PACED: Provenance-based Automated Container Escape Detection

Mashal Abbas[1]§, Shahpar Khan[1]§, Abdul Monum[1]§, Fareed Zaffar[1], Rashid Tahir[2]
David Eyers[3], Hassaan Irshad[4], Ashish Gehani[4], Vinod Yegneswaran[4], Thomas Pasquier[5]
[1]LUMS, [2]University of Prince Mugrin, [3]University of Otago, [4]SRI International, [5]University of British Columbia

***Abstract*—The security of container-based microservices relies heavily on the isolation of operating system resources that is provided by namespaces. However, vulnerabilities exist in the isolation of containers that may be exploited by attackers to gain access to the host. These are commonly referred to as container escape attacks. While prior work has identified vulnerabilities in namespace isolation, no general container escape detection and warning system has been presented. We present PACED, a novel, realtime system to detect container-escape attacks. We define what constitutes a cross-namespace event and how such events can be used to detect a container escape attack. We develop a provenance-based approach to isolate cross-namespace events and propose a rule—*privileged_flow*—to detect attacks on Docker and Kubernetes environments. We evaluate our detection method on a suite of contemporary CVEs with container escape exploits, bad container configurations, and benchmarks. PACED achieves near-perfect accuracy with no false negatives. We release our implementation and datasets as free, open source software.**

I. INTRODUCTION

The need for a lightweight alternative to full Virtual Machines (VMs) has led to the resurgence of intra-operating system isolation, i.e., container-based solutions. Container-based microservices are seeing adoption due to their efficient resource usage, scalability, and maintainability at low computational and complexity cost. Lightweight virtualization on operating systems (OSs) allows multiple isolated containers to run over a single kernel, thus reducing resource costs.

For the rest of this paper, we focus on container technology that runs over a Linux kernel. Namespaces and CGroups in the Linux kernel play a vital role in isolating groups of processes and their access to system resources. Namespaces and CGroups are used to define containers' isolation boundaries. However, while this guarantees some isolation, a large part of the kernel remains shared. As a result, container isolation is fragile and can leave the applications running on top of it vulnerable [13, 60].

In this paper, we focus on data flowing beyond containers' isolation boundaries, called *cross-namespace flow*. Although cross-namespace flows are not all malicious, they can be a sign of an attack. A malicious container may be able to access files and tamper with its host or other containers through systematically orchestrated container escape attacks stemming from these cross-namespace flows.

There is a lack of systematic methods to scan for vulnerabilities which leads to tedious detection efforts by the developer community. Currently developers report vulnerabilities in the system and software maintainers implement fixes in newer software versions. This requires community members to be particularly vigilant regarding vulnerabilities, and fixes are only made when a problem has come to attention. This leaves the microservices running on containers vulnerable to attacks that have not yet been reported.

To detect these attacks, provenance has emerged as a useful technique due to its fine-grained tracking of the system activity [7]. Whole-system provenance [17, 40, 53] is the representation of all activities within a system as a directed graph. The graph generated covers interactions between activities (i.e., processes), entities (e.g., files, sockets, pipes etc.) and agents (i.e., users) [51]. Whole-system provenance records have been used to automatically detect intrusion [2, 16, 21, 23, 27, 35, 58] or to perform forensic analysis [15, 18, 24, 30, 33]. Those approaches require relatively complex analysis and significant computing resources. We demonstrate that we can successfully detect anomalous cross-namespace information flow leveraging simple graph queries.

We propose PACED: a solution to detect cross-namespace attacks by collecting provenance and analyzing *cross-namespace events*. Cross-namespace events are those that have a read-write flow from one set of namespaces to another. This means that there exists data transfer between: (i) two containers, or (ii) a container and the host.

After the collection of data from pre-existing Docker CVEs and benchmarks, we develop a single rule to identify these cross-namespace attacks within provenance records. We then explore if our rule is extendable to other container orchestration engines such as Kubernetes. We find that there too our rule yields a recall of 100%.

This paper makes the following contributions:

- We define, in terms of data provenance, what constitutes a cross-namespace event and highlight how such events can be used to detect security breaches caused by containers.
- We design a mechanism to automatically detect container-escape attacks in Docker and Kubernetes runtime environments. We achieve near-perfect accuracy with this rule for detecting such attacks.
- We open-source our extensions to the SPADE provenance system, and our other software artifacts (see Availability Section).
- Finally, we make our datasets openly available (see Availability Section).

§Equal contribution

II. Background

We introduce a number of concepts and technologies necessary to understand the rest of the paper. First, we describe how Linux uses namespace and CGroups to create containers. Second, we explain container escape vulnerabilities. Finally, we provide a brief introduction regarding system provenance.

A. Linux Support for Containerization

Namespaces. A *namespace* is a Linux kernel feature that allows a set of processes to have a private view of the global resources of the host system. A namespace at its root is an attribute that the OS assigns to a process. This attribute limits which resources and abstractions are accessible to the process. Processes should not interact or be affected by other processes that are outside the scope of their assigned namespaces [7]. Namespaces are used to partition system resources, such as file systems, processes, users, and hostnames. Linux namespaces serve as the fundamental OS abstraction that is leveraged by container technologies to provide each container with a unique a view of system resources. The Linux kernel supports eight namespaces at the time of writing. While namespaces are crucial building blocks to control resource access to containers, not all system resources are namespace-aware [57].

Control Groups (CGroups). *CGroups* are another Linux kernel abstraction that limits resource utilization such as central processing unit (CPU) runtime, system memory, input/output (I/O), and network bandwidth [57]. CGroups and their resource limits are organized hierarchically. Every OS process sits somewhere within the CGroups hierarchy. Containers employ the CGroups feature to define a per-CGroup resource limit policy associated with container instances. This ensures accountability of resource usage and prevents a single container from hijacking or exhausting host system resources [13].

Containers. There are multiple container management and orchestration frameworks that allow users to deploy and manage containers. Docker is an open-source platform that is used to build, deploy, and manage containers. It makes use of the Linux kernel namespaces to create an isolation boundary between the containers. The Docker engine is responsible for managing the containers with the help of the underlying Docker daemon which services the API calls and administers the Docker objects [11]. Kubernetes is a container orchestration framework that popularly serves as a cluster manager. A Kubernetes cluster contains a control plane and at least one worker node. Each worker node may contain multiple pods and each pod may contain multiple containers. The hierarchy between the pods and containers is based on the namespace and CGroups associated to them [31].

