Houdini: Security Benchmarking of Container Confinement

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 privileged and unprivileged 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 are now a standard part of our cloud infrastructure. Unlike hardware virtual machines, containers are extremely lightweight, incurring essentially no overhead versus just running multiple applications on the same host. Containers, however, allow workloads to be precisely replicated by including all userspace dependencies of an application in one unified image. While containers help solve many development and deployment problems, they also create a new problem of container confinement.

Linux cgroups, namespaces and separate mountpoints create the illusion of separation under normal conditions. Yet there are numerous ways for this separation to fail, and when this happens, attackers can disrupt the operation of other containers or even take control of the host. In principle, containers can be secured using a variety of technologies including SELinux, AppArmor, and seccomp. These technologies are complex, however, and the confinement problem is subtle. Best practices for securing containers [CITES] can help; a guide, however, provides no assurance that the resultant configuration prevents container escapes. Container confinement is considered so problematic that the cloud industry has now developed multiple technologies for running containers within specially secured virtual machines [CITE gVisor, kata containers]. But would such solutions be necessary if we could check to see whether we were properly confining containers?

Test suites are regularly used to verify that systems meet various requirements. Performance benchmarks are used to compare the performance of hardware and software. [LIST COMMON BENCHMARKS LIKE SPEC?] In software development, tests are used to verify that important functionality has not been broken by code changes. In many organizations, it is not possible to check in code without it first passing a battery of tests. Test suites are also used to verify the functionality of production systems, for example with uptime monitors that verify that services are performing their designated functions. While used less frequently, functional test suites are also used to check for security issues in deployed systems. Network scanners, in particular, are often used to proactively find accidentally enabled services, firewall issues, and insecure software versions.

Here we present the Houdini, the first test suite for verifying container confinement. Given a docker container description (including both kernel version and specification of userspace filesystem), Houdini will instantiate the container in a standalone QEMU-based virtual machine and perform multiple tests (tricks, in our parlance) to see whether the configuration is vulnerable to known container escape methods. Houdini is written in Rust and is easily extensible, providing an extension language for writing tricks. While it can be used to check general host security, Houdini is specialized to the specific requirements of container confinement.

[I WANT TO LIST CONTRIBUTIONS BUT I'M NOT

SURE WHAT TO WRITE]

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 3. Section 4 explains the tricks Houdini currently implements and their associated vulnerabilities. In Section 5 we present case studies showing how Houdini can detect basic misconfigurations. Section 6 describes related work, Section 7 discusses the contributions, limitations, and our plans for future work. Section 8 concludes.

2 Container Confinement Problem

- explain what a container actually is
- what you get with namespaces & cgroups
- define containers and the game of escaping containers

3 Design and Implementation

3.1 Design Goals

With the creation of `Houdini`, we sought to rectify a critical perceived gap in existing security testing frameworks. In particular, we noticed that no existing testing framework satisfied our specific use case: testing the isolation guarantees of a research artifact in a highly reproducible manner, independent of the underlying container runtime and the implementation details of the enforcement mechanism. (Refer to Section 6 for a detailed comparison with existing work on container security evaluation.)

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

1. *Reproducible Results.* Results of a test should be reproducible over time and across various system configurations. Reproducibility is a critical property of the scientific method and quintessential to the integrity of academic research evaluations. Our hope is that reproducible security evaluations will promote the use of `Houdini` to enable fair and unbiased comparisons of container security artifacts based on their security properties.

2. *Controlled Environment.* `Houdini` should be capable of evaluating a security artifact independently of the underlying platform (e.g. the container runtime, host operating system, and whatever security measures are in place, if any). Platform independence helps to ensure that the focus of a security evaluation is on the effectiveness of whatever protections are being tested as opposed to differences in the underlying platform (e.g. bugs, configuration differences, or version specific behaviors).

3. *Container Specificity.* While a number of security testing frameworks exist, very few target the container specific use case we want to support with `Houdini`: container security. This motivated us to consider an implementation which focuses specifically on the container security use case.

4. *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.

