Towards Improving Container Security by Preventing Runtime Escapes

Michael Reeves*¶, Dave (Jing) Tian†, Antonio Bianchi†, Z. Berkay Celik†
* Sandia National Laboratories, mreeve@sandia.gov
†Purdue University, {daveti, antoniob, zcelik}@purdue.edu

***Abstract*—Container escapes enable the adversary to execute code on the host from inside an isolated container. These high severity escape vulnerabilities originate from three sources: (1) container profile misconfigurations, (2) Linux kernel bugs, and (3) container runtime vulnerabilities. While the first two cases have been studied in the literature, no works have investigated the impact of container runtime vulnerabilities. In this paper, to fill this gap, we study 59 CVEs for 11 different container runtimes. As a result of our study, we found that five of the 11 runtimes had nine publicly available PoC container escape exploits covering 13 CVEs. Our further analysis revealed all nine exploits are the result of a host component leaked into the container. We apply a user namespace container defense to prevent the adversary from leveraging leaked host components and demonstrate that the defense stops seven of the nine container escape exploits.**

***Index Terms*—containers, security study, vulnerability analysis**

I. Introduction

Since containers are lightweight OS-level virtualization, they enable organizations to scale and develop software solutions in the cloud-native era. Containers accomplish this by defining a preconfigured environment enabling developers to focus on writing applications without worrying about dependencies or complicated software setups. In addition, containers are designed to securely run untrusted (unverified and possibly malicious code) code. To constrain this untrusted code, containers leverage kernel isolation and kernel security mechanisms to prevent application vulnerabilities from impacting container hosts. For example, if a web server running in a container has a Remote Code Execution (RCE) vulnerability, exploiting the vulnerability will only gain code execution inside the container and not on the host. However, there exist high severity vulnerabilities that can be used to defeat the isolation and security mechanisms constraining the container. Exploits that target these vulnerabilities are known as "container escapes" since an adversary is able to execute code on the host from inside the container.

There are three main types of vulnerabilities that adversaries exploit to escape a container: (1) Profile misconfiguration, (2) Linux kernel bugs, and (3) Container runtime bugs. The profile misconfiguration vulnerabilities occur when the isolation and security mechanisms of the containers are not properly secured or are dropped entirely to ease the deployment process (e.g., running a privileged Docker container [1]). Exploiting this vulnerability enables the adversary to escape easily and execute code on the host since a container lacking constraints has the same host access as a normal system process [2]. Linux kernel bugs occur when the adversary can leverage a kernel exploit from inside the container to execute arbitrary code in the kernel, disable the container constraints, and escape the container [3]. Lastly, container runtime bugs occur when the adversary is able to exploit a vulnerability in the container runtime itself to escape. For example, CVE-2019-5736 exploits an error in the runC container runtime's mishandling of /proc file descriptors to overwrite the runC binary on the host. When the host attempts to start a new container, the malicious code in the new runtime binary is executed, and the adversary successfully executes code on the host. Thus, a successful "container escape" is executed (See Section IV-D for a full description). There are no prior works that have explored container runtime vulnerabilities, but misconfiguration and kernel escapes have been studied previously [4]–[6].

Contributions. In this paper, we make three contributions. First, we conducted a security study over 11 container runtimes and their corresponding 59 CVEs to understand why container runtime vulnerabilities occur and what impact the corresponding exploits have on the container host. Since CVEs do not include technical details but only a short paragraph description of the vulnerability, we determined a working Proof of Concept (PoC) exploit for a CVE is required to fully analyze the impact of the CVE on the container host. After querying publicly available search services and repositories, we found only 28 CVEs that have PoC exploits. To analyze the cause and effect of each CVE's PoC exploit, we designed a framework to analyze each PoC exploit according to its "steps", the high-level commands executed by the adversary.

Second, we created a seven-class taxonomy of the 28 CVEs and determined the underlying cause of container escapes. To create the taxonomy, we categorized 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. After further investigation of the nine PoC escape exploits, we found the cause of these container escapes is a host component exposed in the container via the vulnerability.

Third, leveraging the insight gained from the security study, we deployed a defense to prevent adversaries from utilizing the host components leaked into a container. To implement the defense, we applied the user namespace kernel feature to

¶All work was conducted during the author's MSc at Purdue University.

map the container's root user as a regular user on the container host. As each host component is owned by a user on the host, the container should not be able to access any host component. Applying this defense to the nine escape exploits, seven escapes are stopped.

II. Background

A. Container 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 the consumption of system resources by grouping processes into "control" groups.

Namespaces. Namespaces (NS) confine processes into groups to limit access to 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 [7]. They isolate different aspects of a process. Specifically, they isolate a process': Cgroups, Inter-process Communication (IPC) objects, network stack, mount points, Process ID (PID), 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., the processes running in that container). Figure 1 shows how PID namespaces can be used to isolate the processes of separate containers by assigning each container a unique PID namespace. The container with host *PID 10040* (designated by the runC process 10040) which spawns in *PID NS 1* cannot see the init process (or any other process) of the container with host *PID 11040* spawned in *PID NS 2*. In addition, both containers in *PID NS 1* or *PID NS 2* cannot see any processes in the original host namespace.

