Houdini: Security Benchmarking of Container Confinement

Huzaifa Patel
Carleton University

David Barrera
Carleton University

Anil Somayaji
Carleton University

Abstract

While container-based workloads are now a standard part of our cloud infrastructure, container security remains a challenging problem. Container confinement is a particularly pressing problem, as without it, a single vulnerable application can be used to compromise entire clusters of containers. While we have many technologies that can be used to secure containers, currently there is no easy way to determine whether a given Docker configuration provides even a basic level of protection. Here we present `Houdini`, a security benchmark for container confinement. Much as network scanning tools can help catch misconfigured firewalls, `Houdini` can check whether a given Docker container configuration properly enforces confinement. While it can be used to test deployable containers, `Houdini` is optimised for testing and comparing container confinement technologies. Here we present the motivation, design, implementation, and initial results on running `Houdini` on a set of different container configurations. By providing a benchmark framework by which container confinement technologies can be evaluated, we believe `Houdini` can help foster the development of next-generation container confinement technologies.

1 Introduction

Container-based workloads have become a cornerstone of modern cloud infrastructure, which has revolutionized how applications are developed, deployed, and scaled. Unlike hypervisor-based virtual machines, which require a full operating system for each instance and rely on a hypervisor to manage hardware virtualization, containers are lightweight, share the host system's kernel, and incur minimal overhead. This efficiency has made containers an attractive choice for organizations seeking to optimize resource utilization and reduce operational costs. Furthermore, containers enable seamless portability by encapsulating all userspace dependencies into a unified image. Despite container's numerous benefits, they introduce new security challenges, particularly around container confinement.

Linux features such as cgroups, namespaces, and mount-points create the appearance of isolation for containers. These mechanisms ensure that they are confined to their own resource, process, and network boundaries. However, this separation can fail due to kernel vulnerabilities, misconfigurations, or even sophisticated attacks like container escapes, allowing attackers to disrupt other containers or compromise the host system. Technologies like SELinux, AppArmor, and seccomp provide additional layers of security but are notoriously complex to configure and prone to misconfiguration. A best practices guide for container security [1] can help with configuring a container, but it offers no guarantee that the resulting configuration performs as intended. For instance, how can we confirm that cgroups and namespaces are restricting containers as intended? As container adoption skyrocketed across industries, particularly in multi-tenant cloud environments, these security challenges have become more pronounced. To address this challenge, the cloud industry has developed technologies like gVisor and Kata Containers, which isolate containers further by running them within lightweight virtual machines. While this approach is effective, it unnecessarily increases complexity and overhead. By developing methods to audit and validate confinement mechanisms, container security could be simplified by reducing reliance on such solutions.

Test suites play a crucial role in ensuring that systems meet requirements and function correctly under defined conditions. For instance, the SPEC (Standard Performance Evaluation Corporation) benchmarks measure CPU performance and power efficiency, while the TPC (Transaction Processing Performance Council) benchmarks assess database performance. The Phoronix Test Suite is another widely used option, particularly in Linux-based systems. Other notable benchmarks include Geekbench, a cross-platform tool for evaluating CPU and GPU performance, and IOzone, which focuses on file system I/O performance. In software development, regression tests play a critical role in ensuring that code changes do not inadvertently break existing functionality. Many organizations enforce policies requiring code to pass these tests before

integration. Test suites also ensure the functionality of production systems, such as uptime monitors that verify whether services are operating as intended. Additionally, functional test suites, though less frequently used, help identify security vulnerabilities in deployed systems. Tools like network scanners are especially valuable for detecting misconfigurations, insecure software versions, and unintended service exposures.

Despite the widespread use of test suites in these areas, a significant gap remains in the context of containers. There are currently no standardized or widely adopted test suites designed specifically to evaluate container security, functionality, or isolation effectiveness. This absence leaves organizations relying on ad hoc methods, which are often inadequate for uncovering subtle vulnerabilities or ensuring robust confinement. Given the growing reliance on containerized environments in modern infrastructure, the development of comprehensive test suites tailored for containers is imperative. Such test suites would not only address these critical gaps but also provide a reliable foundation for improving trust and security in containerized systems.

