data alone would limit the investigation analysis, the investigation focused on the 28 CVEs with publicly available proof of concept (PoC) exploits. To facilitate this analysis, each exploit was broken down into a series of high-level commands executed by the adversary called "steps". Using the steps of each CVE's corresponding exploit, a seven-class taxonomy of these 28 vulnerabilities was constructed revealing that 46% of the CVEs had a PoC exploit which enabled a container escape. Since container escapes were the most frequently occurring category, the nine corresponding PoC exploits were further analyzed to reveal that the underlying cause of these container escapes was a host component leaking into the container. This survey provides new insight into system vulnerabilities exposed by container runtimes thereby informing the direction of future research.

1. INTRODUCTION

1.1 Overview

Containers enable organizations to scale and develop software solutions in the cloud-native era. Containers accomplish this scale by replacing the need to focus on developing specific operating systems or complicated software frameworks and by enabling developers to focus on writing applications within a pre-defined environment configured in a container. Because of this capability to easily abstract independent processes away from the host environment, many containers are run as micro-services. Through this approach, each container is responsible for maintaining and running a specific application. Thus, containers are designed to create lightweight and isolated applications that can be run on any server that supports containers.

In addition to being a development platform, containers are also designed to run untrusted code securely (Untrusted code has not been verified by the person executing it and therefore may be assumed to include malicious content until proven safe.) To constrain this untrusted code, the container leverages kernel isolation and kernel security mechanisms. When an application vulnerability exists in the container or the container itself is malicious, the host's isolation and security mechanisms are designed to prevent any malicious code from executing in the container and impacting the host. For example, if a web server has a Remote Code Execution (RCE) vulnerability, because it is running in a container, exploiting the vulnerability will not lead to a compromise of the underlying server. When any type of vulnerability is discovered in an application, the container can be killed and restarted with a patched version without having to restart the physical machine. However, there are more complex security issues that arise when there are vulnerabilities accessible in the container. These vulnerabilities can be used to defeat the isolation and security mechanisms constraining the container. They have a high impact on the integrity of the container host system so adversaries continue to search for opportunities to exploit these types of vulnerabilities. Exploits that target these vulnerabilities are known as "container escapes" since an adversary is able to execute and control code on the host from within the container.

There are three main categories of vulnerabilities adversaries exploit to escape a container: (1) Profile misconfiguration, (2) Linux kernel, and (3) Container runtime. The first occurs when the isolation and security mechanisms applied to containers are not properly secured or are dropped entirely to ease the development process (e.g., running a privileged Docker container [1]). When adversaries compromise poorly secured containers, they can easily break out (e.g., how to escape a privileged Docker container [2]). The second occurs when there is a Linux kernel vulnerability that is exploitable inside a container. Because containers are still Linux processes that share the same kernel as the host, any kernel vulnerability accessible to the container process is exploitable in the container. By leveraging the appropriate kernel exploit for an accessible vulnerability, the adversary can execute code as root, and modify the properties of the container such that it is no longer secured or isolated from the host [3]. These two kinds of escapes have been studied in the literature [4] [5] [6]. The third vulnerability, container runtime, unlike the two previously studied vulnerability types, has not been previously studied.

1.2 Contributions

In this thesis, a survey over 11 container runtimes and their corresponding 59 CVEs was conducted. The survey's goal was to gain insight into why container runtime vulnerabilities occur, and what impact the corresponding exploits have on the container host. Since CVEs do not include technical details and only a short paragraph description of the vulnerability, a working PoC exploit for the CVE is required to fully analyze the impact of the CVE on the container host. After querying publicly available search services and repositories, only 28 CVEs were found to have PoC exploits. To analyze the cause and effect of each CVE's PoC exploit a formal framework was required. In the framework, each PoC exploit is analyzed according to its "steps", or the high-level commands executed by the adversary. Using these exploit "steps", a seven-class taxonomy was created to categorize the 28 CVEs based on the cause and impact of each vulnerability's exploit. After completing the taxonomy of the 28 CVES, we found the largest category, CVEs with PoC exploits leading to container escapes, comprised 46% or 13 of the 28 CVEs. These 13 CVEs had nine associated PoC exploits: seven

PoC exploits covering one CVE, and two covering three separate CVEs. These nine exploits were further investigated to determine the underlying causes of container escapes that exploit container runtimes. This final investigation phase revealed that the major reason these escapes occur is due to unintended host components exposed in the container. This survey motivates the direction for future research seeking to secure containers against escapes.

2. BACKGROUND

This chapter discusses the background topics of container escapes, traditional container isolation and security, and container architecture design.

2.1 What is a Container and How is it Escaped?

For this work, A container is defined as a secured and isolated application. For example, a container might be dedicated to running a web server, or database, or may be offered as a "Container as a Service" (CaaS) [7] [8]. In some cases, A vulnerability may exist in the application that enables an adversary to gain execution within the container or the adversary already has access to execute code in the container (e.g., CaaS). Even with this ability to execute code in the container, the adversary usually cannot attack other resources on the container host because the container is isolated from the container host. In addition, if the container is properly secured, the isolation of the container should be unbreakable. However as discussed previously, an adversary can exploit three kinds of vulnerabilities accessible from within the container (misconfiguration, runtime, and kernel vulnerabilities) to break the security mechanisms of the container, violate the container's isolation, and run code on the host. The next sections describe the isolation and security mechanisms provided by the Linux kernel.

2.2 Kernel Isolation Mechanisms

The Linux kernel provides two isolation mechanisms for processes: (1) Namespaces, an abstraction layer on each global system resource where processes within a given namespace appear to have their own unique instance of that resource, and (2) Cgroups, which limit consumption of system resources by grouping processes into "control" groups.

Namespaces. Namespaces confine processes into groups to limit access to changes in global system resources (e.g., the container namespace should not have the same access to network interfaces as the host). There are currently eight namespaces as defined by the Linux man page [9]. They help isolate process's: cgroup, IPC (Inter-process Communication) objects,

network stack, mount points, PIDs (Process Id), time data, user and group ids, and host and domain name. For example, when a container process is put into its own PID namespace it can only see processes that also share its PID namespace (i.e., only processes that are running in that specific container).

Cgroups. Cgroups are used to limit each process's usage of system resources. Each cgroup defines limits for specific system resources and upon full consumption of system resources, all other processes in the same group are prevented from accessing more [10]. For containers, the three main cgroups are CPU, memory, and I/O, which limit the number of CPUs, amount of memory, and input/output operations on block devices respectively. Container runtimes each set these cgroups using their internal methods. For example, Docker enables a user to specify in a container runtime configuration the limits for CPU and memory usage [11].

2.3 Kernel Security Mechanisms

The kernel provides three security mechanisms that help secure the host against malicious execution in containers: (1) Mandatory Access Control (MAC), (2) Seccomp, and (3) Capabilities. These mechanisms are highly important as they are the main protections the container host can use to defend itself from attacks that originate in containers.

Access Control Protection. In general, policy frameworks such as mandatory access control (MAC) and discretionary access control (DAC), constrain actions users can take on systems using rule sets. If an action is permitted by all rules in the policy framework, it is allowed, otherwise, the action is stopped. Thus, the security of the system is reliant on the proper building of the rules and reliability of the framework. In DAC, the policy is defined by simple rules where users and groups of users can only perform r/w/x (read/write/execute) permissions on files they own, however, there are many ways around such a simplified rule system. For example, a file with root permissions and execution vulnerabilities that is executable by anyone in the system could be used as an underprivileged user to execute code as root. In MAC, the policy frameworks like SELinux [12] and Apparmor [13] define in-depth rules deeply embedded in the kernel as Linux security modules (LSM [14]) and only permit actions defined by the configured policy. However, if the policy isn't clearly defined, or there

are bugs in the policy modules themselves, adversaries can disable the MAC frameworks or bypass them entirely through flaws in policy logic. These frameworks are non-trivial to implement properly on any highly modular system (e.g., home or corporate computing environment).

Seccomp. Seccomp filters enable the kernel to firewall system calls for any process. Each process is assigned various seccomp settings that can be either very complicated or as simple as a black/white list of enabled/disabled system calls [15]. For example, a simple list of allowed or denied system calls can be defined in LXC [16], and Docker sets a default seccomp profile of the allowed system calls for each container when one is not provided in configuration [17].

Capabilities. Before Linux kernel version 2.2[18], the Linux root user had full permissions over all aspects of the Linux operating system. Either a user was a superuser able to perform any actions or a regular user who could not perform any privileged actions (e.g., open raw sockets or mount directories). Capabilities were created to separate root permissions into 38 separate units and create a more fine-grained permission model for the Linux kernel [18]. For example, CAP_NET_ADMIN gives a process the permissions to edit all levels of the network stack, including the system firewall, while CAP_SYS_ADMIN gives general administration permissions such as mounting file systems. To properly isolate containers, the capabilities assigned to the container must minimize the allowed functionality, otherwise, the adversary can abuse these permissions as pointed out by grsecurity [19]. Granting a container CAP_NET_ADMIN or CAP_SYS_ADMIN is quite permissive and dangerous, especially for something designed to be untrusted such as a container.

