WHITE PAPER–OCTOBER 2021

Container Network Security Primer for Network Security Professionals

Protecting Modern Applications

vmware®

Applications and Workloads
An application is a collection of workloads. A workload provides a specific functionality (for example, database functionality) to the application.

Three types of workloads exist – virtual machines, physical servers, and containers. In the case of virtual machines and physical servers, the term workload includes the operating system on which the functionality runs.

Containers & Containerized Workloads
Containers are software packages that include the workload logic (for example, a database) and everything else that the logic needs to run on an operating system. Recently, containers have become a popular method for packaging software to create workloads.

Containers differ from virtual machines in that containers virtualize the operating system such that multiple workloads can run on a single operating system. In contrast, virtual machines virtualize the physical server such that multiple operating system instances can run on the physical server.

A workload that is packaged and run as a container is called a containerized workload.

Docker
Docker is an open-source project that provides a set of tools to create containers. Docker Inc. is a commercial company that supports the open-source project and sells a commercial version of the Docker toolset.

The Docker toolset runs on Linux, Windows, and macOS. As of this writing, Docker is the most popular means to create containers.

Kubernetes
Kubernetes is an open-source orchestrator for containerized workloads. Kubernetes automates the process of running workloads on a set of physical or virtual machines. As of this writing, Kubernetes is the most popular tool to manage Docker containers.

Introduction

Modern applications are increasingly built using a micro-service architecture. These applications are split up, decoupled from the underlying infrastructure, and delivered as a service in a way that optimizes the application experience. The micro-services model also means that each component can be independently developed and managed—creating operational flexibility and elasticity. Containers are used to deploy each of the micro-services.

Medium and large enterprises typically have a network security team responsible for securing their private and public cloud networks. Such a team also secures the intra-application (east-west) network traffic. When modern applications are deployed, the network security team may be tasked with providing security or assist with securing the modern application.

Network security teams are, of course, fluent in network security terminology and adept at firewall administration. However, often they are not familiar with container terminology or the container-specific mechanisms for securing modern applications at run time.

In this paper, we bridge the gap between traditional network security concepts and network security for containers. Our goal is to provide network security professionals with a conceptual template to secure modern applications.

Note that the sidebar contains an explanation of non-security technical terms used in this paper. See [1] for additional background on these terms.

An example of modern application

Figure 1 shows an example of a modern application that we will use and refer to throughout this paper. The application has four modules – one front-end module (M0) and three additional modules (M1, M2, M3). All user requests are routed to M0, and the results are served back to the user via M0.

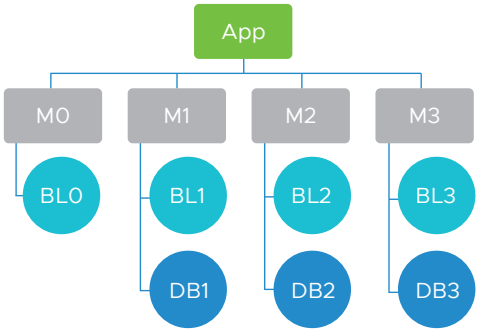

FIGURE 1: Logical Modules in a Modern Application

A real-life e-commerce application can be modeled with these four logical modules. For example, the web front-end would map to M0, product search would map to M1, order processing would map to M2, and payment processing would map to M3. We'll use the abstract modern application to keep the exposition independent of the idiosyncrasies of the specific application.

M1, M2, M3 are identical in structure, even though they implement different functionality. Each of these modules has business logic (BL) and a database (DB). In contrast, M0 only has business logic.

We can think of each type of business logic and database as a micro-service. Thus, there are a total of seven micro-services in our application: one for M0 and two for each of M1, M2, M3.

All business logic can communicate with other business logic. However, only the business logic associated with a module can communicate with the database related to that module. Figure 2 shows the allowed communication patterns at the micro-service level abstraction. Figure 3 shows the communication pattern that we wish to prohibit.

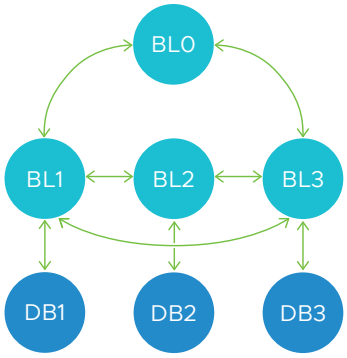

FIGURE 2: Micro-service-level Allowed Communication Pattern

Kubernetes Cluster
A Kubernetes Cluster is a set of physical or virtual machines on which an organization can run containerized workloads. These machines may be deployed in an organization's private cloud (data center) or in the public cloud (e.g., Amazon Web Services).