In this paper, we describe the implementation of `Houdini`, the first test suite for verifying container confinement. Given a Docker container configuration (including both the kernel version and the docker version), Houdini will instantiate the container in a standalone QEMU-based virtual machine and perform multiple tests (or tricks, as we call them) to evaluate whether the configuration is susceptible to various forms of container misconfigurations or failures. These tests explore vulnerabilities that could disrupt container functionality, such as improper resource allocation, inadequate isolation, or insecure filesystem setups. Houdini is written in Python and is easily extensible, providing an extension language for writing tricks. Houdini is designed to assess and enhance the security of containerized environments. It tests the isolation of containers, ensuring they remain properly confined and separated from the host system and other containers. Houdini specializes in identifying container-specific vulnerabilities, misconfigurations, and ensuring adherence to security best practices.

A standard methodology for assessing container confinement is crucial due to the complexity of modern systems and the intricate vulnerabilities they present. A formal approach involving the rigorously evaluating and analyzing the security of the container system might seem ideal, however, it is impractical because container environments are highly complex, with interactions between kernels, runtimes, orchestration tools, and external dependencies that are difficult to model comprehensively. Moreover, many vulnerabilities stem from subtle details that a formal model might overlook. Instead, an empirical approach offers a practical alternative. Though it will never be exhaustive, the goal is not comprehensive coverage but rather establishing a baseline set of tests that demonstrate effective container confinement. This approach is achievable and can evolve over time as new vulnerabilities are identified, allowing the methodology to remain current. Starting with tests for isolation, privilege escalation prevention, resource limitation enforcement, and resistance to known exploits provides a solid foundation. By focusing on demonstrating security effectiveness in critical areas, this methodology can enhance confidence in container confinement while enabling continuous improvement and adaptation to the evolving threat landscape.

In this paper we describe Houdini's motivation, design, and implementation. We also present the results of case studies showing how Houdini can be used to detect misconfigured containers that allow for privilege escalation attacks on the host system. Our hope with Houdini is that it will help with the deployment of better confined containers and will support the development of new technologies to more reliably confine containers.

The rest of this paper proceeds as follows. In Section 2, we explain the container confinement problem and associated technologies. We describe Houdini's design and implementation in Section 5. Section **??** explains the tricks Houdini currently implements and their associated vulnerabilities. In Section 7 we present case studies showing how Houdini can detect basic misconfigurations. Section 3 describes related work, Section **??** discusses the contributions, limitations, and our plans for future work. Section 8 concludes our paper.

2 Container Confinement Problem

Container confinement can be defined as the set of mechanisms and strategies used to isolate a containerized application from the host system and other containers. Effective confinement is critical for maintaining system security and stability, as it ensures that containers operate within defined boundaries by restricting their access to system resources, processes, and networks.

The level of confinement of a container is not fixed; it can vary based on the specific requirements or intentions of the user. In other words, the degree to which a container is isolated from the host system or other containers is customizable to meet the user's needs. The optimal configuration, however, is one that strikes a balance between maximum confinement (restricting access to system resources as much as possible) and functional flexibility (enabling the container to perform its intended tasks). Achieving this balance is the core goal of container confinement.

There is a semantic gap in containerized systems, where the intended security policies and the actual behavior of the system don't always align. The semantic gap refers to the difference between what a system is meant to do and how it actually behaves in practice. In other words, it's the disconnect between the theoretical design of the system and its real-world execution. This gap can occur for various reasons, such as miscommunications, misunderstandings, or limitations in the system's design or configuration.

For example, when setting up a security system, an administrator may want to ensure that containers in a cloud environment are completely isolated from each other and the host system. The goal is to set up strict access controls so that containers cannot interact with each other or the host. However, due to complex settings, misconfigurations, or flaws in how security mechanisms (like namespaces, cgroups, SELinux, etc.) interact, the system might not enforce these isolation policies correctly. As a result, the actual performance may not align with the intended security model.

This gap can happen in many systems, but it is especially common in complex environments like containers, where multiple interdependent components (e.g., the container runtime, kernel features, and security tools) must work together smoothly. Misunderstandings or mistakes in configuring these components can lead to unexpected issues or security vulnerabilities, even when the system seems correctly set up. The semantic gap highlights the challenge of ensuring that the real-world behavior of the system matches the user's or designer's intentions. Houdini helps bridge this gap in container confinement by providing an empirical framework to test and verify whether a container's security mechanisms are functioning as intended.

3 Related Work

At the time of this writing, to our knowledge, `Houdini` is the first comprehensive approach for evaluating and comparing *container confinement mechanisms*. Much of the container security literature to date has focused on either vulnerability scanning of container images, building offensive/attack tools for container escapes, or proposing best practices for securing container runtimes. While these papers, tools, and documents do not directly share our research objectives, they still broadly fit into the container security landscape. The remainder of this section does not aim to exhaustively enumerate all container security tools and systems, but rather to highlight the various salient methods.