2.4 Container Architectures

This section discusses how containers are created and executed by exploring how an OCI (open container initiative) [20] compliant container is created, using the Docker container ecosystem as a case study. For the most part, container runtimes follow the OCI specification for containers, except LXC which is not compliant but has wrappers to enable inter-operability [21]. This defines how both containers are managed and executed.

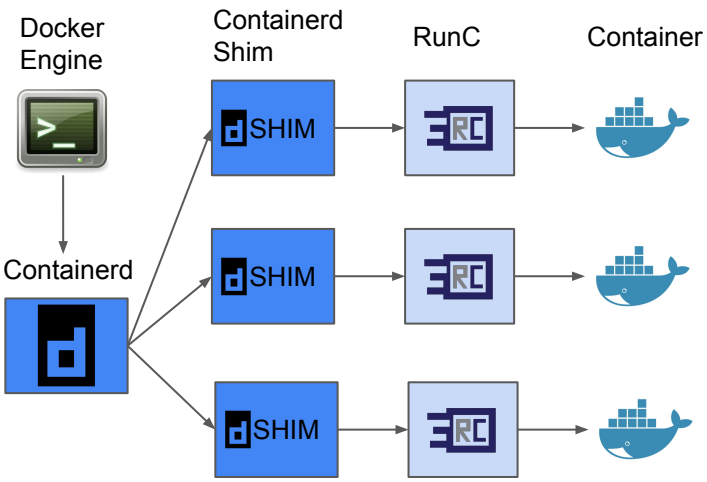

Figure 2.1. The Docker container architecture

OCI containers. In 2015, container developers realized there was not a unified format for specifying how containers should be created and run so they started the OCI. This organization defines a spec of requirements for container runtimes and container images to create an industry-standard container [22]. A figure is provided to aid in the visualization of the typical container architecture (Figure 2.1). The default runtime for the spec is runC [23], which performs the initialization of the actual container process. runC has a simple job, take in an OCI compliant image, and execute the functionality as defined by the OCI image. While runC is mainly concerned with the physical execution of the container, containerd [24] focuses on managing all the metadata and image files for containers. Hence containerd draws its name from being a "container daemon". So containerd organizes and gathers all required container metadata to build and execute an OCI container image. Keeping these facts in mind, when a user executes the command "**docker run ubuntu**" to build an Ubuntu container, this sends a request to the Docker engine. The Docker engine coordinates with Dockerhub [8] to download all required image information. Then, this data is processed, stored, and sorted by containerd, which builds out an OCI image. Next, containerd starts a shim to handle setting

up the new container to be executed by runC. Lastly, runC executes the OCI image as a container process. The encapsulation of various container responsibilities enables developers to focus on securing each container component by its specific responsibilities. This thesis focuses on the container execution portion where the runtime is executed and investigates vulnerabilities exposed in the container through the container runtime. In this case, all the runtimes investigated in this survey replace runC as the process to execute the container.

3. RELATED WORK

In this chapter, previous work related to container security is explored. These works are broken into two categories: (1) Container measurement studies, and (2) Vulnerability analysis. The container measurement studies explore vulnerabilities related to attacks against containers. The vulnerability analysis works investigate the threat landscape of a system from a CVE perspective. For each work, we highlight the specific aspects that are related to the container runtime vulnerability survey.

3.1 Container Measurement Studies

These measurement studies focus on container vulnerabilities. While this thesis focuses on container runtime vulnerabilities, container measurement studies about container vulnerabilities are discussed to define the current literature in container vulnerability research.

A measurement study on exploit execution effectiveness in containers was conducted highlighting the impact of kernel escape type exploits on containers [4]. This measurement study examined 88 exploits based on user-space application and kernel vulnerabilities. They found that 50/88 exploits successfully ran on containers. Furthermore, they claim 4 of the 30 kernel exploits achieved privilege escalation and successfully ran the in the container. These exploits all followed a similar attack chain utilizing *commit_creds* to achieve privilege escalation, so they designed a 10 line code defense into the commit_creds kernel function to prevent these exploits from running.

Anton investigates risks and benefits of user-namespace security applied to container vulnerabilities, notably dirtycow (CVE-2016-5195), socksign (CVE-2017-7308), and runC (CVE-2019-5736). For dirtycow and socksign, user-namespaces do not prevent the exploit from succeeding. For the runC vulnerability, remapping the root user of a container to an unprivileged user effectively mitigates the vulnerability. This study shows that efforts can be taken to mitigate some of the vulnerabilities discussed here.

A different work explores vulnerabilities within the Docker ecosystem [6]. This research focuses on Docker misconfiguration and image vulnerabilities, while container runtime vulnerabilities are listed, but not discussed. In contrast, this thesis chooses to focus on container

runtime vulnerabilities across 10 other container runtimes and explores the vulnerabilities exposed by the Docker runtime more fully.

In this work, Flauzac *et al.* conducted a comparison over technical aspects of container solutions [25]. This work compares LXC, Singularity, runC, Kata, and gVisor container runtimes using four feature sets, but the most applicable part of this work is the isolation and security capability comparisons. From the security features comparison, cgroup isolation is an optional setting across all the container runtimes except gVisor. In addition due to design choices, Kata containers do not support Apparmor or seccomp filters, and gVisor supports seccomp filters, but not Apparmor. From the isolation comparison, LXC has more robust default isolation settings over Singularity or runC. Most significantly LXC runs containers in a separate user namespace by default, while the others do not. In addition, Kata containers run containers inside a hypervisor, and gVisor provides a kernel isolation mechanism. These insights are helpful when comparing vulnerabilities across container runtimes.

Another research group measures the ability of containers to limit resources by stress testing container cgroup limits on an Amazon cloud instance [26]. They present five case studies where cgroup limits can be defeated. This paper does not study the case of Denial of Service attacks, so while this novel attack is interesting, we consider Denial of Service attacks out of scope, as they do not enable the attacker to control code on the container host. DoS attacks limit the availability of system resources and do not compromise the integrity or confidentiality of the host.

3.2 Vulnerability Analysis

These next two works are good examples in the literature that analyze and compare vulnerabilities based on CVEs. The first paper compares vulnerabilities by analyzing their exploits and the second uses CVEs to provide insight into the security issues exposed in trusted execution environments.

Allodi and Masscci conducted a vulnerability risk assessment by comparing discovered CVEs to actively used exploits [27]. They find that a high CVSS score is not as indicative of the risk of the exploitability of a vulnerability. Rather, the existence of a working proof

of concept (42%) or Black Market exploit sale (80%) is associated with a higher risk of exploitation. This insight motivates the survey to only investigate vulnerabilities where a publicly available PoC exploit can be found. Due to lack of resources, it was not possible to investigate black market exploit sales.

Cerdeira *et al.* perform a vulnerability analysis of popular TrustZone Trusted Execution Environments by leveraging publicly documented exploits and vulnerabilities to gain insight into critical vulnerabilities that exist in these systems [28]. The relevant contribution to this thesis is the vulnerability study that was conducted over TEE implementations. They classified the vulnerabilities based on the severity of the CVSS score: critical, severe, medium, low. This study shows that useful insight can be gained from studying the CVEs within the NVD, just as performed in the container runtime CVE survey. The TEE survey differs from the container runtime survey in that it motivates reverse-engineering the TEE binaries, while the container runtime survey breaks down exploits into high-level attacker commands called "steps".

4. THREAT MODEL

For the system, we assume an administrator is operating a Linux-based container host where each container runs a specific application (*[1]* in Figure 4.1). For example, this may be a micro-service type environment hosting external corporate services (e.g., a container is responsible for hosting a web server and another a database) or a CaaS host providing container instances for clients ([7], [8]). The exact hardware used for this host is unimportant as all hardware is part of the trusted computing base (TCB), as well as all the software in the Linux kernel (*[5]* in Figure 4.1). The administrator controls all software on the container host and secures the containers so that the adversary cannot take advantage of misconfiguration vulnerabilities (*[3]* in Figure 4.1). For the adversary, we assume they have gained code execution inside the containers the administrator hosts (*[2]* in Figure 4.1). This could happen in one of three ways: (1) there is a remote code execution vulnerability present in a container application with a high probability of exploitability [27], (2) the service allows the adversary to execute code on the container as a user in the environment