As another example, the user namespace isolates the container from the host by mapping the root ID of the container to a regular user on the host. This can prevent a malicious process that gains root within the container from executing with root permissions on the host. Applying this user namespace to a malicious container would prevent the runC CVE-2019-5736 (as discussed in the Introduction) from executing. We discuss the user namespace defense in further detail in Section V.

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 resources [8]. For containers, the three main cgroups are CPU, memory, and I/O, limiting 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 [9].

B. Container Security Mechanisms

The Linux kernel provides three security mechanisms to secure the host against malicious execution in containers:

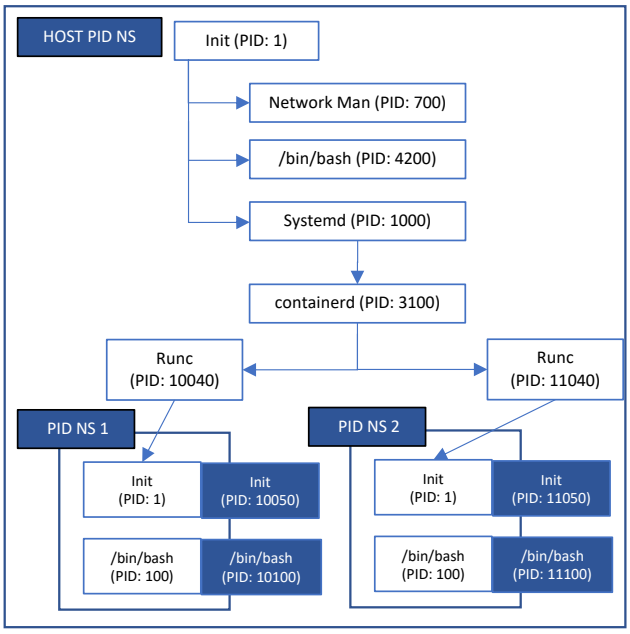

Fig. 1 Visualization of PID namespaces. Each new namespace has its own hierarchy of PIDs and cannot see the other PIDs in other PID namespaces.

(1) Mandatory Access Control (MAC), (2) Seccomp, and (3) Capabilities. These are the main protections the container host can use to defend itself from attacks originating in containers.

MAC. MAC policy frameworks are implemented as Linux Security Modules (LSM) [10]. 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 the reliability of the framework. Policy frameworks like SELinux [11] and Apparmor [12] define in-depth rules deeply embedded in the kernel as LSMs and only permit actions defined by the configured policy. Yet, if the policy is not 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.

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 block/allow list of enabled/disabled system calls [13]. For example, a simple list of allowed or denied system calls can be defined in LinuX Containers (LXC [14]), and Docker sets a default seccomp profile of the allowed system calls for each container when one is not provided in the configuration [15].

Capabilities. In modern Linux versions, capabilities (in addition to the traditional user-based permission system) can constrain what users can do. Specifically, they separate root permissions into 38 separate units and create a more fine-grained permission model for the Linux kernel [16]. For example, CAP_NET_ADMIN gives a process the permissions to edit all levels of the network stack, including the system firewall. To isolate containers properly, the capabilities assigned to the container must minimize the allowed functionality.

C. Least Privilege

Used together, container isolation and security mechanisms enable an effective Linux administrator to enforce the principle of least privilege over container applications. The least privilege ensures that each entity in a system is granted the minimum system resources and authorizations that the entity needs to perform its function [17]. If least privilege was applied to each container, all container applications would run as a unique low-privileged user by being: constrained with low CPU consumption in cgroups, limited by unique resource identifiers with namespaces, monitored and controlled through mandatory access control, prevented from running insecure system calls by seccomp, and inhibited with the smallest number of capabilities. In practice, these restrictions are not used optimally; for instance, 58% of containers still run as the default root user [18].

D. Container Architectures

This section discusses how containers are created and executed by exploring how an open container initiative (OCI [19]) compliant container is created, using the Docker container ecosystem as a case study. Since containers are constrained processes executed on the container host, understanding how containers are created and executed will aid in understanding exploits which attack container runtime vulnerabilities. For the most part, container runtimes follow the OCI specification, except LXC, which uses wrappers to enable inter-operability with the specification [20]. This organization defines a specification of requirements for container runtimes and container images to create an industry-standard container [21].

The default runtime for the specification is runC [22], which performs the initialization of the container process. runC initializes an OCI compliant image, and executes the functionality defined by the OCI image's configuration file. While runC executes the container, containerd [23] focuses on managing all the metadata and image files for the container. 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 runs the command "**docker run ubuntu**" to execute an Ubuntu container, this sends a request to the Docker engine. The Docker engine coordinates with Dockerhub [24] 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. In this case, all the runtimes investigated replace runC as the process to execute the container.