B. Container Escape

Containers heavily depend on namespaces to achieve virtualization at the OS level. However, the implementation of namespaces in the Linux kernel is incomplete [13]. Inspecting the Linux kernel code, we can find that context checks are missing for existing namespaces, and some Linux subsystems are not (fully) namespaced [13]. This raises significant security and privacy concerns if multiple containers from different tenants share the same OS kernel [14, 60]. These concerns include in-container information leakage channels that may expose information about the host OS and co-resident containers [13]. Furthermore, popular container management and orchestration systems, such as Docker and Kubernetes respectively, have also suffered from severe vulnerabilities which, combined with the incomplete isolation of containers, have led to containers escaping their boundaries and taking control of the host machine in the most severe cases. A malicious adversary can then proceed to take control of the entire multi-tenant cloud. We discuss and evaluate our PACED detection capabilities against such attacks in § VI.

C. Provenance and its Capture

Data provenance—also called data lineage—refers to a comprehensive record of the origin, history, and evolution of data objects. This record is captured in a graph structure that describes the relationships among all the entities including their source, context, and dependencies that led to their existence and transformations [5]. Whole-system provenance extends these properties further by capturing file metadata and transient system objects [53]. Furthermore, it collects a complete provenance history of the system from its initialization to shutdown [53]. Provenance can therefore be very useful for forensic analysis and system security use cases. These include detection of security violations after execution, guaranteeing the reproducibility of computational experiments to verify compliance with legal regulations, deletion and policy modification, intrusion detection, and fault injection [51].

To realize these use cases, provenance frameworks are integrated into OSs where they are responsible for capturing the provenance of a data workflow and streaming it to relevant processing components. There are many tools and frameworks to capture and perform queries on provenance graphs [3, 7, 19, 32, 34, 51–53].

The provenance tools that we make use of in the entirety of our work are CamFlow [51] and SPADE [19]. CamFlow is a whole-system provenance capture mechanism for Linux that finely tunes the provenance captured to meet application requirements [51]. Similarly, Linux's Audit subsystem is capable of monitoring and tracking system calls and file accesses throughout the system and provides `auditd` as a userspace component to write the information to disk [1]. SPADE can ingest this audit data to construct a provenance graph. It can also directly ingest provenance graphs as generated by CamFlow. Further, SPADE can convert this data into relevant graph formats (e.g., W3C-PROV [39]), and store the graph in persistent storage; it also facilitates filtering, querying, and transforming the graph objects.

III. Threat Model

We follow a similar threat model to existing provenance-based systems [2, 21, 23, 27, 35, 58, 58]. As in previously published work, we assume that the underlying OS, the provenance capture system and PACED analysis code are

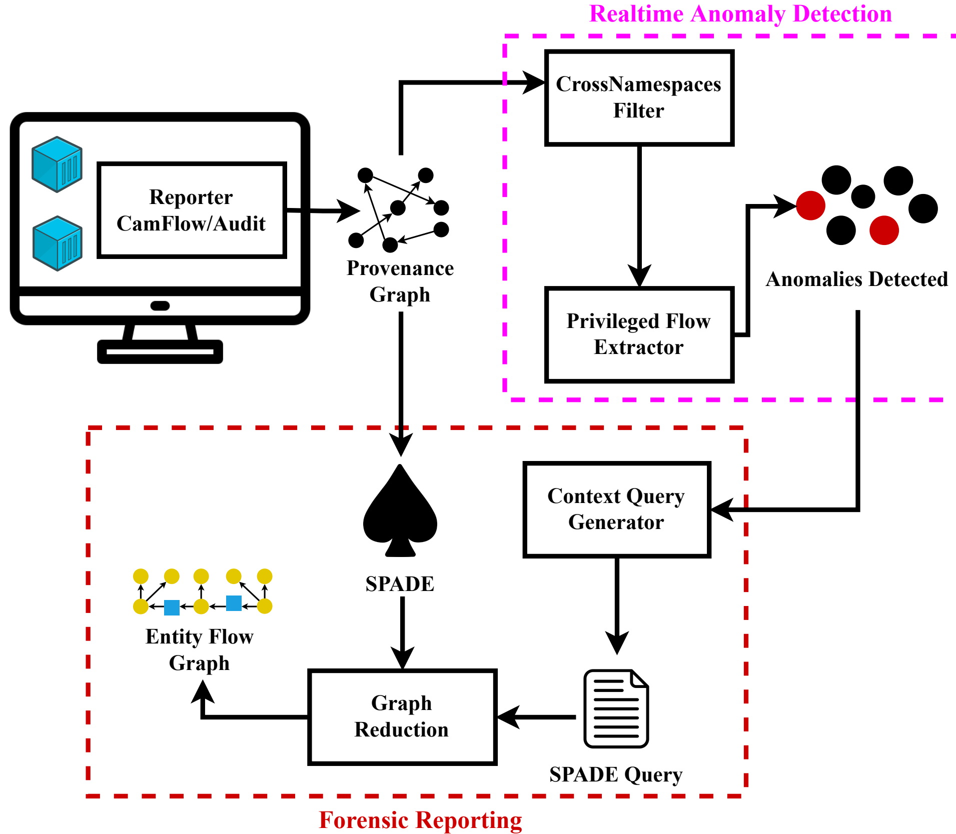

Fig. 1. **PACED Architecture** consists of two major phases: (1) Realtime Anomaly Detection, (2) Forensic Reporting

trusted. We assume that measures are in place to protect the audit mechanism [3, 51]. We do not consider hardware-level vulnerabilities as they are beyond the scope of current provenance capture systems. Finally, we assume the integrity of the provenance records. We leverage existing secure provenance systems [3, 51] and tamper-evident logging techniques [49, 50] to ensure log integrity and detect any malicious interference with provenance logs.

IV. DESIGN

We describe the design of PACED, our new anomaly detection system for container escape attacks. We proceed by first defining the components of PACED in order to explain how they interact to effectively detect malicious cross-namespace events. Our design makes use of open-source provenance tools [19, 51] and a canonical rule to effectively detect privileged flows during container escape attacks.

Fig. 1 provides a high-level overview of the architecture that also illustrates the workflow of provenance analysis from raw provenance to anomaly detection. The pipeline architecture may be broken up into two major phases: Realtime Anomaly Detection (§ IV-A) and Forensic Reporting (§ IV-B).

A. Phase 1: Realtime Anomaly Detection

The Realtime Anomaly Detection phase is divided into two steps. First, we use the *CrossNamespaces Filter* to extract all cross-namespace events. Then we perform anomaly detection to classify malicious cross-namespace events.

