WEBINAR SUMMARY

March 11, 2021

Securing Containers and Kubernetes-Orchestrated Environments

Sheila Berta, Head of Security Research, Dreamlab Technologies
Haim Helman, CTO, Carbon Black App Security, VMWare

KEY TAKEAWAYS

- Docker containers can be secured in multiple ways.
- Distroless multi-stage builds and Docker Content Trust are useful tools for securing Docker images.
- Kubernetes clusters can be secured through TLS certificates, API authentication and authorization, secrets management solutions, security context, and network policies.
- In cloud-native environments, containers present application, network group, and container privilege-related risks.
- Kubernetes is a powerful API that appeals to attackers.
- New security tools are needed for Kubernetes across the application life cycle.

in partnership with

OVERVIEW

Containers have revolutionized the software industry. Developers can package applications that behave exactly as tested and enterprises can split them into scalable microservices. Securing locally deployed containers seems neither complex nor complicated. The picture changes, however, when the ecosystem grows and thousands of containers must be orchestrated to maintain availability.

The complexity of these large environments enlarges the attack surface exponentially. Moving containerized applications to cloud-native environments also introduces new security concerns. To address these issues, leading organizations are adopting a variety of tools and techniques to secure Docker containers, Docker images, and Kubernetes clusters in the cloud.

CONTEXT

Sheila Berta and Haim Helman discussed different approaches for protecting large, containerized ecosystems. They shared advanced security features to secure the Docker daemon and its core components, container execution, and Kubernetes-orchestrated environments.

KEY TAKEAWAYS

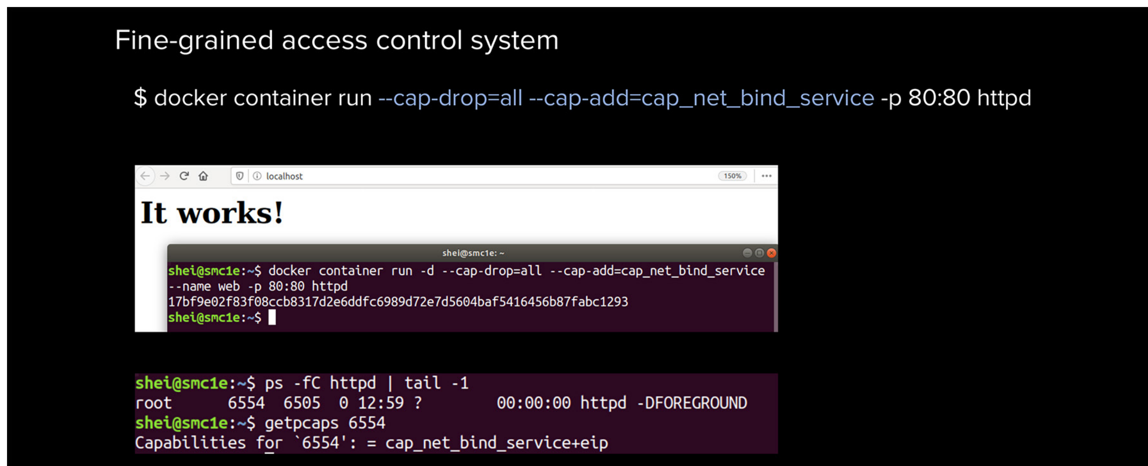
Docker containers can be secured in multiple ways.

Containers are processes that run on the host machine. They are the first isolation layer in the Docker platform. Six techniques for securing Docker containers are:

1. **Kernel namespaces.** Docker container runtimes use Linux kernel namespaces to isolate processes from other containers or processes on the host machine. Docker uses six kernel namespaces to provide the first layer of isolation at container runtime:
   - UTS isolates system identifiers.
   - PID isolates the PID space.
   - IPC isolates IPC resources.
   - NET isolates network interfaces.
   - USER isolates user and group ID spaces. This is disabled by default.
   - Mount isolates the set of filesystem mount points.
2. **Kernel capabilities.** Docker drops most kernel capabilities for containers. By default, the root within a container has fewer privileges than the root of the host machine. Additional information about the capabilities allowed by default in Docker is available here.

