

Docker Security

BY KNOX ANDERSON, PRODUCT MARKER, SYSDIG

CONTENTS

- > OVERVIEW
- > CI/CD AND PRE-DEPLOYMENT SECURITY
- > RUNTIME CONTAINER SECURITY
- > INCIDENT RESPONSE
- > CONCLUSION

OVERVIEW

Are Containers insecure? Not at all. Features like process isolation with user namespaces, resource encapsulation with cgroups, immutable images, and shipping the minimal software and dependencies reduce the attack vector providing a great deal of protection.

Container security tools are becoming hot topics in the modern IT world as the early adoption fever is transforming into a mature ecosystem. Security is an unavoidable subject to address when we plan to change how we architect our infrastructure.

This Refcard will lay out the basics of the container security challenge, give you hands-on experience with basic security options, and also spell out some more advanced workflows.

We'll split container security into three sections covering what to do at each step of your container security lifecycle.

- CI/CD and pre-deployment security
- Run-time security
- Incident response and forensics

CI/CD AND PRE-DEPLOYMENT SECURITY

CONTAINER IMAGE AUTHENTICITY

There are plenty of Docker images and repositories on the Internet for every type of application under the sun, but if you are pulling images without using any trust and authenticity mechanism, you are basically running arbitrary software on your systems.

- Where did the image come from?
- Do you trust the image creator? Which security policies are they using?
- Do you have objective cryptographic proof that the author is actually that person?
- How do you know nobody has been tampering with the image after you pulled it?

Docker will let you pull and run anything you throw at it by default, so encapsulation won't save you from this. Even if you only consume your own custom images, you want to make sure nobody inside the organization is able to tamper with an image. The solution usually boils down to the classical PKI-based chain of trust.

Best practices:

- The regular Internet common sense: do not run unverified software from sources that you don't explicitly trust.
- Deploy a container-centric trust server using some of the registry servers.
- Enforce mandatory signature verification for any image that is going to be pulled or run on your systems.

Example: Deploying a full-blown trust server is beyond the scope of this card, but you can start signing your images right away.

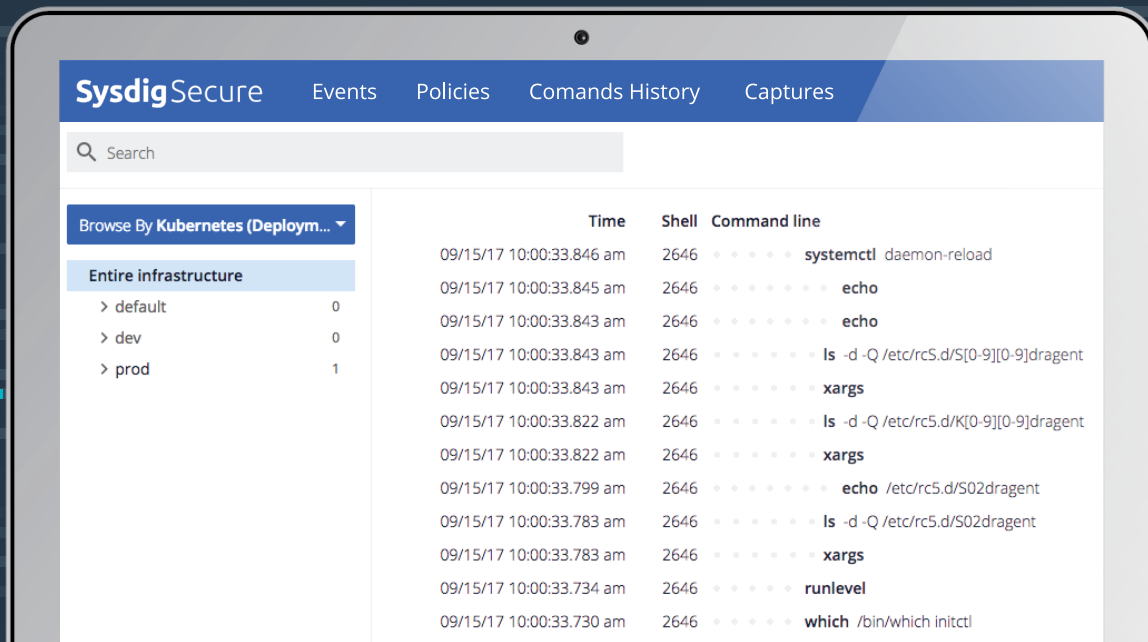
1. Get a Docker Hub account if you don't have one already.
2. Create a directory containing the following trivial Dockerfile:

20 Docker Security Tools Compared



What's Inside Your Containers?

Sysdig Secure. Container security and forensics platform.



The screenshot shows the Sysdig Secure web interface. At the top, there are navigation tabs: SysdigSecure, Events, Policies, Comands History, and Captures. Below the navigation is a search bar. On the left, there is a sidebar with a 'Browse By Kubernetes (Deploy...)' dropdown menu. Underneath, it shows 'Entire infrastructure' with a tree view containing 'default' (0), 'dev' (0), and 'prod' (1). The main content area displays a table of events with columns for Time, Shell, and Command line.

Time	Shell	Command line
09/15/17 10:00:33.846 am	2646	systemctl daemon-reload
09/15/17 10:00:33.845 am	2646	echo
09/15/17 10:00:33.843 am	2646	echo
09/15/17 10:00:33.843 am	2646	ls -d -Q /etc/rc5.d/S[0-9][0-9]dragent
09/15/17 10:00:33.843 am	2646	xargs
09/15/17 10:00:33.822 am	2646	ls -d -Q /etc/rc5.d/K[0-9][0-9]dragent
09/15/17 10:00:33.822 am	2646	xargs
09/15/17 10:00:33.799 am	2646	echo /etc/rc5.d/S02dragent
09/15/17 10:00:33.783 am	2646	ls -d -Q /etc/rc5.d/S02dragent
09/15/17 10:00:33.783 am	2646	xargs
09/15/17 10:00:33.734 am	2646	runlevel
09/15/17 10:00:33.730 am	2646	which /bin/which initctl



“All in all, we found that Sysdig is the only one who has unified performance monitoring and security, and done it in a low-resource and cost-effective way.

- Sun Run

BLOCK ATTACKS

Don't just detect an attack - block it. Automatically kill containers you suspect to be compromised based on application, container, or network activity. All without per-pod instrumentation or code changes.

RESOLVE BREACHES

See pre- and post-attack activity. Drill down from policy violation, to open connections, all the way down to the actual data written to file. Our forensics allow you inspect data outside of production, even if the containers are long gone.

COMPLETE AUDIT & GOVERNANCE

Capture a complete audit trail of user actions, container activity, and command-line arguments. Review any policy violation with rich, service-oriented context. Send audit trails downstream into a SIEM or any compliance software.

Visit sysdig.com and find out.

Sysdig

```
# cat Dockerfile
FROM alpine:latest
```

Build the image:

```
# docker build -t <youruser>/alpineunsigned .
```

Log into your Docker Hub account and submit the image:

```
# docker login
[...]
```

```
# docker push <youruser>/alpineunsigned:latest
```

Enable Docker trust enforcement:

```
# export DOCKER_CONTENT_TRUST=1
```

Now, try to retrieve the image you just uploaded:

```
# docker pull <youruser>/alpineunsigned
```

You should receive the following error message:

```
Using default tag: latest
Error: remote trust data does not exist for docker.
io/<youruser>/alpineunsigned:
notary.docker.io does not have trust data for docker.
io/<youruser>/alpineunsigned
```

Now that `DOCKER_CONTENT_TRUST` is enabled, you can build the container again and it will be signed by default.

```
# docker build --disable-content-trust=false -t
<youruser>/alpineunsigned:latest .
```

Now, you should be able to push and pull the signed container without any security warning. The first time you push a trusted image, Docker will create a root key for you and you will also need a repository key for the image. Both will prompt you for a user-defined password.