Vulnerability Scanners. Many free, open, and closed-source tools exist for identifying the presence of known vulnerabilities in container images. Shu *et al.* [2] developed a vulnerability scanning tool to scan Docker Hub container images at scale. They found that on average, official and community images have concerning amounts of vulnerabilities (180+), and that these vulnerabilities remain unpatched for hundreds of days. Their study utilized the Docker Image Vulnerability Analysis (DIVA) framework, which automates the discovery, download, and analysis of over 300,000 Docker images. The findings revealed that more than 80% of both official and community images contained at least one high-severity vulnerability, and many images were not updated for extended periods. Moreover, vulnerabilities were often inherited from parent images to child images, further increasing security risks.

Best Practices. Users in search of (often high-level) advice on how to improve the security of their container deployments can consult one of many best practices guides available online. One such guide is the Center for Interent Security (CIS) Docker Benchmark [1]. In our experience, guides tend to offer generic advice (e.g., "*Only allow read access to the root filesystem*". This is problematic because it oversimplifies container security. Generic advice like "Only allow read access to the root filesystem" may seem straightforward, but it doesn't account for the specific context or unique configuration of an individual deployment. Such one-size-fits-all recommendations can lead users to believe that simply following these broad guidelines is enough to secure their systems, potentially leaving critical vulnerabilities unaddressed. In reality, effective container security requires a nuanced, tailored approach that considers the particular needs and threat models of each environment. Houdini fills the gap between generic advice and practical security by enabling you to objectively evaluate the effectiveness of your container confinement measures. In the evaluation section, we leverage the best practices guide [1] with `Houdini`, to rigorously assess whether container isolation mechanisms achieves confinement based on different components that make up the Docker environment.

Offensive Tools. The offensive security community has developed a range of specialized tools aimed at identifying and exploiting vulnerabilities within containerized environments. Containers, while providing strong isolation, are not immune to attacks, and these tools focus on testing the robustness of container defenses. CDK[1] and DEEPCE[2] are two widely used, open-source container penetration testing toolkits that leverage a collection of known exploits to gain persistence, escape the container, and gather sensitive information about the container environment. These tools aim to bypass isolation mechanisms like namespaces, cgroups, and SELinux, often using a variety of aggressive techniques to identify weaknesses in container security. CDK specifically uses any available method to circumvent security measures, providing a broad toolkit to test the limits of a container's defense. Meanwhile, DEEPCE, developed by stealthcopter, focuses on privilege escalation and container enumeration, attempting to exploit specific weaknesses to achieve container escapes, thereby compromising the host system. In contrast, Houdini takes a more systematic and structured approach to container security testing. Rather than focusing solely on exploiting vulnerabilities, Houdini allows for the direct comparison of defensive techniques by clearly defining the testing environment, outlining the exploit steps, and documenting the expected outputs.

[1] https://github.com/cdk-team/CDK
[2] https://github.com/stealthcopter/deepce

4 Linux Confinement Mechanisms

On Linux, process containment relies on several key technologies, including Unix, cgroups, namespaces, Linux capabilities, apparmor, selinux, and seccomp. Below, we provide an overview of each of these mechanisms.

cgroups Cgroups, or control groups, are a Linux kernel feature that allows system administrators to allocate and manage system resources (such as CPU, memory, disk I/O, and network bandwidth) for processes or groups of processes. When you create a cgroup, you are essentially grouping together processes and imposing resource constraints on them. This is crucial for resource management and isolation in multi-user or containerized environments. For instance, cgroups allow you to set a limit on the amount of memory a set of processes can use, preventing any single process from consuming excessive resources and affecting other processes or the system as a whole. Additionally, cgroups allow for the prioritization of resources, ensuring that critical processes receive more CPU or memory than less important ones. They also offer monitoring capabilities to track how much of each resource a process or cgroup is consuming. This feature is extensively used in containerization technologies like Docker, where containers are assigned their own resource limits via cgroups to ensure fairness and prevent resource hogging.

namespaces Namespaces are another crucial Linux kernel feature that provides process isolation. A namespace isolates certain aspects of a system for a process or group of processes, ensuring that they have their own independent environment. There are several types of namespaces, including PID, network, mount, user, and UTS namespaces. For example, a PID namespace allows processes within it to have their own process IDs, meaning that processes in different PID namespaces can have the same process ID without interfering with each other. Similarly, network namespaces allow processes to have their own network stack, which includes IP addresses, routing tables, and network interfaces. This isolation is fundamental for containers, as it ensures that processes running in one container cannot see or interact with processes in another, even though they may be running on the same physical host. Namespaces, combined with cgroups, provide the foundation for building lightweight, secure containers in Linux systems.