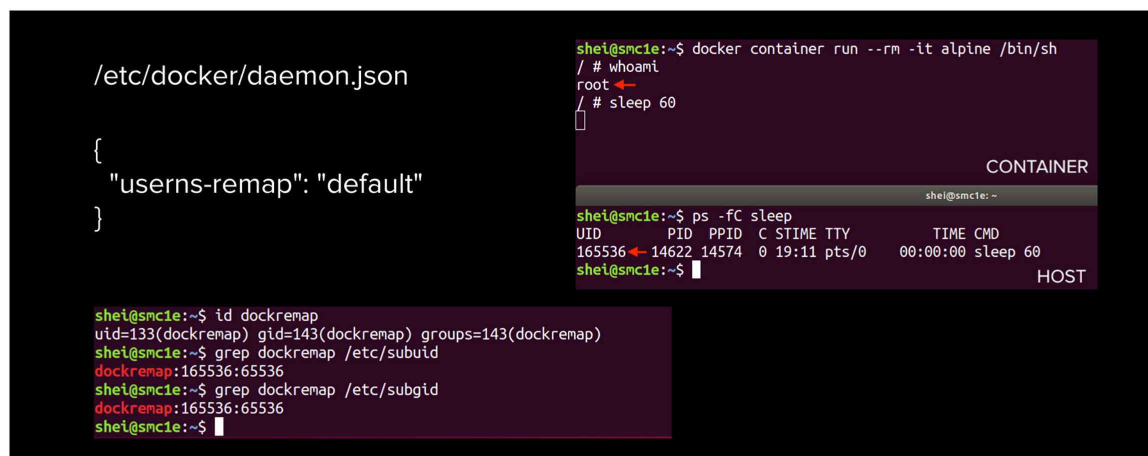
   Containers can be run with the dash-dash privilege cap, which allows all current capabilities. This is dangerous, however, because it breaks all isolation layers. If attackers compromise the container, they can control the host machine easily. Fortunately, the core also supports fine-grained access control. The cap-drop flag can be used to drop all capabilities, while the cap-add flag can be used to add the necessary capabilities.

Figure 1: Kernel Capabilities – Fine-Grained Access Control



3. **System calls restriction.** Docker supports Seccomp, a kernel feature that determines whether processes can execute system calls based on the Seccomp profile. By default, Docker blocks several system calls. This can be addressed through custom Seccomp profiles that are specified using the security ops flag at container execution. The goal is to allow only the necessary application system calls and to block the rest.
4. **Mandatory access control.** Docker supports mandatory access control through AppArmor or SELinux profiles. These restrict processes from accessing or performing certain operations on system objects. Although Docker uses a default profile, custom profiles can be created for applications. Kubernetes also supports Seccomp and AppArmor.
5. **Container UID and GID management.** When a command is executed within a container, the user running the command is root. To address this issue, itis possible to specify a non-privileged user ID, so the user running processes in the container isn't root.
6. **User namespace remap.** This feature uses the Linux user namespace to remap the root user in the container to a less privileged user in the host machine. This feature is disabled by default. To use it, a line must be added to the Docker daemon config file.

Figure 2: User Namespace Remap



Distroless multi-stage builds and Docker Content Trust are useful tools for securing Docker images.

Even if organizations aren't using Docker at container runtime with Kubernetes, they are most likely still using Docker images. As a result, it is important to properly secure Docker images. Obvious security best practices include not writing sensitive information in Docker files. It is also advisable to specify a non-privileged user and to scan the images for vulnerabilities by integrating a scanner in the continuous integration and continuous deployment (CICD) process.

Selecting the base image is one of the most important decisions when creating a Docker file for an application. One option is using Distroless base images which leverage Docker multi-stage builds. In the build stage, application artifacts are built and copied to the runtime image. Since the final image contains only the application and its runtime dependencies, the attack surface is reduced.

Docker also offers a feature called Docker Content Trust (DCT), which enables developers to digitally sign their images. Image publishers and image consumers follow the processes outlined in Figure 3. DCT prevents users from running poisoned Docker images.