Your private keys are in the `~/.docker/trust` directory; safeguard and back them up.

The `DOCKER_CONTENT_TRUST` is just an environment variable and will die with your shell session. But trust validation should be implemented across the entire process, from the images building and the images hosting in the registry to images execution in the nodes.

DOCKER CREDENTIALS AND SECRETS

Your software needs sensitive information to run: user password hashes, server-side certificates, encryption keys, etc. This situation is made worse by the nature of containers; you don't just "set up a server" — there's a large number of distributed containers that may be

constantly created and destroyed. You need an automatic and secure process to share this sensitive info.

Best practices:

- Do not use environment variables for secrets; this is a very common yet very insecure practice.
- Do not embed any secrets in the container image. Read this IBM post-mortem report: "The private key and the certificate were mistakenly left inside the container image."
- Deploy a Docker credentials management software if your deployments get complex enough. Do not attempt to create your own "secrets storage" (curl-ing from a secrets server, mounting volumes, etc.) unless you really know what you are doing.

Examples: First, let's see how to capture an environment variable:

```
# docker run -it -e password='S3cr3tp4ssw0rd' alpine sh
/ # env | grep pass
password=S3cr3tp4ssw0rd
```

It's that simple, even if you su to a regular user:

```
/ # su user
/ $ env | grep pass
password=S3cr3tp4ssw0rd
```

Nowadays, container orchestration systems offer some basic secret management. For example, Kubernetes has the secrets resource. Docker Swarm has also its own secrets feature, which will be quickly demonstrated here.

Initialize a new Docker Swarm (you may want to do this on a VM):

```
# docker swarm init --advertise-addr <your_advertise_addr>
```

Create a file with some random text, your secret:

```
# cat secret.txt
This is my secret
```

Create a new secret resource from this file:

```
# docker secret create somesecret secret.txt
```

Create a Docker Swarm service with access to this secret; you can modify the uid, gid, mode, etc:

```
# docker service create --name nginx --secret
source=somesecret,target=somesecret,mode=0400 nginx
```

Log into the Nginx container; you will be able to use the secret:

```
root@3989dd5f7426:/# cat /run/secrets/somesecret
This is my secret
root@3989dd5f7426:/# ls /run/secrets/somesecret
```

```
-r----- 1 root root 19 Aug 28 16:45 /run/secrets/
somesecret
```

This is a minimal proof of concept. At the very least, now your secrets are properly stored and can be revoked or rotated from a central point of authority.

CONTAINER RESOURCE ABUSE

Containers are much more numerous than virtual machines on average. They are lightweight and you can spawn big clusters of them on modest hardware. That's definitely an advantage, but it implies that a lot of software entities are competing for the host resources. Software bugs (such as memory leaks), design miscalculations, or a deliberate malware attack can easily cause a Denial of Service if you don't properly configure resource limits.

To add to the problem, there are several different resources to safeguard: CPU, main memory, storage capacity, network bandwidth, I/O bandwidth, swapping... there are some kernel resources that are not so evident, and even more obscure resources such as user IDs (UIDs) exist.

Best practices: Limits on these resources are disabled by default on most containerization systems; configuring them before deploying to production is basically a must. There are three fundamental steps:

- Use the resource limitation features bundled with the Linux kernel and/or the containerization solution.
- Try to replicate the production loads on pre-production. Some people use synthetic stress tests, and others choose to "replay" the actual real-time production traffic. Load testing is vital to knowing where the physical limits are and where your *normal* range of operations is.
- Implement Docker monitoring and alerting. You don't want to hit the wall if there is a resource abuse problem. Malicious or not, you need to set thresholds and be warned before it's too late.

