*electronics*

MDPI

*Article*

# A Container Escape Detection Method Based on a Dependency Graph

**Kai Chen [1,*], Yufei Zhao [1], Jing Guo [2], Zhimin Gu [2], Longxi Han [1] and Keyi Tang [3]**

[1] China Electric Power Research Institute, No. 15 Xiaoying East Road, Qinghe, Haidian District, Beijing 100192, China; zhaoyufei@epri.sgcc.com.cn (Y.Z.); hanlongxi@epri.sgcc.com.cn (L.H.)
[2] State Grid Jiangsu Electric Power Co., Ltd., Research Institute, Nanjing 210000, China; muye@gwjsdky.wecom.work (J.G.); rena@gwjsdky.wecom.work (Z.G.)
[3] School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China; tangkeyi@seu.edu.cn
* Correspondence: chenkai1@epri.sgcc.com.cn

**Abstract:** With the rapid advancement in edge computing, container technology has gained widespread adoption. This is due to its lightweight isolation mechanisms, high portability, and fast deployment capabilities. Despite these advantages, container technology also introduces significant security risks. One of the most critical is container escape. However, current detection research is incomplete. Many methods lack comprehensive detection coverage or fail to fully reconstruct the attack process. To address these gaps, this paper proposes a container escape detection method based on a dependency graph. The method uses various nodes and edges to describe diverse system behaviors. This approach enables the detection of a broader range of attack types. It also effectively captures the contextual relationships between system events, facilitating attack traceability and reconstruction. We design a method to identify container processes on the dependency graph through label generation and propagation. Based on this, container escape detection is implemented using file access control within the graph. Experimental results demonstrate the effectiveness of the proposed method in detecting container escapes.

**Keywords:** dependency graph; container escape; security threat detection

check for updates

**Citation:** Chen, K.; Zhao, Y.; Guo, J.; Gu, Z.; Han, L.; Tang, K. A Container Escape Detection Method Based on a Dependency Graph. *Electronics* **2024**, *13*, 4773. https://doi.org/10.3390/electronics13234773

Academic Editor: Zbigniew Kotulski

Received: 17 October 2024
Revised: 1 November 2024
Accepted: 2 November 2024
Published: 3 December 2024

**Copyright:** © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

In recent years, edge computing has advanced rapidly. This progress has led many large enterprises and institutions to adopt edge-based models [1]. Domestically, industries such as government, energy, finance, and small to medium-sized businesses have moved much of their infrastructure to the edge. Edge computing uses virtualization technology to combine resources like computing, storage, networking, and applications over the internet. It provides on-demand services to users. This model improves resource utilization, reduces operational costs, and increases overall efficiency.

Container technology is widely used in edge computing infrastructures. It is employed for packaging, isolating, and reusing applications due to its lightweight isolation mechanisms. Containers are increasingly replacing hypervisor-based virtual machines (VMs). They offer faster startup times, lower resource consumption, and better I/O performance [2]. In this context, container security becomes crucial.

The Linux kernel's *namespace* and *cgroup* mechanisms form the foundation of container technology [3]. These mechanisms provide lightweight isolation and resource control. They allow containers to run applications independently while being isolated from the host and other containers. However, the kernel is shared with the host, making this isolation weaker than expected. This leaves containers vulnerable to threats such as container escape [4,5]. Container escape occurs when an attacker exploits vulnerabilities within the container to take control of the host system. For instance, the CVE-2019-5736 vulnerability enables attackers to execute code under specific conditions by exploiting a flaw in runc, granting

them control over the host machine. This incident highlights the inherent risks posed by the shared kernel between containers and the host. Another notable example is CVE-2021-22922, which pertains to file permission configurations in Docker, allowing attackers to access sensitive host data via malicious operations in containers. Such cases illustrate that container escapes can result in not only data breaches but also significant service disruptions and other security incidents.

This is a serious security threat. Once the attacker controls the host, they can access sensitive data, modify applications, or launch attacks like Distributed Denial-of-Service (DDoS). These activities pose severe risks to the host's integrity. Therefore, improving container security and preventing container escape is essential.

Container escapes can be classified into three categories. They can result from insecure configurations, vulnerabilities in related components, or kernel vulnerabilities. However, most research only focuses on one or two types of escape. This leaves gaps in comprehensive detection coverage. Further research is needed to develop mechanisms that detect all types of container escapes.

This paper proposes a container escape detection method based on a dependency graph. This method addresses all three types of container escapes. First, we introduce a method for identifying container processes on the dependency graph through label generation and propagation. Second, we propose a dependency threat model. Finally, to overcome the limitations of existing detection methods, we present a container escape detection approach based on file access control within the dependency graph.

## 2. Related Work

Several methods have been proposed to detect specific container escape attacks. Zhiqiang Jian et al. [6] observed that after a container escape, the process operates in a different namespace from its parent. They used this as a basis for detection. Ke Xu et al. [7] suggested mitigating container escape damage by using mandatory access control mechanisms. These mechanisms restrict escaped processes from accessing files illegally. Smith [8] conducted a comprehensive study on container escape vulnerabilities in Docker environments. The study focused on analyzing these vulnerabilities and proposed detection methods to mitigate container escape threats. He et al. [9] analyzed cross-container attacks in edge environments using eBPF, demonstrating how attackers could exploit eBPF to bypass container isolation. They proposed a new permission framework to address these security vulnerabilities. M. Reeves et al. [10] studied 59 CVEs across 11 container runtimes. They recommended using user namespace-enabled containers to prevent attackers from exploiting vulnerable host components. M. Abbas et al. [11] used a dependency graph to detect container escape attacks. Their approach flagged read/write operations from low-privilege namespaces to high-privilege namespaces as illegal. This method extended beyond the Docker environment to Kubernetes. Tao Zhang et al. [12] modeled container escape behaviors caused by kernel vulnerabilities. They selected key process attributes as observation points and used privilege escalation as the detection criterion. They minimized the dependency graph size by recognizing container boundaries and built a heterogeneous observation chain based on the Open Provenance Model (OPM). VS D P et al. [13] examined precaution levels and mitigation strategies for container security, offering insights into potential vulnerabilities and the current state of research. They also identified key areas for future exploration, particularly in server-based and serverless containers.

Despite these advances, there are two main limitations in current Docker container escape detection research. First, container escapes can be categorized into three types: those caused by insecure configurations, component vulnerabilities, or kernel vulnerabilities. Existing studies typically address only one or two types, failing to cover all three. Second, in real-world environments, container escapes often involve multi-stage, continuous attacks. Each stage may have different objectives and impacts. However, most current methods focus on detecting individual escape behaviors. They lack the ability to reconstruct the entire attack process.

Provenance-based Intrusion Detection Systems (PIDSs) [14] offer a promising approach to address these issues. PIDSs detect intrusions using provenance graphs, also known as dependency graphs. These graphs contain various nodes and edges, representing diverse system behaviors. They can detect a broader range of attack types. Moreover, they capture the contextual relationships between system events during an attack. This enables attack sequence reconstruction using directed graphs. Most PIDS research currently focuses on detecting Advanced Persistent Threats (APTs).