We wrote `Houdini` in just over 3300 lines of Rust code[1]. Rust offers several desirable properties for a security benchmarking application like `Houdini`. First, Rust has a rich ecosystem of libraries (called "crates") that enable us to easily work with external APIs such as the Docker daemon. Moreover, Rust has excellent support for serializing and deserializing YAML and JSON to and from Rust enums and data structures using powerful macros exposed by the `serde` crate. Finally, Rust's `tokio` crate provides a powerful asynchronous framework that enables us to easily parallelize and synchronize `Houdini` tricks and their steps.

[TODO: talk about security and how we shouldn't need to worry about the tricks being used maliciously (we're testing armour and brining our own weapons and test range, we're not really worried terrorists coming in and blowing us up)]

3.2 Houdini Tricks

Exploits that `Houdini` can test are broken up into individual files called *tricks* (Design Goal 4). Each `Houdini` trick is written in YAML and consists of a metadata section followed by one or more *steps*. `Houdini` comes with a stock set of tricks (c.f. Section 4), tested in continuous integration to ensure they always work by default under vulnerable configurations.

While these default tricks can provide decent exploit coverage for initial tests, it is also possible for the user to write their own tricks by defining their own YAML files following the expected format. This modular design enables `Houdini`

[1] Source lines of code, i.e. not counting whitespace, comments, and dependencies.

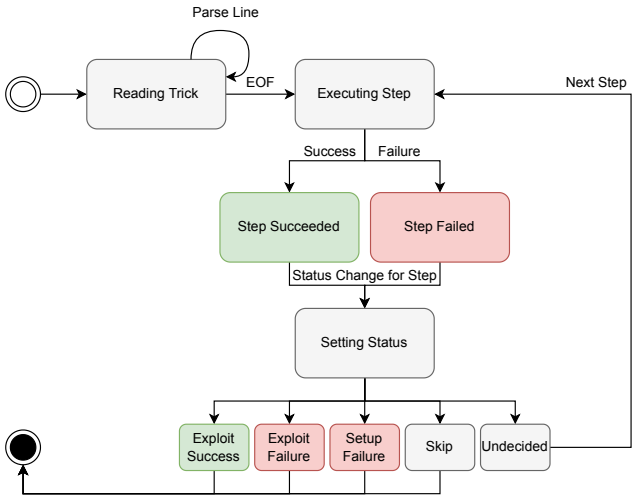

Figure 1: A state machine diagram of Houdini running a Trick. Note that aside from the "Undecided" status, all other statuses are considered final. That is, a step reporting such a status terminates execution of the Trick.

to test essentially any container escape while providing the ability for the user to modify existing tricks should the need arise (e.g. to test a specific aspect of a defense mechanism in a more targeted fashion).

Listing 1 provides a simple example of a `Houdini` trick that tests whether a mounted Docker socket in a container can be used to communicate with the Docker daemon running on the host. Steps in a `Houdini` trick consist of an ordered list of parameterized, well-defined actions. In our current implementation, there are six distinct types of step:

- `VersionCheck` enforces a version check on the host operating system version, docker daemon version, and container runtime version (e.g. `runc`). This helps to verify that a trick is indeed running under an expected configuration where it would ordinarily succeed without any additional protections in place.
- `SpawnContainer`
- `KillContainer`
- `Host`
- `Container`
- `Wait`

3.3 Reproducible Tricks

[TODO: WILLIAM: Houdini design and implementation details here.]

[TODO: WILLIAM: Introduce the architecture diagram below somehow (what is the best way to do this?)]

Listing 1: A YAML definition of a Houdini trick with three steps. The first step spawns a container from the "bash" Docker image, downloading it if necessary. As part of the exploit setup, we mount `/var/run/docker.sock` from the host into the container. The second step installs curl into the bash container. Finally, the third step attempts to abuse the mounted Docker socket to make API requests to Docker. Note the status codes on success and failure for each step in the Trick.

```yaml
name: mounted-docker-socket
steps:
  - spawnContainer:
      name: bash
      image: bash
      cmd: sleep infinity
      volumes:
      - "/var/run/docker.sock:/docker.sock"
      failure: setupFailure
  - container:
      name: bash
      script:
      - command: apk
        args:
        - add
        - curl
      failure: setupFailure
  - container:
      name: bash
      script:
      - command: curl
        args:
        - "--unix-socket"
        - "/docker.sock"
        - "http://localhost/_ping"
      failure: exploitFailure
      success: exploitSuccess
```