CrossNamespaces Filter. In theory, the namespaces feature of the Linux kernel should provide a security isolation that curtails information flow from one namespace to another. In reality, numerous CVEs describe how adversaries can successfully compromise the host from inside a container. In such exploits, a container process with its own set of namespaces interacts with entities that are used by host processes with a different set of namespaces. We capture all occurrences of processes from two different sets of namespaces interacting with the same system object (in particular inodes, e.g., files) by implementing a streaming *CrossNamespaces Filter* in SPADE. We call such events *cross-namespace events*. The *CrossNamespaces Filter* reports (1) the threads involved; (2) the system object involved (e.g., a file); and (3) the interactions between those objects (e.g., `read`/`write` system calls).

Anomaly Detection. Namespaces are organized in a hierarchy. Containers are spawned on the host, their namespace is lower in the hierarchy representing lower resource access and privileges. We expect isolation to prevent information to flow from a lower-privileged namespace to a higher-privileged namespace (e.g., overwriting host configuration files). However, our analysis of contemporary CVEs (see § VI-A) demonstrates that container escape attacks typically involve one or more cross-

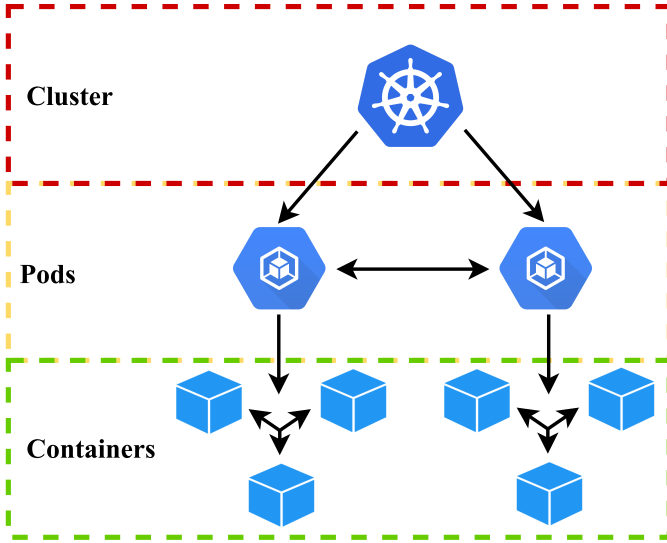

Fig. 2. Lattice structure illustrating the three levels of privilege in a Kubernetes environment. Cluster has the highest privilege level followed by Pods and Containers.

namespace events that violate this assumption. We characterize the following flow as a violating flow in the ordered sequence:

1) Write from a process in a container (less privileged namespace)
2) Read from a process on the host (more privileged namespace)

We refer to such a violating flow as *privileged_flow*.

Kubernetes Extension. We also devised a strategy to map our *privileged_flow* rule to Kubernetes. With minor modifications, it is possible to run our rule on provenance collected within a Kubernetes environment. There are three levels of privilege in a Kubernetes environment:

- Host/Cluster—highest privilege
- Pod—moderate privilege
- Container—lowest privilege

The lattice structure in Fig. 2 provides a visual representation of the privileges at each level. To distinguish the types of *privileged_flows*, we define the following two policies:

- Policy 1: Ban low to higher privilege flows and inter-pod flows
- Policy 2: Ban only low to higher privilege flows

The administrator must decide how to define *privileged_flows* in the context of Kubernetes because the use case differs across projects. For example, if the application is spread over different pods on a cluster and all of the pods must communicate with one another then Policy 1 would generate a lot of false positives. This changes how we define *privileged_flow* explicitly:

1) Write from a level n (less privileged namespace)
2) Read from a level $>= n$ (more privileged namespace)

With *privileged_flow* defined for both Docker and Kubernetes, we can automatically extract cross-namespace events

TABLE I
ENTITY FLOW GRAPH ELEMENTS

Element	Description
Entity Vertex	The entity through which the cross-namespace flow occurs.
Task Vertices	The threads interacting with the entity.
Process Memory Vertices	Vertices representing the process memory associated with the threads.
Path Vertices	The path associated with the processes and the entity.
Argv Vertices	The command line arguments passed to the processes.

Algorithm 1 EFG Creator to Generate Entity Flow Graph

N, R, W, M, P, A—Set of all entities, readers, writers, process_memory, path, argv; G—Output graph

▷ Connect entities with readers and writers
$G \leftarrow N_V \cup R_V \cup W_V$
$G \leftarrow G \cup (N_E \cap R_E) \cup (N_E \cap W_E)$

▷ Add process memories connected to readers and writers
for m in M **do**
  $G \leftarrow G \cup m_V \cup (m_E \cap R_E)$
  $G \leftarrow G \cup m_V \cup (m_E \cap W_E)$

▷ Add paths connected to entities and process memories
for p in P **do**
  $G \leftarrow G \cup p_V \cup (p_E \cap N_E)$
  $G \leftarrow G \cup p_V \cup (p_E \cap M_E)$

▷ Add argvs connected to process memories
for a in A **do**
  $G \leftarrow G \cup a_V \cup (a_E \cap M_E)$

return G

and identify anomalous ones. However, security analysts require more information to understand the root cause of any problem, and identify malicious behaviors from false positives. This leads to the second phase of PACED—Forensic Reporting.

B. Phase 2: Forensic Reporting

The Forensic Reporting phase is divided into two steps. First, we use SPADE queries to retrieve the context of cross-namespace event in the form of a graph. Second, we use graph reduction techniques to compress this graph and present the final output in the form of an entity flow graph, as illustrated in Fig. 4.

SPADE Query. The SPADE *QuickGrail* query interface enables execution of scalable graph operations and generation of provenance graphs while abstracting the underlying database semantics [15]. Our query is automatically generated based on the output of the anomaly detector. This output represents cross-namespace events, and as previously described, contains the entity through which an anomalous flow occurs (e.g., a file), the two threads reading and writing to this entity, and finally the relations connecting those entities (e.g., read/write system calls).

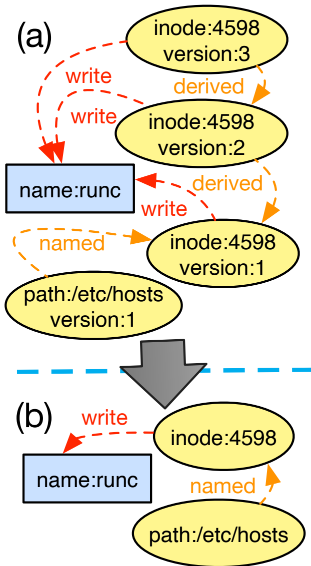

Fig. 3. Illustration of graph reduction

We use the algorithm presented in Alg. 1 to retrieve the context of a cross-namespace event. In particular, we aim to retrieve 1) the path of the entity through which the cross-namespace flow occurs; 2) the paths corresponding to the executable program for each of the two threads; and 3) the arguments that were passed to those executables. The graph vertices we retrieve are summarized in Table I.