III. Threat Model

We assume an administrator operates a Linux-based container host where each container runs a specific application. 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 Container as a Service (CaaS) host providing container instances for clients [25], [24]. All hardware is part of the trusted computing base (TCB), as well as all the software in the Linux kernel. The administrator controls all software on the container host and secures the containers so that adversaries cannot take advantage of misconfiguration vulnerabilities or Linux kernel bugs. For the adversary, we assume they have gained code execution inside the containers the administrator hosts. 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 [26], (2) the service allows the adversary to execute code on the container as a user in the environment (e.g., CaaS), and (3) the administrator, by mistake, downloads and executes a malicious container image. The adversary aims to gain code execution on the container host by exploiting a vulnerability available in the container runtime and successfully escaping the container. The adversary can exploit any vulnerability in the container's runtime accessible within the container. Side-channel and DoS attacks do not impact the integrity of the container host (i.e., they do not enable adversary command execution outside the container), and therefore are out of scope [27]–[30].

TABLE I CVE vulnerabilities for each runtime from 2016 to March 2021. The PoC number designates how many of the CVEs had publicly available proof of concepts.

Runtime	Critical	High	Med	Low
LXC [31]	1 (1 PoC)	3 (0 PoC)	0 (0 PoC)	2 (2 PoC)
Docker [32]	1 (1 PoC)	11(5 PoC)	7 (1 PoC)	0 (0 PoC)
runC [22]	0 (0 PoC)	4 (3 PoC)	0 (1 PoC)	0 (0 PoC)
CRI-O [33]	0 (0 PoC)	1 (0 PoC)	1 (1 PoC)	0 (0 PoC)
Singularity [34]	1 (0 PoC)	8 (3 PoC)	1 (0 PoC)	0 (0 PoC)
gVisor [35]	1 (1 PoC)	0 (0 PoC)	2 (1 PoC)	0 (0 PoC)
rkt [36]	0 (0 PoC)	3 (3 PoC)	0 (0 PoC)	0 (0 PoC)
crun [37]	0 (0 PoC)	1 (0 PoC)	0 (0 PoC)	0 (0 PoC)
Podman [38]	0 (0 PoC)	1 (0 PoC)	2 (0 PoC)	0 (0 PoC)
containerd [23]	0 (0 PoC)	1 (0 PoC)	2 (1 PoC)	0 (0 PoC)
Kata [39]	0 (0 PoC)	3 (3 PoC)	2 (1 PoC)	0 (0 PoC)
Total	4 (3 PoC)	36 (17 PoC)	17 (6 PoC)	2 (2 PoC)

IV. Container Runtime Security Study

In this section, we detail our container security study. First, we describe the collection process of the container runtimes list and their corresponding CVEs. Then, we motivate the need to focus on runtime vulnerabilities with PoC exploits in an initial analysis. To classify each CVE, we create a seven-class taxonomy to understand the cause and impact of each CVE's exploit based on high-level adversary commands called "steps". We find the majority of PoC exploits enable container escapes, and further investigate these nine PoC exploits covering 11 container runtimes. Our study reveals the underlying cause of container escapes exploiting container runtimes: the host component exposed in the container.

A. Data Collection Overview

Container Runtimes. To analyze as many runtimes as possible, we gathered a list of popular runtimes from the recent literature and searching Google with queries for popular container runtimes (e.g., "popular container runtimes"). Table I presents

the list of container runtimes along with respective links to their code repositories.

CVE Data Sources. The NIST National Vulnerability Database (NVD) [40] publishes all the data regarding common vulnerability enumerations (CVE) [41] 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 security study was conducted. To parse this data effectively, we leveraged nvdtools [42], an open-source tool 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" [43]. Thus, we fed the list of runtimes in CPEID format yielding 59 CVEs.

Initial Analysis. The distribution of these CVEs is displayed in Table I. The Critical, High, Medium, and Low categories designate the severity of a vulnerability determined by its respective Common Vulnerability Scoring System (CVSS) score [44]. The ranges for each category are as follows: Critical (> 9), High ($>= 7, < 9$), Medium ($>= 4, < 7$), and Low ($> 0, < 4$). The higher the severity of the vulnerability, the greater the vulnerability's impact on the security of the container host. In addition, every runtime has at least one vulnerability that is high severity or greater, while Docker and Singularity have the greatest number, twelve and nine high and critical vulnerabilities, respectively. The green text in Table I is an overview of all container runtime vulnerabilities, and includes seven DoS vulnerabilities and two repeat vulnerabilities that associated CVE-2019-5736 with the LXC and Docker runtimes. 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.