Han et al. [15] explored the opportunities and challenges of PIDSs. Li et al. [16] proposed a PIDS framework with modules for data collection, data management, and threat detection. They evaluated recent approaches and discussed future research trends. M. Zipperle et al. [14] provided a comprehensive literature review. They also emphasized the importance of benchmark datasets for future research. The earliest use of dependency graphs for intrusion detection dates back to 2003. Wong et al. [17] performed threat modeling on the container ecosystem using STRIDE and surveyed existing mitigation strategies, assessing their strengths and weaknesses. L. Zhao et al. [18] proposed a robust soliton distribution-based zero-watermarking method for securing semi-structured power data, ensuring data integrity and tamper detection in power systems. Y. Yang et al. [19] introduced EPA-GAN, a model utilizing Generative Adversarial Networks (GANs) for anonymizing electric power data and balancing privacy and data utility. King et al. [20] introduced Backtracker, which traced the origins of an attack using event dependency graphs. In recent years, PIDS has gained increasing attention from researchers. M. Du et al. [21] developed DeepLog. This method uses multi-classifiers to predict subsequent events based on previous sequences. It applies LSTM to detect anomalies. M. Garchery et al. [22] introduced ADSAGE. This system models application log sequences with Recurrent Neural Networks (RNNs). It predicts future events and uses Feedforward Neural Networks (FFNNs) to assess event validity and predict anomaly scores. Y. Song et al. [23] developed a hierarchical dynamic risk assessment framework for the power data lifecycle, improving security through scenario-adaptive methods. M. Hossain et al. [24] proposed SLEUTH. This method detects intrusions on a dependency graph through label propagation. It assigns trust and confidentiality labels to nodes. Predefined policies detect intrusions, such as when a low-trust entity accesses a high-confidentiality object. However, SLEUTH faces the issue of dependency explosion. The team later addressed this by introducing MORSE [25], which reduces the impact of dependency explosion through a label decay strategy.

These dependency-graph-based intrusion detection methods do not account for container-specific provenance data during graph generation. Hassaan et al. [25] proposed CLARION, a namespace- and container-aware solution. It identifies container boundaries using clone and unshare calls. It also detects container initialization patterns. However, this solution only supports Linux kernels up to version 5.7.

## 3. Methods

Container technology is widely used in edge computing environments. However, it faces significant security challenges, especially the threat of container escape. This paper proposes a container escape detection method based on a dependency graph to address these challenges. Current detection methods have issues such as limited coverage and low detection accuracy. These issues complicate the traceability of the full attack process and hinder detection.

To overcome these limitations, our approach uses a dependency graph. This enables comprehensive detection of the three main types of container escape attacks. The method not only improves detection accuracy but also enhances the ability to reconstruct complex attack chains.

This section presents a comprehensive overview of the method's architecture and design, focusing on the global framework and the relationships between its key modules. It also details the construction of the dependency graph and explains the core principles guid-

ing the design of the container escape detection process, aimed at providing an innovative and effective solution for container security.

### *3.1. Overall Architecture*

The overall architecture of this solution is shown in Figure 1. It consists of two core components:

1. Container process identification based on label generation and propagation within the dependency graph: First, by analyzing the process behaviors of the container, relevant container attributes (such as *container-id*, *container-dir*, etc.) are generated and propagated to the corresponding process nodes. Then, by generating nodes and edges, a dependency graph for the container is constructed. This process provides the foundational data structure for subsequent security threat detection.
2. Container image vulnerability detection based on file access control: This component constructs a security threat model for the container and uses the dependency graph to track the associations between file nodes and process nodes, thereby detecting potential security threats. This module primarily analyzes file access behaviors inside and outside the container to determine if there are potential escape behaviors or other security vulnerabilities.

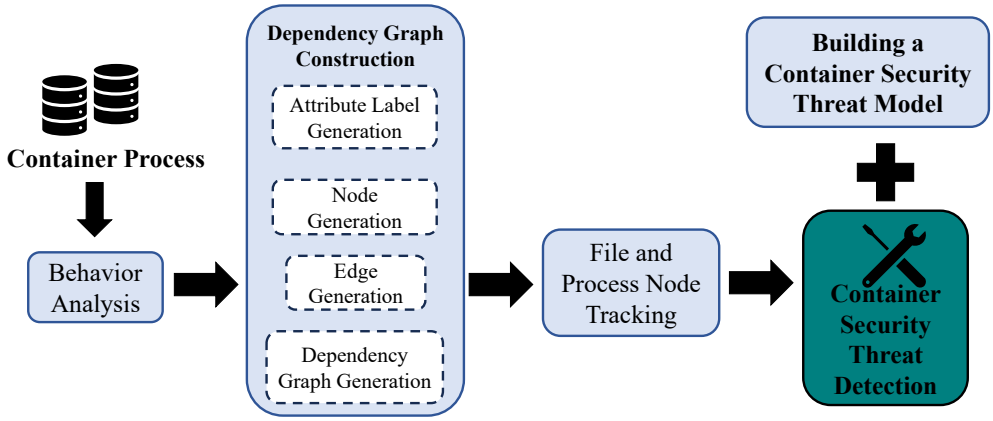



**Figure 1.** The Overall Architecture Diagram.

### *3.2. Dependency Graph Design*

#### 3.2.1. Container Process Behavior Analysis

The Linux kernel uses a Namespace mechanism to isolate container processes. It restricts the resources that containers can access, such as processes, file system mount points, and network stacks. The *cgroup* mechanism further limits resources like CPU, memory, and network bandwidth. Additionally, security mechanisms such as AppArmor, Seccomp, and SELinux apply restrictions to container processes to ensure security.

Aside from these mechanisms, container processes do not fundamentally differ from other system processes.

By analyzing the container startup process, it is observed that containers follow a fixed procedure. The behavior of processes within the container falls within a defined range of activities.

The container startup process is shown in Figure 2, and the specific steps are as follows:

1. The client sends a request to the daemon to create a container.
2. After receiving the request, the daemon (*dockerd*) completes operations such as configuring the container working directory and sends instructions to the container runtime engine (*containerd*) via gRPC.
3. *containerd* starts a *containerd-shim* process for each container, which is responsible for creating the new container.

4. *containerd-shim* invokes the *runc* process to initialize the container. The parameters passed to *runc* specify the configuration path of the container (i.e., the location of config.json), and the root path of the container is also prepared. The container startup process formally begins.
5. The *runc* child process, *runc:[0:PARENT]*, replicates another child process, *runc:[1:CHILD]*, which creates a new namespace via the unshare system call.
6. *runc:[1:CHILD]* spawns further child processes, *runc:[2:INIT]*, to complete container initialization, such as setting up */rootfs*, */proc*, and network stacks.
7. Finally, *runc:[2:INIT]* executes the *execve* system call to run the container's ENTRYPOINT program (such as *sh* or *apache*), which becomes the container's init process, i.e., process 1.

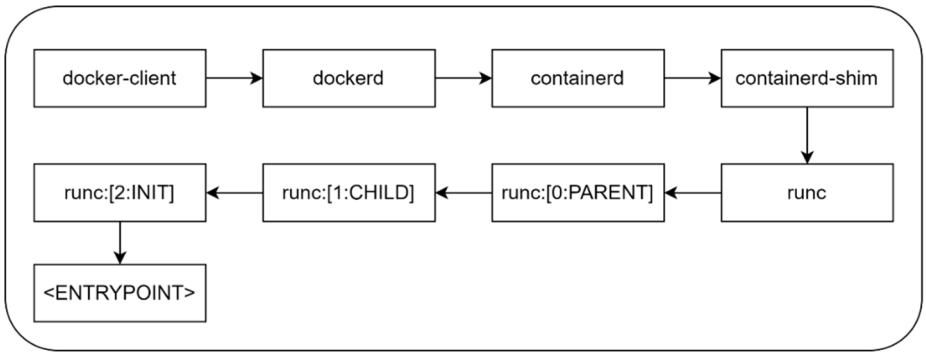



**Figure 2.** Container Startup Process.

In summary, the container is started by *containerd-shim*, which invokes *runc* to launch the container. After the container starts, *runc* exits, and *containerd-shim* becomes the parent process of the container. It is responsible for collecting the container's process status and reporting it to *containerd*. When the container's initial process (i.e., process 1) exits, *containerd-shim* cleans up the remaining child processes within the container to prevent zombie processes. During the container's runtime, the *ps* command on the host shows *containerd-shim* as the parent process for each container.