Figure 3: Docker Content Trust (DCT)

DOCKER CONTENT TRUST – SIGNED IMAGES

Image Publisher side:

Step 1: $ DOCKER_CONTENT_TRUST=1

Step 2: $ docker trust key generate <your_name>

Step 3: $ docker trust signer add --key <your-key.pub> <your-name> <your-repo>

```
shei@smc1e:~$ docker tag hello-world unapibageek/demo:latest
shei@smc1e:~$ docker -D push unapibageek/demo:latest
The push refers to repository [docker.io/unapibageek/demo]
9c27e219663c: Pushed
latest: digest: sha256:90659bf80b44ce6be8234e6ff90a1ac34acbeb826903b02cfa0da11c82cbc042 size: 525
Signing and pushing trust metadata
DEBU[0015] reading certificate directory: /home/shei/.docker/tls/notary.docker.io
```

Image Consumer side:

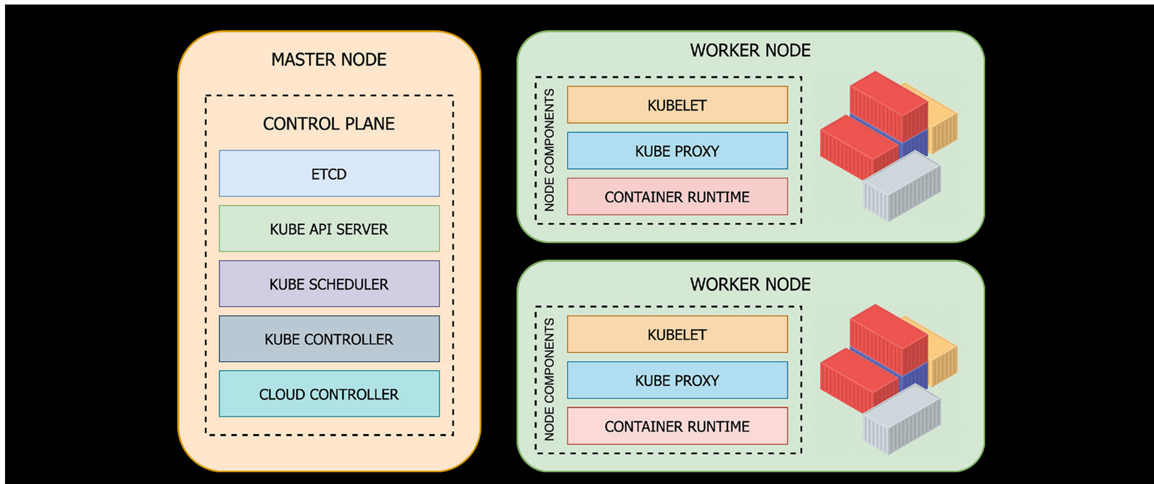
$ DOCKER_CONTENT_TRUST=1

```
shei@smc1e:~$ export DOCKER_CONTENT_TRUST=1
shei@smc1e:~$ docker pull unapibageek/ctfr
Using default tag: latest
Error: remote trust data does not exist for docker.io/unapibageek/ctfr: notary.docker.io
does not have trust data for docker.io/unapibageek/ctfr
shei@smc1e:~$ docker pull unapibageek/demo
Using default tag: latest
Pull (1 of 1): unapibageek/demo:latest@sha256:90659bf80b44ce6be8234e6ff90a1ac34acbeb82690
3b02cfa0da11c82cbc042
```

Kubernetes clusters can be secured through TLS certificates, API authentication and authorization, secrets management solutions, security context, and network policies.

A Kubernetes cluster is a set of nodes that runs containerized applications. The master node contains the control plane where cluster information is stored. If a cloud provider is used, the master node will include a cloud controller. Application workloads reside in worker nodes. Each node contains a Kubelet, which is a Kubernetes agent. Kubelets start the KUBE API Server or master node to obtain information, execute actions, and do other things.