Mapping CVEs to Exploits. To analyze the impact of the remaining 50 container runtime vulnerabilities, more information must be explored, as CVEs do not contain technical details besides 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 due to exploiting the vulnerability (e.g., gaining host code execution, host network access, or privilege escalation). To gather a list of all PoC for each vulnerability, we built an initial list of exploits from those available in the NVD. Each CVE's JSON data contains referral 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's list of corresponding links. 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 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 (Table I).

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 reasons: (1) Adversaries hold them privately for use as zero-days and do not disclose the exploits after the vulnerabilities are discovered, 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. The vulnerability is still valid in either case, but as discussed previously, it has limited use for further analysis. Therefore, only the 28 CVEs with PoC exploits are explored further. To analyze the 28 CVEs, they were categorized by the cause and impact of their associated exploits.

B. Exploit Analysis Framework

To 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 [45]. If a full binary analysis was used for the framework, access to the exploit's binary 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 important vulnerabilities from the analysis. On the other hand, a higher-level framework would provide more semantic information, such as where the exploit is executed (e.g., inside the container or on the container host), However, important technical commands that are required for an exploit would not be included in the analysis, as the higher-level framework would produce categorical results, losing some detailed information that would be important for analysis (e.g., the exact commands executed by the adversary).

The framework created here takes the middle ground between low-level and high-level. The framework used in this analysis breaks each exploit into high-level adversary commands called "steps". These steps capture the details missing in the high-level adversarial framework but do not require the compiled exploits such as in binary analysis. For further examples in the Appendix, we break down each of the final 13 PoC CVEs into their steps.

C. CVE Taxonomy

Using the steps created with the exploit analyzer framework, we categorized each CVE into one of seven classes based on its cause (i.e., How does the vulnerability enable the adversary to gain the capabilities associated with the impact?) and the impact of its exploit (i.e., What are the capabilities gained by

the adversary from executing the exploit associated with the vulnerability, such as the host code execution, host network access, or host privilege escalation?). This section details six of the seven categories in the CVE taxonomy. As the exploits of the vulnerabilities in these categories do not result in adversary-controlled code execution on the host, we use the term "non-escape" to refer to these six categories. The exploits of the vulnerabilities in the seventh category all result in adversary-controlled code execution on the host, so we call them "container escapes". We explore these escape exploits in Section IV-D. We separate the six non-escape and container escape categories since 46% of the CVEs in the taxonomy fall into the container escape category.

MAC Disabling. 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 [46]. The exploit stops Apparmor from securing the container. However, other security mechanisms (e.g., seccomp) still greatly limit the capabilities of the adversary, so it is not considered a container escape.

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 [47].

Container to Host Network Access. This category includes vulnerabilities that enable the adversary to gain host network access from inside the container. There is one vulnerability in this category: CVE-2019-14891 [48]. This CVE enables the adversary to gain networking capabilities on the container host by gaining control of the resources allocated for the CRI-O container monitoring process.

Unpatched System. This category includes vulnerabilities for runtime packages that regress to old vulnerabilities by faulty or mistaken patches when patched. There are two vulnerabilities in this category: CVE-2020-14298 [49], and CVE-2020-14300 [50]. Since these CVEs result from failed patches that reintroduce old vulnerabilities, they are put into their own category even though the original vulnerabilities lead to container escapes.

Limited Container to Host Access. This category defines vulnerabilities that enable the adversary to gain limited control over the container host from inside the container (e.g., create network interfaces, enable/disable hardware, or discover privileged file paths). There are four exploits in this category: CVE-2017-5985 [51], CVE-2018-10892 [52], CVE-2018-16359 [53], CVE-2018-6556 [54].

Host Privilege Escalation. This category is the second-largest category. It defines vulnerabilities that enable an adversary to gain privilege escalation *assuming they have access to the container host.* There are six CVEs in this category: CVE-2019-11328 [55], CVE-2018-19295 [56], CVE-2020-13847 [57], CVE-2019-13139 [58], CVE-2018-15514 [59], CVE-2020-27151 [60].

Non-Escape Categories Summary. Overall, these six categories cover 15 CVEs that 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 still gain significant capabilities, especially in the host privilege escalation category, where each vulnerability enables a *local user on the host* to get root permissions.

D. Container Escape Exploits

In this section, we detail the remaining category from the CVE taxonomy, the container escape exploits. For this category, the remaining 13 CVEs correspond to nine exploits. The main reason container escapes are able to exploit container runtimes is an exposed host component inside the container. We find 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. Table II 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 the Appendix.

Mishandled File Descriptor. For this issue, the exposure occurs when a file descriptor remains accessible inside the container via the /proc filesystem, which gives the adversary read/write access to the host filesystem. For example, in CVE-2019-5736 [64], the adversary is able to create a reference to the file descriptor of the container runtime. To exploit this CVE, the adversary configures a 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. The malicious.so file overwrites the file descriptor of the executing process that loads it. When the malicious container executes, the symlinked entry points to the host's runC binary since that is where /proc/self/exe points. Thus, this causes the host's runC binary to execute in the container context and load the malicious.so. Then, malicious.so overwrites the host runC binary referenced by /proc/self/exe with a malicious backdoor. Finally, when the administrator next spawns new containers, the malicious backdoor executes on the host signaling the adversary has successfully escaped the container.