Example: Control groups, or *cgroups*, are a feature of the Linux kernel that allow you to limit the access processes and containers have to system resources. We can configure some limits directly from the Docker command line:

```
# docker run -it --memory=2G --memory-swap=3G ubuntu bash
```

This will limit the container to 2GB main memory, 3GB total (main + swap). To check that this is working, we can run a load simulator; for example, the **stress** program present in the Ubuntu repositories:

```
root@e05a311b401e:/# stress -m 4 --vm-bytes 8G
```

You will see a "FAILED" notification from the **stress** output. If you tail the syslog on the hosting machine, you will be able to read something similar to:

```
Aug 15 12:09:03 host kernel: [1340695.340552] Memory
cgroup out of memory: Kill process 22607 (stress) score
210 or sacrifice child
```

```
Aug 15 12:09:03 host kernel: [1340695.340556] Killed
process 22607 (stress) total-vm:8396092kB, anon-
rss:363184kB, file-rss:176kB, shmem-rss:0kB
```

Using Docker stats, you can check current memory usage and limits. If you are using Kubernetes, you can actually book the resources that your application needs to run properly and define maximum limits using requests and limits on each pod definition:

```
[...]
- name: wp
  image: wordpress
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
[...]
```

STATIC VULNERABILITY SCANNING

Containers are isolated black boxes: if they are doing their work as expected, it's easy to forget which software and version is specifically running inside. Maybe a container is performing like a charm from the operational point of view, but it's running version X.Y.Z of the web server, which happens to suffer from a critical security flaw. This flaw was fixed long ago upstream, but not in your local image. This kind of problem can go unnoticed for a long time if you don't take the appropriate measures.

Best practices: Picturing the containers as immutable atomic units is really nice for architecture design, but from the security perspective, you need to regularly inspect their contents:

- Update and rebuild your images periodically to grab the newest security patches. Of course, you will also need a pre-production testbench to make sure these updates are not breaking production.
- Live-patching containers is usually considered a bad practice. The pattern is to rebuild the entire image with each update. Docker has declarative, efficient, easy-to-understand build systems, so this is easier than it may sound at first. Use software from a distributor that guarantees security updates. Anything you install manually out of the distro, you have to manage security patching yourself.
- Docker and microservice-based approaches consider progressively rolling over updates without disrupting uptime a fundamental requisite of their model.
- User data is clearly separated from the images, making this whole process safer.
- Keep it simple. Minimal systems expect less frequent updates. Remember the intro: less software and moving parts equals

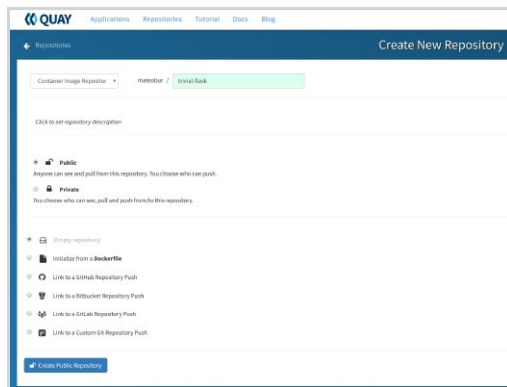
less attack surface and updating headaches. Try to split your containers if they get too complex.

- Use a vulnerability scanner. There are plenty out there, both free and commercial. Try to stay up-to-date on the security issues of the software you use subscribing to the mailing lists, alert services, etc.
- Integrate this vulnerability scanner as a mandatory step of your CI/CD and automate where possible; don't just manually check the images now and then.

Example: There are multiple Docker images registry services that offer image scanning. For this example, we decided to use CoreOS Quay, which uses the open-source Docker security image scanner Clair. Quay is a commercial platform but some services are free to use. You can create a personal trial account by following [these instructions](#).

Once you have your account, go to Account Settings and set a new password (you need this to create repos).