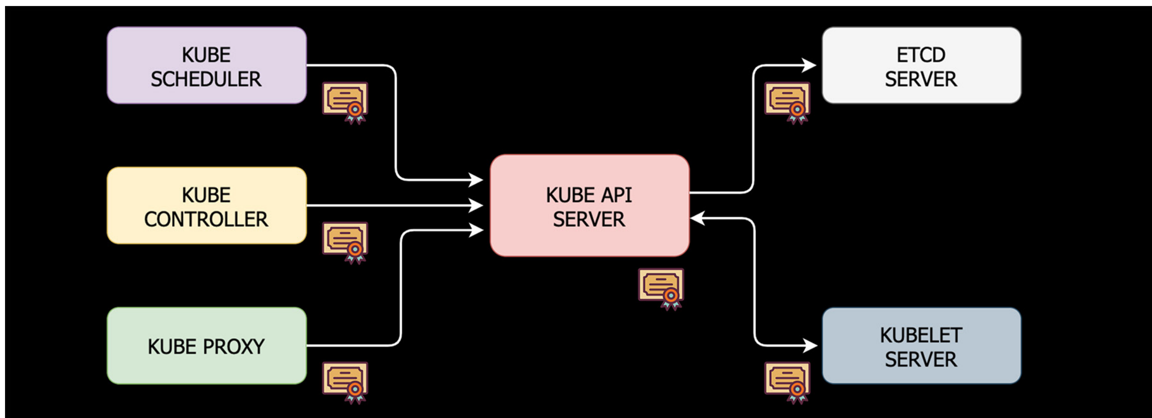
Figure 4: The Kubernetes Architecture



Six mechanisms for securing Kubernetes clusters are:

1. **Securing components communication.** All actions within a Kubernetes cluster go through the KUBE API server. The communication within various cluster components can be secured by creating a TLS certificate for each component. Typically, these certificates are automatically created. It is also possible to create and use custom certificates. These can be specified in each component's configuration file.

Figure 5: Securing Components Communication



2. **API authentication.** Kubernetes API authentication mechanisms include basic (user/password or token), TLS certificates, LDAP, Kerberos, and others. Most organizations avoid basic authentication techniques and instead use TLS certificates for on-premise clusters. The first step is to generate a key for the user, then create a certificate signing request. The final step is to sign previous records using the cluster certificate authority. With cloud-based clusters, the cloud provider will provide tokens to specify instead of client certificates.

3. **API authorization.** Kubernetes supports various authorization mechanisms including node, attribute-based access control (ABAC), role-based access control (RBAC), and WebHook. RBAC is the preferred method for users, groups, and service accounts. There are four RBAC objects: role, role binding, cluster role, and cluster role binding. Roles and cluster roles define the set of permissions. They can be bound to users, groups, and service accounts using the binding objects.

4. **Kubernetes secrets management.** Kubernetes encodes secrets using Base64. This is problematic, since Base64 isn't true encryption. Most organizations implement third-party open-source solutions for secret management, such as Hashicorp Vault, CyberArk Conjur, or Confidant. Cloud providers also offer secrets managers like AWS Secrets Manager, GCP KMS, and Azure Key Vault.
5. **Security context.** Many Docker container security features can be enforced in Kubernetes through the security context. A security context defined at the pod level applies to all containers within the pod. In addition, security contexts can be defined at the container level. Kubernetes also offers security policies. These include options for defining the security contexts applied to a cluster. Security policies are useful in large production environments.
6. **Network policies.** By default, Kubernetes allows all traffic. As a result, all deployments can talk to each other. A best practice is to implement a network policy that denies all ingress and egress traffic. On top of that, organizations can create new rules that allow specific communications. Three different selectors exist for network traffic: namespace, pod, and CIDR. These can be combined and used in ingress and egress traffic. After defining the selectors, the ports and protocols for the rules must be specified.

> Kubernetes encodes secrets using Base64. Since this isn't true encryption, many companies turn to third-party open-source solutions for secrets management.
>
> *Sheila Berta, Dreamlab Technologies*

In cloud-native environments, containers present application, network group, and container privilege-related risks.

Many organizations would like to run mission-critical and sensitive applications in a cloud-native environment, since it offers better scalability, agility, and resilience. At the same time, they don't want to compromise security.

When enterprises run applications on top of containers, security risks come from three primary sources:

1. **The applications themselves.** Applications often have vulnerabilities and containers may contain malware. Malware is particularly problematic with third-party container images. Containers don't provide perfect isolation like virtual machines. Applications, either developed in-house or from a third-party, may use an excessive amount of resources. As a result, the availability of other applications in the cluster gets disrupted.
2. **Network groups.** Kubernetes doesn't provide a default network policy, so all containers can communicate with one another. Lateral movement between applications in a Kubernetes cluster is easy unless steps are taken to control it.
3. **Container privileges.** Containers may share host namespaces or aspects of the host file system directory. Once a container is compromised, these privileges can help attackers take control of the host and then move to other containers and hosts.