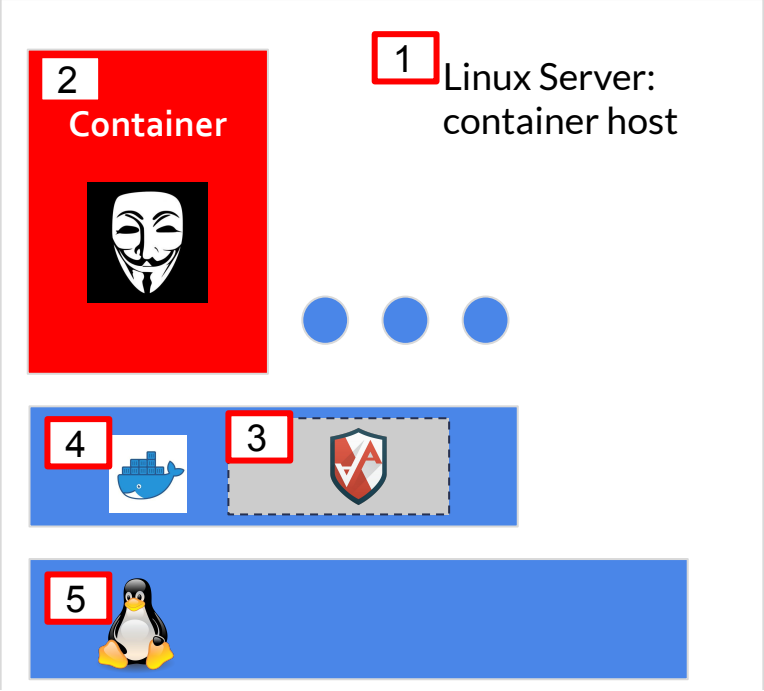

Figure 4.1. Visual of the threat model with labeled components

(e.g., CaaS), and/or (3) the administrator by mistake downloads and executes a malicious container image. The adversary's goal is to own the whole container host by exploiting a vulnerability available in the container to execute a container escape. For this study, the adversary can exploit any vulnerability in the container's runtime accessible within the container (*[4]* in Figure 4.1). Any side-channel attacks that may be possible only impact the confidentiality of the system and not integrity (i.e., they don't enable adversary command execution outside the container), and therefore are out of scope for this study ([29], [30], [31]). As well as side channels, attacks targeting the availability of the system, denial of service (DoS), are considered out of scope (e.g., a container that hogs all system resources). These attacks do not affect the container hosts' integrity or confidentiality and have been studied in previous works [26].

5. CONTAINER RUNTIME SURVEY

In this chapter, we detail the entirety of the container survey. First, we describe the collection process of the container runtimes list and their corresponding CVEs. After collection, initial analysis motivates the need to focus on runtime vulnerabilities with PoC exploits. To classify the PoC exploits, a seven-class taxonomy is created to understand the cause and impact of each PoC exploit. Since the major category of this taxonomy was found to enable container escapes, these nine PoC exploits covering 13 container runtime CVEs were further investigated. This final analysis determined the underlying cause of container escapes exploiting container runtimes: a host component exposed in the container.

5.1 Data Collection Overview

Container Runtimes Analyzed. To analyze as many runtimes as possible, a list of popular runtimes was gathered from the literature review and search engine queries (e.g., "popular container runtimes"). The full list of container runtimes along with respective links to their code repositories is featured in Table 5.1. We note that while rkt is no longer maintained (per the Github repository), analyzing vulnerabilities found in the project is still relevant, as rkt was an actively used container runtime.

Table 5.1. Distribution of CVE vulnerabilities for each runtime from 2016 to March 2021

Runtime	Critical	High	Medium	Low
LXC [32]	1.0	3.0	0.0	2.0
Docker [33]	1.0	11.0	7.0	0.0
runC [23]	0.0	4.0	0.0	0.0
CRI-O [34]	0.0	1.0	1.0	0.0
Singularity [35]	1.0	8.0	1.0	0.0
gVisor [36]	1.0	0.0	2.0	0.0
rkt [37]	0.0	3.0	0.0	0.0
crun [38]	0.0	1.0	0.0	0.0
Podman [39]	0.0	1.0	2.0	0.0
containerd [24]	0.0	1.0	2.0	0.0
Kata [40]	0.0	3.0	2.0	0.0
Total	4.0	36.0	17.0	2.0

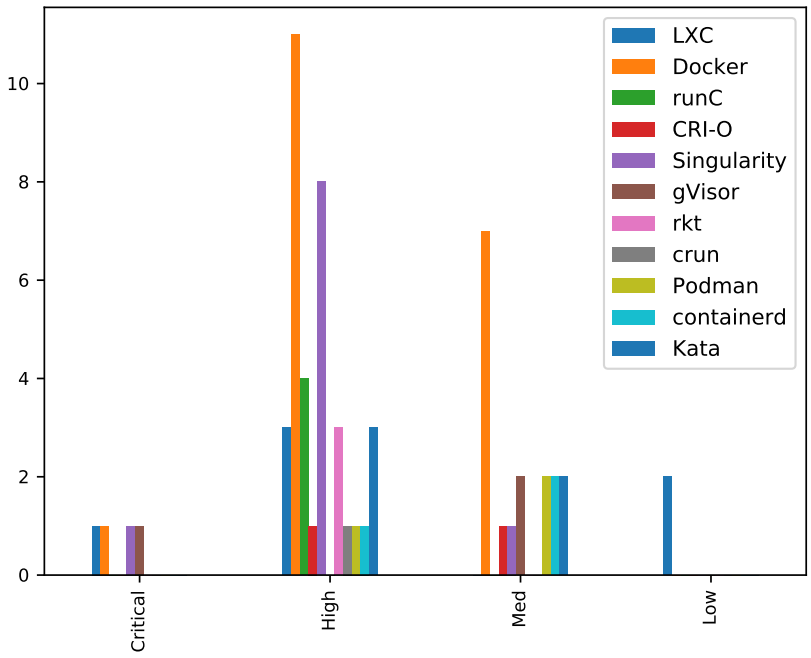

Figure 5.1. Container runtime CVEs by CVSS score

CVE Data Sources. The NIST National Vulnerability Database (NVD) [41] publishes all the data regarding common vulnerability enumerations (CVE) [42] that are submitted to Mitre. For this study, we investigated all CVEs published from January 1st, 2016 to March 2021. 2016 was chosen since this was one year after the foundation of the OCI, while March 2021 was when the survey was conducted. To parse this data effectively, we leveraged nvdtools [43], an open-source golang library that downloads and queries the published NVD JSON data from the NVD public API. The NVD data is designed to be queried via common platform enumeration ids (CPE), which to quote NIST is "a structured naming scheme for information technology systems, software, and packages" [44]. Accordingly, the container runtime list in Table 5.1 was transformed into a list of CPEIDs (see appendix A.1), so all CVEs for each runtime could be collected. After building this CPEID list, the nvdtools utility cpe2cve collected all relevant CVEs for each respective runtime CPEID yielding 59 CVEs in total.

Initial Analysis. The distribution of these CVEs is displayed in Table 5.1. The Critical, High, Medium, Low designate the severity of a vulnerability determined by its respective Common Vulnerability Scoring System(CVSS) score [45]. The ranges for each category are as follows: Critical (> 9), High ($>= 7, < 9$), Medium ($>= 4, < 7$), Low ($> 0, < 4$). The higher the severity of the vulnerability the greater the impact the vulnerability has on the confidentiality, integrity, and availability of the container host. Looking further at Figure 5.1, we can see that 40 of the vulnerabilities are high or critical severity, which demonstrates the high impact container runtime vulnerabilities can have on the container host. In addition, at least every runtime has a high severity vulnerability, while Docker and Singularity have the greatest number: 12 and 9 high/critical vulnerabilities respectively. As this table is an overview of all container runtime vulnerabilities, it includes seven DoS vulnerabilities and two repeat vulnerabilities. These nine vulnerabilities were removed from the study as they fall outside the scope of the threat model. While having this high severity distribution of CVEs demonstrates the critical nature of container runtime vulnerabilities, CVEs alone do not provide enough detail to understand the impact of the vulnerabilities on the container host. We discuss the issues with CVEs in the next section by pivoting the investigation to focus on CVEs with PoC exploits.

5.2 Mapping CVEs to Exploits

Finding Exploits. To analyze the impact of the remaining 50 container runtime vulnerabilities, more information must be explored, as CVEs do not contain technical details only a textual description of the vulnerability. The CVE descriptions miss technical details from two aspects: (1) the step-by-step technical process an adversary can take to exploit the vulnerability, and (2) the exact privileges gained by the adversary as a result of exploiting the vulnerability (e.g., gaining host code execution, host network access, or privilege escalation). For example, the description for CVE-2018-19295 for the Singularity runtime is only one sentence: "Sylabs Singularity 2.4 to 2.6 allows local users to conduct Improper Input Validation attacks" [46]. This CVE description lacks both kinds of details (technical steps and the adversary gained capabilities).