Click on the + symbol on your top right and create a new public repo:

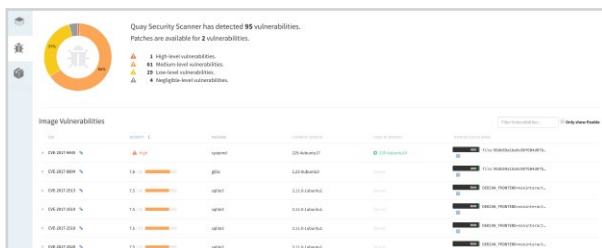


We go for an empty repository here, but you have several other options, as you can see in the image above.

Now, from the command line, we log into the Quay registry and push a local image:

```
# docker login quay.io
# docker push quay.io/<your_quay_user>/<your_quay_image>:<tag>
```

Once the image is uploaded into the repo, you can click on its ID and inspect the image security scan, ordered by severity, with the associated CVE report link and upstream patched package versions.



RUNTIME CONTAINER SECURITY

DOCKER INTRUSION DETECTION

In the previous sections, we covered the static aspects of Docker security: vulnerable kernels, unreliable base images, capabilities that are granted or denied at launch-time, etc. But what if, despite all these, the image has been compromised during runtime and starts to show suspicious activity?

Best practices:

- All the previously described static countermeasures do not cover all attack vectors. What if your own in-house application has a vulnerability? Or attackers are using a 0-day not detected by the scanning? Runtime security can be compared to Windows anti-virus scanning: detect and prevent an existing break from further penetration.
- Do not use runtime protection as a replacement for any other static up-front security practices: Attack prevention is always preferable to attack detection. Use it as an extra layer of peace-of-mind.
- Having generous logs and events from your services and hosts, correctly stored and easily searchable and correlated with any change you do, will help a lot when you have to do a post-mortem analysis.

Example: Sysdig Falco is an open-source behavioral monitoring software designed to detect anomalous containerized activity. Sysdig Falco works as an intrusion detection system on any Linux host, although it is particularly useful when using Docker since it supports container-specific contexts like `container.id`, `container.image`, Kubernetes resources, or namespaces for its rules.

Falco comes with a **default ruleset** to spot behaviors like binary directories being modified, writes below, and many other activities that could be a sign some type of malicious activity is occurring. Falco rules can trigger notifications on multiple anomalous activities. Let's show a simple example of someone running an interactive shell in one of the production containers.

First, we will install Falco as a container on the host: `docker pull sysdig/falco`

```
docker run -i -t --name falco --privileged -v /var/run/docker.sock:/host/var/run/docker.sock -v /dev:/host/dev -v /proc:/host/proc:ro -v /boot:/host/boot:ro -v /lib/modules:/host/lib/modules:ro -v /usr:/host/usr:ro sysdig/falco
```

And then we will run an interactive shell in a Nginx container:

```
# docker run -d --name nginx nginx
# docker exec -it nginx bash
```

On the hosting machine, tail the `/var/log/syslog` file and you will be able to read:

```
Aug 15 21:25:31 host falco: 21:25:31.159081055: Debug
Shell spawned by untrusted binary (user=root shell=sh
parent=anacron cmdline=sh -c run-parts --report /etc/
cron.weekly pcmdline=anacron -dsq)
```

Sysdig Falco doesn't need to modify or instrument containers in any way! This is just a trivial example of Falco capabilities; check out these examples to learn more.

DETECTING AND BLOCKING ATTACKS ON ORCHESTRATED MICROSERVICES

Containers are the base building block of a service that is managed by popular orchestrators like Kubernetes or Docker Swarm. There are many cases where the same image will be used in different areas of your infrastructure depending on what that specific service that image is providing to an application — think load balancer images like HAProxy or Nginx.

Often, teams apply standard security policies to an image and don't differentiate at the application and orchestration layer, which can leave holes in the service by trying to protect everything via blanket policies. This is where tight integrations with an orchestrator and the metadata they provide is needed to differentiate between the services that are running an image.