Linux capabilties Linux capabilities are a fine-grained security mechanism that splits the traditional superuser (root) privileges into smaller, more manageable pieces. Instead of giving a process full root access, Linux capabilities allow for the assignment of specific privileges to processes, such as the ability to bind to a network port or change the system time, without granting full administrative control. This is particularly useful for increasing system security, as it limits the potential damage caused by vulnerabilities in applications. For example, a web server might require the ability to bind to port 80 but should not need full root privileges. By assigning only the necessary capabilities to a process, the security surface area is reduced, making it harder for an attacker to exploit. This approach is commonly used in containerized environments, where processes are run with just the capabilities they need to function.

Seccomp (Secure Computing Mode) Seccomp is a security feature in the Linux kernel that allows administrators to filter and restrict the system calls that processes can make. It provides a mechanism to limit the actions that a process can perform, which is particularly useful in reducing the attack surface of applications, especially in containerized environments. With seccomp, you can create a whitelist or blacklist of system calls, effectively controlling which parts of the operating system a process can access. For instance, you could block potentially dangerous system calls like execve, which is used to execute programs, or restrict the ability to open files by denying access to certain file descriptors. This reduces the risk of an attacker exploiting a process to gain control over the system. Seccomp is often used in combination with other security mechanisms like AppArmor and SELinux to provide an additional layer of protection.

App Armor (Application Armor) AppArmor is a Linux security module that provides Mandatory Access Control (MAC) by enforcing security policies on individual programs. It restricts the actions that applications can perform by defining profiles that specify which files, directories, and capabilities the program can access. AppArmor operates by using a set of predefined or custom profiles, which are attached to programs, to limit their access to system resources. For example, an AppArmor profile for a web server might restrict the server's access to specific directories and prevent it from executing arbitrary binaries or making network connections to external IP addresses. AppArmor is easier to configure and manage than some other security modules, and it provides a level of protection by ensuring that even if an application is compromised, the attacker has limited access to the system.

selinux SELinux is a security module that provides more granular control than traditional discretionary access control (DAC) by using mandatory access control (MAC). It enforces policies that define which processes or users can access specific files or resources, based on security contexts. These contexts are applied through SELinux policies, which restrict access even if a process has standard permissions. SELinux is commonly used in high-security environments, such as government or financial systems, to prevent unauthorized access. While powerful, it requires careful configuration to avoid restricting legitimate processes and reduce the risk of privilege escalation.

5 Design

Houdini is built using Python 3 and leverages a QEMU VM to provide a controlled testing environment for container security validation. Instead of running containers directly on the host system, Houdini spins up a dedicated VM, ensuring that any

security vulnerabilities or container escapes remain isolated and do not affect the host system. This setup allows for highly reproducible security tests while minimizing unintended side effects on the underlying infrastructure.

At the core of Houdini's architecture, a Flask-based server runs on the host operating system (not inside the VM). This server acts as a bridge between the host and the QEMU virtual machine, facilitating communication between them. The Flask server is responsible for sending commands and configuration data to the guest VM and receiving trick execution results. When a trick is initiated, the Houdini client on the host instructs the VM to start a new test. The VM then launches a container with the specified security configurations and executes the trick inside it.

Once the trick completes, the VM collects the results and sends them back to the host via the Flask server. The host then processes this data, logs the results, and determines whether the container security mechanisms functioned as expected. This structured workflow ensures that Houdini remains independent of the host system's container runtime while providing a reliable and repeatable environment for testing container confinement mechanisms.

5.1 Design Goals

`Houdini` is designed to provide a systematic and reproducible approach to testing container confinement mechanisms. One of its primary goals is to evaluate whether container security features, such as namespaces, cgroups, seccomp, and Linux capabilities are effectively enforcing isolation. Many security tools rely on theoretical guarantees or static analysis, but Houdini focuses on empirical validation by executing controlled security tests, known as "tricks", that verify the correct enforcement of security mechanisms. By doing so, `Houdini` ensures that container security mechanisms are correctly configured.