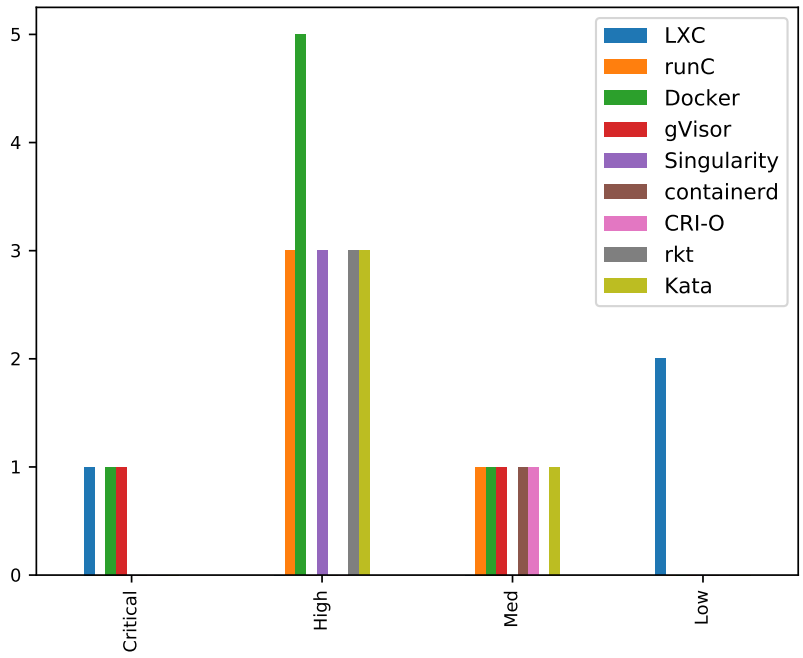

Figure 5.2. Container runtime CVEs with PoC exploits by CVSS score

To gather a list of all PoC for each vulnerability, we began by building the initial list of exploits from those available in the NVD. Each CVE's JSON data contains reference links that point to additional information associated with each CVE. If exploits were available in a CVE's NVD entry, they were denoted with the "exploit" tag in the reference link list. By filtering for all appropriate "exploit" flags, we identified 12 CVEs in the NVD with publicly available PoCs.

To ensure we covered the other CVEs missing PoC exploits in the NVD database, we leveraged Github and Google to query for publicly available PoC exploits. Example search strings used to validate available public PoCs for a CVE include "[CVE-NUM] PoC" and "[RUNTIME_NAME] [CVE-NUM] exploit". From these manual search queries, we identified an additional 16 CVEs with a public PoC exploit. In total across the NIST NVD and public searches on Google and Github, we curated a list of 28 CVEs with PoC exploits (Figure 5.2).

Completeness. While only the CVEs with known PoC exploits were analyzed further, this does not mean other vulnerabilities cannot be exploited, nor that they lack working exploits.

The exploits may not be accessible for two main reasons: (1) Adversaries hold them privately for use as zero-days and do not disclose the exploits after the vulnerabilities are discovered or the zero-days are found exploiting the vulnerability in the wild, or (2) The vulnerabilities may have been disclosed responsibly to the affected vendors, and such vendors desire the researchers to keep the exploits private. Thus, the exploits are never made publicly available. In either case, the vulnerability is still valid, but as discussed previously has limited use for further analysis. So only the 28 CVEs with PoC exploits are explored further. To analyze the 28 CVEs further, they were categorized by the cause and impact of their associated exploits.

5.3 Exploit Analysis Framework

To analyze and compare exploits across runtimes, a systematic framework must be created. Frameworks of choice could be: (1) comparison of raw exploit binaries, or (2) comparison using a high-level adversarial framework like Mitre ATT&CK [47]. If a full binary analysis was utilized for the framework, access to the exploit's binary or source code would allow deep technical analysis (e.g., comparing the exact syscalls used in an exploit). However not all PoC URLs for a given CVE provide binary-level code or even source code. Limiting the analysis to this low level would exclude these important vulnerabilities from the analysis. In addition, this analysis can miss important semantic information that is more easily detailed with a higher level framework such as where the exploit is executed (e.g., inside the container or on the container host).

On the other hand, if a high-level adversarial framework was utilized, analyzing PoC exploits would not require access to full source code or the PoC binary; a detailed PoC blog-like explanation of the exploit would suffice. However, important technical commands that are required for an exploit could not be included in the analysis. For example, the use of the mount command to mount a host directory within a container would be considered "T1006: Direct Volume Access" in the MITRE ATT&CK framework. Using this classification would lose the semantic information of the exact commands executed by the adversary.

So the framework created here takes a middle ground between full binary analysis of the exploits and the high-level Mitre ATT&CK framework [47]. The framework used in this

Table 5.2. Number of CVEs in each category

Category	Exploits
Container-to-Host-Escape	13
Host-Privilege-Escalation	6
Container-to-Host-Limited-Host-Access	4
Unpatched-System	2
Container-to-Host-Network-Access	1
Container-Privilege-Escalation	1
MAC-disable	1

analysis breaks each exploit into high-level adversary commands called "steps". These steps can capture the details missing in the high-level adversarial framework, but do not require the original source of the exploit like in binary analysis.

5.4 Exploit Taxonomy

Using the steps created with the exploit analyzer framework, we created a seven-class taxonomy based on the cause and impact of each exploit. This section details six of the seven categories in the taxonomy. The last category, full container escape exploits, is explored in the next section since 46% of the CVEs in the taxonomy fall into this category.

For all seven categories, each CVE is categorized based on the associated exploit's impact (i.e., the capabilities the adversary gains from executing the exploit such as host code execution, host network access, or host privilege escalation) and the cause of the exploit (i.e., how does the vulnerability enable the adversary to gain the capabilities associated with the impact). The overall distribution of each exploit category is visualized in Figure 5.3. In the rest of this section, the findings for each category are presented and analyzed according to the associated exploits. As stated previously, the largest exploit category, a full container escape, is associated with the most CVEs, so it is presented in the next section.

MAC Disable. This category includes exploits that disable the MAC framework used to constrain the container (e.g., Apparmor or SELinux). There is only one exploit in this category. CVE-2019-16884 [48] enables an adversary to disable the Apparmor MAC constraints securing runC container runtime. This issue was the result of runC, not properly validating

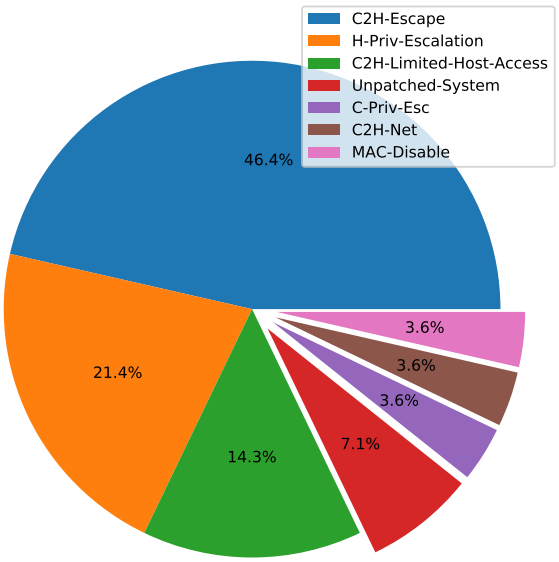

Figure 5.3. Breakdown of CVE exploits by category

runtime mount paths. To exploit, the adversary first creates a fake /proc directory. When the container runs, the host /proc is mounted over this fake directory, and Apparmor is fooled into running the container process "unconfined" (i.e., the container is no longer constrained by app armor). This exploit only stops Apparmor from securing the container, but other security mechanisms (e.g., seccomp) still greatly limit the capabilities of the adversary.

Container Privilege Escalation. This category includes vulnerabilities that achieve container privilege escalation, but do not escape the container (e.g., inside the container a regular user can escalate privileges to root). There is only one exploit in this category. CVE-2019-19333 enables, "...unprivileged processes in the sandbox to read and write the memory of other, more highly privileged processes in the sandbox" [49]. Thus, a regular user can gain root privileges, but because of gVisor's sandbox design gaining root in the container does not enable a full container escape.

Container to Host Network Access. This category includes vulnerabilities that enable the adversary to gain host network access from within the container and demonstrates that container constraint mechanisms can expose the host to new container vulnerabilities. There is one vulnerability in this category, CVE-2019-14891 [50] for the CRI-O runtime. This CVE enables the adversary to gain full networking capabilities on the container host, by gaining control of resources allocated for the CRI-O container monitoring process. When CRI-O starts up, a monitoring process for all CRI-O containers runs in the background. The vulnerability exists because CRI-O spawns all containers under the same cgroup including the initial CRI-O monitoring process. Thus, a malicious actor can consume all of the memory allocated for the CRI-O container cgroup space. In doing so, multiple times, a malicious CRI-O container is able to gain a namespace reference to the old host monitoring PID, which shares the host NET, UTS, and IPC namespaces. This enables the adversary to gain access to the host's network interface, and thus the host network. However, this vulnerability does not directly enable a container to escape onto the container host.

Unpatched System. This category includes vulnerabilities for runtime packages which when patched regress to old vulnerabilities by faulty or mistaken patches. Missing proper patches can introduce discovered vulnerabilities into an otherwise secure system. Redhat published two CVEs to track incorrect runC versions that were pushed to Redhat Enterprise Linux seven (RHEL7), CVE-2020-14298 [51], and CVE-2020-14300 [52]. The exploits for CVE-2019-5736 [53] and CVE-2016-9962 [54] (corresponding to the unpatched CVEs 14298 and 14300 respectively) will be able to exploit the unpatched runC software in the unfixed RHEL7. Since the 14298 and 14300 vulnerabilities are the results of failed patches, they are not included in the Container-to-Host escape category. The real exploits are covered by the original CVE corresponding to the two runC vulnerabilities (5736 and 9962). As each of these vulnerabilities is a full container escape they are explored further in the next section.