Graph Reduction. Graph reduction techniques are a common strategy to reduce the complexity of a provenance graph [26, 59, 62]. To improve interpretability and simplify analysis of the provenance we apply common provenance reduction techniques. Provenance capture systems, such as CamFlow, represent system objects as multiple versions to guarantee graph acyclicity [40] (in essence, a new version is created when an object is modified, e.g., a node representing a new version of a file is created when a process writes to the file). We collapse these versions into a single vertex. A second reduction is to remove redundant edges. Redundant edges occur when a process performs the same operation repeatedly (e.g., reading data from a file), and can be replaced with a single representative edge. Fig. 3 illustrates how graphs are transformed through reduction.

Final Output. We show in Fig. 4 the final output as presented to a security analyst. It contains the minimum amount of information to understand the processes involved in a cross-namespace event and the entity through which it occurred. Further analysis can be performed to fully understand the impact of the malicious behavior by issuing queries to SPADE. We refer readers interested in provenance-based investigation to work by Gehani et al. [15].

V. Experimental Setup

We describe the tools and techniques we employ to execute the container escape attacks that we subsequently detect. We

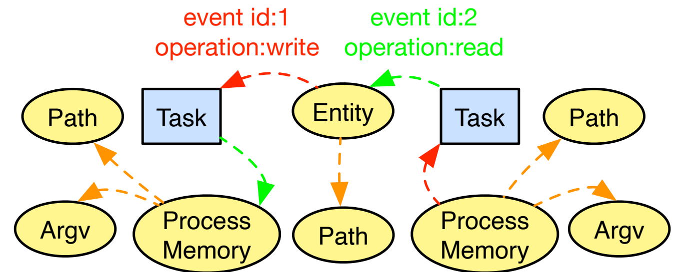

Fig. 4. **Entity Flow Graph**: Illustrates the write edge on the *Entity*, followed by a read edge on the same *Entity*. The graph also provides information about the writing *Task*, and the reading *Task* in the form of their *Argv*, *Path*, and *Process Memory* vertices.

also describe the method that we use to collect provenance and parse it to make it suitable for PACED. We highlight the experimental environment used to execute these attacks, including description of the modules we use to support our experiments.

Provenance Collection. To collect provenance, we use either SPADE's *Audit Reporter* or *CamFlow Reporter*. Even though both provenance capture tools use different provenance models, with the help of the SPADE query client, we can employ the same anomaly detection approach. In our experiments, we use provenance collected from CamFlow to perform our analysis. However, we have verified that our approach can be applied to provenance collected using SPADE's *Audit Reporter*.

In our experiments, we use the camflow-dev repository `v8.1`. Our analysis requires provenance graph edges to be reported in temporal order. However, CamFlow output does not guarantee such an ordering, and we describe below how our experimental setup accommodates this.

A vertex is considered out-of-order if it is reported after an edge that is incident upon it. This is resolved by using CamFlow's 'duplicate' flag, which ensures that the endpoint vertices of an edge are emitted before each edge even if one or both had previously been reported. This duplication inflates the size of the provenance log but is necessary for our analysis to minimize the state maintained.

An edge, *e1*, is out-of-order when it is reported after another edge, *e2*, where the corresponding event for *e1* occurred before the corresponding event for *e2*. This is resolved by sorting edges in the provenance log on the edge attribute `relation id`. (CamFlow's `relation id` attribute provides a total ordering between edges.)

CamFlow may drop provenance elements when under high memory pressure. The missing provenance elements can cause our analysis to be incomplete. This was resolved by an update to CamFlow that allows the user to set the amount of memory allocated to CamFlow's internal buffers. In our experiments, we increased CamFlow's internal buffer size from its default setting to of 8MB to 1GB.

Environment Setup. In our experiments, we execute the

container CVEs that exploit Docker and Kubernetes hosted on Linux. We also need to run our provenance capture tool—CamFlow—in the background to audit the system activity and report its provenance.

For each Docker and Kubernetes exploit, there is a vulnerable version mentioned in the CVE database provided by the National Vulnerable Database [41]. Setting up the environment according to each CVE can be an arduous undertaking. However, for most CVEs, we were able to use Metarget [38] to configure the vulnerable environment. Metarget provides automated construction and swift deployment of the vulnerable environments within cloud native infrastructure. Metarget saves us time in the deployment phase of the CVE and allows us to focus our efforts on our exploit detection solution [61]. After setting up the environment and the provenance capture tool, we take the following steps to execute each CVE:

1) Start *CamFlow Reporter*
2) Build container image(s)
3) Run the exploit code
4) Stop *CamFlow Reporter*

We obtain a provenance log that has system activity records of the complete execution of exploit on our system.

We follow the same steps for a set of known bad container configurations, ensuring these configurations are changed just before the exploit is run. Similarly, for benchmarking we run the workload instead of the exploit code in step 3.

VI. Evaluation

We evaluate the ability of PACED to effectively detect real-world vulnerabilities. In § VI-A, we describe the CVEs we exploited and what PACED detects. In § VI-B, we describe common container misconfiguration that can be exploited to perform container escape attack. We also report what PACED is able to detect. In § VI-C, we describe workloads not containing any attack we used to ensure that PACED doesn't raise false alarms. Finally, we summarize PACED detection ability in § VI-D.

A. CVEs

We evaluate the capacity of PACED to detect the exploitation of seven Dockers and Kubernetes vulnerabilities (see Table II). In the rest of this section, we describe those vulnerabilities and what PACED detects.

TOCTOU bug in `docker cp` [45]: The function `FollowSymlinkInScope()`, used by the Docker daemon, is vulnerable to a TOCTOU (Time To Check To Time Of Use) vulnerability. When a user tries to copy something into the container filesystem using the `docker cp` command, the Docker daemon process executes the `FollowSymlinkInScope()` function. This function resolves the path as if the process was inside the container. Once an actual non-symlink path is found, the Docker copy command adds this resolved path to the container mount point, and then simply copies. However, an attacker can use the time window after the path resolution and right before the write operation to create a symlink that will be resolved in the context of the host's root directory. Through this exploit, the Docker process can thus potentially overwrite any file on the host.