Based on the above analysis, in the dependency graph, the *containerd-shim* process node can be seen as the starting point of a container. Since it contains container ID information, we can use this node to assign container attribute labels to the *containerd-shim* process and its child processes. This helps in identifying container processes. The specific method will be introduced in the following sections.

#### 3.2.2. Node and Edge Design in the Dependency Graph

1. Node Design

Based on the analysis above, we can distinguish which process nodes in the dependency graph belong to a specific container by following the behavior patterns of Docker container processes. The process nodes contain the attribute labels shown in Table 1. This table summarizes key attributes used to identify and track process nodes in a dependency graph, with each attribute playing a specific role in distinguishing processes and recording critical metadata. The *Type* attribute differentiates between node types, such as processes, files, or network sockets. Name records the name of the process, while *Pid* and *Ppid* store the Process ID and Parent Process ID, respectively, allowing for the identification of processes and their hierarchical relationships. *Uid* and *Gid* log the user and group associated with the process, providing insight into access control and permissions. The *Exe* attribute captures the executable file that initiated the process, linking it to specific binaries. For containerized environments, *Container-id* tags the container to which a process belongs, and *Container-dir* indicates the container's file directory on the host system. Finally, *Command-line* records the command executed by the process, offering a detailed view of the process's actions. These

attributes are essential for tracking processes within containers, ensuring accurate monitoring and detection of potential security risks or anomalous behaviors in edge computing and containerized environments.

**Table 1.** Process Node Attribute Labels.

| Attribute Name | Meaning | Role |
|---|---|---|
| Type | Node type | Distinguishes process, file, and network socket nodes |
| Name | Process name | Records the process name |
| Pid | Process ID | Records the process ID |
| Ppid | Parent process ID | Records the parent process ID |
| Uid | User | Records the user of the process |
| Gid | Group | Records the group of the process |
| Exe | Executable | Records the executable file of the process |
| Container-id | Container ID | Tags the container to which the process belongs |
| Container-dir | Container file directory | Tags the directory location of the container on the host |
| Command-line | Command line | Records the command executed by the process |

Table 2 presents the key attributes for file nodes within a dependency graph. *Type* is used to distinguish node types, such as process, file, or network socket nodes. *Inode* records the file's inode, a critical identifier in the file system that stores metadata about the file. *Path* logs the file's location in the directory structure, allowing for easy tracking of where the file resides. *Permissions* captures the access rights associated with the file, such as read, write, or execute permissions (e.g., 0644). These attributes are essential for monitoring file behavior and ensuring proper access control in the system.

**Table 2.** File Node Attribute Labels.

| Attribute Name | Meaning | Role |
|---|---|---|
| Type | Node type | Distinguishes process, file, and network socket nodes |
| Inode | File inode | Records the file's inode |
| Path | File path | Records the file's path |
| Permissions | File permissions | Records the file's permissions (e.g., 0644) |

Table 3 lists key attributes for network socket nodes in a dependency graph. *Type* distinguishes the node type, while *IP* records the socket's IP address and *Port* captures the socket's port number, providing essential information for network-related processes.

**Table 3.** Network Socket Node Attribute Labels.

| Attribute Name | Meaning | Role |
|---|---|---|
| Type | Node type | Distinguishes process, file, and network socket nodes |
| IP | Socket IP address | Records the socket's IP address |
| Port | Socket port | Records the socket's port |

## 2. Edge Design

Edges represent relationships between nodes. This paper focuses on four types of edges, corresponding to the Open Provenance Model (OPM): *used*, *wasTriggeredBy*, *wasGeneratedBy*, and *wasDerivedFrom*. In addition to the type attribute, edges also have the attributes *eventId* (event ID) and *time* (the time of the event) to determine the event's sequence. The attributes *syscall* (system call that triggered the event) and *operation* (operation corresponding to the event) are used to record the system call and operation of the event, as shown in Table 4.

**Table 4.** Edge Attribute Labels.

| Attribute Name | Meaning | Role |
|---|---|---|
| Type | Edge type | Records the relationship between nodes, including used, wasTriggeredBy, wasGeneratedBy, and wasDerivedFrom |
| EventId | Event ID | Serves as a unique identifier and helps determine the event sequence |
| Time | Time of the event | Helps determine the event sequence |
| Syscall | System call of the event | Records the system call that triggered the event |
| Operation | Operation of the event | Records the operation of the event |

#### 3.2.3. Label Propagation in the Dependency Graph

Each *containerd-shim* process represents a Docker container. Any processes running inside the container are derived from the *containerd-shim* process. Based on this, the paper proposes a method for generating and propagating container attribute labels. These labels are used to identify container processes during the generation of the dependency graph.

First, the method generates a container-id label for each *containerd-shim* process node. This label indicates that these *containerd-shim* processes represent different containers. The container-id is a 64-character string composed of lowercase letters and digits. It can be obtained by querying the command-line of the *containerd-shim* process and matching it using regular expressions.

Once the *container-id* is obtained, a *container-dir* (container directory) label is generated for the process node. This label indicates the file directory of the container on the host. From the host's perspective, Docker container files have specific paths. These paths can be viewed by executing the command docker inspect <container-name> or docker inspect <container-id>.

Finally, processes derived from *containerd-shim* inherit its container attribute labels. This indicates that these processes originate from the same container. Figure 3 summarizes the label generation and propagation process.