Runtime Component Missing Access Control. For this issue, 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 [68], the adversary gains access to the containerd abstract UNIX socket. To exploit this CVE, 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 Linux capabilities. With this newly spawned container (i.e., root process), the adversary now has access to the host system. This exploit occurred mainly from the unsecured abstract UNIX API socket and was patched by running the API socket as a normal UNIX socket.

Adversary-controlled Host Execution. For this issue, 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

TABLE II Container runtime exploits listed by the runtime, highlighting the cause of the escape (✓), the non-causes (✗), the leaked host component, the results of applying user namespaces (✓ for success on exploit failure, and ✗ for exploit success), and the reason the exploit fails or still succeeds.

Runtime	CVE	File Descriptor Mishandling?	Component Missing Access Control?	Host Execution in Container Context?	Host Component Leaked	Exploit Fails	Reason
LXC	2016-8649 [61]	✓	✗	✗	/proc fd	✓	fd access denied
runC	2016-9962 [62]	✓	✗	✗	/proc fd	✓	fd access denied
Docker	2018-15664 [63]	✗	✗	✓	symlink of host file path	✗	chroot executes as root user outside container
runC	2019-5736 [64]	✓	✗	✗	/proc/self/exe	✓	fd acccess denied
runC †	2019-19921 [65]	✗	✓	✗	/proc via a container volume	✗	volume mounts still allow /proc mounting
Docker	2019-14271 [66]	✗	✗	✓	host /proc	✓	fails to mount host /proc
rkt	2019-10147 [67]	✗	✗	✓	host filesystem device	✓	mount fails
	2019-10144 [67]						
	2019-10145 [67]						
containerd*	2020-15257 [68]	✗	✓	✗	abstract UNIX socket	✓	UNIX socket access denied
kata	2020-2023 [69]	✗	✓	✗	container to host shared directory	✓	mknod syscall fails
	2020-2025 [70]						
	2020-2026 [71]						

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

malicious symlink. For example, in CVE-2019-101(44–47) [67] the `rkt-enter` runtime utility executes a command in a rkt container (e.g., "bash") without the constraints of cgroups, seccomp, and Linux capabilities. This enables the adversary to control the execution of this process and escape the container. To exploit these vulnerabilities, 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 access to read/write the host filesystem from inside the container, and thus the adversary escapes the container.

V. Defeating Container Runtime Escape Exploits

A. User Namespace Defense

The Linux user namespace creates arbitrary levels of security-related identifiers on the container host [72] (e.g., userid (UID), groupid (GID), and Linux capabilities). By applying this mechanism to containers using a user namespace, the container user differs from inside and outside the container. It would be simpler to run the container as a non-root user; yet, we assume the adversary is able to gain root inside the container (Section III). To demonstrate how user namespaces separate permissions inside and outside the container, we show how the user namespace prevents the `mknod` syscall from successfully executing on the host in Figure 2.

By default, a container runs as the same user that initializes it. As shown at the top box in Figure 2, when a container running as the root user executes the `mknod` syscall to create a reference to the host file system, the syscall succeeds. However, the regular user will fail since they do not have the proper capability (i.e., CAP_MKNOD), while the root does. In the bottom box in Figure 2, both containers cannot access the filesystem block device even though the user in container A

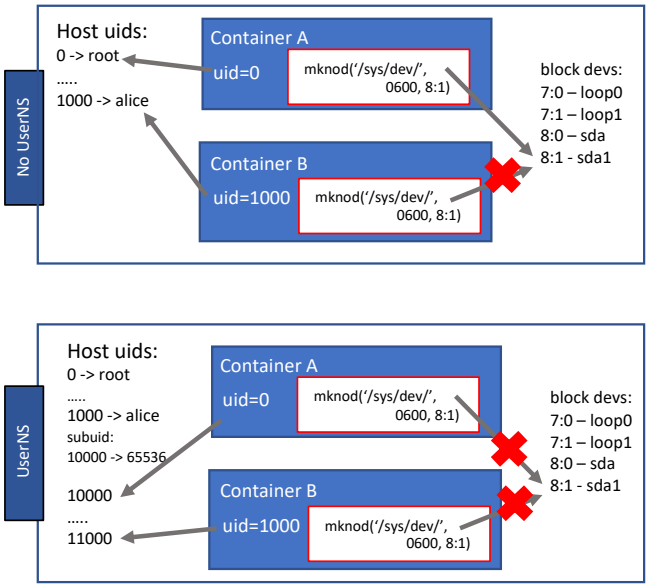

Fig. 2 Visualization of user namespaces (userNS) protecting access to host device files. Without a userNS, the container root ID is equivalent to the host root, but inside a userNS, the root ID is equivalent to a regular host user.

executes as root. When the container UIDs are checked for permissions on the host, they are transformed into the real UID at evaluation time. Therefore, as container A attempts to create a reference to the host filesystem, the host checks the capabilities of UID 10000. Since UID 10000 has no special capabilities, the syscall fails. The same goes for UID 1000 in container B that transforms to 11000 at evaluation time.