Best practices:

- Differentiate between images by using orchestration and container labels for more granular policies.
- Take actions like killing or pausing a container when a policy has been violated. Often, the orchestrator will spin up a new container bringing the environment to a safe state.
- Rely on orchestration metadata rather than trying to manually update container image hashes.

Example: Let's look into preventing data exfiltration in Kubernetes with Sysdig Secure. Sysdig Secure is a container security platform that provides run-time security and forensics. Sysdig Secure installs as a privileged container on the host and instruments the underlying kernel to see all system calls that are happening on the host. The container also integrates with Kubernetes APIs to tag all file opens, processes, etc. from being passed through the data pipeline.

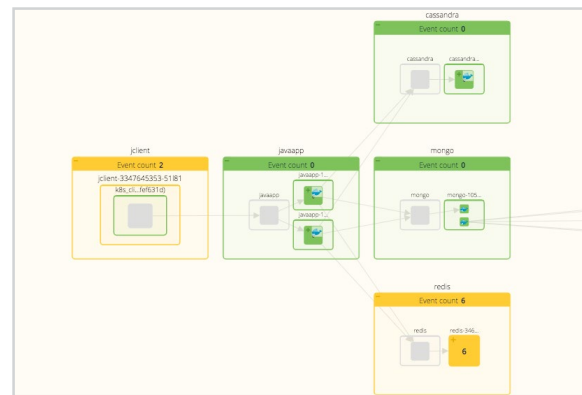
Let's first build a policy to detect outbound connections from a Redis data service. This policy is scoped by `kubernetes.deployment.name = redis`, meaning that pods/containers that fall under that specific deployment will have this policy applied regardless of what region or host that container resides on. Next, apply a rule to detect any outbound connections coming from that service.

Description	Detect Outbound Network Connections for redis deployment, which should not have any
Severity	High Medium Low
Scope	kubernetes.deployment.name = redis
Apply to	Hosts and containers Hosts only Containers only
Rule	No Outbound Connection

The step above will detect any outbound connection from the Redis deployment, but what about taking actions to stop the data exfiltration event? Actions can be taken based on any policy violation in Sysdig Secure. In this case, we'll want to stop the container to prevent data from leaving, and then Kubernetes will spin up a new container to return the Redis service to a steady state. We'll cover figuring out how data exfiltration, intrusions, and other events occur in the next section, Incident Response.

Rule	No Outbound Connection
You can edit security rules in the Rules Editor.	
Actions	<input checked="" type="checkbox"/> Stop the container <input type="checkbox"/> Pause the container

These events can also be explored easily through topology maps to quickly see where policy violations are occurring in your infrastructures, as well as any dependencies the Kubernetes services have on each other.



INCIDENT RESPONSE

POST-MORTEM ANALYSIS

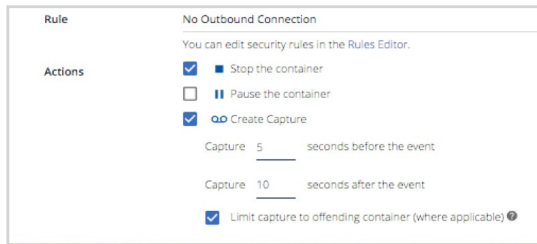
Incident response becomes harder in container environments because of their ephemeral nature. A bad actor can delete a container after an attack to remove all traces of any file access or network activity. Not to mention, gaining visibility into what is happening inside a container in the first place is very difficult.

Best practices:

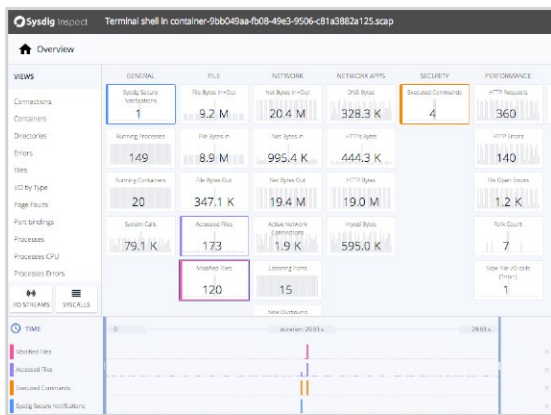
- Log everything that's from stdout in containers across your infrastructure.
- Capture network, file, system call, and user data from inside the container around any policy violation.
- Commit containers that have policy violations so that they can be examined and ran outside of production.