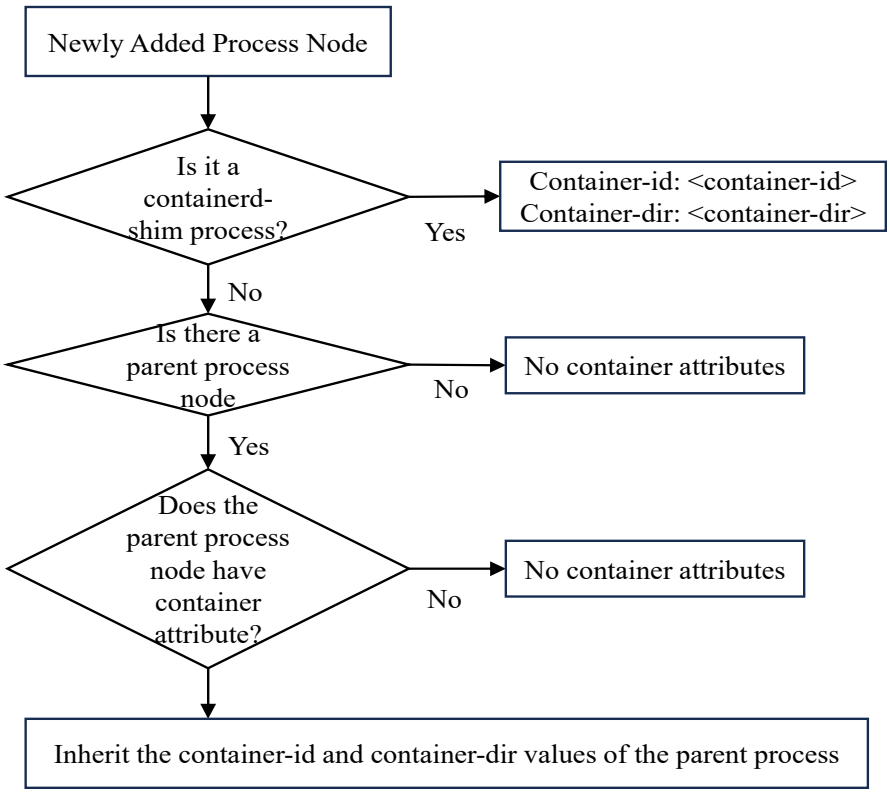



**Figure 3.** The Container Attribute Label Generation and Propagation Flowchart.

In the dependency graph, when a new process node is added, its container attributes are determined. If the process is a *containerd-shim* process, the *container-id* (i.e., the container it belongs to) can be obtained from the command-line property. The *container-dir* (container directory) can be retrieved using the command docker inspect *--format = {{.GraphDriver.Data.MergedDir}} <container-id>*.

If the process is not a *containerd-shim* process, the method checks if it has a parent process. If no parent process exists, the new process node does not have container attributes. However, if the parent process has container attributes, the new process inherits those attributes.

### *3.3. Container Escape Detection*

#### 3.3.1. Container Escape Model

Currently, there is no unified definition of container escape. However, this paper defines it with two main objectives: (1) Gaining command execution capability on the host, and (2) gaining access to files on the host.

After escaping, container escape behavior typically involves the escaped process accessing files on the host. As shown in Figure 4, container process 1 gains the ability to execute commands on the host after escaping. When running processes (escaped processes) on the host, it needs to load the binary files of the process.

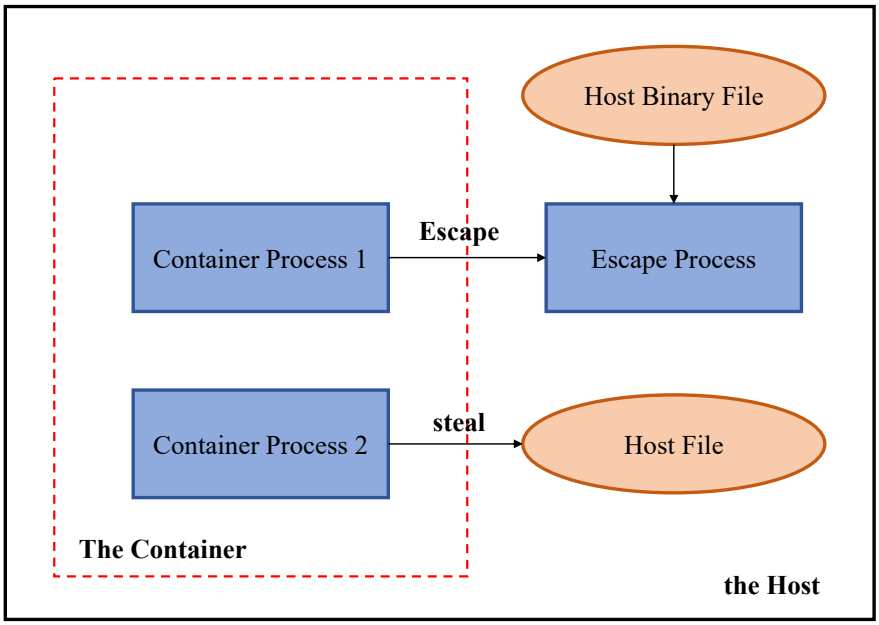



**Figure 4.** Container Escape Model.

Similarly, container process 2 gains access to files on the host after escaping. It then performs operations, such as opening files, to steal data from the host.

#### 3.3.2. Container Escape Detection Method

This section proposes a container escape detection method based on file access control within the dependency graph. The method determines if container escape has occurred by detecting whether processes from the container have accessed files outside the container. Specifically, when a container process node in the dependency graph is associated with any file node (via an edge of type *used* or *wasGeneratedBy*), the method checks if the file and the process belong to the same container. If they do, the association is considered legitimate. If they do not, it is considered a container escape.

Figure 5 shows a simple dependency graph with three process nodes and four file nodes. Process 2's parent is Process 1, and Process 1's parent is the *containerd-shim* process.

According to the container attribute label generation and propagation method described earlier, all three processes are labeled as belonging to the same container. Four files are associated with Process 2.

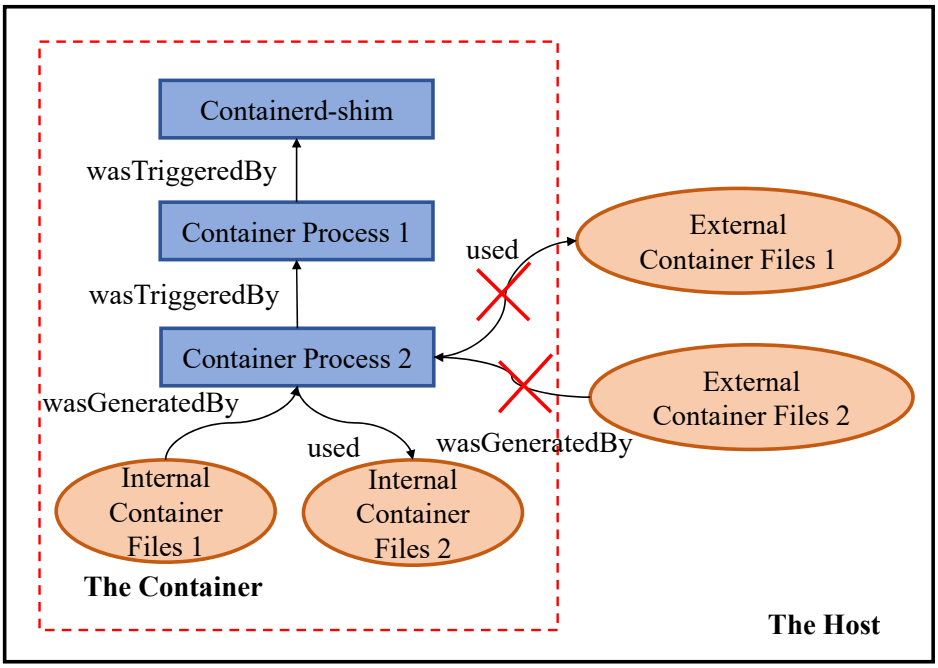



**Figure 5.** Container Escape Detection Model.

Among these, container-internal file 1 and container-internal file 2 belong to the same container as Process 2. Therefore, these associations are legitimate. However, container-external file 1 and container-external file 2 do not belong to the same container as Process 2. Their association is illegal and represents container escape behavior.

Linux Audit records do not indicate whether an event occurred on the host or inside a container. Even if the event happened inside a container, the audit record cannot directly identify which container was involved. For instance, if the */etc/passwd* file is read inside a container, the audit record of *type* = *PATH* shows *name* = "*/etc/passwd*". However, it does not specify whether the file belongs to a container or the host. To resolve this, the paper uses the *inode* (index node) provided by the Linux Audit record to determine the file's location.