Kubernetes Namespace
Kubernetes Namespaces are a way to organize Kubernetes clusters into virtual sub-clusters. They can be helpful when different teams or projects share a Kubernetes cluster. Any number of namespaces are supported within a cluster, each logically separated from others but able to communicate with each other. Namespaces cannot be nested within each other.

Kubernetes Pod
Kubernetes Pods are the smallest, most basic deployable objects in Kubernetes. Pods contain one or more containers (usually Docker containers). When a pod runs multiple containers, the containers are managed as a single entity and share the pod's resources, including its IP address for communication outside the pod.

As of this writing, most workloads are designed to have one container per pod.

Container Network Interface
Containers, when initially created, are not "on the network." Network interfaces need to be added to these containers to make them accessible. A Container Network Interface (CNI) inserts the required (virtual) network interface into a container to enable basic networking for the container. A CNI can also include functionality to support network policies and encryption.

Notwithstanding its name, CNI as a concept applies at the pod level in a Kubernetes cluster. A pod with multiple containers has only one CNI. To get traffic in or out of multiple containers in a pod, a CNI implements network address translation since the containers do not have routable IP addresses of their own.

CNIs are also known as Network Plugins.

We will use this application to demonstrate network security concepts as applied to containers.

East-west firewalling for intra-application traffic

Let's assume that we want to deploy the application in a Kubernetes cluster of its own.

We associate a Kubernetes namespace with each micro-service. Kubernetes pods (one for each business logic or database micro-service) map to their respective namespaces. For example, BL1 has a pod P_BL1 in the namespace N_BL1 (see Figure 4).

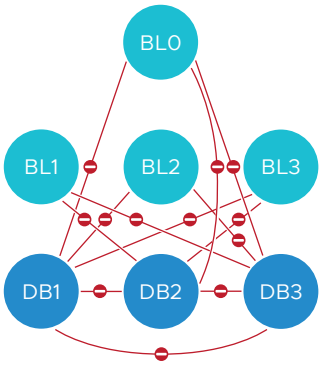

FIGURE 3: Micro-service-level Prohibited Communication Pattern

Within each pod, the business logic or database runs as a container, and each pod houses a CNI. Network security policies for intra-application traffic are enforced at this CNI.

The prevention of communication between two database micro-services (DB1 and DB2, for example) is enforced at the CNI of the database pods. Similarly, allowance of communication between the business logic micro-services (BL1 and BL2, for example) is enforced at the CNI of the business logic pods. This enforcement is the equivalent of having a hypervisor resident distributed firewall for virtualized workloads [2].

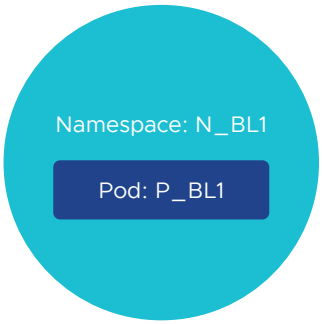

FIGURE 4: Namespace and Pod for BL1

Generally, communication within a namespace if there are multiple pods is allowed by the CNI, and communication between namespaces is prohibited unless the CNI has been configured to let the communication through. This concept is sometimes called "namespace isolation."

Figure 5 illustrates the containers C_BL1 and C_BL2 inside the pods P_BL1 and P_BL2, respectively, with communication permitted between the pods by their respective CNIs.

Figure 6 shows all the namespaces, which map to the micro-services from Figures 2 and 3.

Encryption for intra-application traffic

Now let's say that we want intra-application communication to be encrypted. A service mesh can be employed to enforce service-identity-based TLS encryption.

Note that a service mesh is implemented by injecting a sidecar proxy container within the appropriate pods of the applications. A sidecar proxy takes control of all communication in and out of a pod. The proxy exchanges keys via mTLS with the appropriate pods for other micro-services, encrypts outgoing traffic, and decrypts incoming traffic. Service meshes have other capabilities, such as application-level observability and application-level policy control. We focus on encryption in this paper for the sake of brevity.

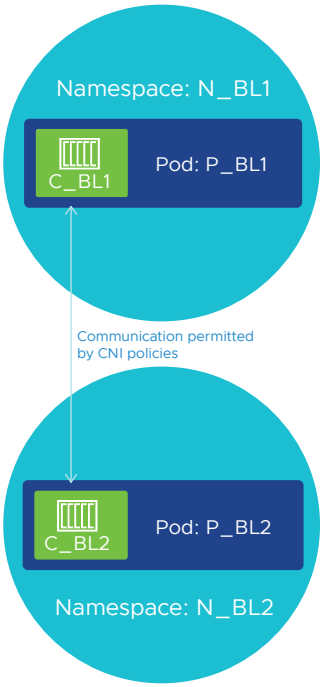

FIGURE 5: Communication Between BL1 and BL2 Permitted by CNI

Network Policies
Network Policy is a Kubernetes concept that specifies the network entities that a pod is allowed to communicate with. Such entities are identified through a combination of the following three identifiers:

- Pods
- Namespaces
- IP address ranges

In other words, network policies are a way to implement access control in a Kubernetes cluster.

