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 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 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.

We define container confinement 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. To confine a container means to enforce strict isolation between the container and the host system as well as between containers themselves, ensuring that the container operates within predefined boundaries without interfering with or compromising other containers or the host. The goal of container confinement is to limit the container's access to system resources, prevent unauthorized actions, and ensure that it only interacts with the host environment in an optimal manner. Addressing this challenge is essential as the adoption of container technology continues to grow across industries.

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. This process requires some semantic knowledge about the system and is referred to as the semantic gap issue. This gap often arises because the mechanisms designed to secure a system (e.g., namespaces, cgroups, SELinux policies) may not always function as expected due to misconfigurations, lack of understanding, or emerging vulnerabilities. To bridge this semantic gap, a formal approach 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 3, 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 2 describes related work, Section **??** discusses the contributions, limitations, and our plans for future work. Section 9 concludes.

2 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.

3 Container Confinement Problem

A container is a lightweight, portable unit of software that encapsulates an application and all its dependencies, such as libraries, configurations, and binaries, which ensures consistent behavior across various environments. Unlike virtual machines, containers do not require a full operating system to run; instead, they share the host operating system's kernel, which makes them highly efficient in terms of resource usage and startup time. Containers achieve isolation using Linux mechanisms such as namespaces, which provide separate views of system resources (e.g., process IDs, filesystems, and network interfaces) to each container, and control groups (cgroups), which limit and prioritize resource usage (like CPU, memory, and I/O). This design allows containers to run independently in isolated environments while sharing the same underlying operating system. This approach makes containers much more lightweight and faster than hypervisor-based virtual machines, with significantly lower overhead. As a result, containers have become an essential tool for modern cloud-native applications, microservices architectures, and continuous integration/continuous deployment (CI/CD) pipelines.

Confining containers is essential for maintaining both system security and stability by limiting the potential damage they can cause if compromised. Several mechanisms are employed to isolate containers and restrict their interaction with the host system. Linux namespaces are the foundation of container isolation. It creates separate environments for processes, network interfaces, and filesystems that ensures containers cannot interfere with each other or the host system. Control groups (cgroups) complement namespaces by limiting resource usage, such as CPU, memory, and disk I/O, by preventing a single container from starving the system

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

or other containers of essential resources. Mandatory access control (MAC) systems like SELinux and AppArmor further restrict containers by defining and enforcing strict security policies on what resources they can access, such as files, devices, and network resources. Seccomp (Secure Computing Mode) adds another layer of security by filtering and controlling the system calls that containers can make, thus preventing unauthorized or dangerous operations that could lead to exploits. Linux capabilities are another mechanism, which is used to grant containers only the specific privileges they need, which reduces the attack surface by limiting elevated privileges. Container runtimes like Docker and containerd provide additional hardening by supporting image signing, vulnerability scanning, and enforcing container runtime security policies. Despite these defenses, unintentional vulnerabilities can still emerge, especially when misconfigurations occur.

Container confinement is complicated by the numerous and sometimes overlapping Linux mechanisms used to protect Docker containers. These mechanisms, such as namespaces, control groups (cgroups), SELinux, AppArmor, seccomp, and capabilities, each contribute to the isolation and security of containers. However, while these mechanisms are designed to work together to restrict access and resource consumption, the interaction between them is complex and not always fully understood, especially when misconfigurations or edge cases arise. For instance, while namespaces ensure process and filesystem isolation, cgroups manage resource limits, and security policies like SELinux enforce mandatory access controls. Yet, the way these layers of security interact is not always transparent, and subtle vulnerabilities may exist in the integration of these components. Moreover, as Docker and other container runtimes continue to evolve, new features and mechanisms are added, further complicating the security landscape. Even seasoned security experts sometimes struggle to keep track of how these protections interrelate or to predict how they will behave in certain scenarios. This lack of complete understanding introduces the possibility of security gaps, as attackers can exploit unintended interactions between these mechanisms. Consequently, the very complexity that makes containers so powerful and efficient also makes securing them a challenging and ongoing problem.

4 Linux Confinement Mechanisms

5 Design

5.1 Design Goals

Houdini is a security testing framework specifically designed to evaluate container isolation within Docker environments. Its primary goal is to systematically measure and evaluate a system's performance or quality against a set of predefined standards or metrics. In the context of security benchmarking for container confinement, benchmarking involves testing the effectiveness of a container's isolation mechanisms—such as namespaces, cgroups, seccomp, and other security features—by simulating real-world attack scenarios and assessing whether these mechanisms successfully prevent unauthorized access, privilege escalation, or container escapes. Essentially, it is a process of objectively comparing how well container security measures perform, identifying strengths and weaknesses, and providing a reference point for improving overall security posture. Unlike general-purpose security tools, Houdini focuses exclusively on container isolation, ensuring that tests produce reliable and reproducible results across different Docker setups. By simulating real-world attack scenarios, Houdini also verifies whether Docker's security mechanisms successfully block actions that could lead to container escapes or privilege escalation. The framework is modular and extensible, allowing users to define new tests (tricks) to cover emerging risks, and it is built to adapt to different Docker configurations and container runtimes. Additionally, Houdini supports automation, enabling seamless integration into continuous integration (CI) pipelines for ongoing security assessments. The tool also prioritizes reproducibility by isolating tests in controlled environments, ensuring consistent results across configurations. Furthermore, it provides detailed reports that highlight the success or failure of security measures, offering actionable insights to improve container security practices.