Kubernetes is a powerful API that appeals to attackers.

Kubernetes is more than an orchestration layer. It is a powerful API that enables organizations to control their entire infrastructure in a uniform, cloud-agnostic way. This is very attractive to attackers.

The most famous kind of Kubernetes attack is unsecured API servers that allow hackers to run their own workload (such as crypto mining) in the target cluster. Kubernetes makes it very easy to expose workloads to the Internet. If a service doesn't have strong authentication and authorization, attackers can use it to exfiltrate data or manipulate data.

Secrets are another attractive target for attackers. Kubernetes provisions workload tokens and credentials. If a cluster is compromised, attackers can access a trove of tokens and access sensitive data. Kubernetes also provides containers with runtime privileges. If attackers can run workloads with high privilege, they can eventually escape through the host and access other services outside the cluster.

> Kubernetes isn't just about defining which container to run; it's also about provisioning storage, managing secrets, provisioning the network, and defining access control. This powerful API is very appealing to attackers.
>
> *Haim Helman, Carbon Black App Security*

New security tools are needed for Kubernetes across the application life cycle.

Existing security tools aren't well suited for Kubernetes for two reasons:

1. **The technology stack.** Cloud-native environments are highly dynamic. In addition, containers are standalone. No affinity exists between workloads, IP addresses, or servers. When defining policies or behavior models, a new identity is needed—the workload identity.
2. **Organizational change.** As organizations move to agile and a continuous integration and deployment model, they no longer have the time to conduct security audits. Security tools must become part of continuous integration (CI) and continuous deployment (CD) processes.

The top Kubernetes security use cases include:

- **Posture management.** As organizations consider possible security solutions for cloud-native environments, they must assess the risks associated with workloads and clusters, and then prioritize them.
- **Continuous compliance.** Policies must be defined that prevent additional risks from entering the environment.
- **Threat investigation and correlation.** Teams need tools that perform runtime threat detection.
- **DevSecOps enablement and automation.** When tools are embedded in CICD pipelines, it is easier to identify potential risks early.

As organizations evaluate security tools for Kubernetes, it can be helpful to look at the application life cycle. During the build phase, vulnerability scanning is crucial to analyze images, code, or both. It is also important to harden Kubernetes configurations.

In the deploy phase, teams need tools to enforce the policy, facilitate compliance reporting, and enable visibility and hardening. During the operate phase, threat and anomaly detection and response capabilities are important, as well as event monitoring and zero-trust networking.

Figure 6: Container Security with VMware Carbon Black

BIOGRAPHIES

Sheila A. Berta

Head of Security Research, Dreamlab Technologies

Sheila A. Berta is an offensive security specialist who started at 12 years old by learning on her own. At the age of 15, she wrote her first book about web hacking, published in several countries. Over the years, Sheila has discovered vulnerabilities in popular web applications and software, as well as given courses at universities and private institutes in Argentina. She specializes in offensive techniques, reverse engineering, and exploit writing and is also a developer in ASM (MCU and MPU x86/x64), C/C++, Python, and Golang. As an international speaker, she has spoken at important security conferences such as Black Hat Briefings, DEF CON, HITB, Ekoparty, IEEE ArgenCon, and others. Sheila currently works as Head of Research at Dreamlab Technologies.

Haim Helman

CTO, Carbon Black App Security, VMWare

Haim Helman is CTO, Carbon Black App Security at VMware. He Previously was co-founder at Octarine, now part of the Security BU in VMware. Octarine created a continuous security platform for applications in Kubernetes. He has been leading enterprise infrastructure innovation for the last 20 years. Before Octarine, he co-founded XIV, an enterprise storage company which was acquired by IBM. Following his time at IBM he was a member of the founding team at Robin Systems, where he first encountered the promise and challenges of cloud-native architecture. He holds a BSc in Math, Physics and Computer Science from the Hebrew University in Jerusalem and an MSc in Computer Science from Tel Aviv University.

© 2021 Informa. All rights reserved.