Scientific and Practical Cyber Security Journal (SPCSJ) 4(1): 87 – 92 ISSN 2587-4667 Scientific Cyber Security Association (SCSA)

ELIMINATING PRIVILAGE ESCALATION TO ROOT IN CONTAINERS RUNNING ON KUBERNETES

[1]Linetskyi Artem, [2]Babenko Tetiana, [3]Myrutenko Larysa, [4]Vialkova Vira
1-4 Faculty of Information Technology, Taras Shevchenko National University of Kyiv, Ukraine
artem.linetskiy@gmail, babenkot@ua.fm, myrutenko.lara@gmail, veravialkova@gmail.com

ABSTRACT: containerization and orchestration tools like Kubernetes allow enterprises to automate many aspects of application lifecycle, especially deployment, significant business benefits. However, these new deployments are also vulnerable to attacks and introduce new exploits from hackers and insiders, making Kubernetes security a crucial component for every deployment. We perform a study analyzing privilege escalation to root in containers running on Kubernetes. Based on the results we create a solution that can eliminate this type of attack.

KEYWORDS*: kubernetes; privilage escalation; cybersecurity*, *security layer.*

I. INTRODUCTION

Lately, microservice based architecture has proven to be the most reliable and scalable approach to developing an application. With the popularity of small apps, the solution to maintain them was found - Kuberntes. Kubernetes is an orchestration tool that can easily manage thousands deployments and scale and container-based workload. With wide adoption by many organization and major cloud companies, it has exploded onto the technology scene over the last couple of years. Once Kubernetes has become the new industry standard, hackers and malicious organisations also did't miss the chance. Attacks for crypto mining, ransomware, service disruption and data stealing will continue to be launched in the new container based virtualized environments in both public and private clouds. In the recent days even such large company as Tesla was affected by hacker's interest to Kubernetes clusters. "The hackers had infiltrated Tesla's Kubernetes console which was not password protected," RedLock researchers wrote. "Within one Kubernetes pod, access credentials were exposed to Tesla's AWS environment which contained an Amazon S3 (Amazon Simple Storage Service) bucket that had sensitive data such as telemetry" [1,2].

II. BACKGROUND

A. Kubernetes basics

Kubernetes is tool, which automates the updates, deployment and monitors containers. Kubernetes is supported by all major container management and cloud platforms such as Red Hat OpenShift, IBM Cloud, AWS EKS and Google Cloud. Here are some of the key things to know about Kubernetes:

- **Master Node**. A server that manages the deployment of pods the Kubernetes worker node cluster.
- **Worker Node**. Servers that typically run the app containers or other Kubernetes components, they are also called slaves or minions sometimes.
- **Pods**. A single deployment and addressability unit in Kubernetes. A pod can have one or more containers and their own IP addresses.
- **Services.** A service functions as a proxy to its underlying pods and requests can be load balanced across replicated pods.
- **System Components.** Key components that are used to manage a Kubernetes cluster include the API Server, Kubelet, and etcd. Each of these components are the most likely targets for the attack.

B. Kubernetes Networking Basics

Kubernetes' main concept in networking is that every pod has routable IP address (Fig. 1). Plug-ins for Kubernetes take care of routing requests internally between host machines to appropriate pods. Kubernetes pods can be accessed from outside only through a load balancer, service or ingress controller, which routes the traffic to the correct pod.

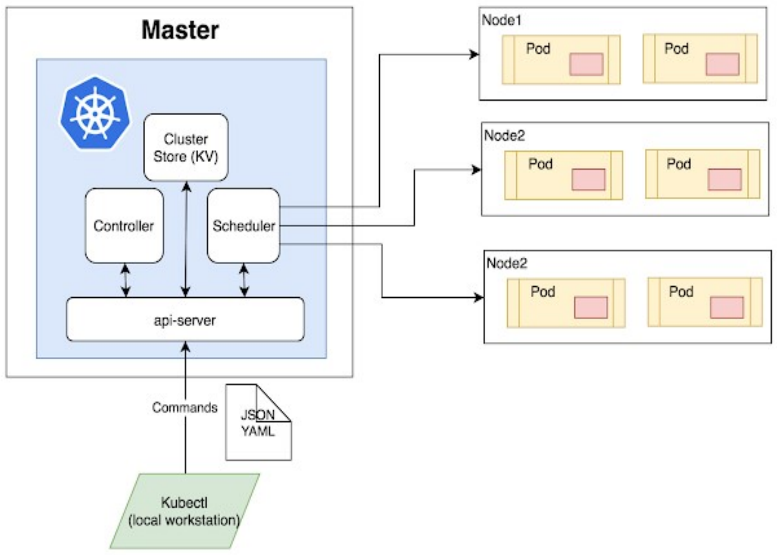


Fig. 1. *Kubernetes architecture diagram*

DNAT and Load balancing help to make connections to the correct pod. Pods communicate with each other over the network overlay. Encapsulation is used for packets, where appropriate headers are added to get them to the appropriate destination, where the encapsulation is removed.