One of the goals of `Houdini` is not only for testing container isolation but also as a means to optimize container configurations. By systematically simulating various tasks within Docker containers, Houdini can help determine the minimal set of privileges required for a container to perform its intended function. For example, administrators can start with a container configured with broad privileges and then incrementally reduce permissions, such as Linux capabilities, seccomp filters, or resource limits, while monitoring task success. Houdini's testing framework will identify the point at which functionality is maintained while unnecessary privileges are removed, thereby pinpointing the least privilege configuration that still allows the container to operate effectively. This approach is highly beneficial because it adheres to the principle of least privilege. Minimizing the privileges granted to a container reduces its attack surface, making it significantly harder for an attacker to exploit vulnerabilities.

The following are additional goals of Houdini:

Verifying Whether Security Mechanisms Are Truly Working as Expected: Many security mechanisms in containerized environments, such as Linux namespaces, cgroups, seccomp filters, and capabilities—are intended to enforce process isolation and prevent privilege escalation. However, just because a security feature is enabled does not necessarily mean it is functioning correctly. Misconfigurations, improper runtime enforcement, and subtle implementation flaws can lead to unexpected security gaps. Houdini helps verify whether these security mechanisms are actually enforcing the intended restrictions by executing controlled tests (tricks) that attempt to bypass or manipulate confinement policies. If a test succeeds in breaking out of isolation, it provides direct evidence that the security mechanism is not functioning as expected.

Identifying Gaps Between Intended Confinement Policies and Actual Enforcement: There is often a discrepancy between security policies that administrators configure and what is actually enforced at runtime. For example, a security policy may specify that a container should not have network access, but due to a misconfigured Docker setting, the container may still be able to establish outbound connections. Similarly, a container may be configured with resource constraints (e.g., CPU and memory limits via cgroups), but in practice, those constraints may be ineffective or bypassable. Houdini systematically tests these security policies against real execution environments to uncover where enforcement fails. By detecting these discrepancies, organizations can adjust their configurations and improve security postures, ensuring that containers remain properly confined under expected conditions.

Challenging Misleading Security Assumptions with Empirical Testing: Container security is often discussed in terms of theoretical guarantees, with many security best practices based on assumptions rather than direct testing. For instance, documentation may state that a container running in unprivileged mode cannot escalate to root, but Houdini actively tests whether that is actually the case under different container configurations. By running real-world security validation instead of relying solely on security claims, Houdini helps challenge misleading assumptions and exposes where security mechanisms fail in practice. This is particularly important as container escapes and privilege escalation exploits continue to emerge, demonstrating that security controls do not always work as expected. By grounding security claims in measurable results rather than theory, Houdini helps researchers and practitioners develop more robust container security strategies.

In order to achieve the aforementioned goals, we designed `Houdini` with the following in mind:

1. *Reproducible Results.* A key goal of `Houdini` is to ensure that tests yield consistent and repeatable results. By controlling system variables and maintaining a structured testing environment, Houdini allows researchers

and practitioners to compare results across different system configurations and security policies.

2. *Separation from the Host.* `Houdini` is designed to run tests inside a controlled containerized environment within a virtual machine (VM), ensuring that even if a test triggers a security vulnerability, it does not compromise the host system running the tests. This design choice enhances the safety of security evaluations, preventing unintended side effects on the underlying infrastructure.

3. *Test Case Expressiveness.* `Houdini` test cases (called "tricks") should be maximally expressive, such that some combination of steps can be used to achieve and test any desired result. It should be possible to define a new `Houdini` trick and modify existing tricks without modifying the `Houdini` binary. Moreover, it should be clear from reading a defined trick precisely what steps are involved, the consequences of each step passing or failing, and the overall nature of the exploit being tested.

4. *Focus on Observing Failures, Not Preventing Them.* `Houdini` is built to identify security weaknesses rather than defend against attacks.

5.2 Security of the Testing Environment

If Houdini were compromised during testing, it would not necessarily pose a threat to the host system due to the design of the testing environment. Houdini is intentionally designed to operate within a controlled, containerized environment inside a VM. This means that even if a test were to exploit a vulnerability within the containerized environment, the impact would be contained within the VM, isolated from the host system and other containers. The VM itself serves as an additional layer of security, providing a buffer that prevents the compromise from reaching the host infrastructure.

5.3 Design of Tricks

In containerized environments, security challenges often arise from vulnerabilities within specific components of the architecture. As such, understanding the relationships between the different components—such as container registries, images, runtime, namespaces, cgroups, and various security modules like eBPF, seccomp, and Linux capabilities—is crucial for identifying potential attack surfaces. This is where a comprehensive architecture diagram comes into play.