Houdini is not only a tool for testing container isolation but also a means to optimize container privilege 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 or escalate privileges in the event of a breach. Additionally, this fine-tuning helps prevent unintended interactions between the container and the host system, leading to a more robust and secure container environment.

In order to achieve the aforementioned result, we designed `Houdini` with the following goals 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 sys-

tem 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 cgroups, are more frequently targeted in real-world attacks

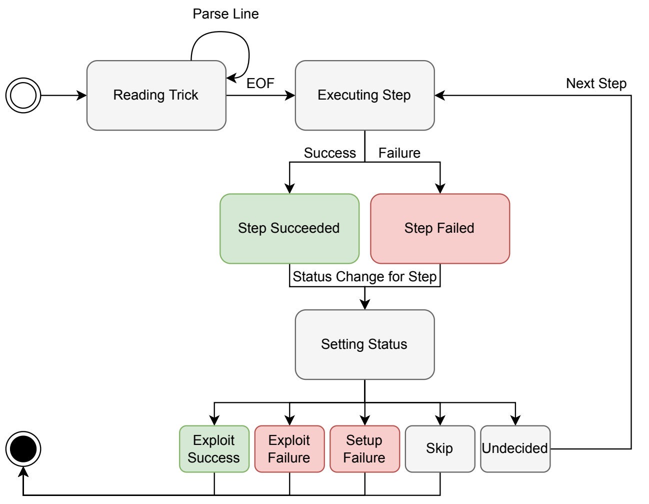

Figure 1: A state machine diagram of Houdini running a Trick.

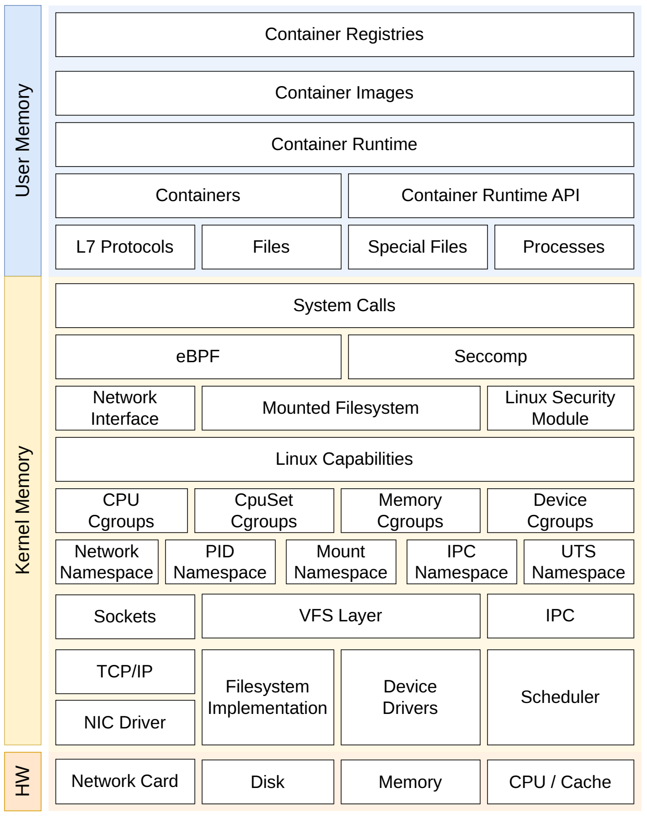

Figure 2: An architectural diagram of a container deployment environment depicting the attack surface created by its various components.

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

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}")
```

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.

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

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	✗

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.

8 Summary of Usecases

Bridging the Gap Between Theoretical and Practical Container Security.

Houdini provides an objective and practical way to measure container security by testing the actual security properties being enforced, rather than relying on assumptions or theoretical guarantees about what should be enforced.

Many security mechanisms, such as namespaces, cgroups, seccomp, and capabilities are designed to isolate and restrict containers. However, their effectiveness depends on proper implementation and configuration. In many cases, security policies may appear to be correctly applied but fail under real-world attack scenarios.

Houdini ensures that security claims are backed by empirical testing rather than assumptions. It allows researchers and practitioners to:

1. Verify whether security mechanisms are truly working as expected.

2. Identify gaps between intended confinement policies and actual enforcement.

3. Challenge misleading security assumptions, ensuring that claims about container security are based on real, measurable behavior rather than theoretical models.

9 Conclusion

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.