An *inode* is one of the most important disk structures in the Linux file system. It describes metadata such as file size, permissions, and the location of file blocks. Unfortunately, an *inode* does not directly provide the file name or path. While it is possible to find the *inode* using a search (e.g., with the shell command *find* <*path*> *-inum* <*inode*>), this method is inefficient. However, the *inode* can be retrieved directly by using the file name.

Based on this, when a container process node in the dependency graph is associated with any file node, and the *container-dir* property (the container's directory on the host) and the file node's path and *inode* are known, the process is straightforward. It is only necessary to check whether "<*container-dir*>/<*path*>" exists and whether its *inode* matches the file node's *inode*. This method determines whether the file and process belong to the same container, and thus, whether container escape has occurred.

There are exceptions for default Mount Namespace mount points within the container. These can be viewed by running the command *cat/proc/self/mountstats* inside the container, as shown in Figure 6. A whitelist can be created to exclude directories like */proc*, */sys*, and */dev*, as well as files such as */etc/resolv.conf*, */etc/hostname*, and */etc/hosts* from path checks. This whitelist can be customized based on the actual scenario.

```
# cat /proc/self/mountstats
device overlay mounted on / with fstype overlay
device proc mounted on /proc with fstype proc
device tmpfs mounted on /dev with fstype tmpfs
device devpts mounted on /dev/pts with fstype devpts
device sysfs mounted on /sys with fstype sysfs
device cgroup mounted on /sys/fs/cgroup with fstype cgroup2
device mqueue mounted on /dev/mqueue with fstype mqueue
device shm mounted on /dev/shm with fstype tmpfs
device /dev/vda3 mounted on /etc/resolv.conf with fstype ext4
device /dev/vda3 mounted on /etc/hostname with fstype ext4
device /dev/vda3 mounted on /etc/hosts with fstype ext4
device devpts mounted on /dev/console with fstype devpts
device proc mounted on /proc/bus with fstype proc
device proc mounted on /proc/fs with fstype proc
device proc mounted on /proc/irq with fstype proc
device proc mounted on /proc/sys with fstype proc
device proc mounted on /proc/sysrq-trigger with fstype proc
device tmpfs mounted on /proc/acpi with fstype tmpfs
device tmpfs mounted on /proc/kcore with fstype tmpfs
device tmpfs mounted on /proc/keys with fstype tmpfs
device tmpfs mounted on /proc/timer_list with fstype tmpfs
device tmpfs mounted on /proc/scsi with fstype tmpfs
device tmpfs mounted on /sys/firmware with fstype tmpfs
```

**Figure 6.** Container Startup Process.

## 4. Evaluation

In the previous section, we provided a detailed explanation of the overall architecture of our solution. This included the design and generation of the dependency graph, as well as the detection of container escape and the reconstruction of the attack process. The use of a dependency graph makes the detection process, especially the attack reconstruction, more structured and comprehensive. This approach allows for the detection of all three types of container escapes. It also enables the full reconstruction of multi-stage attack processes. This section focuses on evaluating the effectiveness of the detection and reconstruction capabilities claimed by our method.

### *4.1. Experimental Environment*

The purpose of the experiments is to verify the following aspects of our method:

1. The ability of the dependency graph generation method to identify container processes and its performance overhead;
2. The detection capability for all three types of container escape;
3. The effectiveness of the attack reconstruction method.

To validate these capabilities, we set up the experimental environment with the following specifications: Ubuntu 22.04 as the operating system, a 5.19.0-38-generic kernel, Docker 18.03.1 as the container runtime, Kubernetes 1.23.1 for orchestration, and Neo4j 4.1.1 as the graph database. Additionally, Linux audit rules were configured as "*-a always,exit -F arch = b64 -S fork -S vfork -S clone -S execve*". Due to the high kernel version (>=5.7), kernel modules do not support namespace recognition, so we did not consider the unshare and setns system calls.

### *4.2. Container Escape Detection Experiments*

To test the effectiveness of our container escape detection method, we replicated six common container escape attack techniques in the experimental environment. We then verified the method's effectiveness.

As this paper focuses on container escape threats, we assume that the experimental environment contains only risks related to container escape. All other components, including the Linux Audit system and all code (both native SPADE and our modifications), are assumed to be secure. Additionally, it is assumed that Linux Audit logs remain intact

and uncompromised. The Neo4j database is also assumed to be untampered with. Vulnerabilities in other layers, such as web applications or hardware, are beyond the scope of this study.

Container escape attacks typically fall into three categories: kernel vulnerabilities, insecure configurations, and vulnerabilities in container-related components. The detection experiments for these three types are described below.

#### 4.2.1. Escape via Insecure Configurations

Insecure configurations, such as dangerous mounts and permissions, can lead to container escape. Unlike software vulnerabilities, these risks are often caused by human error during the container setup. In development environments, developers or system administrators may use improper configurations for convenience. Attackers can then exploit these misconfigurations.

This section lists one common example of container escape due to insecure configurations, along with the test results. Table 5 shows a common insecure configuration leading to container escape.

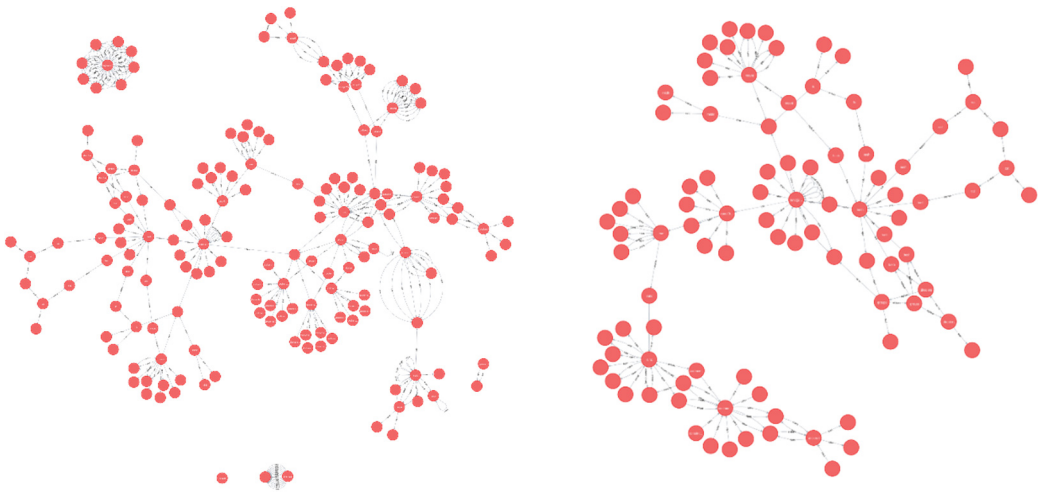
**Table 5.** Common Insecure Configurations Leading to Container Escape.

| Type | Container Configuration Parameter | Exploitation Method |
|---|---|---|
| Privileged Mode | --Privileged | Mount host disk |

The privileged mode was initially designed to enable Docker-in-Docker functionality, but due to its extensive privileges, it poses significant security risks to the host. In privileged mode, all capabilities are enabled for the container, and security mechanisms like AppArmor, Seccomp, and SELinux are disabled, exposing host devices to the container.

A privileged container can mount the host's disk, bypassing the isolation of the file system and enabling access to host files.

We replicated a privileged container escape in the experimental environment, generating the provenance graph shown in Figure 7. (a) contains all the events on the host during the experiment, while (b) focuses on container-related events.