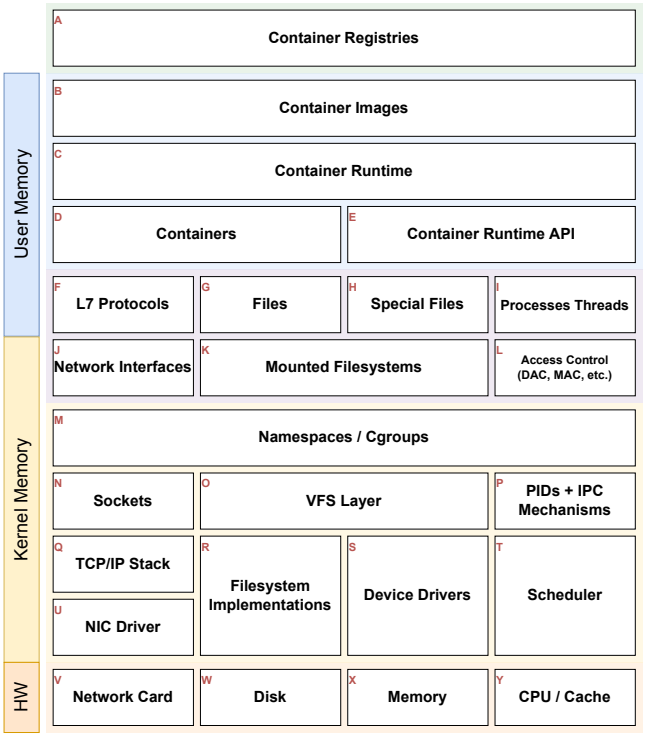

Figure 2: An architectural diagram of a container deployment environment depicting the attack surface created by its various components. Components are organized hierarchically with lower level subsystems toward the bottom, including data structures and abstractions that reside in kernel memory and the hardware itself.

- security problems arise from vuln in specific components
- if we're going to be testing, we need to have an idea of what components we're testing
- ideal coverage will touch everything
- some components are targeted more frequently than others in attacks commonly used in practice

4 Testing Confinement

[TODO: KEVIN: Explain some technical details for each exploit. Refer back to Table 1 as necessary.]

CVE-2019-5736. This CVE[**?**] is a remote code execution vulnerability in the `runc` binary. The PoC exploits the way `runc` creates a process in the container called `runcInit` to run a specified command. The `procfs` file system contains a symlink to the binary being executed in `/proc/self/exe` which points to the `runc` executable on the host. A malicious container overwrites its own `/bin/sh` binary to the `/proc/self/exe` interpreter `#!/proc/self/exe`. Once a user executes the overwritten `/bin/sh` binary, the interpreter calls the `runc` binary whose pid is then captured. Malicious code will then obtain the file descriptor from `/proc/runc-pid/exe` and overwrite the contents of the `runc` binary on the host.

CVE-2021-42013. This is a remote code execution vulnerability in the Apache HTTP server container (versions 2.4.49 and 2.4.50). The exploit is a path traversal vulnerability that takes advantage of the server being unable to detect the path traversal characters "../" in a URL. This may occur when the second dot is replaced by its unicode representation "%2e". By requesting a URL with several path traversal sequences and a binary to execute, contents of files can be retrieved from the container's local filesystem.

CVE-2022-0492. A bug in the kernel's cgroup_release_agent_write cgroupv1 filesystem code can be exploited to enable a container escape. A file called release_agent gets executed when a process in the cgroup terminates if notify_on_release is enabled. The exploit involves mounting the cgroups filesystem on the host to the container and creating a directory which creates a new cgroup. Creating the notify_on_release file in the new directory and writing a 1 to it tells the host to run the release_agent file when a process in the custom cgroup terminates. Modifying the release_agent file with a script that executes a malicious binary on the container from the host's filesystem allows the malicious container to successfully escape upon termination of a process in the custom cgroup.

CVE-2022-0847. This is an escalation of privilege Linux kernel vulnerability caused by a bug in the kernel's pipe.c code introduced in version 5.8. The exploit utilizes the pipe buffers flag PIPE_BUF_FLAG_CAN_MERGE which allows for data to be overwritten in a page cache from a spliced file if the pipe buffer is prepared in a specific way. The flag gets set by filling the buffer with random data. The pipe buffer can then be emptied and then a file can be spliced into the pipe. Writing data into the pipe buffer now overwrites the data in the file. In order to escape a container with this exploit, the setup is similar to that of CVE-2019-5736 above where a malicious container waits for runc to execute in the container. The file descriptor is grabbed from the procfs and overwritten using the dirty pipe vulnerability.