For a real-world example, the user namespace prevents the abstract UNIX socket from being accessed by a malicious container in CVE-2020-15257 [68]. Without the defense, the exploit can access the root-owned abstract UNIX socket since the root UID in the container is equivalent to the root UID on the host. However, with the defense, the root user of the container maps to an unprivileged user on the host, and any attempt to access the UNIX socket will fail as the adversary will not have appropriate host permissions.

B. Defense Effectiveness

While the user namespace creates an effective security paradigm for containers, 58% of containers still run as the default root user [18]. Based on this fact and the study results, we decided to apply the user namespace to determine if the vulnerabilities could have been exploited when they were discovered. We found that seven of the nine exploits were stopped by applying user namespaces in each affected runtime. The results are presented in the last two columns of Table II. These results demonstrate that applying user namespaces is a simple and effective defense against container runtime escapes. Yet, two of the exploits continue to succeed after applying the defense because a runtime utility on the host executes as root while depending on adversary-controlled container objects.

As an example, one of the two successful exploits, CVE-2019-19921 [65], utilizes a time-of-check to time-of-use vulnerability (TOCTOU) [73] in runC's volume mount operation. This TOCTOU vulnerability occurs before the container is fully initialized, so the user namespace defense cannot prevent the attack. To execute the exploit, the adversary requires access to two containers and takes advantage of a bug in runC's volume mounting procedure by tricking the container to mount a symlink over the */proc* directory. 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 process filesystem (procfs) at */evil/level1/~/level2*. On success, container A will have access to the host procfs through the */evil/level1/~/level2* directory. With this, the adversary is able to fully escape the container by overwriting files such as */proc/sys/kernel/core_pattern* since the command within *core_pattern* will be executed as root after a process segfaults [74].

We propose the same solution as the runC developers [75], and it requires a fix to the software architecture of runC by modifying runC to mount directories using the file descriptor of the specified mount, rather than a string of the mounted file path. Therefore, the entire race condition is avoided because the file descriptor will always point to the correct file path (and thus, runC will ignore the malicious symlink).

VI. Related Work

A measurement study on exploit execution effectiveness in containers was conducted, highlighting the impact of kernel exploits on containers [4]. The exploits all followed the same attack chain executing `commit_creds` to achieve privilege escalation, so the authors designed a defense based on 10 lines of code into the commit_creds kernel function to prevent these exploits from executing.

Anton also investigates risks and benefits of user-namespace security applied to system vulnerabilities, notably dirtycow (kernel), socksign (web), and runC 2019-5736 (container) [76]. Another work explores vulnerabilities within the Docker ecosystem [6]. This research focuses on Docker misconfiguration and image vulnerabilities. Container runtime vulnerabilities are listed but not discussed. Lastly, Flauzac et al. conducted a comparison over technical aspects of container solutions [77]. These three works all explore vulnerabilities or comparisons under limited settings while we investigate 11 different runtimes and their 59 CVEs. Finally, Allodi et al. conducted a vulnerability risk assessment by comparing discovered CVEs to actively used exploits [26].

VII. Conclusions

As more companies pivot their technology stacks to leverage the benefits of containers, adversaries will continue to exploit available vulnerabilities. In this paper, we conduct a security study over 11 container runtimes and their 59 vulnerabilities. We then present a seven-class taxonomy over the 28 CVEs with publicly available PoC exploits, revealing that the main cause of container escapes is a host component leaked into the container. Finally, we demonstrate that applying a user namespace defense to container runtimes stops seven of the nine escape exploits. This paper presents the first thorough study of container runtime vulnerabilities and demonstrates that applying user namespaces to containers may effectively prevent container escapes.

VIII. Acknowledgments

We want to thank our reviewers and colleagues at Sandia National Labs for their insightful comments and time improving this work. This research was supported in part by Sandia National Laboratories [¶] Critical Skills Masters Program (CSMP). This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

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. New York, NY, USA: Association for Computing Machinery, 2018, p. 418–429. [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.