TABLE II
CVEs with CVSS Score

CVE	CVSS Score	Technologies
2018-15664 [45]	7.5	Docker
2019-5736 [48]	8.6	Docker
2019-14271 [43]	9.8	Docker
2019-1002101 [46]	5.5	Kubernetes, Docker
2020-15257 [44]	5.2	Docker, containerd
2021-30465 [47]	8.5	Kubernetes, Docker
2022-0492 [42]	7.0	Docker

PACED successfully captures the symlink exchange happening to hit this race condition. The attack is captured on a link object that has both paths (i.e., path on the host and symlink path) associated with it.

Overwrite `runC` binary [48]: This CVE exploits a vulnerability that allows users to gain root access to a host through a malicious container by overwriting the `runC` binary residing on the host. The malicious user is able to initiate an attack in the following two cases: by running a malicious Docker image, or by running an `exec` command on an already running and compromised container.

To run the CVE, a `Dockerfile` is used that appends a `run_at_link.c` code to the `libseccomp` library file. It then copies and compiles the `overwrite_runc` file, then creates a symlink `/proc/self/exe` → `/entrypoint` and sets the entrypoint to `/entrypoint` [4]. When the container runs, the `run_at_link` code sends the `runC` file descriptor to `overwrite_runc`. The `overwrite_runc` file then overwrites the `runC` using the file descriptor with "`evil_runc`".

PACED successfully captures the execution of the instance of `overwrite_runc` when it modifies the contents of the `runc` binary file.

Code injection via shared libraries [43]: In this vulnerability an attacker can gain root access to the host when the `docker cp` command to copy files to or from the container is executed. The vulnerability stems from a shared library (specifically `libnss_*.so`). Docker is implemented in Go, and in the Golang `v1.11` release, shared libraries are loaded at runtime. Normally, the libraries are loaded from the host namespace, but since `docker-tar` momentarily applies a `chroot` into the container during the execution of the `cp` subcommand, `docker-tar` loads its shared libraries from the container namespace. An attacker can replace this library to execute arbitrary code.

In our scenario, the attacker prepares a malicious `libnss` shared library where it injects a malicious function in one of the `libnss` source files. The function verifies that it runs in the context of `docker-tar` and executes a malicious binary inside the container which mounts the host filesystem onto the container [55]. The attacker replaces the `libnss` source library with the malicious one and attack is completed when

user runs the `docker cp` command.

PACED successfully captures the overwriting of the original `libnss` library with the malicious `libnss` library.

TOCTOU bug in `runC` binary [47]: The host can be compromised due to a race condition that resides in the `SecureJoinVFS()` function in `runC`. Kubernetes, when mounting shared volumes, trusts the source but not the specified target, therefore, it relies on `SecureJoinVFS()` to ensure that the target ends up within the container. The target is evaluated safely unless a symlink is created before the `SecureJoinVFS()` function returns. An attacker can exploit this vulnerability through a race condition,

To run the exploit, we need to configure the pod in such a way that we can run a race binary before starting some container where we wish to hit the TOCTOU bug. In order to do that, we can configure the pod in the following way: (1) Create a pod with 10 containers and upon initiation run only 1 container; (2) Create the necessary mount points to hit the race condition through the `SecureJoinVFS()` call; (3) Run the race binary in the first container; (4) Boot up the remaining containers to try to hit the race condition. Moreover, we create 4 mount points for each container and execute 4 instances of the race binary code for each mount point to increase our chances of success.

We then update the image of the subsequent containers which automatically launches them and carries out the mounts specified in the configuration. After all the containers are launched we can iterate and see which container successfully hit the race condition.

PACED successfully captures the symlink exchange to hit the race condition for each one of the 4 mount points in all 10 containers.

Exposed abstract Unix domain socket [44]: This CVE involves a vulnerability within the `containerd-shim` API in `containerd`, the standard container runtime used by the Docker engine, which allows a container with access to the host network namespace to escape from the container to the host machine. When we run a container using the `docker` command, `containerd` executes the `containerd-shim` binary which is responsible for the execution of the container lifecycle. The `containerd-shim` exposes its functions to `containerd` via the `containerd-shim` API. The `containerd-shim` API is exposed via an abstract Unix domain socket that is accessible on the host system's network namespace. An abstract Unix domain socket is a kind of Linux socket that is not bound to a file path in the filesystem. However it is associated to the network namespace of the process. These sockets have no access control and the only thing that is verified is that the user that listens to these sockets must be the same user as `containerd-shim`. Therefore, if a container is running as root, i.e., UID 0, and with host network privileges, it can connect to the `containerd-shim` API socket and lead to the container taking over the host machine. This includes arbitrary reads and writes to the host filesystem, executing any command on the host machine as the root user, and creating and starting up new containers.

We exploit this vulnerability by running a container with the desired configuration and the exploit we used overwrites a host file by running a command as root user on the host machine as well as obtaining a reverse shell connection.

PACED successfully captures the cross-namespace escape flow for both the reverse shell execution and when the malicious command is executed on the host.

Vulnerable CGroup `release_agent` [42]: This CVE exploits a simple privilege escalation vulnerability in CGroups implementation within the Linux kernel. The vulnerability specifically lies in the CGroups v1 architecture which is still widely used. The `release_agent` file of CGroups v1 allows administrators to configure a "release agent" program that would run upon the termination of a process in the CGroup. The release agent program runs with root capabilities in the initial namespaces. The vulnerability stems from a missing verification. The kernel simply does not verify that the process modifying the `release_agent` file must also have administrative privileges (e.g, the `CAP_SYS_ADMIN` capability).

The vulnerability is therefore exploited if one is able to write to the `release_agent` file, then you can force the kernel into invoking a malicious binary with highest privileges and compromise the entire machine. Since Linux sets the owner of the `release_agent` file to root, only root can modify it. In a scenario where the root process doesn't have full control over the machine, like the root of a container, it can write to the file and gain access to the host machine leading to a container breakout [8].

A malicious container that wants to exploit CVE-2022-0492 must mount another, writable `cgroupfs`. Mounting a `cgroupfs` requires the `CAP_SYS_ADMIN` capability in the user namespace hosting the current cgroup namespace. We can either provide the `CAP_SYS_ADMIN` capability directly to the container or through the `unshare()` syscall, where containers can create new user and CGroup namespaces which possess the `CAP_SYS_ADMIN` capability and can mount a `cgroupfs`. Since the container process now has the required administrative privilege, it can write to the `release_agent` file.