Limited Container to Host Access. This category defines vulnerabilities that enable the adversary to gain limited control over the container host from within the container (e.g., create network interfaces, enable/disable hardware, or discover privileged file paths). There are four exploits in this category. The first exploit corresponds to CVE-2017-5985 [55]

where the LXC suid binary [56] (lxc-user-nic) [57] did not check the target net namespace utilized by the user executing the binary to create a network interface. This enabled any user to create a host network interface. The second exploit corresponds to CVE-2018-10892 [58] where a malicious Docker container can modify host hardware exposed through /proc/acpi. For example, the container can disable/enable Bluetooth, or modify screen brightness. The third exploit corresponds to CVE-2018-16359 [59] where gVisor did not block the renameat syscall, so any container user was able to change filenames for paths on the host. The fourth exploit corresponds to CVE-2018-6556 [60] where the LXC suid binary lxc-user-nic enables an unprivileged user to open arbitrary files on the host (but does not allow read/write to those files).

Host Privilege Escalation. This category is the second-largest category and defines vulnerabilities that enable an adversary to gain privilege escalation assuming they have access to the container host. These vulnerabilities do not enable a container escape, as they require the ability to run code as a host user prior to the adversary exploiting the vulnerability. The first three CVEs correspond to the Singularity container runtime: CVE-2019-11328 [61], CVE-2018-19295 [62], CVE-2020-13847 [63]. CVE 11328 and 19295 identify issues with the suid binary Singularity uses to initialize and run containers. Both vulnerabilities are the result of overly permissive directories containing the namespace information for spawned containers. These directories can be manipulated by a local user executing the suid binary to point to arbitrary mount, NET, IPC, PID namespaces. Thus, the user can spawn a container that to the host appears as a root process. The third Singularity CVE 13847 occurs because Singularity did not validate the header/descriptor fields of the SIF (Singularity image format [64]) container image when checking the SIF image hash. Normally, the entire SIF file should be hashed and validated by the image validation signature. In this case, the header and descriptor fields can be modified after generating the validation signature for a given image, and the new modified image will still validate based on the old signature of the unmodified image. This enables an adversary to put arbitrary commands in the header of a secure image to be executed during container build time, enabling the adversary to gain privilege escalation.

The next two CVEs in this category correspond to Docker. The first Docker CVE, CVE-2019-13139 [65], is a bug in "Docker build" when reading from a malicious Github URL that includes a command injection string. When the URL is set up as a local path, "Docker build" does not cleanse the input, and the included command will be executed by "Docker build". Thus, the adversary can gain code execution as whatever user is performing the "Docker build". Usually, this is the root user since running "Docker build" as a non-root user requires special configuration [66]. The second Docker CVE, CVE-2018-15514 [67], impacts Windows systems as it is a deserialization vulnerability present on the Windows Docker client. While this survey focuses on vulnerabilities impacting a Linux container host, this vulnerability is included here for completeness, as there is a publicly available PoC exploit for the vulnerability.

The last CVE is CVE-2020-27151 [68] which impacts the Kata container runtime. If a local user defines a malicious container configuration, then any binaries designated in the configuration will execute with the same permissions as the Kata runtime. Since at the time, Kata-runtime ran with root permissions, this was a privilege escalation issue. The six vulnerabilities in this class demonstrate how container runtime vulnerabilities can enable local users on the container host to gain privilege escalation.

Non-Escape Categories Summary. Overall, these six categories cover 16 CVEs which the adversary can exploit to gain various levels of host access. While the adversary cannot escape the container with any of these vulnerabilities, they can gain still gain significant capabilities especially in the host privilege escalation category, where each vulnerability enables a local user to get root permissions.

5.5 Container Escape Exploits

While the taxonomy from the last section revealed that container escapes were the highest occurring issue, it did not detail how an adversary exploits these 13 CVEs to escape containers. In this section, we further investigate the container runtime exploits that lead to a container escape, and how this leak occurs for each exploit. The main reason container escapes are able to exploit container runtimes is an exposed host component inside

Table 5.3. Container runtime exploits listed by the runtime, highlighting the cause of the escape (✓), the non-causes (✗), and the leaked host component

Runtime	CVE	File Descriptor Mishandling?	Component Missing Access Control?	Host Execution in Container Context?	Host Component Leaked
LXC	2016-8649	✓	✗	✗	/proc fd
runC	2016-9962	✓	✗	✗	/proc fd
Docker	2018-15664	✗	✗	✓	host chroot
runC	2019-5736	✓	✗	✗	/proc/self/exe
runC †	2019-19921	✗	✓	✗	/proc via a container volume
Docker	2019-14271	✗	✗	✓	/proc
rkt	2019-10144	✗	✗	✓	host filesystem device
	2019-10145				
	2019-10147				
containerd *	2020-15257	✗	✓	✗	abstract UNIX socket
Kata	2020-2023	✗	✓	✗	container to host shared directory
	2020-2025				
	2020-2026				

* the exploit requires access to the host network namespace, † the exploit requires control of two containers

the container. We determined that the host component exposure occurs from three different issues: (1) a mishandled file descriptor, (2) a runtime component missing access control, or (3) adversary-controlled host execution.

For issue 1, the exposure occurs by a file descriptor remaining open inside the container via the /proc filesystem, which gives the adversary read/write access to the host filesystem. For issue 2, the exposure occurs because the adversary gains access to a host runtime component that failed to implement fine-grained access control. For example in CVE-2020-15257, the adversary gains access to the containerd abstract UNIX socket. For issue 3, the exposure occurs because a host binary executes in the context of the container which enables the adversary to manipulate execution through either a malicious shared object or a malicious symlink. Table 5.3 lists each exploit sorted by CVE date. The runtime and CVEs associated with each exploit, the cause of the leak (denoted by a green checkmark), and the leaked host component are all listed in the respective column. For reference, the steps of each exploit are included in Appendix A.2.

5.5.1 File Descriptor Mishandling

This section details the three exploits that lead to container escapes due to mishandled file descriptors exposed in the container. These exploits are associated with CVE-2016-8649, CVE-2016-9962, and CVE-2019-5736 respectively.

CVE-2016-8649. This CVE was identified in LXC [69] and enables a container escape by exploiting a race condition created during the lxc-attach [70] command to access an instance of a host file descriptor. The lxc-attach command enables the administrator to get a shell in the context of the currently executing container. To exploit the vulnerability, the adversary constructs a fake proc file system in the container that ensures capabilities are dropped for new processes. When the administrator executes lxc-attach to get a container shell, the adversary executes a malicious binary to ptrace lxc-attach. Through this ptrace command, the adversary is able to copy the host file descriptor of the lxc-attach process, and use the execve system call on the file descriptor from the container context to execute a new process. This new process will have all capabilities enabled due to the hijacked /proc set up by the adversary. Thus the adversary escapes the container and gains unconstrained code execution on the host.

CVE-2016-9962. This CVE was identified in runC [54], and enables a container escape by exploiting a race condition during the initialization of a runC container to access an instance of a host file descriptor. This exploit is similar to CVE-2016-8649 in that the init process of the runC container has an open file descriptor on the host. However, to exploit this race condition the adversary does not need to create a fake /proc filesystem in the container. Upon container initialization, the adversary uses ptrace on the init process on container startup. This enables the adversary to gain a copy of an open file descriptor on the host. Using the open file descriptor to the host, the adversary is able to read/write files on the host, signifying a full container escape.

CVE-2019-5736. This CVE was identified in runC [53] and enables an adversary to overwrite the host runC binary and therefore execute arbitrary code on the host. To exploit this CVE, the adversary configures the malicious container with two properties: (1) a symlink from the container's entry point (usually /bin/bash) to /proc/self/exe, and (2) a malicious.so

file to be loaded by /proc/self/exe (in this case the host runC) later. This malicious.so file overwrites the file descriptor of the executing process that loads it, and so will overwrite the host runC. When the malicious container executes, the symlinked entry point will point to the host's runC binary, as this is what /proc/self/exe refers to. Then, this /proc/self/exe will execute in the context of the container and load the malicious.so. The malicious.so then overwrites the host runC binary referenced by /proc/self/exe with a malicious backdoor. Finally, when the administrator spawns new containers, the malicious backdoor is executed signaling the adversary can successfully escape the container. This exploit succeeds because it is able to gain a reference to the host runC file descriptor. Using this file descriptor, the adversary can execute the host runC in the context of the container and force the newly executing runC (which is the host binary) to load the adversary's malicious.so.

5.5.2 Runtime Component Missing Access Control

This section details the three exploits that lead to container escapes due to runtime components that fail to implement robust access control. These exploits demonstrate that access to any properties of the container runtime should be properly authenticated, otherwise they present a threat to the container host's security. These exploits are associated with CVE-2019-19921, CVE-2020-15257, and CVE-2020-23,25,26 respectively.

CVE-2019-19921. This CVE was identified in runC [71] and relies on a race condition between two runC containers that share a volume mount. This exploit attacks a vulnerability in runC where it does not properly control how /proc is mounted in the container allowing the adversary to fully escape by mounting the /proc filesystem using a directory on the volume mount. To exploit this vulnerability, the adversary prepares two containers A and B. This complies with the threat model as we do not limit the number of containers in which an adversary may gain execution.