We selected the exploits above based on several criteria. (1) The exploits must have a proof of concept readily available online; (2) the exploits must cover various components of the container deployment environment (see Section 3.3); and (3) the exploits must be high impact and severity defined by the CVS standards. The exploits chosen all fit within these criteria and can be tested repeatedly with different system configurations using `Houdini`.

The exploits are run in privileged and unprivileged mode with one of seccomp or apparmor securing the container. Most exploits are blocked successfully by using these security systems, but these security systems might not be configured or

Table 1: **[TODO: table caption]**

No.	Exploit		CVE No.	Attack Vectors (See Section 3.3)
1	Mounted Docker Socket	CITE	—	C, D, E, N
2	Mounted `/etc/passwd`	CITE	—	C, D, G, K
3	Pipes Read-Only Overwrite	CITE	CVE-2022-0847	O, S
4	Cgroup Release Agent Code Execution	CITE	CVE-2022-0492	M, O, R
5	Apache Path Traversal	CITE	CVE-2021-42013	D, I
6	`runc` Binary Overwrite	CITE	CVE-2019-5736	A, B, C, D, H, K

enabled by default. For example, kernels can come with apparmor on the system but it is disabled by default on boot and the service must be re-enabled by manipulating GRUB boot parameters. Using Docker info on a system lets a user know what container security mechanisms are available to the user. If apparmor is not installed on the system or the service is not enabled on boot through GRUB, then seccomp is usually the only security mechanism that can be deployed.

The default policy files for both apparmor and seccomp are used during testing. The default policy for apparmor most importantly restricts write permissions in the sensitive `/proc` and `/sys` file sysytems, as well as denying mounting operations. The default seccomp profile allows for all but 44 of the system calls to pass through and can allow more based on the container's capabilities. The system calls that are filtered out and restricted are focused around sensitive operations that can interact directly with the host kernel and potentially modify its behaviour (kernel modules, system time, reboot, etc.).

Tests were run on Ubuntu 22.04 with kernel version 5.15.0-50-generic. The docker engine is version 18.09.1 and runc version is 1.1.0+dev. For virtualization we used QEMU version 6.2.0 and buildroot version 2022.02.

We manually created a trick yaml file for each exploit outlining the environment and the commands for the host and container to perform. If required for the exploit, a new environment is created using the buildroot to generate a new virtual machine with a configurable linux kernel and filesystem. The versions for the kernel and specific packages that are required to run the exploit successfully, such as docker and runc, that are specified in the exploit yaml are passed to buildroot as parameters. Houdini is also included as a package and launches when the virtual system boots. QEMU runs the virtual environment and creates a virtual socket (vsock) connection between the host and the VM for status reporting and management. Once `Houdini` on the VM boots, a connection is made to `Houdini` running on the host. The trick is then sent via the vsock connection and parsed by the `Houdini` client on the VM and for the remainder of the trick, the VM is acting as the host. Once the trick finishes execution, the results are sent back to the host `Houdini` using the vsock connection and the VM is shut down.

The exploit CVE-2019-5736 is set up with a malicious container that runs a script on start to wait for the /bin/sh binary to execute and then in turn execute its own binary to overwrite the host runc. The vulnerable runc version being tested is version 1.0.0-rc6. On the host side, the user spins up the container and attempts to exec into the container using "docker exec -it ID sh". The exploit is considered a success if the runc binary on the host system has been compromised.

CVE-2021-42013 has a different threat model where the container image is vulnerable but not necessarily malicious. The exploit begins with a vulnerable apache httpd image being launched and then on the host side, the exploit binary is called using the container's IP address and a command as input. The exploit is considered a success if the command runs and data is received containing the response from the container.

Unlike the previous two exploits, CVE-2022-0492 does not utilize any malicious binaries. The exploit takes place with commands executed entirely within the container. A privileged container is launched and the cgroup rdma folder from the host cgroup filesystem is mounted in the container allowing for manipulation of the cgroup's release agent file. The exploit is considered successful if all commands execute successfully with no errors.