PACED successfully captures the malicious write to the `release_agent` file from the container process and then the file being read from the host context when the release agent program is run.

Directory traversal vulnerability in `kubectl cp` [46]: This CVE stems from vulnerability in the `kubectl cp` architecture which leads to a directory traversal to the host machine via a symlink. This effectively allows a compromised container to write to any file on the host machine. When copying files from a container, `kubectl cp` adds the source container files to a TAR archive and unzips it on the destination host machine. For this action, `kubectl cp` relies on the `tar` binary inside the container and if the `tar` binary is malicious, an adversary can execute arbitrary commands on the host machine. The method of copying files from the container is written in the `copyFromPod` function in `cp.go` and this

function is vulnerable to follow or create symlinks from the TAR headers to virtually any path on the host machine.

We prepare a malicious TAR archive which contains the payload and symlink to a target host directory. The `tar` binary inside the container is replaced with the malicious binary which outputs the contents of the malicious TAR we created. Finally, when a user runs the `kubectl cp` command to copy files from the container, the compromised binary is run which extracts the malicious TAR archive on the host machine and runs the malicious payload [10].

In this exploit, PACED successfully captures the `tar` binary inside the container being compromised and then `kubectl cp` using the compromised `tar` binary to copy the target files from the container. It also captures the malicious TAR archive created from the container and used in the host context when it is extracted onto the destination host directory.

B. Bad Container Configurations

In addition to CVEs, a badly configured Docker container can easily suffer reduced levels of security, sometimes leading to container escape. The same idea applies to a Kubernetes pod as well, where a pod can escalate privileges. For ease of development, some developers and system administrators use these misconfigurations for convenience, which can provide an adversary the attack surface required to exploit these vulnerabilities. Here, we briefly mention some of the common examples of container misconfiguration that we used within our evaluation. To run these exploits, we used *CDK* [6], a container penetration toolkit which automates running different exploits by placing their binary inside a container and escaping the container using a supported list of exploits.

Mounting `docker.sock` into a container: A process which has access to the Docker socket (usually at `/var/run/docker.sock`) has the same privileges as the Docker service, which effectively allows access to the host system, as Docker service runs as root. Therefore, mounting a Docker socket inside a container can allow any process inside the container to execute commands on the host machine.

PACED is able to identify a write to a file with host path from the container process and the file then read from the host context.

Mounting the `proc` filesystem into a container: The `procfs` pseudo-filesystem contains sensitive information about all the processes running in the system. Mounting the host `procfs` directory inside the container allows the container to access all the processes on the host and thus allows an attacker to take over the host machine. To achieve an exploit, we can point any arbitrary shell command to the host's `/sys/kernel/core_pattern` file, and then use a runtime segment fault to trigger a core dump inside the container which effectively executes our shell code on the host.

PACED is able to identify the attack flow on the file which contained the malicious shell command which was then read from the host context.

Mounting host CGroup directory into a container: Similar to `procfs`, mounting the CGroup pseudo-filesystem inside the container allows the container access to the `release_agent` file into which it can inject malicious shell code that can be run using the CGroup `notify_on_release` feature.

PACED is able to identify the attack flow on the malicious shell file which contained the malicious shell command which was then read from the host context.

Mounting the `/var/log` directory into a Kubernetes Pod: The `kubelet` service allows a Kubernetes pod to access files in the host's `/var/log` directory via an API. If the pod creates a symlink and passes it to the API, the symlink is resolved in the underlying host filesystem's hierarchy. The `kubelet` service should check that the resolved path remains within the `/var/log` hierarchy if the request is from a pod, but it does not.

The host can mount `/var/log` into the pod. If this path is writable, the pod can create a symbolic link in `/var/log` that points to the host's root directory. It can then proceed to construct a malicious `kubelet` request containing the symbolic link, resulting in the ability to access the entire host filesystem. This is exploited by a Python script that creates symlinks to the host paths where private keys and the Kubernetes service account token reside. The script uses the `kubelet` service to transfer the files into the pod.

When the `kubelet` service passes the above files to the pod, they result in a low privileged write by the receiver. On the host, `kubelet` runs at high privilege and is responsible for polling the target directory and reading requested files. The combination of the write and the poll results in the triggering of the *privileged_flow* rule.

C. Benchmarks

In addition to detecting malicious behaviors, we want to ensure that PACED does not raise unworkable numbers of false alarms. To do so, we use the scenarios from the well established microservices benchmark DeathStarBench [12].

D. Results

We report results in Table III. PACED detects malicious flows occurring on entities in Docker with 100% accuracy. We further note that PACED does not raise any alarms when running the benchmarks.

The extension of our *privileged_flow* rule to Kubernetes CVEs and bad pod configurations gives a recall score of 100%, which again indicates that no malicious flow was left undetected. A Kubernetes cluster can be configured with numerous parameters. For instance, a cluster can be set up with different Container Network Interface (CNI) plugins. We do see a false positive in our work where we used the flannel plugin (CNI) to set up a cluster.

Running PACED on the Kubernetes Benchmark led to 2 false positives out of the 62 entities. The entities on which these false positives are reported are internal to Kubernetes, for example the `xtables.lock` access which occurs as a false positive throughout the benchmark and CVEs. We leave further filtering these isolated false positives to future work.

TABLE III
EVALUATION RESULTS OF PACED

Evaluation type	Name	Total Entities	TP	TN	FP	FN
Docker CVEs	2018-15664 [45]	18	1	17	0	0
	2019-5736 [48]	24	1	23	0	0
	2019-14271 [43]	16	2	14	0	0
	2020-15257 [44]	10	3	7	0	0
	2022-0492 [42]	16	1	15	0	0
Bad Docker Configurations	Sock Pwn	17	1	16	0	0
	Mount CGroup	2	1	1	0	0
	Mount Procfs	1	1	0	0	0
Kubernetes CVEs	2019-1002101 [46]	9	6	1	2	0
	2021-30465 [47]	2584	7	2573	4	0
Bad Pod Configurations	Mount `/var/log`	48	31	15	2	0
Docker Benchmarks	Hotel Reservation	113	0	101	0	0
	Media Microservices	448	0	436	0	0
	Social Network	378	0	366	0	0
Kubernetes Benchmarks	Hotel Reservation	62	0	60	2	0

VII. RELATED WORK

Below we identify prior research that motivated our work and describe how ours differs from it.