Having all that overlay, being dynamically handled by Kubernetes, it is extremely difficult to monitor network traffic, and even more difficult to secure it [3].

C. Container Inspection

Attacks in containerized environments usually utilize malicious processes and privilege escalations to execute an attack or spread it. Exploits of vulnerabilities in the Linux kernel, libraries, packages, or applications can result in suspicious activity inside the container.

Each application in a container has a defined set of functions and it is built with only the needed libraries and packages, detecting suspicious file system activity and processes should be much more accurate.

Inspecting container file system activity, processes and suspicious behavior is crucial in container security. Reverse shells, port scanning or privilege escalations should all be detected at the beginning. Some of these detections should be built-in and also include a baseline behavioral learning process which can identify unusual processes based on previous activity [3].

III. ESCALATION CAUSE ANALYSIS

Since every container running in Kubernetes is running on Docker, there's always a possibility of other libraries being compromised. Hence, "compromised container = compromised cluster" we should take seriously protection of Docker containers running on production grade environments like Kubernetes. The best and simplest security measure that could be taken is avoid running containers as root user. Let's break down why is it important.

A. Non-root containers

When application is running on the host machine, it is normally understood why isolation between the root user and non-privileged users is required. If application is run as a root, if breached, it can easily wreak the system, by modifying system files, stopping or launching privileged processes and so on. A lot of Linux daemons remove privileges in the early setup stages, for example the Nginx daemon forks and runs as the unprivileged www-data user.

Containers have just simplified the privilege separation. Separate runtime is given to each container and it's isolated by Linux namespaces with a set of capabilities. Every application is run in a separate container, and can be considered to be safe to use in the container.

Therefore it has accidently become a common practice to run application with root user and most of the Docker images don't change to unprivileged user. However, a very important security layer is eliminated in such cases. It's even more true, considering that there's no real reasoning to why, such images should be run as root.

On the other hand, there are container platforms that run all their containers as unprivileged users by default. OpenShift is a great example, that only allows its users to use images with support for a random, non-root user.

Some containerized applications can remove root privilege, changing the root user to a non-privileged straight after the setup. This allows them to rely on file permissions, based on users and prevents access to sensitive information (e.g. configuration files, processes) in the containers. Such preventions, really limit the damage, that can be done to a compromised container. A great example of such design is Jenkins image. [4].

Nonetheless, using a non-root user in containers is a crucial security precaution to prevent container breakout.

In most container runtimes root user is shared between the host machine and containers. This is not a problem, because the container processes are sandboxed using Linux capabilities and Linux namespaces such as PID, net, mount and IPC. However, when kernel doesn't distinct the user or group IDs of the host and the container, vulnerabilities in the container, that exposes everything from the host to a container process, creates real trouble. Simply speaking, a container breakout is more likely to happen for processes running as root.

In this situation, any attacker's process will be able to change files on the system, without further escalating privileges to the root user. [5].

IV. Root privilages escalation method

Root-privilege escalation can be performed in a number of different ways. These are some of them:

- System libraries' (runC, libC) vulnerabilities exploitation
- Improper following of symbolic links
- Usage of same volumes for different containers
- Go programming language vulnerabilities
- Docker vulnerabilities
- Default unsecure configuration problem
- Natively Kubernetes security issues

In this particular case we'll look at root privilege escalation using known Kubernetes vulnerability in kubelet v1.13.6 and v1.14.2. This versions are still broadly used by both cloud providers and on-premise deployments.

A. So What Is Kubelet?

Kubelet is an agent, that's present on every Kubernetes server. Kubelet takes a PodSpec which is a description of pod resource. PodSpec is usually described in a form of a YAML file. Kubelet makes sure that every file with pod description is successfully transformed into a healthy running pod.