First, container A creates a symlink from /proc to /evil/level1 and specifies a volume mounted to /evil. At the same time, Container B also has a volume mounted to /evil. Then, Container B swaps /evil/level1 with /evil/level1~ on a continuous loop. Finally, container A continuously reruns and tries to access the host procfs at /evil/level1~/level2. On success,

container A will have access to the host procfs through the /evil/level1/~/level2 directory. This access enables the adversary to escape the container.

CVE-2020-15257. This CVE was identified in containerd [72], and enables the adversary to fully escape the container by leveraging the containerd API UNIX abstract socket from within the container. While the exploit requires that the container executes with host networking, this is often enabled if the administrator wants to increase the network performance of the executing container [73]. As the container has access to host networking, the adversary can connect to the containerd abstract socket, and issue API commands to containerd. Using this channel to control containerd, the adversary sends create/start API commands to spawn a new container in the host namespace, unconstrained by Apparmor, seccomp, and running with all capabilities. With this newly spawned container (i.e., root process) the adversary now has access to the full system. This exploit is mainly an issue with the unsecured abstract UNIX API socket and was patched by running the API socket as a normal UNIX file path socket.

CVE-2020-2023-25-26. These three CVEs were identified in Kata [74] [75] [76]. This exploit is the most complicated of the nine since the container executes in a virtual machine and is constrained by kernel isolation/security mechanisms on the virtual machine (VM). To fully escape, the attacker has to first escape the container on the guest, then onto the host. Accordingly, this exploit takes place in three stages, one for each CVE: (1) to escape the container onto the virtual guest (2023), (2) to infect the underlying VM image (2025), and (3) to escape the VM through a shared host directory (2026). To exploit the first CVE (2023), the adversary overwrites a binary on the guest filesystem from within the container running on the virtual guest. To accomplish this, the adversary first determines the virtual guest's root filesystem block device numbers. Using the mknod syscall the adversary creates a device file for the guest root filesystem on the container. Then, the adversary uses debugfs to overwrite the Kata-agent binary on the container with faulty data and the "shutdown" binary with a malicious exploit. Next, to clear the working Kata-agent binary from the VM page cache, the adversary allocates memory until the Kata-agent binary in memory is replaced with the faulty one the adversary overwrote on the filesystem. Once this replacement

occurs, the VM attempts to shut down and loads the malicious "shutdown" binary which executes indicating the adversary has control over the guest machine.

Next, by modifying the virtual guest during the previous exploit, the second CVE is already exploited(2025), since Kata VM images are reused in future container execution with whatever modifications occurred during the previous execution. Because the adversary controlled the modification in the previous exploit, they can use this ability to modify the underlying guest operating system to control the next initialization of the Kata container VM.

Finally, the last CVE enables the adversary to escape the VM during VM initialization (2026). Since the adversary has already compromised the VM guest image through the previous two steps, the adversary is able to control Kata container initialization performed by Kata-agent on the guest and Kata-shim on the host. Kata agent is responsible for managing the containers on the virtual guest, while Kata-shim communicates container commands to the Kata-agent on the virtual guest. To exploit this CVE, the adversary grabs the container id sent to Kata-agent during container initialization. This allows the adversary to predict a path to the shared directory that communicates files between the host Kata-shim and the container Kata-agent. The adversary creates a symlink from a file path on the shared host directory (/run/kata-containers/shared/containers/$CONTAINER_ID/) to an arbitrary path on the host (/ for the root file system). Continuing the startup process, the Kata-shim then attempts to bind mount the symbolic link set in the shared directory to the VM guest. This symbolic link is followed to the root file system, and the Kata-shim mounts the host filesystem inside the VM. Thus the adversary gains read/write access to the root host filesystem and fully escapes the virtualized container. Even though the container executes in a VM because the container host syncs resources between the VM and the host this opens up a threat on the host the adversary can exploit to gain access to the host file system (i.e., abusing the shared directory to read/write the host filesystem).

5.5.3 Host Execution in Container Context

This section details the three exploits that lead to container escapes due to container utilities that execute in the context of the container. These exploits are associated with CVE-2018-15664, CVE-2019-14271, and CVE-10144,45,47 respectively.

CVE-2018-15664. This CVE was identified in Docker [77] and enables the adversary to escape due to a TOCTOU (time of check, time of use [78]) vulnerability involving a container file path. Docker cp is a container utility that enables administrators to move files into and out of a container and is designed to mirror the Linux utility "cp" [79]. If during the execution of Docker-cp the container target is changed to a symlink, the symlink is evaluated on the host. Thus by manipulating the target on the container, the adversary can gain read/write to an arbitrary host target and escape the container. To detail the vulnerability, first, the adversary executes a malicious binary in the container to run the TOCTOU attack against a directory the administrator may wish to copy out of the container (e.g., "/var/www/html"). The attack runs a continuous while loop to swap the path /var/www/html with a symlink to /. The attack will succeed if Docker-cp opens the /var/www/html path and later writes on the same path, which has been changed into a symlink pointing to "/" causing the write to happen on the host rootfs mounted at "/". In the next step, the administrator runs a command to copy the ./html directory on the host into the container. (e.g., Docker-cp ./html/ webserver:/var/www/html/). If during execution the TOCTOU attack succeeds, then the adversary controls a write onto the host and successfully escapes the container.

CVE-2019-14271. This CVE was identified in Docker [80] and enables the adversary to fully escape the container because the Docker-cp host binary is executed in the context of the container. This is relatively similar to CVE-2019-5736, however, the execution of the host binary is a flaw in Docker-cp and not the result of a leaked file descriptor. To exploit this vulnerability, the adversary sets up a malicious libnsss.so inside the container which executes a malicious binary inside the container (e.g., /evil). Then, when the administrator executes Docker-cp, Docker-cp executes a subprocess (Docker-tar) in the context of the container, loading the malicious libnsss.so and executing the /evil binary with the same permissions as the Docker-cp host binary (most likely root permissions). Next, the /evil binary mounts the

host /proc filesystem, giving the adversary the ability to read/write files arbitrarily on the host. With access to the host /proc, the adversary can escape the container.

CVE-2019-10144,45,46. These three CVEs were identified in rkt [81], and enable the adversary to fully escape the container by controlling the execution of the "rkt enter" utility in the container. Rkt enter executes a specified binary on the container (the entry point by default) without the constraints of cgroups, seccomp, and all 38 Linux capabilities. Since the adversary can control the content of any binary in the container, this means they can escape by editing the binary used by the administrator (e.g., "/bin/bash"), or editing a shared object file the binary would load during the execution. To detail the exploit, first, the adversary modifies libc.so.6 so that when loaded, it mounts the host filesystem. Then, when the administrator executes "rkt enter" (the /bin/bash command by default) to spawn a new shell in the container, the new shell loads libc.so. This triggers the exploit embedded in libc.so to create a block device of the host root filesystem on the container using the mknod syscall. This is possible since the shell is not constrained by security or isolation primitives. Finally, the exploit finishes execution by mounting the host filesystem in the container. This gives the adversary full access to read/write the host filesystem from within the container and thus the adversary escapes the container.

5.6 Survey Results and Conclusions

In summary, this survey investigated 11 runtimes and their 59 CVEs from the NVD database from which denial of service and duplicate CVEs were removed leaving a remainder of 50 CVEs. Because CVE descriptions lack technical details, only the 28 CVEs with publicly available PoC exploits were further analyzed. These 28 CVEs were categorized into a seven-class taxonomy based on the cause and impact of each CVE's respective exploit. This taxonomy revealed that the major category of container runtime vulnerabilities yields a full container escape. In the full container escape category, the 13 CVEs and their corresponding nine exploits all occur because of host components that are exposed in the container. We analyzed each exploit and discovered that the host component exposure occurs in one of three ways: (1) file descriptor mishandling, (2) runtime components missing access control,

and (3) host execution in the context of the container. In the next chapter, we propose two possible defenses which cover each of these three issues to prevent exposing host components in the container.

6. DISCUSSION

The results of the container runtime vulnerability survey show that while containers provide working security and isolation primitives there are still issues that result in complete container escapes. The main reason container runtime escapes occur is due to a host component exposed in the container. To prevent this exposure, we propose two possible defenses for future research: (1) user namespace isolation, and (2) host file monitoring.

6.1 User Namespace Isolation

Running each container in its separate user namespace should prevent container runtime exploits from executing successful escapes. Running a container in its user namespace maps the root user of the container to a non-root user on the host [9]. This would prevent two of the three issues from exposing a host component in the container: (1) Host file descriptors would be owned by a different user on the host and therefore not accessible by the container process. (2) Host runtime component access could not occur within the container as a different user would run them on the host than the user attempting to access them in the container. (3) Host execution in the container context would not be prevented directly by using user namespaces to isolate the container since the adversary always has control over the container contents. Integrating the user namespace defense is non-trivial to implement as over 58% of containers execute as the root user by default [82], and LXC is the only runtime to execute containers as a non-root user by default [83]. Despite this setback, this defense was demonstrated by Anton [84] to mitigate CVE-2019-5736. He showed in his thesis that user namespaces could be used to stop this vulnerability, however, he did not show it could stop the five other applicable exploits explored in this thesis. Applying this defense to the six corresponding exploits would define a well-rounded defense.