A. Container Security

Research Taxonomies: Since containers have become widely popular due to the lightweight isolation provided, researchers have studied their security implications and challenges. Sultan *et al.* [57] provides a comprehensive survey on container security and proposes a research taxonomy of use cases for: (1) in-container protection from applications, (2) inter-container protection, (3) host protection from containers, and (4) container protection from malicious hosts. Gao *et al.* [13, 14] investigated information leakage channels in containers. These channels can leak host information and can provide a path for adversaries to launch advanced attacks. Their work focused on power attacks where, based on the leaked information of a host machine, an active adversary can cause a power outage that can affect the reliability of data centers. They also propose a power-based namespace for fine-grained energy usage at the container level.

Container Vulnerability Analysis: In 2015, Gummaraju *et al.* [20] found that 30% of all images on Docker Hub contained high priority security vulnerabilities. Shu *et al.* [56]'s Docker image vulnerability analysis framework showed that official and community images contain over 180 vulnerabilities on average. Martin *et al.* [36]'s vulnerability analysis spanned the Docker ecosystem, identifying exploits and possible fixes. Kabbe [29] demonstrated Docker-based attacks DirtyCow (CVE-2016-5195), Shellshock (CVE-2014-6271), Heartbleed (CVE-2014-0160), and Fork-bomb were possible in production environments. The *Torpedo* [37] extension of the *syzkaller* fuzzing framework searches for sequences of system calls that break process isolation constraints to check for vulnerabilities in popular containerization platforms. Deng *et al.* [9] focus on container interference in the cloud and describe a way to mitigate it with a deep reinforcement learning approach for container placement.

Container Escape: Jian and Chen [28] consider the case where a container escape results in a root shell with different namespaces from its parent, using this as the basis for detection. Our approach is more general and detects a superset of cases. Reeves *et al.* [54] studied 59 CVEs in 11 container runtimes. Focusing on 28 for which they found a proof-of-concept (PoC), they determined that 13 occurred because a host component was exposed to a container. They devised a defense that prevented 7 of the 9 PoCs used to exploit these 13 CVEs. Their approach aims to mitigate a subset of the cases that PACED detects.

B. Provenance-based Security

Data provenance has been used in prior work for security applications [7, 24, 52], including intrusion detection [21–23, 25, 27, 58], but not specifically for identifying container escapes. Below we describe relevant efforts.

Intrusion Detection: *Unicorn* [21] is a real-time anomaly detection system that uses CamFlow's whole-system provenance to detect Advance Persistent Threats (APTs). *TRACE* [27] is a combination of SPADE and Unit-Based Selective Instrumentation (UBSI), a form of execution partitioning [32]. It was used in DARPA's Transparent Computing program to provide provenance for APT detection. *SIGL* [23] uses deep learning over provenance graphs to build a model of normal software installations. This is then leveraged to detect when malicious software is installed, facilitating intrusion detection. *NoDoze* [25] and *ProvDetector* [58] are closed-source systems that use internal NEC data for provenance-based threat alert triage and stealthy malware detection, respectively. None of these systems focus on container environments.

Container Security Applications: *Winnower* [24] modifies SPADE to implement a security system for Docker Swarm container clusters. It accelerates attack investigation by grammatical inference over graphs. *Clarion* [7] adds namespace-

awareness to SPADE. This ensures the soundness and clarity of emitted provenance for downstream security analyses. In contrast to Clarion, PACED's analysis is automated. *CamQuery* [52] builds on CamFlow to support both real-time and forensic provenance analysis of systems, including ones with containerized environments. None of these prior systems support real-time container escape detection.

VIII. CONCLUSION

Container escape attacks allow attackers to circumvent the container's isolation boundary and take control of the host system. In this paper, we propose a novel provenance-based container escape detection system called PACED that detects the malicious entities responsible for bypassing the container boundary. We present PACED's architecture and data processing pipeline that queries, transforms, and detects malicious entities on the cloud systems. We test PACED over multiple Docker and Kubernetes container escape CVEs, benchmark applications and common misconfiguration scenarios and report the results. PACED detects container escape exploits with 100% accuracy for containers hosted on Docker and with near-perfect accuracy for containers hosted on Kuberenetes, with no attack being left undetected.

ACKNOWLEDGMENTS

We thank Rizwan Shahid, Hira Jamshed, Mahnoor Jameel, and Saad Ullah for their scripting of three CVE exploits and analysis of the resulting data provenance records. This material is based upon work supported by the U.S. National Science Foundation under Grant ACI-1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

AVAILABILITY

Extensions to SPADE are available at https://github.com/ashish-gehani/SPADE. The software artifacts used to detect cross namespace flows and our datasets are available at https://github.com/PACED-prov.

REFERENCES

[1] AuditD - Linux Manual Page. https://man7.org/linux/man-pages/man8/auditd.8.html.

[2] Mathieu Barre, Ashish Gehani, and Vinod Yegneswaran. Mining Data Provenance to Detect Advanced Persistent Threats. In *Workshop on the Theory and Practice of Provenance (TaPP'19)*. USENIX, 2019.

[3] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux Kernel. In *Security Symposium (USENIX Sec'15)*. USENIX, 2015.

[4] Breaking out of Docker via runC – Explaining CVE-2019-5736. https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/.

[5] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, and Andy Hopper. A primer on provenance. *Communication of the ACM*, 2014.

[6] CDK - Zero Dependency Container Penetration Toolkit. https://github.com/cdk-team/CDK.

[7] Xutong Chen, Hassaan Irshad, Yan Chen, Ashish Gehani, and Vinod Yegneswaran. Clarion: Sound and Clear Provenance Tracking for Microservice Deployments. *30th USENIX Security Symposium*, 2021.

[8] New Linux Kernel cgroups Vulnerability Could Let Attackers Escape Container. https://thehackernews.com/2022/03/new-linux-kernel-cgroups-vulnerability.html.

[9] Qiqing Deng, Xinrui Tan, Jing Yang, Chao Zheng, Liming Wang, and Zhen Xu. A secure container placement strategy using deep reinforcement learning in cloud. In *International Conference on Computer Supported Cooperative Work in Design (CSCWD'22)*. IEEE, 2022.

[10] Disclosing a directory traversal vulnerability in Kubernetes copy – CVE-2019-1002101. https://unit42.paloaltonetworks.com/disclosing-directory-traversal-vulnerability-kubernetes-copy-cve-2019-1002101/.

[11] Docker overview. https://docs.docker.com/get-started/overview/.

[12] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (APSLOS'19)*. ACM, 2019.