Why use Houdini.

Container hardening techniques through confinement mechanisms require both knowledge of the application code and security best practices. Ensuring that their container environment is not vulnerable to known common exploits is a step that many developers will either save for the last part of their development cycle or forget about all together. Testing frameworks allow developers to save time by automatically running a suite of some tests in order to give the developer more insight into how their application behaves. Existing exploit testing frameworks are not designed with the goal of exposing problematic areas in a container's confinement policy or environment. `Houdini` allows the user to run a series of known vulnerabilities against their container environment to expose issues for the developer to quickly asses and act upon. The security mechanisms that are used to lockdown the container can easily be tested repeatedly throughout the development lifecycle without much overhead for the developer. `Houdini` also allows for the testing of new confinement mechanisms by academics and researchers with very little required customization to the trick files themselves. `Houdini`

Table 2: **[TODO: table caption, replace success/failure text with glyphs. re-order, grouping unprivileged rows and privileged rows.]**

Security	**CVE-2019-5736**	**CVE-2021-42013**	**CVE-2022-0492**	**CVE-2022-0847** (See Section 3.3)
unprivileged	success	success	failure	success
privileged	success	success	success	success
unprivileged + apparmor	failure	success	failure	failure
privileged + apparmor	failure	success	failure	failure
unprivileged + seccomp	failure	success	failure	success
privileged + seccomp	success	success	success	success

can test any part of the system for vulnerabilities including the kernel, runtime, and the container configuration which all contribute towards making the container vulnerable. This makes `Houdini` a powerful tool when assisting in the development of container confinement policies and environment testing.

What makes running the exploit possible in apparmor.

Apparmor and seccomp are integrated with docker to allow a user to further confine their resources and capabilities. When apparmor is enabled on a host kernel, docker will apply a default apparmor policy to a container. The policy name is docker default and it restricts: mounting, most cases of writing to /proc, and most cases of writing to /sys. This rather limited profile does a relatively good job of blocking common container escapes and privilege escalations. Issues mainly arise when a user tries to modify the existing policy, or develop their own without proper consideration. An apparmor profile is developed by specifying resources to allow or deny access to on each line. The profile must then be ran through the apparmor parser which checks for the proper syntax to be followed. Once the parser successfully reads and loads the profile, docker can use it as a value passed to their "–security-opts apparmor=PROFILE_NAME" flag.

The apparmor parser will not check for whether a specified system path exists or not and if contradictory statements exist, it uses the statement that is more restrictive (i.e. if "mount" and "deny mount" exist in the same profile, it will always use "deny mount" no matter the declared order).

In order for CVE-2022-0492 to successfully operate with the apparmor security enabled on a privileged container, the only line that needs to be modified in the default profile is "deny mount". If a user were to change the "deny mount" to "mount" for either their own usage or as a simple mistake, then the exploit would be able to run successfully.

CVE-2019-5736 requires a little more modification to the docker default profile. Write access to `/proc/PID` must be allowed and full use of ptrace must be allowed. The exploit can successfully overwrite its own shell binary without any modifications to the default profile. If only write access to `/proc/PID` is given, the exploit can grab a hold of the runc process ID, but access to ptrace is required as it is used for process control by the system when modifying the `/proc/PID/exe` file. These two changes to the default apparmor profile allow CVE-2022-0492 to successfully overwrite the runc binary even without the privileged flag.

Using older versions of the container runtime RunC before version 1.0.0-rc8 allows for a different bypass of the apparmor confinement policy. Since apparmor is based off of path names, a malicious user can get around the default apparmor's policy that is locking down a specific path name by simply remounting it to a different path. An example of this is the procfs filesystem mounted on `/proc`. The Apparmor default profile locks down the procfs filesystem at the defined `/proc` location. By remounting the procfs filesystem to a different location, this bypasses the pathname based confinement that apparmor offers.

What do the results tell us.