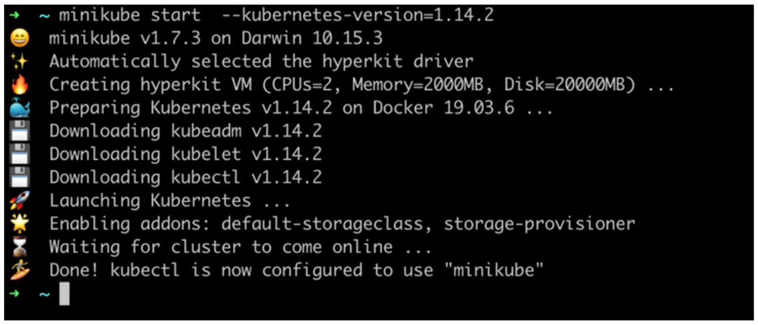
B. Exploitation process

Every container that is deployed in Kubernetes environment is usually created with a user that is specified in the respective Dockerfile. Hovewer, Kubernetes includes native handling for this and gives a possibility to specify a user in PodSpec file or configure the container to avoid running as root user. So the regular behavior is for Kubernetes to start the container with user in the dockerfile if nothing else was specified in PodSpec (like *runAsUser: <uid>* or *mustRunAsNonRoot:true).* But sometimes the image can be run as root instead. These are some of the scenarios:

- explicitly specified *runAsUser* pods are unaffected and continue to work properly
- *runAsUser* setting that is forced by *podSecurityPolicies* keeps containers unaffected and works properly
- If *mustRunAsNonRoot:true* is specified, the container will refuse to start as uid 0, this can affect availability
- If *runAsUser* or *mustRunAsNonRoot:true are not specified, pods* will run as uid 0 on restart or if the image was pulled to the node previously

C. Steps to reproduce

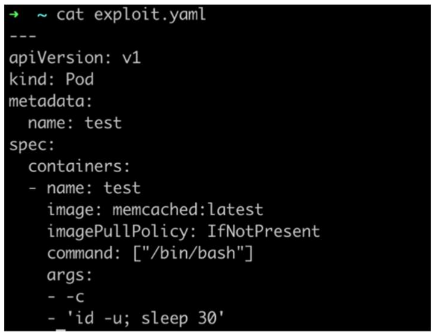
In order to reproduce this issue, security analyst or malicious user needs to have access to pod resources and ability to create them in the cluster. He can have access to any given namespace. To reproduce this problem minikube will instance will be created (Fig 2)



```
➜  ~ minikube start  --kubernetes-version=1.14.2
😄  minikube v1.7.3 on Darwin 10.15.3
✨  Automatically selected the hyperkit driver
🔥  Creating hyperkit VM (CPUs=2, Memory=2000MB, Disk=20000MB) ...
🐳  Preparing Kubernetes v1.14.2 on Docker 19.03.6 ...
💾  Downloading kubeadm v1.14.2
💾  Downloading kubelet v1.14.2
💾  Downloading kubectl v1.14.2
🚀  Launching Kubernetes ...
🌟  Enabling addons: default-storageclass, storage-provisioner
⌛  Waiting for cluster to come online ...
🏄  Done! kubectl is now configured to use "minikube"
➜  ~ █
```

Fig. 2. *Minikube instance creation process*

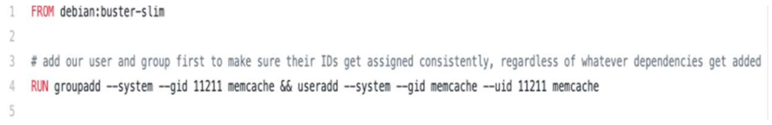
After environment for reproduction is ready, we need to prepare a test pod resource YAML file. Example of a YAML file (Fig 3):



```
➜  ~ cat exploit.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - name: test
    image: memcached:latest
    imagePullPolicy: IfNotPresent
    command: ["/bin/bash"]
    args:
    - -c
    - 'id -u; sleep 30'
```

Fig. 3. *Example of exploit YAML file*

It's a simple pod resource that has a base image of memcached:latest, which has it's process run as memcached user. Snippets of Dockerfile confirming that (Fig 4, Fig 5):

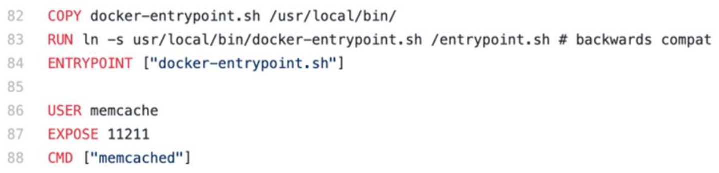


```
FROM debian:buster-slim

# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever dependencies get added
RUN groupadd --system --gid 11211 memcache && useradd --system --gid memcache --uid 11211 memcache