[6] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366417300956

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

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

[¶]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. - SAND2021-11135J

[9] Docker, "Runtime options with memory, cpus, and gpus." [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/
[10] Wikipedia, "linux security modules." [Online]. Available: https://en.wikipedia.org/wiki/Linux_Security_Modules
[11] R. Developers, "What is selinux?" [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-selinux
[12] A. developers, "Apparmor." [Online]. Available: https://www.apparmor.net/
[13] L. Kernel, "Seccomp linux man page." [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html
[14] Systutorials, "lxc.container.conf." [Online]. Available: https://www.systutorials.com/docs/linux/man/5-lxc.container.conf/
[15] M. project Github contributors, "seccomp_default.json." [Online]. Available: https://github.com/moby/moby/blob/master/profiles/seccomp/default.json
[16] L. Kernel, "capabilities man page." [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html
[17] NIST, "Nist least privilege glossary," 2021. [Online]. Available: https://csrc.nist.gov/glossary/term/least_privilege
[18] Sysdig, "2020 container security snapshot." [Online]. Available: https://sysdig.com/blog/sysdig-2020-container-security-snapshot/
[19] O. C. Initiative, "About the open container initiative." [Online]. Available: https://opencontainers.org/about/overview/
[20] L. developers, "lxcri github readme." [Online]. Available: https://github.com/lxc/lxcri
[21] O. C. Initiative, "Open container initiative runtime specification." [Online]. Available: https://github.com/opencontainers/runtime-spec
[22] Open Container Initiative, "runc github." [Online]. Available: https://github.com/opencontainers/runc
[23] Containerd developers, "containerd github." [Online]. Available: https://github.com/containerd/containerd
[24] Docker, "Docker products," 2021. [Online]. Available: https://www.docker.com/products
[25] Google, "Google kubernetes engine," 2021. [Online]. Available: https://cloud.google.com/kubernetes-engine/
[26] 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. [Online]. Available: https://doi.org/10.1145/2630069
[27] 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.
[28] 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.
[29] 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.
[30] 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, p. 1073–1086.
[31] L. developers, "Lxc github." [Online]. Available: https://github.com/lxc/lxc
[32] D. developers, "Docker github." [Online]. Available: https://github.com/docker/engine
[33] C.-O. developers, "Cri-o github." [Online]. Available: https://github.com/cri-o/cri-o
[34] S. developers, "Singularity github." [Online]. Available: https://github.com/hpcng/singularity
[35] G. developers, "Gvisor github." [Online]. Available: https://github.com/google/gvisor
[36] R. developers, "rkt github." [Online]. Available: https://github.com/rkt/rkt
[37] C. developers, "crun github." [Online]. Available: https://github.com/containers/crun
[38] P. developers, "Podman github." [Online]. Available: https://github.com/containers/podman
[39] K. developers, "Kata github." [Online]. Available: https://github.com/kata-containers/kata-containers
[40] NIST, "National vulnerbility database," 2021. [Online]. Available: https://nvd.nist.gov/
[41] MITRE, "Terminology," 2021. [Online]. Available: https://cve.mitre.org/about/terminology.html
[42] N. developers, "Nvd tools," https://github.com/facebookincubator/nvdtools, 2021.
[43] NIST, "Official common platform enumeration (cpe) dictionary," 2021. [Online]. Available: https://nvd.nist.gov/products/cpe
[44] FIRST, "Common vulnerability scoring system version 3.1: Specification document," 2021. [Online]. Available: https://www.docker.com/products
[45] MITRE, "Mitre att&ck," 2021. [Online]. Available: https://attack.mitre.org/
[46] L. Schabel, "Cve-2019-16884," https://github.com/opencontainers/runc/issues/2128, 2019.
[47] M. Justicz, "Cve-2018-19333," 2018. [Online]. Available: https://justi.cz/security/2018/11/14/gvisor-lpe.html
[48] Capsule8, "Cve-2019-14891," 2019. [Online]. Available: https://capsule8.com/blog/oomypod-nothin-to-cri-o-bout/
[49] R. Security, "Cve-2020-14298," 2020. [Online]. Available: https://access.redhat.com/security/cve/CVE-2020-14298
[50] RedHat Security, "Cve-2020-14300," 2020. [Online]. Available: https://access.redhat.com/security/cve/CVE-2020-14300
[51] J. Horn, "Cve-2017-5985," 2016. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1654676
[52] P. security team, "Cve-2018-10982," 2018. [Online]. Available: https://github.com/moby/moby/pull/37404
[53] J. Horn, "Cve-2018-10982," 2018. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1632
[54] M. Gerstner, "Cve-2018-6556," 2018. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1783591
[55] Matthias Gerstner, "Cve-2019-11328," 2019. [Online]. Available: https://www.openwall.com/lists/oss-security/2019/05/16/1
[56] M. Gerstner, "Cve-2018-19295," 2018. [Online]. Available: https://www.openwall.com/lists/oss-security/2018/12/12/2
[57] A. Hughes, "Cve-2019-13847," 2019. [Online]. Available: https://github.com/hpcng/singularity/security/advisories/GHSA-m7j2-9565-4h9v
[58] E. Stalmans, "Cve-2019-13139," 2019. [Online]. Available: https://staaldraad.github.io/post/2019-07-16-cve-2019-13139-docker-build/
[59] S. Seeley, "Cve-2020-15514," 2018. [Online]. Available: https://bit.ly/3zeq5kk
[60] C. de Dinechin, "Cve-2020-27151," 2020. [Online]. Available: https://bugs.launchpad.net/katacontainers.io/+bug/1878234
[61] R. Fiedler, "Cve-2016-8649," 2016. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1639345
[62] A. Sarai, "Cve-2016-9962," 2016. [Online]. Available: https://bugzilla.suse.com/show_bug.cgi?id=1012568
[63] A. Sarai, "Cve-2018-15664," 2018. [Online]. Available: https://bugzilla.suse.com/show_bug.cgi?id=1096726
[64] B. P. Adam Iwaniuk, "Cve-2019-5736," 2019. [Online]. Available: https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html
[65] L. Schabel, "Cve-2019-19921," 2019. [Online]. Available: https://github.com/opencontainers/runc/issues/2197
[66] 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/
[67] Y. Avrahami, "Cve-2018-10144,10145,10147," 2018. [Online]. Available: https://unit42.paloaltonetworks.com/breaking-out-of-coresos-rkt-3-new-cves/
[68] J. Dileo, "Cve-2020-15257," 2020. [Online]. Available: https://bit.ly/385nKMI
[69] Y. Avrahami, "Cve-2020-2023," 2020. [Online]. Available: https://github.com/kata-containers/community/blob/master/VMT/KCSA/KCSA-CVE-2020-2023.md
[70] Y. Avrahami, "Cve-2020-2025," 2020. [Online]. Available: https://github.com/kata-containers/community/blob/master/VMT/KCSA/KCSA-CVE-2020-2025.md
[71] Y. Avrahami, "Cve-2020-2026," 2020. [Online]. Available: https://github.com/kata-containers/community/blob/master/VMT/KCSA/KCSA-CVE-2020-2026.md
[72] M. Kerrisk, "user_namespaces linux manual page," 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html
[73] Wikipedia, "Time of check time of use," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use
[74] C. M. Peñalver, "Apport (see core_patter)," 2021. [Online]. Available: https://wiki.ubuntu.com/Apport
[75] A. Sarai, "crun follows symlinks when creating mount points," 2019. [Online]. Available: https://github.com/containers/crun/issues/111#issuecomment-536495867