[TODO: docker default security profiles can be overly permissive. Why?] [TODO: seccomp by default allows almost 90% of systemcalls] [TODO: apparmor allows all network capability] [TODO: default profiles are needed to be generic and more likely overly permissive] [TODO: generally do a good job catching existing exploits by locking down key parts] [TODO: Compromise between security and usability] [TODO: developing profiles can be difficult even with tools such as aa-easyprof] [TODO: testing confinement is made easy]

5 Evaluation

Houdini on:

- default docker container unpriv
- default docker container priv
- same as above, but with AppArmor
- find "how to secure a container" policies?
- make sure to pick apart standard docker policies, try to see which pieces contribute what protection

6 Related Work

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 approaches and contrast them to Houdini.

Repeatable Exploit Frameworks. TestREX [**?**] - repeatable exploits in different software versions.

Vulnerability Scanners and measurement studies. Many free, open and closed source tools exist for identifying the presence of known vulnerabilities in container images. Shu *et al.* [**?**] 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. This motivates the need for container confinement. Lin *et al.* [**?**] collected a dataset of 223 container exploits to create a taxonomy of attacks and propose a lightweight mitigation strategy.

Many commercial tools aim to automate the search for known vulnerabilities (often those reported **[TODO: through the CVE process]**). IBM Vulnerability advisor[2] scans images when they are uploaded to IBM's container registry. Red Hat's Clair[3] statically inspects images layer by layer. Anchore's container registry scanner[4] can monitor a range of container registries, and block deployment if the image does not meet a security policy. Javed and Toor [**?**] evaluate the accuracy of several commercial scanners and find that many vulnerabilities can go undetected due to the static analysis nature of the tools.

Houdini analyzes and reports on container confinement, so application-specific vulnerabilities that do not result in a container escape are not in scope. However, we expect Houdini trick developers will largely base their contributions on the CVE framework like the tools described above.

Offensive Tools. The offensive security community has developed tools to attack containers. CDK[5] and DEEPCE[6] are two popular open source container penetration testing toolkits that attempt a collection of known exploits to gain persistence, escape the container, or gather information about the environment. Houdini takes a more systematic approach that allows direct comparison of defensive techniques by clearly defining the environment, exploit steps, and output, whereas CDK's goal is to use any possible technique to bypass the environment's defenses.

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 guides [**?**, **?**, **?**] available online. In our experience, guides tend to offer generic advice (e.g., "*Only allow read access to the root filesystem*" [**?**] or "*Use Linux Security Module (seccomp, AppArmor, or SELinux)*" [**?**]), making the advice difficult to follow. Guides, by their nature also tend to lack sufficient context to assist administrators in deploying the advice within their specific environments.

[TODO: Houdini could be used to systematically compare best practices from several sources.... We leave this for future work.]

[TODO: a new section: security experimentation]

[TODO: look at CSET conference]

7 Discussion

Discussion points

- talk about how similar all confinement solutions are: methods for restricting what happens in response to an upcall (system call). (Address space confinement is a standard part of the process model.)
- can vary with whether a hypervisor, kernel, or both are called. Note here that many concurrent security tools can be used (mandatory access control, system call filtering, capabilities, etc.)
- can switch back to userspace as part of servicing upcall (user mode drivers)
- whether to allow the upcall or not is made by the hypervisor and/or kernel.
- Heart of problem: virtualization of resources does not guarantee separation of confinement. Policy is defined, it is just whether it is implicit or explicit. If we have explicit rules, we can test to see whether the rules are being respected; if they are implicit, we may not even be sure what the rules are.
- Houdini helps keep us honest, lets us talk about the security properties that are actually being preserved rather than the ones that *should* be preserved.
- rather than use hypervisors to implicitly define rules that are then bypassed with guest additions/hypervisor modifications, we should explicitly define rules using the least amount of code as possible (to make it efficient and more easily verifiable)

[2] https://www.ibm.com/docs/en/cloud-private/3.2.0?topic=guide-vulnerability-advisor

[3] https://www.redhat.com/en/topics/containers/what-is-clair

[4] https://anchore.com/container-registry-scanning/

[5] https://github.com/cdk-team/CDK

[6] https://github.com/stealthcopter/deepce

- why don't things like `Houdini` exist for other problems in security — why aren't we doing this more in security?
- experimental apparatus vs. big data sets
- what about fuzzers? well how do we know the fuzzers are fuzzing properly (correctness)? where is the reproducibility?
- `Houdini` opens the door to implementing a policy fuzzer (permute issues / bugs in the security policy, then test it with houdini) N.B. this might be interesting for Kevin's thesis

8 Conclusion