(**a**) (**b**)

**Figure 7.** Provenance Graph for Privileged Container Escape Detection. (**a**) Full Provenance Graph; (**b**) Subgraph of Container Events.

Since the original graph generated by Neo4j is too complex to show the attack process clearly, we manually extracted the core steps and marked where the container escape

detection rules were triggered, as shown in Figure 8. Subsequent experiments follow a similar approach.

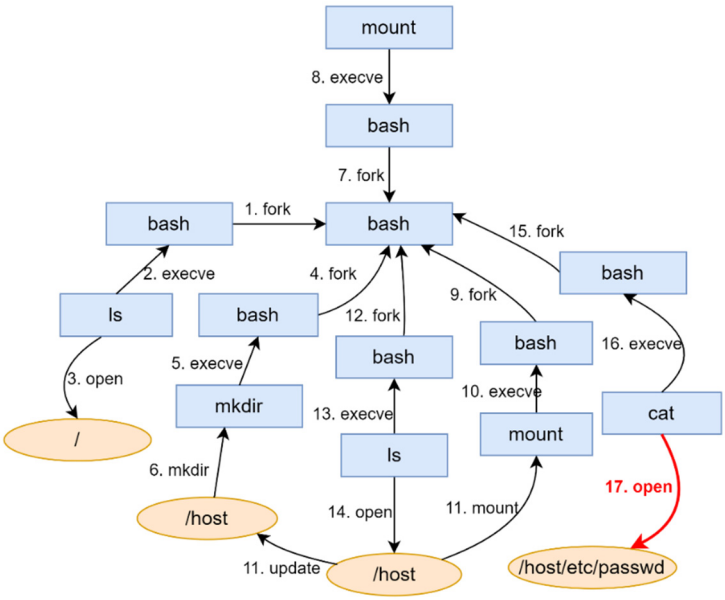



**Figure 8.** Core Steps of Privileged Container Escape in the Provenance Graph.

The experiment shows that privileged container escape can be detected. After mounting the host disk, access to host files triggered the detection rules. Figure 8 provides a clearer view of the core steps of the container escape. In steps 1–6, the current directory in the container was inspected, and a host folder was created. Steps 7–11 show how the host device was mounted to the host folder using the privileged container. Steps 12–17 demonstrate file access, including reading files on the host. In step 17, the cat process in the container accessed the host's */etc/passwd* file, triggering the detection rule.

#### 4.2.2. Escape via Component Vulnerabilities

Container clusters in production environments involve many components, which may have vulnerabilities, including container-related software. This section lists one common component vulnerability, as shown in Table 6. We replicated this vulnerability and tested the effectiveness of our container escape detection method.

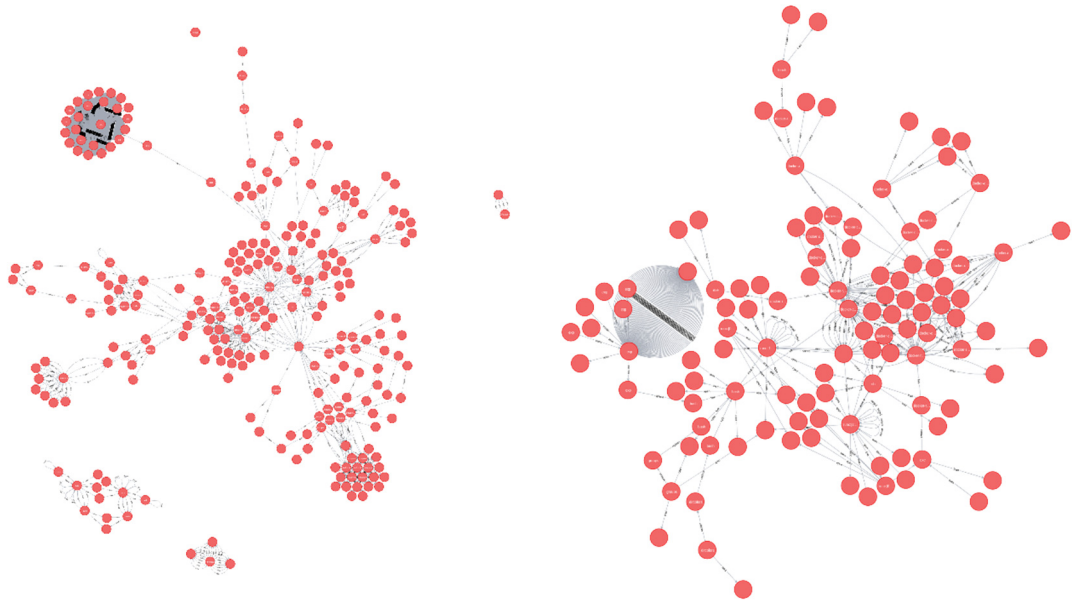
**Table 6.** Component Vulnerabilities Leading to Container Escape.

| CVE Name | CVSS3 Score | Component |
|---|---|---|
| CVE-2019-5736 | 8.6 [26] | runc |

CVE-2019-5736 is a well-known vulnerability in *runc*, where an attacker can overwrite the *runc* binary on the host to execute arbitrary commands.

The vulnerability allows the attacker to obtain the file descriptor (fd) of the *runc* process via the */proc/[PID]/exe* file and inject a payload into *runc*, which is executed the next time *runc* is run. The attack requires interaction between the container and the host's *runc* process.

We replicated this vulnerability, and the resulting provenance graph is shown in Figure 9. (a) shows all events, while (b) focuses on container processes. The dense links between nodes in (b) represent the exp process trying to write payloads into *runc*.



(**a**) (**b**)

**Figure 9.** Provenance Graph for CVE-2019-5736 Escape Detection. (**a**) Full Provenance Graph; (**b**) Subgraph of Container Events.

The experiment shows that CVE-2019-5736 can be detected. In steps 1–10, shown in Figure 10, the exploit is downloaded. Steps 11–13 execute the exploit, rewriting */usr/bin/sh* inside the container. Steps 14–15 involve capturing *runc*'s *PID*, and step 16 involves writing a malicious payload to *runc*. Steps 17–22 show the execution of *runc*, now modified to run the payload, resulting in file creation on the host in step 22 (touch */tmp/pwn-success*).

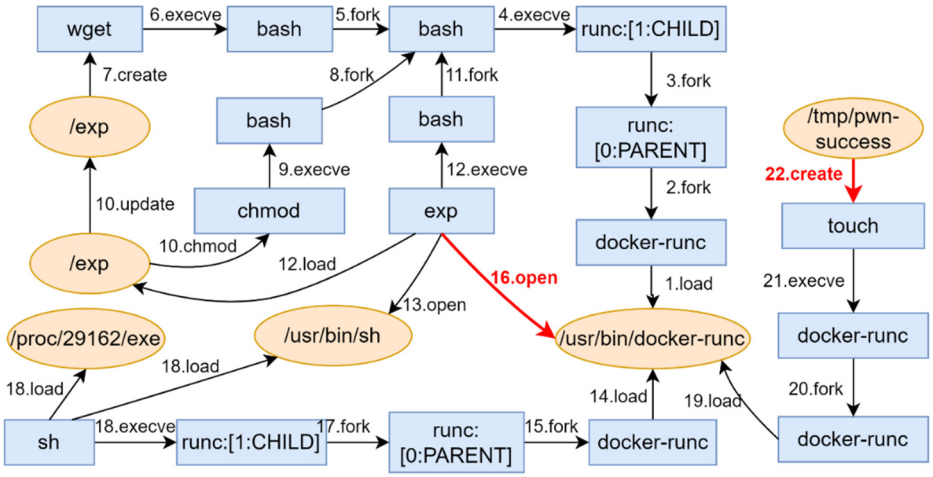



**Figure 10.** Core Steps of CVE-2019-5736 Escape in the Provenance Graph.

Steps 16 and 22 triggered the detection rules: step 16 represents the container's exp process overwriting *runc* on the host, and step 22 represents the execution of the payload.