```

Fig.4. *First snippet of memcached Dockerfile*

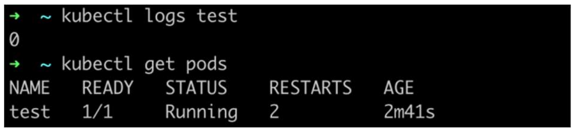


```
COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # backwards compat
ENTRYPOINT ["docker-entrypoint.sh"]

USER memcache
EXPOSE 11211
CMD ["memcached"]
```

Fig.5. *Second snippet of memcached Dockerfile*

So, it's clear that the container should be executed with memcache user, specified in the Dockerfile. According to the reproduction steps, if the container will restart, next time it will be executed with uid 0, the command with args in the Pod YAML file simulates just that. It displays the uid of the user that container is running under and then waits 30 seconds and restarts. After the restart it should be executed by root. The proof of this concept is available on Fig 6.



```
➜  ~ kubectl logs test
0
➜  ~ kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
test   1/1     Running   2          2m41s
```

Fig. 6. *Proof of exploit*

The container was restarted 2 times and according to the logs it is executed with uid 0, so we can verify that expoit works, which verifies that some security measures should be taken in order to protect any existing Kubernetes cluster.

V. Mitigations

A. Avoidance

This problem exists on the level of the core Kubernetes component, so this is Kubernetes development team responsibility to patch it in the next release. However there are some security measures that can be taken before the patch version release:

- Specify *runAsUser* directives in pods to control the uid a container runs as
- Specify *mustRunAsNonRoot:true* directives in pods to prevent starting as root (note this means the attempt to start the container will fail on affected kubelet versions)
- Downgrade kubelets to v1.14.1 or v1.13.5 as instructed by your Kubernetes distribution.

VI. Conclusion

Containerization brings a lot of benefits to the industry, speeds up the development and deployment processes and improves the workflow. Needless to say how Kubernetes has changed production deployment management, improved usability and scalability of various applications and became popular among small businesses and large enterprise companies like Google and Tesla.

However, all the rapid development brings new horizons for hackers and security breaches. Therefore, it’s always important to treat security seriously, take minimal measures to ensure no stupid misconfiguration or mistake will result in resource loses [10].

Therefore in recently created environments, like Kubernetes, security measures have to be taken even more seriously and every plugin or configuration for a more secure environment should be inspected and enforced.

References

[1] "Production-Grade Container Orchestration", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/. [Accessed: 30- Oct- 2019].

[2] D. Goodin, "Tesla cloud resources are hacked to run cryptocurrency-mining malware", *Ars Technica*, 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/. [Accessed: 30- Sep- 2019].

[3] F. Huang and G. Duan, "The Ultimate Guide to Kubernetes Security - Threats, Tips, and Ebook", *NeuVector*, 2018. [Online]. Available: https://neuvector.com/container-security/kubernetes-security-guide/. [Accessed: 30- Oct- 2019].

[4] C. Meléndez, "The top Kubernetes security best practices - Sqreen blog", *Sqreen Blog*, 2019. [Online]. Available: https://blog.sqreen.com/kubernetes-security-best-practices/. [Accessed: 30- Sep- 2019].

[5] A. Zelivansky, "Non-Root Containers, Kubernetes CVE-2019-11245 and Why You Should Care", *Unit42*, 2019. [Online]. Available: https://unit42.paloaltonetworks.com/non-root-containers-kubernetes-cve-2019-11245-care/. [Accessed: 11- Nov- 2019].

[6] C. Gilbert, "9 Kubernetes Security Best Practices Everyone Must Follow - Cloud Native Computing Foundation", *Cloud Native Computing Foundation*, 2019. [Online]. Available: https://www.cncf.io/blog/2019/01/14/9-kubernetes-security-best-practices-everyone-must-follow/. [Accessed: 30- Nov- 2019].

[7] Y. Avrahami, "Breaking out of Docker via runC – Explaining CVE-2019-5736", *Unit42*, 2019. [Online]. Available: https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/. [Accessed: 09- Nov- 2019].

[8] A. Martin, "11 Ways (Not) to Get Hacked", *Kubernetes.io*, 2018. [Online]. Available: https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/. [Accessed: 30- Oct- 2019].

[9] "Extending your Kubernetes Cluster", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/ extend-cluster/. [Accessed: 19- Oct- 2019].

[10] S. Prodan, "Scanning Kubernetes resources with Kubesec", *Stefanprodan.com*, 2018. [Online]. Available: https://stefanprodan.com/2018/scanning-kubernetes-deployments-with-kubesec/. [Accessed: 30- Sep- 2019].