Figure 2 outlines the key components of a containerized system, highlighting the critical paths and interactions that security tests should focus on. Security testing should aim to cover these components extensively to ensure that vulnerabilities in any of these layers are adequately addressed. However, some components, like network interfaces, system calls, and

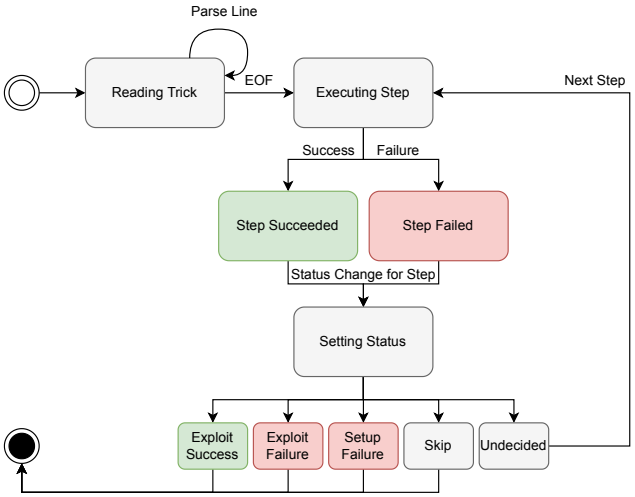

Figure 1: A state machine diagram of Houdini running a Trick.

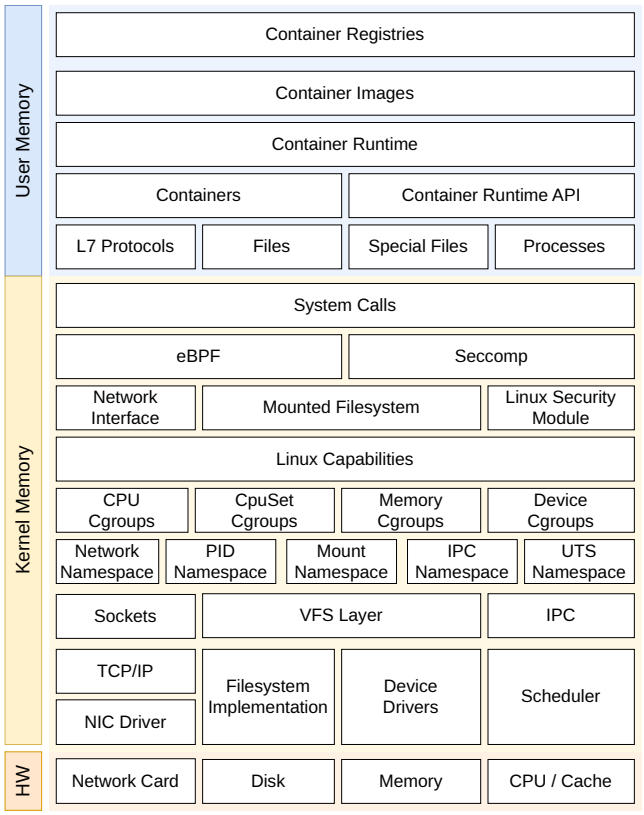

Figure 2: An architectural diagram of a container deployment environment depicting the attack surface created by its various components.

cgroups, are more frequently targeted in real-world attacks and thus warrant more attention in testing. The diagram will guide our understanding of which components to prioritize during testing, ensuring that our coverage is both thorough and effective.

6 Implementation of Houdini Tricks

Exploits that Houdini can test are divided into three separate files: a configuration file, a Python file, and a Dockerfile. Each Houdini trick begins with a configuration file (see Listing 1), which contains the Docker container's settings and environment configuration. The configuration file defines various Docker-specific parameters that are directly fed to the Docker API to start a container with the desired settings. Once the configuration is parsed, Houdini communicates these settings to the Docker API, using Docker's docker run command to initiate the container with these parameters. This ensures that the container is launched with the exact environment specified, allowing for consistent testing of vulnerabilities such as CVE-21616.

Listing 1: Configuration file for CVE-2024-21616.

```
name: CVE-21616
  name: CVE-21616
  dockerfile:
    - path: Dockerfile
  dependencies:
    - server: False
  docker_config:
    - network_mode: bridge
    - read_only: False
    - security_opt: ["no-new-privileges"]
    - pid_mode: null
    - cpu_shares: null
    - volumes : {"/proc": {"bind": "/host_proc", "mode": "ro"}}
    - mem_limit: null
    - cpuset_cpus: null
    - cpu_quota: null
    - cpu_period: null
    - cap_add: []
    - cap_drop: []
    - privileged: False
    - user: root
    - pids_limit: null
  trick:
    - path: /houdini/tricks/HostMount
```

The second component is a Python file that contains the actual exploit for the trick. This Python file is executed inside the container and is responsible for carrying out the trick scenario, such as testing privilege escalation or container breakout attempts.