#### 4.2.3. Escape via Kernel Vulnerabilities

Kernel vulnerabilities are highly impactful, especially those affecting shared resources like containers. Containers and the host share the same kernel, meaning any kernel vulnerability affects all containers running on the host. However, not all kernel vulnerabilities can be exploited for container escape. This section replicates two harmful kernel vulnerabilities, shown in Table 7, and tests their detection.

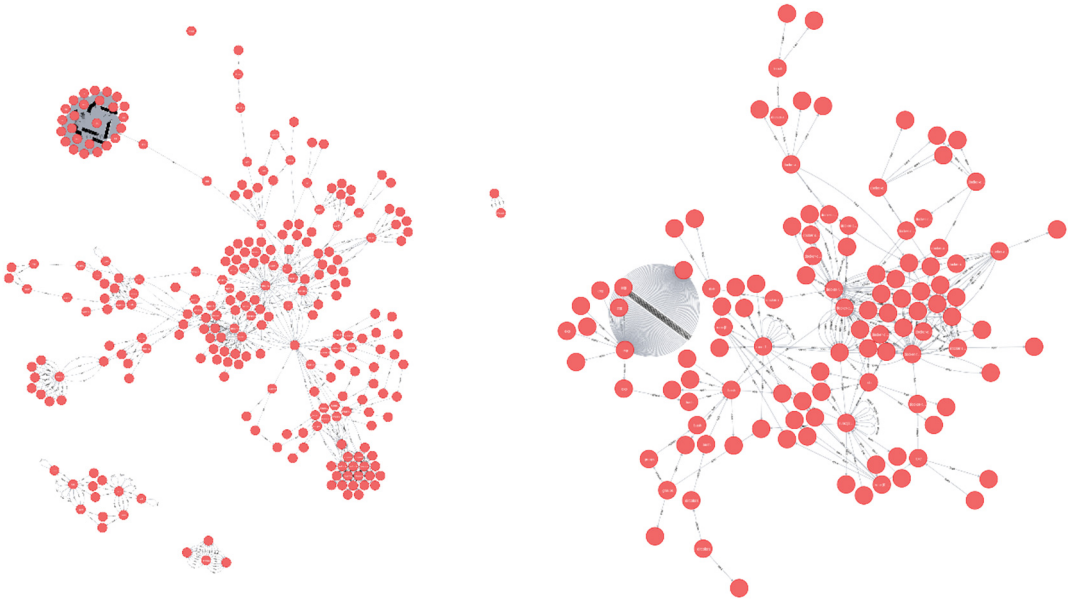
**Table 7.** Kernel Vulnerabilities Leading to Container Escape.

| CVE Name | CVSS3 Score | Component |
|---|---|---|
| CVE-2022-0847 | 7.8 [27] | pipe |

CVE-2022-0847, or *DirtyPipe*, is a file overwrite vulnerability affecting Linux kernel 5.8 and above. It allows any user to overwrite arbitrary files, similar to *DirtyCOW*.

*DirtyPipe* allows the modification of the file cache via a pipe's buffer, enabling file overwrites. While *DirtyPipe* cannot directly cause container escape, it can be combined with CVE-2019-5736 to overwrite *runc* and achieve escape.

We replicated the *DirtyPipe* vulnerability by overwriting *runc*, generating the provenance graph shown in Figure 11. (a) shows all events, while (b) focuses on container processes. The dense links between nodes represent the exp process writing payloads into *runc* using *DirtyPipe*.



(**a**) (**b**)

**Figure 11.** Provenance Graph for CVE-2022-0847 Escape Detection. (**a**) Full Provenance Graph; (**b**) Subgraph of Container Events.

The experiment demonstrates that using CVE-2022-0847 to overwrite *runc* for container escape can be detected. Although the process of overwriting *runc* via the kernel cannot be detected, the subsequent execution of *runc* triggered the detection rules. Steps 1–7, shown in Figure 12, involve executing the exp process in the container, which rewrites */bin/sh*. Steps 8–11 show the interaction with *runc*. Steps 11–15 show the execution of *runc*, now modified by *DirtyPipe*, resulting in file creation on the host.

This section validates the effectiveness of the dependency graph-based method for detecting container escape attacks through experimental results, illustrating its ability to identify and reconstruct various attack types. Initially, the experiments evaluated the method's capacity to generate dependency graphs, showcasing its outstanding performance in recognizing container processes, particularly within insecure configurations. The method's flexibility and scalability allow it to dynamically adapt to emerging or evolving escape techniques. Specifically, in scenarios involving collaborative attacks that exploit kernel vulnerabilities to obscure the *runc* process, the method can update the dependency graph in real time, monitor potential anomalies, and swiftly modify detection strategies, thereby reducing reliance on traditional detection systems. This adaptability enhances

the response to novel attacks and provides effective support against increasingly complex cybersecurity threats.

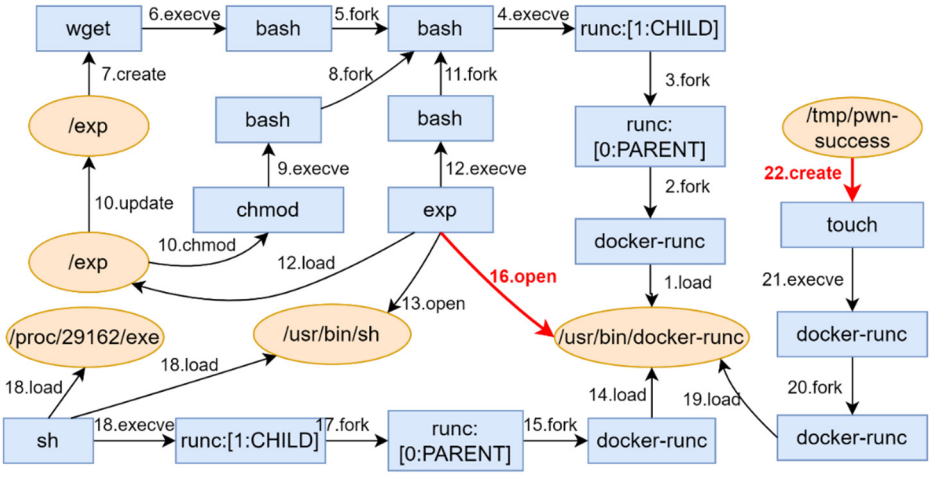



**Figure 12.** Core Steps of CVE-2022-0847 Escape in the Provenance Graph.

In testing component vulnerabilities, the method successfully reproduced CVE-2019-5736, capturing the process by which attackers exploit the *runc* vulnerability for container escape. This finding affirms the method's detection capabilities and highlights the security risks associated with real-world deployments, urging development and operations teams to prioritize security updates for software components. Additionally, experiments focused on kernel vulnerabilities, particularly CVE-2022-0847, underscore the significant implications of these vulnerabilities for container security, emphasizing the critical need for heightened awareness of kernel security when implementing container technologies.