6.2 Host File Monitoring

Normally, MAC constraints prevent containers from accessing host files, however in the case of the container escapes demonstrated in Section 5.5, kernel MAC mechanisms fail

to protect against illicit host access. In the case of this failure, container-aware system monitoring tools like sysdig [85] could monitor sensitive directories (e.g., /bin, /proc) on the host and alert on illicit activity. However, no current open source solutions exist to easily take these logs and stop malicious activity as its detected. Building this type of prevention system would stop all three issues highlighted in the survey from executing: (1) Host file descriptors used by containers could be closed when accessed by container processes. (2) Runtime components could be monitored and all processes accessing them without authorization could be killed. (3) Host binaries that execute in container namespaces could be killed before running, to prevent host execution in a container context. Implementing an active defense that kills malicious activity as it is detected would be a useful defense to prevent future container escapes like those identified by the survey.

7. CONCLUSIONS

We present a survey over 11 container runtimes and their corresponding 59 CVEs to provide insight into why container runtime vulnerabilities occur and the impact their corresponding exploits have on the container host. As CVE descriptions alone do not provide enough details for further analysis, only the 28 container runtime CVEs with PoC exploits were analyzed further. To analyze each CVE's PoC, the exploit was broken into high-level attacker commands called steps. By examining each of the 28 CVEs in terms of their corresponding exploit steps, we constructed a 7 class taxonomy that revealed that 46% of the CVEs had exploits that lead to a container escape. Since container escapes were the most occurring category covering twelve CVEs, we conducted an additional investigation over the eight PoC container escape exploits in this category. In this final investigation, we demonstrated that the main reason container escapes occur through container runtimes is the exposure of a host component in the container. This occurs in three ways: (1) File descriptor mishandling, (2) Runtime components missing access control, and (3) Host execution in the context of the container. By highlighting the vulnerabilities exposed in container runtimes, especially container escapes, this survey provides a useful resource for future container security research.

REFERENCES

[1] A. Braun, *Privileged docker containers*, 2016. [Online]. Available: http://obrown.io/2016/02/15/privileged-containers.html.

[2] T. of Bits, *Understanding docker container escapes*, 2019. [Online]. Available: https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/.

[3] N. Stoler, *The route to root: Container escape using kernel exploitation*, 2019. [Online]. Available: https://www.cyberark.com/resources/threat-research-blog/the-route-to-root-container-escape-using-kernel-exploitation.

[4] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18, San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 418–429, ISBN: 9781450365697. DOI: 10.1145/3274694.3274720. [Online]. Available: https://doi.org/10.1145/3274694.3274720.

[5] Y. Wu, L. Lei, Y. Wang, K. Sun, and J. Meng, "Evaluation on the security of commercial cloud container services," in *Information Security*, W. Susilo, R. H. Deng, F. Guo, Y. Li, and R. Intan, Eds., Cham: Springer International Publishing, 2020, pp. 160–177, ISBN: 978-3-030-62974-8.

[6] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018, ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2018.03.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366417300956.

[7] Google, *Google kubernetes engine*, 2021. [Online]. Available: https://cloud.google.com/kubernetes-engine/.

[8] Docker, *Docker products*, 2021. [Online]. Available: https://www.docker.com/products.

[9] L. Kernel, *Namespaces linux man page*. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[10] L. Kernel, *Cgroups linux man page*. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[11] Docker, *Runtime options with memory, cpus, and gpus*. [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/.

[12] R. Developers, *What is selinux?* [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-selinux.

[13] A. developers, *Apparmor*. [Online]. Available: https://www.apparmor.net/.

[14] Wikipedia, *Linux security modules*. [Online]. Available: https://en.wikipedia.org/wiki/Linux_Security_Modules.

[15] L. Kernel, *Seccomp linux man page*. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html.

[16] Systutorials, *Lxc.container.conf*. [Online]. Available: https://www.systutorials.com/docs/linux/man/5-lxc.container.conf/.

[17] M. project Github contributors, *Seccomp_default.json*. [Online]. Available: https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.

[18] L. Kernel, *Capabilities man page*. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html.

[19] G. S. Forums, *False boundaries and code execution*. [Online]. Available: https://forums.grsecurity.net/viewtopic.php?f=7&t=2522.

[20] O. C. Initiative, *About the open container initiative*. [Online]. Available: https://opencontainers.org/about/overview/.

[21] L. developers, *Lxcri github readme*. [Online]. Available: https://github.com/lxc/lxcri.

[22] O. C. Initiative, *Open container initiative runtime specification*. [Online]. Available: https://github.com/opencontainers/runtime-spec.

[23] O. C. Initiative, *Runc github*. [Online]. Available: https://github.com/opencontainers/runc.

[24] C. developers, *Containerd github*. [Online]. Available: https://github.com/containerd/containerd.

[25] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," *Procedia Computer Science*, vol. 175, pp. 157–164, 2020.

[26] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 1073–1086, ISBN: 9781450367479.

[27] L. Allodi and F. Massacci, "Comparing vulnerability severity and exploits using case-control studies," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 1, Aug. 2014, ISSN: 1094-9224. DOI: 10.1145/2630069. [Online]. Available: https://doi.org/10.1145/2630069.

[28] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1416–1432. DOI: 10.1109/SP40000.2020.00061.

[29] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "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.

[30] Y. Wang, Q. Wang, X. Chen, D. Chen, X. Fang, M. Yin, and N. Zhang, "Containerguard: A real-time attack detection system in container-based big data platform," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2020. DOI: 10.1109/TII.2020.3047416.

[31] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A study on the security implications of information leakages in container clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 174–191, 2021. DOI: 10.1109/TDSC.2018.2879605.

[32] L. developers, *Lxc github*. [Online]. Available: https://github.com/lxc/lxc.

[33] D. developers, *Docker github*. [Online]. Available: https://github.com/docker/engine.

[34] C.-O. developers, *Cri-o github*. [Online]. Available: https://github.com/cri-o/cri-o.

[35] S. developers, *Singularity github*. [Online]. Available: https://github.com/hpcng/singularity.

[36] G. developers, *Gvisor github*. [Online]. Available: https://github.com/google/gvisor.

[37] R. developers, *Rkt github*. [Online]. Available: https://github.com/rkt/rkt.

[38] C. developers, *Crun github*. [Online]. Available: https://github.com/containers/crun.

[39] P. developers, *Podman github*. [Online]. Available: https://github.com/containers/podman.

[40] K. developers, *Kata github*. [Online]. Available: https://github.com/kata-containers/kata-containers.

[41] NIST, *National vulnerbility database*, 2021. [Online]. Available: https://nvd.nist.gov/.

[42] MITRE, *Terminology*, 2021. [Online]. Available: https://cve.mitre.org/about/terminology.html.

[43] N. developers, *Nvd tools*, https://github.com/facebookincubator/nvdtools, 2021.

[44] NIST, *Official common platform enumeration (cpe) dictionary*, 2021. [Online]. Available: https://nvd.nist.gov/products/cpe.

[45] FIRST, *Common vulnerability scoring system version 3.1: Specification document*, 2021. [Online]. Available: https://www.docker.com/products.

[46] NIST, *Cve-2018-19295 detail*, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-19295.

[47] MITRE, *Mitre att&ck*, 2021. [Online]. Available: https://attack.mitre.org/.

[48] L. Schabel, *Cve-2019-16884*, https://github.com/opencontainers/runc/issues/2128, 2019.

[49] M. Justicz, *Cve-2018-19333*, 2018. [Online]. Available: https://justi.cz/security/2018/11/14/gvisor-lpe.html.

[50] Capsule8, *Cve-2019-14891*, 2019. [Online]. Available: https://capsule8.com/blog/oomypod-nothin-to-cri-o-bout/.

[51] R. Security, *Cve-2020-14298*, 2020. [Online]. Available: https://access.redhat.com/security/cve/CVE-2020-14298.

[52] R. Security, *Cve-2020-14300*, 2020. [Online]. Available: https://access.redhat.com/security/cve/CVE-2020-14300.

[53] B. P. Adam Iwaniuk, *Cve-2019-5736*, 2019. [Online]. Available: https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html.

[54] A. Sarai, *Cve-2016-9962*, 2016. [Online]. Available: https://bugzilla.suse.com/show_bug.cgi?id=1012568.

[55] J. Horn, *Cve-2017-5985*, 2016. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1654676.

[56] M. Hales, *Suid binaries*, 2019. [Online]. Available: https://recipeforroot.com/suid-binaries/.

[57] D. L. Christian Brauner Serge Hallyn, *Lxc-user-nic*, 2021. [Online]. Available: https://linuxcontainers.org/lxc/manpages/man1/lxc-user-nic.1.html.

[58] P. security team, *Cve-2018-10982*, 2018. [Online]. Available: https://github.com/moby/moby/pull/37404.

[59] J. Horn, *Cve-2018-10982*, 2018. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1632.