[13] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *International Conference on Dependable Systems and Networks (DSN'17)*. IEEE/IFIP, 2017.

[14] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. A study on the security implications of information leakages in container clouds. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[15] Ashish Gehani, Raza Ahmad, Hassaan Irshad, Jianqiao Zhu, and Jignesh Patel. Digging Into "Big Provenance" (With SPADE). *Communications of the ACM*, 64(12), 2021.

[16] Ashish Gehani, Basim Baig, Salman Mahmood, Dawood Tariq, and Fareed Zaffar. Fine-Grained Tracking of Grid Infections. In *International Conference on Grid Computing (GRID'10)*. ACM/IEE, 2010.

[17] Ashish Gehani, Minyoung Kim, and Jian Zhang. Steps Toward Managing Lineage Metadata in Grid Clusters. In *Workshop on the Theory and Practice of Provenance (TaPP'09)*. USENIX, 2009.

[18] Ashish Gehani, Florent Kirchner, and Natarajan Shankar. System Support for Forensic Inference. *Advances in Digital Forensics V*, 2009.

[19] Ashish Gehani and Dawood Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. *13th ACM/IFIP/USENIX International Middleware Conference*, 2012.

[20] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of official images in Docker Hub contain high priority security vulnerabilities. *Technical Report*, 2015.

[21] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. UNICORN: Runtime Provenance-based Detector for Advanced Persistent Threats. In *Network and Distributed System Security Symposium (NDSS'20)*. Internet Society, 2020.

[22] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo I. Seltzer. Frappuccino: Fault-detection through runtime analysis of provenance. In *Workshop on Hot Topics in Cloud Computing (HotCloud'17)*. USENIX, 2017.

[23] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo Seltzer, and Haifeng Chen. SIGL: Securing Software Installations Through Deep Graph Learning. In *Security Symposium (USENIX Sec'21)*. USENIX, 2021.

[24] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium (NDSS'18)*, 2018.

[25] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed Systems Security Symposium (NDSS'19)*. Internet Society, 2019.

[26] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. Dependence-Preserving Data Compaction for Scalable Forensic Analysis. In *Security Symposium (USENIX Sec'18)*. USENIX, 2018.

[27] Hassaan Irshad, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Kyu Lee, Jignesh Patel, Somesh Jha, Yonghwi Kwon, Dongyan Xu, and Xiangyu Zhang. TRACE: Enterprise-Wide Provenance Tracking For Real-Time APT Detection. *IEEE Transactions on Information Forensics and Security (TIFS)*, 16, 2021.

[28] Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *International Conference on Cryptography, Security and Privacy (ICCSP'17)*. ACM, 2017.

[29] Jon-Anders Kabbe. Security analysis of docker containers in a production environment. In *MS Thesis (Norwegian University of Science and Technology)*, 2017.

[30] Samuel T King and Peter M Chen. Backtracking intrusions. In *Symposium on Operating Systems Principles (SOSP'03)*. ACM, 2003.

[31] Kubernetes overview. https://kubernetes.io/docs/concepts/overview/.

[32] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Network and Distributed System Security Symposium (NDSS'13)*. Internet Society, 2013.

[33] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *Network and Distributed System Security Symposium (NDSS'18)*. Internet Society, 2018.

[34] Shiqing Ma, Xiangyu Zhang, Dongyan Xu, et al. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Network and Distributed System Security Symposium (NDSS'16)*. Internet Society, 2016.

[35] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, 2016.

[36] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem – vulnerability analysis. *Computer Communications*, 2018.

[37] Kenton McDonough, Xing Gao, Shuai Wang, and Haining Wang. Torpedo: A fuzzing framework for discovering adversarial container workloads. In *International Conference on Dependable Systems and Networks (DSN'22)*. IEEE/IFIP, 2022.

[38] Metarget. https://github.com/Metarget/metarget.

[39] Paolo Missier, Khalid Belhajjame, and James Cheney. The W3C PROV family of specifications for modelling provenance metadata. In *International Conference on Extending Database Technology (EDBT'13)*, 2013.

[40] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *Annual Technical Conference (ATC'06)*. USENIX, 2006.

[41] Nvd - national vulnerable database. https://nvd.nist.gov/.

[42] CVE-2022-0492. https://nvd.nist.gov/vuln/detail/CVE-2022-0492.

[43] CVE-2019-14271. https://nvd.nist.gov/vuln/detail/CVE-2019-14271.

[44] CVE-2020-15257. https://nvd.nist.gov/vuln/detail/CVE-2020-15257.

[45] CVE-2018-15664. https://nvd.nist.gov/vuln/detail/cve-2018-15664.

[46] CVE-2019-1002101. https://nvd.nist.gov/vuln/detail/CVE-2019-1002101.

[47] CVE-2021-30465. https://nvd.nist.gov/vuln/detail/CVE-2021-30465.

[48] CVE-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

[49] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller,

and Dave Tian. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *Network and Distributed System Security Symposium (NDSS'20)*. Internet Society, 2020.

[50] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *Conference on Computer and Communications (CCS'20)*. ACM, 2020.

[51] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Symposium on Cloud Computing (SoCC'17)*. ACM, 2017.

[52] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *Conference on Computer and Communications Security (CCS'18)*. ACM, 2018.

[53] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-Fi: collecting high-fidelity whole-system provenance. In *Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[54] Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z. Berkay Celik. Towards improving container security by preventing runtime escapes. In *Secure Development Conference (SecDev'21)*. IEEE, 2021.

[55] Docker Patched the Most Severe Copy Vulnerability to Date With CVE-2019-14271. https://unit42.paloaltonetworks.com/docker-patched-

[60] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. Demons in the shared kernel: Abstract resource attacks against OS-level virtualization. In

the-most-severe-copy-vulnerability-to-date-with-cve-2019-14271/.

[56] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Conference on Data and Application Security and Privacy (CODASPY'17)*. ACM, 2017.

[57] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 2019.

[58] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *Network and Distributed System Security Symposium, (NDSS'202)*. Internet Society, 2020.

[59] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Conference on Computer and Communications Security (CCS'16)*. ACM, 2016.

Conference on Computer and Communications Security (CCS'21). ACM, 2021.

[61] Yutian Yang, Wenbo Shen, Bonan Ruan, Wenmao Liu, and Kui Ren. Security challenges in the container cloud. In *International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA'21)*. IEEE, 2021.

[62] Jun Zengy, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. ShadeWatcher: Recommendation-guided cyber threat analysis using system audit records. In *Symposium on Security and Privacy (S&P'22)*. IEEE, 2022.