The findings of this section confirm the application of dependency graphs in container security detection, indicating that this method excels in detection while also providing structured support for attack tracing and reconstruction through its inherent flexibility and scalability, thereby establishing a foundation for future research in container security.

## 5. Conclusions

To address the issue that existing dependency graph generation methods cannot distinguish between container and non-container processes, we introduced a container process identification module. This module is based on label generation and propagation within the dependency graph. It captures system events and generates audit logs. These logs are used to produce node and edge data for generating the dependency graph.

In addition, we addressed the limited coverage of existing container escape detection methods. We introduced a detection module based on file access control within the dependency graph. This module identifies container escapes by detecting whether a container process accesses files outside its container.

**Author Contributions:** All authors contributed equally to the conceptualization, methodology, data analysis, and writing of the manuscript. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the State Grid Corporation of China Headquarters Management Science and Technology Project: Research on Key Technologies for Vulnerability Mining and Security Assessment of Power Proximity Networks and Edge Devices, grant number 5700-202418245A-1-1-ZN.

**Data Availability Statement:** Data are contained within the article.

**Conflicts of Interest:** Authors Jing Guo and Zhimin Gu were employed by the company State Grid Jiangsu Electric Power Co., Ltd. Research Institute. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## References


1. NSFOCUS. Cloud Native Security Technology Report. Available online: https://www.nsfocus.com.cn/html/2021/92_0113/146.html (accessed on 10 September 2024).
2. Van't Hof, A.; Nieh, J. Androne: Virtual Drone Computing in the Cloud. In Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, 25–28 March 2019; pp. 1–16.
3. Mochafreddo, A. Understanding Docker Containers: Leveraging Linux Kernel's Namespaces and cgroups. Dev.to. 2021. Available online: https://dev.to/mochafreddo/understanding-docker-containers-leveraging-linux-kernels-namespaces-and-cgroups-4fkk (accessed on 10 September 2024).
4. Gao, X.; Gu, Z.; Kayaalp, M.; Zhang, H.; Lin, Z. Containerleaks: Emerging Security Threats of Information Leakages in Container Clouds. In Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, 26–29 June 2017; pp. 237–248.
5. Yang, N.; Shen, W.; Li, J.; Luo, X. Demons in the Shared Kernel: Abstract Resource Attacks Against OS-Level Virtualization. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Seoul, Republic of Korea, 15–19 November 2021; pp. 764–778.
6. Jian, Z.; Chen, L. A Defense Method Against Docker Escape Attack. In Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, Guangzhou, China, 25–28 January 2017; pp. 142–146.
7. Xu, K.; Zhang, X.; Li, X. Research on Docker Container Escape Protection Technology. *Inf. Secur. Res.* **2022**, *8*, 768.
8. Aktolga, İ.T. A Study on Analysis and Detection of Container Escape Vulnerabilities in Docker. Master's Thesis, Middle East Technical University, Ankara, Turkey, 2024.
9. He, Y.; Guo, R.; Xing, Y.; Che, X.; Sun, K.; Liu, Z.; Xu, K.; Li, Q. Cross Container Attacks: The Bewildered {eBPF} on Clouds. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 5971–5988.
10. Reeves, M.; Tian, D.J.; Bianchi, A.; Bagchi, S.; Payer, M. Towards Improving Container Security by Preventing Runtime Escapes. In Proceedings of the 2021 IEEE Secure Development Conference (SecDev), Arlington, VA, USA, 18–19 October 2021; pp. 38–46.
11. Abbas, M.; Khan, S.; Monum, A.; Zhang, L.; Ahmad, R. PACED: Provenance-Based Automated Container Escape Detection. In Proceedings of the 2022 IEEE International Conference on Cloud Engineering (IC2E), San Francisco, CA, USA, 4–7 April 2022; pp. 261–272.
12. Zhang, Y.; Fang, B.; Du, C.; Wu, L. Container Escape Detection Method Based on Heterogeneous Observation Chain. *J. Commun.* **2023**, *44*, 49–63.
13. V S, D.P.; Sethuraman, S.C.; Khan, M.K. Container security: Precaution levels, mitigation strategies, and research perspectives. *Comput. Secur.* **2023**, *135*, 103490. [CrossRef]
14. Zipperle, M.; Gottwalt, F.; Chang, E.; Gruschka, N.; Ren, K.; Sun, L. Provenance-Based Intrusion Detection Systems: A Survey. *ACM Comput. Surv.* **2022**, *55*, 1–36. [CrossRef]
15. Han, X.; Pasquier, T.; Seltzer, M. Provenance-Based Intrusion Detection: Opportunities and Challenges. In Proceedings of the 10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018), London, UK, 11–12 July 2018.
16. Li, Z.; Chen, Q.A.; Yang, R.; Zeldovich, N. Threat Detection and Investigation with System-Level Provenance Graphs: A Survey. *Comput. Secur.* **2021**, *106*, 102282. [CrossRef]
17. Wong, A.Y.; Chekole, E.G.; Ochoa, M.; Zhou, J. On the security of containers: Threat modeling, attack analysis, and mitigation strategies. *Comput. Secur.* **2023**, *128*, 103140. [CrossRef]
18. Zhao, L.; Zou, Y.; Xu, C.; Ma, Y.; Shen, W.; Shan, Q.; Jiang, S.; Yu, Y.; Cai, Y.; Song, Y.; et al. Robust Soliton Distribution-Based Zero-Watermarking for Semi-Structured Power Data. *Electronics* **2024**, *13*, 655. [CrossRef]
19. Yang, Y.; Shen, W.; Guo, Q.; Shan, Q.; Cai, Y.; Song, Y. EPA-GAN: Electric Power Anonymization via Generative Adversarial Network Model. *Electronics* **2024**, *13*, 808. [CrossRef]
20. King, S.T.; Chen, P.M. Backtracking Intrusions. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 19–22 October 2003; pp. 223–236.
21. Du, M.; Li, F.; Zheng, G.; Srikumar, V. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 1285–1298.
22. Garchery, M.; Granitzer, M. ADSAGE: Anomaly Detection in Sequences of Attributed Graph Edges Applied to Insider Threat Detection at Fine-Grained Level. *arXiv* **2020**, arXiv:2007.06985.
23. Song, Y.; Jiang, S.; Shan, Q.; Yang, Y.; Yu, Y.; Shen, W.; Guo, Q. Hierarchical-Based Dynamic Scenario-Adaptive Risk Assessment for Power Data Lifecycle. *Electronics* **2024**, *13*, 631. [CrossRef]
24. Hossain, M.N.; Milajerdi, S.M.; Wang, J.; Borisov, N.; Jha, S. SLEUTH: Real-Time Attack Scenario Reconstruction from COTS Audit Data. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 487–504.

25. Chen, X.; Irshad, H.; Chen, Y.; Tian, D.J. CLARION: Sound and Clear Provenance Tracking for Microservice Deployments. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 11–13 August 2021; pp. 3989–4006.
26. National Vulnerability Database. CVE-2019-5736: Vulnerability in runc. Available online: https://nvd.nist.gov/vuln/detail/CVE-2019-5736 (accessed on 30 October 2024).
27. National Vulnerability Database. CVE-2022-0847: DirtyPipe Vulnerability. Available online: https://nvd.nist.gov/vuln/detail/CVE-2022-0847 (accessed on 30 October 2024).

**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.