[60] M. Gerstner, *Cve-2018-6556*, 2018. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1783591.

[61] M. Gerstner, *Cve-2019-11328*, 2019. [Online]. Available: https://www.openwall.com/lists/oss-security/2019/05/16/1.

[62] M. Gerstner, *Cve-2018-19295*, 2018. [Online]. Available: https://www.openwall.com/lists/oss-security/2018/12/12/2.

[63] A. Hughes, *Cve-2019-13847*, 2019. [Online]. Available: https://github.com/hpcng/singularity/security/advisories/GHSA-m7j2-9565-4h9v.

[64] S. Developers, *The singularity image format (sif)*, 2020. [Online]. Available: https://github.com/hpcng/sif.

[65] E. Stalmans, *Cve-2019-13139*, 2019. [Online]. Available: https://staaldraad.github.io/post/2019-07-16-cve-2019-13139-docker-build/.

[66] D. Developers, *Run the docker daemon as a non-root user (rootless mode)*, 2021. [Online]. Available: https://docs.docker.com/engine/security/rootless/.

[67] S. Seeley, *Cve-2020-15514*, 2018. [Online]. Available: https://srcincite.io/blog/2018/08/31/you-cant-contain-me-analyzing-and-exploiting-an-elevation-of-privilege-in-docker-for-windows.html.

[68] C. de Dinechin, *Cve-2020-27151*, 2020. [Online]. Available: https://bugs.launchpad.net/katacontainers.io/+bug/1878234.

[69] R. Fiedler, *Cve-2016-8649*, 2016. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1639345.

[70] D. Lezcano, *Lxc-attach*, 2021. [Online]. Available: https://linuxcontainers.org/lxc/manpages/man1/lxc-attach.1.html.

[71] L. Schabel, *Cve-2019-19921*, 2019. [Online]. Available: %5Curl%7Bhttps://github.com/opencontainers/runc/issues/2197%7D.

[72] J. Dileo, *Cve-2020-15257*, 2020. [Online]. Available: https://research.nccgroup.com/2020/12/10/abstract-shimmer-cve-2020-15257-host-networking-is-root-equivalent-again/.

[73] D. Developers, *Use host networking*, 2021. [Online]. Available: https://docs.docker.com/network/host/.

[74] Y. Avrahami, *Cve-2020-2023*, 2020. [Online]. Available: https://github.com/kata-containers/community/blob/master/VMT/KCSA/KCSA-CVE-2020-2023.md.

[75] Y. Avrahami, *Cve-2020-2025*, 2020. [Online]. Available: https://github.com/kata-containers/community/blob/master/VMT/KCSA/KCSA-CVE-2020-2025.md.

[76] Y. Avrahami, *Cve-2020-2026*, 2020. [Online]. Available: https://github.com/kata-containers/community/blob/master/VMT/KCSA/KCSA-CVE-2020-2026.md.

[77] A. Sarai, *Cve-2018-15664*, 2018. [Online]. Available: https://bugzilla.suse.com/show_bug.cgi?id=1096726.

[78] Wikipedia, *Time of check time of use*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use.

[79] D. Developers, *Docker cp*, 2021. [Online]. Available: https://docs.docker.com/engine/reference/commandline/cp/.

[80] Y. Avrahami, *Cve-2019-14271*, 2019. [Online]. Available: https://unit42.paloaltonetworks.com/docker-patched-the-most-severe-copy-vulnerability-to-date-with-cve-2019-14271/.

[81] Y. Avrahami, *Cve-2018-10144,10145,10147*, 2018. [Online]. Available: https://unit42.paloaltonetworks.com/breaking-out-of-coresos-rkt-3-new-cves/.

[82] Sysdig, *2020 container security snapshot*. [Online]. Available: https://sysdig.com/blog/sysdig-2020-container-security-snapshot/.

[83] C. Brauner, *Runtimes and the curse of the privileged container*, 2019. [Online]. Available: https://brauner.github.io/2019/02/12/privileged-containers.html.

[84] A. Semjonov, "Security analysis of user namespaces and rootless containers," bachelorThesis, Technische Universität Hamburg, 2020. DOI: 10.15480/882.3089. [Online]. Available: http://hdl.handle.net/11420/7891.

[85] S. Developers, *Welcome to sysdig!* 2021. [Online]. Available: https://github.com/draios/sysdig.

A. APPENDIX

A.1 CPEID LIST

While there are 12 CPEIDs on the list two correspond to the kata container runtime. This is the list that was used to query for all the CVES associated with 11 runtimes

1. `cpe:2.3:a:crun_project:crun:*:*:*:*:*:*:*:*:*`
2. `cpe:2.3:a:docker:docker:*:*:*:*:*:*:*:*:*`
3. `cpe:2.3:a:google:gvisor:*:*:*:*:*:*:*:*:*`
4. `cpe:2.3:a:katacontainers:kata_containers:*:*:*:*:*:*:*:*:*`
5. `cpe:2.3:a:katacontainers:runtime:*:*:*:*:*:*:*:*:*`
6. `cpe:2.3:a:kubernetes:cri-o:*:*:*:*:*:*:*:*:*`
7. `cpe:2.3:a:linuxcontainers:lxc:*:*:*:*:*:*:*:*:*`
8. `cpe:2.3:a:linuxfoundation:containerd:*:*:*:*:*:*:*:*:*`
9. `cpe:2.3:a:linuxfoundation:runc:*:*:*:*:*:*:*:*:*`
10. `cpe:2.3:a:podman_project:podman:*:*:*:*:*:*:*:*:*`
11. `cpe:2.3:a:redhat:rkt:*:*:*:*:*:*:*:*:*`
12. `cpe:2.3:a:sylabs:singularity:*:*:*:*:*:*:*:*:*`

A.2 EXPLOIT STEPS

CVE-2016-8649.

1. attacker constructs fake /proc in container
2. the attacker bind mounts to the fake /proc
3. the administrator executes “lxc-attach”

4. ptrace lxc-attach process to get host file descriptor to entry binary
5. rexec file descriptor with execve

CVE-2016-9962.

1. the container initializes and the init process executes
2. a malicious container executes a ptrace on the init process during container initialization
3. ptrace enables the copy of a host fd
4. read/write files on the host using the open file descriptor

CVE-2018-15664.

1. the attacker embeds malicious executable inside a container that executes a TOCTOU symlink-swap attack against a directory the user seeks to copy (e.g., "/var/www/html").
2. the container starts and executes the symlink swap, which runs a while loop to continuously swap the path "/var/www/html/" with a path on the container and a symlink to "/"
3. the administrator attempts to copy a directory from the container with "docker cp ./html/ webserver:/var/www/html/"
4. during the copy the attacker switches the directory /var/www/html into a symlink "/"
5. the docker-cp utility to write to the symlink on the host

CVE-2019-5736.

1. replace container entry point (bash) to /proc/self/exe
2. save malicious.so with a new function to overwrite host runtime engine onto container
3. /proc/self/exe (the host container runtime binary) executes in the container on startup

4. runtime loads the malicious.so on the container
5. malicious.so overwrites container_runtime with an adversary controlled binary /evil
6. spawning new containers will execute evil, code execution achieved

CVE-2019-19921.

1. rootfs of container A has a symlink /proc -> /evil/level1
2. launch container A specifying volume /evil
3. container B, started before container A, shares this named volume and repeatedly swaps /evil/level1 and /evil/level1
4. container A mounts procfs to /evil/level1 /level2, but when it remounts /proc/sys, it does so at /evil/level1/level2/sys
5. container A has access outside the container

CVE-2019-14271.

1. attacker sets up malicious libnss to execute /evil in the container
2. administrator executes docker-cp (executing docker-tar)
3. docker-tar loads the malicious libnss.so and executes /evil
4. /evil mounts the host /proc filesystem
5. the adversary has arbitrary read/write to the host

CVE-2020-15257.

1. an attacker compromises a container executing in the host network namespace
2. the attacker connects to the containerd abstract socket
3. the attacker issues create/start API commands to spawn a root-id proc on the host (non-ns, non-apparmor, non-seccomp, all-caps)

4. the attacker has root access and controls the system

CVE-2020-10144,45,47.

1. adversary modifies shared object file loaded by container's entry point (evil.so)
2. administrator executes rkt enter
3. bash executes evil in the shared object file
4. evil.so runs mknod to create host filesystem block
5. evil.so mounts the mknod block
6. adversary can edit the host filesystem

CVE-2020-2023.

1. find the guest root filesystem device major and minor numbers
   by inspecting /sys/dev/block.
2. Use mknod to create a device file for the guest root filesystem device
3. access device file and modify guest filesystem with debugfs
4. malloc loop to overwrite files in memory (kata-agent/systemd-shutdown)

CVE-2020-2025.

1. guest images share file changes
2. execute cve-2020-2023 to change the filesystem from the container

CVE-2020-2026.

1. create symbolic link in "/run/kata-containers/shared/containers/${ctrid}/rootfs" to
   host directory path
2. on startup kata-runtime gets directory setup as symbolic link
3. kataruntime mounts /run/kata-containers/shared/sandbox/$ctrid/rootfs
   to the host directory