[76] A. Semjonov, "Security analysis of user namespaces and rootless containers," bachelorThesis, Technische Universität Hamburg, 2020. [Online]. Available: http://hdl.handle.net/11420/7891
[77] 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.

A. EXPLOIT STEPS

CVE-2016-8649.

1) The adversary constructs fake /proc in the container
2) The adversary bind mounts to the fake /proc
3) The administrator executes "lxc-attach"
4) The adversary ptrace lxc-attach process to get host file descriptor to the entry binary
5) The adversary uses rexec on the file descriptor with the execve syscall

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) The adversary reads/writes files on the host using the open file descriptor

CVE-2018-15664.

1) The adversary embeds a 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/ web-server:/var/www/html/"
4) During the copy the adversary switches the directory "/var/www/html" into a symlink "/"
5) The docker-cp utility mistakenly will write to the symlink on the host

CVE-2019-5736.

1) The adversary switches the container entry point (/bin/bash) to /proc/self/exe
2) The adversary saves malicious.so with a new function to overwrite the host runtime engine on the container
3) /proc/self/exe (the host container runtime binary) executes in the container on startup
4) The runtime loads the malicious.so on the container
5) The malicious.so overwrites container_runtime with an adversary controlled binary evil
6) Spawning new containers will execute evil, code execution is achieved

CVE-2019-19921.

1) The rootfs of container A has a symlink /proc to /evil/level1
2) The adversary launches 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 to the host /proc, and can execute code outside the container

CVE-2019-14271.

1) The adversary sets up a malicious libnss to execute /evil in the container
2) The 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) The adversary compromises a container executing in the host network namespace
2) The adversary connects to the containerd abstract socket
3) The adversary issues create/start API commands to spawn a root-id proc on the host (non-ns, non-apparmor, non-seccomp, all-caps)
4) The adversary has root access and controls the system

CVE-2020-10144,45,47.

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

CVE-2020-2023.

1) The adversary finds the guest root filesystem device major and minor numbers by inspecting /sys/dev/block.
2) The adversary uses mknod to create a device file for the guest root filesystem device
3) The adversary accesses the device file and modifies the guest filesystem with debugfs
4) The adversary executes a loop of malloc calls to overwrite files in memory (kata-agent/systemd-shutdown)

CVE-2020-2025.

1) The adversary executes CVE-2020-2023 to change the filesystem on the kata virtual machine
2) The malicious changes will be shared across future deployments of the maliciously modified guest filesystem

CVE-2020-2026.

1) The adversary creates a symbolic link in "/run/kata-containers/shared/containers/${ctrid}/rootfs" to host directory path
2) On startup kata-runtime gets directory setup as symbolic link
3) The kata-runtime mounts /run/kata-containers/shared/sandbox/$ctrid/rootfs to the host directory