Finally, the Dockerfile is used to configure and set up the container's environment. It specifies how the container should be built, including copying necessary files (like the Python file) into the container and installing required dependencies. The Dockerfile also configures the container with the settings

Listing 2

```
import os

# Define the relative path
relative_path = '../../../../../'

# Change the current working directory
try:
    os.chdir(relative_path)
    print(f"Successfully changed directory to: {os.getcwd()}")
except FileNotFoundError as e:
    print(f"Error: {e}")
except PermissionError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
```

defined in the configuration file, ensuring the container is set up correctly to run the exploit.

Listing 3

```
FROM ubuntu:20.04
RUN apt-get update -y
WORKDIR /proc/self/fd/9
CMD ["bash", "-c", "ls
↪  ../../../../../houdini/tricks/HostMount"]
```

This modular design—comprising the configuration file, Python script, and Dockerfile—provides flexibility to define and test various container escapes and vulnerabilities. It also allows users to easily customize or extend existing tricks by modifying any of the three components to suit their needs.

7 Evaluation

To assess the aforementioned security mechanisms, in this section, we evaluate the performance of a series of tricks, such as resource starvation, unauthorized access attempts, and container escapes.

Evaluating DOS Prevention Mechanisms.

With our first trick, we investigate the resilience of Docker containers against fork bomb attacks, a common form of denial-of-service (DoS) attack, where a process continuously replicates itself, and quickly exhaustes system resources. A fork bomb is designed to overwhelm the process table and exhaust available CPU and memory resources. If a container's resource management policies are inadequate, it would render the system unresponsive in the event of such an attack. Docker is able to leverage various Linux kernel mechanisms, most notably control groups (cgroups), to impose limits on process creation. Therefore, we define the success of this trick if the docker security mechanism that is used can prevent or disallow resource exhaustion.

Table 1: Effect of Different Security Configurations in Docker versions < 20.03

Configuration	Result
+NET_BIND_SERVICE | root	✓
-NET_BIND_SERVICE | root	✗
+NET_BIND_SERVICE | Custom Seccomp (deny bind syscall) + root	✗
-NET_BIND_SERVICE | root	✗
+NET_BIND_SERVICE | non_root	✗

Table 2: Effect of Different Security Configurations in Docker versions $\geq$ 20.03

Configuration	Result
+NET_BIND_SERVICE | root	✓
-NET_BIND_SERVICE | root	✓
+NET_BIND_SERVICE | Custom Seccomp (deny bind syscall) + root	✗
+NET_BIND_SERVICE | non_root	✗

In our evaluation of the trick, we applied a value of 10 to the pid_limit mechanism, which sets a maximum cap on the number of processes that can be created within a container. This configuration was successful in preventing a future fork bomb because the container would not be able to create enough processes to exhaust resources.

Assessing Docker Port Forwarding Restrictions

For this trick, we will run an HTTP server and bind it to port 23, which is a privileged port. Our objective is to identify which Docker confinement mechanism and versions permit binding port 23 to a process running an HTTP server.

It turns out that Docker made a change starting with version 20.03. They redefined unprivileged ports to start at 0 instead of 1024, which means that the 'NET_BIND_SERVICE' capability is no longer required to bind to a privlidged port. Thus, we will test one version of docker that is less than 20.03, and one greater than or equal to 20.03.

Starting with the docker version less than 20.03, the results of the tests reveal that binding to a privileged port (port 23) is influenced by several Docker confinement mechanisms. First of all, success was achieved when the 'NET_BIND_SERVICE' capability was enabled and the user was root. If the NET_BIND_SERVICE was activated, but the user was not root, the trick would fail. Other ways the trick could fail is if a custom SECCOMP profile was used to deny the bind system call. This indicates that the ability to bind to privileged ports is primarily dependent on the presence of the 'NET_BIND_SERVICE' capability and root user privilege. The table below describes these occurences.

In Docker versions prior to 20.03, NET_BIND_SERVICE had an effect as explained earlier. However, in Docker versions 20.03 and later, NET_BIND_SERVICE no longer has any impact. However, root user is still needed to bind the HTTP server to a port, and not nessearily a privlidged port. If the user is a non-root, then the trick will not work. Also, if a custom seccomp profile is used to deny the bind syscall, the trick will not work. All of this is described in the table below.