Example: Forensic analysis with Sysdig Inspect. Inspect is an open-source interface for container troubleshooting and security investigation. It can be coupled with Sysdig Secure to do deep analysis on Sysdig captures. A capture is a .scap file of all system calls that happened over a period of time (for anyone who has done network forensic analysis before, this is very similar to a .pcap file). Sysdig Secure allows users to take a buffering capture to collect data pre and post any security violation, which can be tuned on a per-policy basis to decide

the window of data to collect pre and post policy violations. Captures can also be manually initiated from the sysdig open-source tool or via the Sysdig Secure UI.



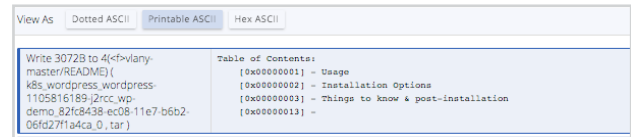
Once a capture has been taken, you can read the capture file with Sysdig, Csysdig, or Sysdig Inspect. In this case, let's look at what a rootkit installation would look like in Inspect. The overview page offers an out-of-the-box, at-a-glance summary of the content of the capture file. Content is organized in tiles, each of which shows the value of a relevant metric and its trend. Tiles are organized in categories to surface useful information more clearly and are the starting point for investigations and drill downs. The timeline can be used to drill into sub-second granularity of all the events happening on the system.



From there, additional filtering can be done to view individual files, network connections, or threads from a process. In this case, we're looking at all files written by the tar process.

BYTES OUT	OPS	OPENS	ERRORS	Container	FILENAME
0	6	3	0	k8s_wordpress_wordpress-1...	/etc/ld.so.cache
31401	6	0	0	k8s_wordpress_wordpress-1...	vlan1-master/README
35141	6	0	0	k8s_wordpress_wordpress-1...	vlan1-master/LICENSE
15654	5	0	0	k8s_wordpress_wordpress-1...	vlan1-master/symbols/exec/execute.c
12482	5	0	0	k8s_wordpress_wordpress-1...	vlan1-master/symbols/backdoor/pam/pam_private.h
16517	5	0	0	k8s_wordpress_wordpress-1...	vlan1-master/install.sh

You can even go a layer deeper within the forensic analysis and inspect contents written to a file even after the offending container is long gone.



Having a quick container-native workflow for forensics investigation is critical in container environments where services are coming and going, while at the same time, container density and the abstraction of services are increasing.

CONCLUSION

In this Refcard, we've walked from the first principles of security before containers even enter production environments all the way up to the more complex analysis of a rootkit installation. In this card, we've used components baked into the Docker runtime, open source tools like Clair and Falco, and even performed deep system call forensic analysis with Sysdig Secure.

As you can see, Docker security can start very simply but grow complex as you actually take containers into production. Get experience early and then grow your security sophistication to what your environment requires.

DZONE.COM/REFCARDZ

Written by Knox Anderson, Product Marker, Sysdig

Knox Anderson is a container aficionado, working in product marketing at Sysdig focused on security and forensic solutions for containers and microservices. Prior to joining Sysdig he first discovered containers as an easy way to demo complex products like distributed SQL databases and has been helping companies of all sizes make their experience of running containers in production easier. Knox holds a BS in Business Management Information Systems and Services from Boston University.

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

DZone, Inc.
 150 Preston Executive Dr. Cary, NC 27513
 888.678.0399 919.678.0300

Copyright © 2017 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.