CVE 2024-21616: the Docker Container Escape Exploit

The CVE-2024-21626 vulnerability exists within Docker and runc and allows malicious containers to escape their isolation layer, enabling attackers to take control of the host machine. This poses a significant security risk for enterprises relying on containerization, as once the vulnerability is exploited, attackers could breach security boundaries to access sensitive data and system resources.

Specifically, the cause of CVE-2024-21626 involves improper setting of the container's working directory. Under certain conditions, if a container's working directory is set to a special file descriptor path, such as /proc/self/fd/<fd> (where <fd> typically points to the /sys/fs/cgroup directory), it may allow for container escape. The affected versions of runc range from v1.0.0-rc93 to 1.1.11.

To evaluate the practical impact of CVE-2024-21616, we leveraged Houdini to systematically assess the effectiveness of Docker's confinement mechanisms in preventing container escapes. By integrating CVE-2024-21616 as a Houdini trick, we were able to analyze under what conditions the vulnerability could be exploited and determine the effectiveness of different security configurations in mitigating the risk.

The experiment was structured to evaluate the default security posture of Docker as well as the impact of applying additional confinement measures. The Houdini trick for CVE-2024-21616 was designed using a combination of a configuration file, which defined the container's security settings, a Dockerfile, which set up the containerized environment, and an exploit script, which attempted to break out of the container and execute arbitrary commands on the host. These files can be seen in section 6.

Our findings revealed that Docker's default security settings were insufficient to prevent exploitation. The only way to prevent it is to use a runc version greater than 1.1.11. This is because the vulnerability lies inside of runc, the container runtime responsible for managing process execution inside containers. Since the flaw is in the runtime itself, Docker's built-in security mechanisms, such as namespaces, cgroups, and seccomp, do not inherently prevent exploitation. As a result, even well-configured containers remain vulnerable if they are running on an affected version of runc.

Killing Host Processes using Process Namespace

By default, Docker containers operate in their own isolated process namespace, meaning each container has its own set of processes that it can see and manage. This isolation is one of the fundamental aspects of Docker's security model, preventing containers from interfering with the host system or other containers. Each container runs as if it's a completely separate environment, with its own process tree and PID (process identifier) space.

Table 3: Effect of Different Security Configurations in Docker versions $\geq$ 20.03

Configuration	Result
pid_mode=host | root	✓
pid_mode=host | non-root	✗
pid_mode=null	✗

When you configure a container to run with pid_mode = host, you effectively remove the process isolation between the container and the host system. In this configuration, the container shares the same process namespace as the host, which means the container can see all processes running on the host, just as a process running natively on the host would. This feature, while useful in some scenarios, also exposes the container to the host's processes, providing the ability to monitor or interact with them.

By leveraging the pid_mode = host setting, a container can gain access to the host's process identifiers (PIDs), enabling advanced use cases such as debugging or monitoring processes on the host from within the container. This is a powerful capability, but it also introduces potential risks. With root privileges, we found that a container can use typical Linux commands, like kill or killall, to target and terminate host processes. For instance, if a process x is running on the host with a specific PID of 343, the container can send a termination signal to process x, effectively killing it from within the container. If the pid_mode does not map to the host, then we are not able to kill a host process from the container.

However, this trick does not work with non-root privileges, as the kill command requires elevated permissions to target processes owned by other users or system processes. Without root access, the container can still see the host's processes but will lack the necessary permissions to terminate them.

8 Conclusion

In this paper, we presented Houdini, a security benchmarking framework designed to evaluate the effectiveness of container confinement mechanisms. As container-based workloads continue to dominate cloud infrastructures, ensuring that containers are properly isolated is paramount for maintaining security. Houdini offers a systematic and empirical approach to test and assess various container security configurations, highlighting potential vulnerabilities and misconfigurations that may compromise container isolation. Our evaluation demonstrated the ability of Houdini to detect critical misconfigurations, including those leading to privilege escalation and container escapes. By providing a reliable and reproducible method for testing container confinement, Houdini helps bridge the gap between theoretical security assumptions and practical, real-world behavior.

As container technologies evolve, it is essential to continuously assess and improve their security mechanisms. Houdini's modular design ensures that it remains adaptable to new security challenges and configurations, allowing for ongoing validation and refinement of container security practices. By offering an open framework for container confinement benchmarking, Houdini aims to support the development of more secure container technologies, ultimately contributing to a safer cloud infrastructure for the future.

References

[1] Cis docker benchmark. In *CIS Docker Benchmark*, 2024.

[2] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.