Sidecar Proxy
A sidecar container is sometimes injected into a Kubernetes pod to handle specific functionality on behalf of a containerized workload in the same pod. In other words, the sidecar acts as a proxy for the containerized workload. Hence, the name Sidecar Proxy.

Service Mesh
A Service Mesh is an abstraction layer that takes care of service-to-service communications, observability, and resiliency in modern, cloud-native applications. It provides a layer of services (at layers 5 through 7 of the OSI network stack) to the containerized workload in a Kubernetes pod. Service meshes aim to offload certain types of networking logic from the containerized workload. Service meshes are usually implemented as sidecar proxies.

Mutual TLS
Mutual TLS (mTLS) is a method for mutual authentication. mTLS ensures that the parties at each end of a network connection are who they claim to be. Service meshes use mTLS to exchange keys and encrypt communication between micro-services. As discussed above, a service mesh is implemented using a sidecar proxy.

Figure 7 shows a close-up of the containers and pods for BL1 and BL2. Containers C_BL1 and C_BL2 are still in place, and pod-to-pod communication goes through the CNI. However, C_BL1 directs all traffic through the sidecar proxy C_BL1_S. Similarly, C_BL2 directs all traffic through the sidecar proxy C_BL2_S. The sidecar proxies establish the identity of the service endpoint that they are communicating with, exchange keys through mTLS, and encrypt/decrypt traffic.

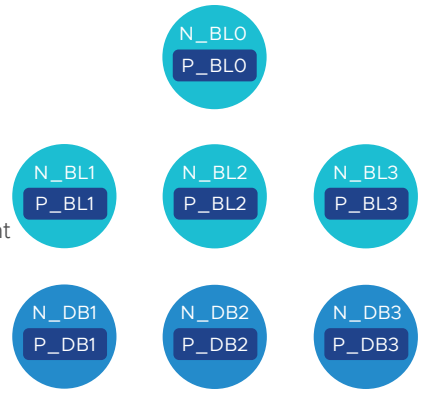

FIGURE 6: All the Namespaces for the Application

Sidecar proxies should be present in all the pods for our seven services to enable encryption across the application. Thus C_BL1_S communicates not just with C_BL2_S but also with C_BL0_S, C_BL3_S, and C_DB1_S.

We should point out that encryption can also be achieved via the CNI – in this case, Figure 5 remains valid, but with encryption for all pod-pod communication.

Since CNI and service mesh operate at different levels of the OSI stack, service mesh encryption is compatible with CNI encryption (for example, you can use the CNI to encrypt pod-to-pod traffic and the service mesh to encrypt service-service traffic for two-levels of encryption).

Load balancing traffic into the application

Kubernetes, CNI, and service mesh allow the creation of multiple instances of any of the seven services discussed in this paper. While a detailed discussion of this horizontal scaling and redundancy is outside the scope of this paper, for illustration purposes, we'll assume that two instances of the user-facing business logic (BL0) have been created.

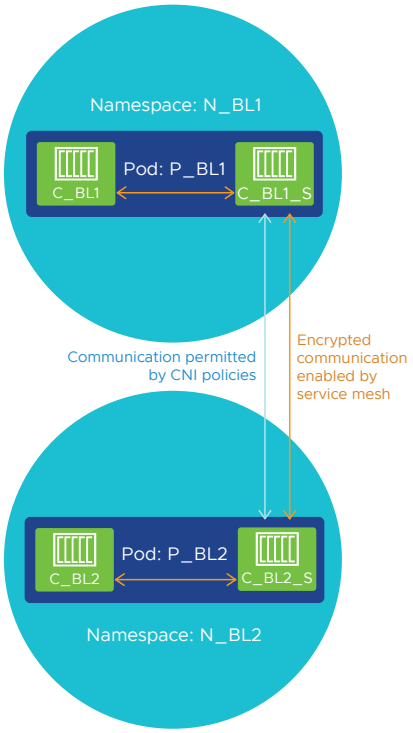

FIGURE 7: Intra-application Traffic Encryption

These two instances of BL0 will run in separate pods. Thus, incoming traffic from the user needs to be load balanced to the two pods. This load balancing can be carried out by Kubernetes' native load balancing primitive – kube-proxy. Kube-proxy exports a virtual IP address to which the user-application traffic is directed. Traffic coming to this virtual IP address is then split between the available pods to service the request. Again, details of the Kubernetes configuration for routing user traffic to the virtual IP address is out of scope for this paper.

Two levels of protection

Let's assume that the application is to be deployed in a private cloud. In this case, the cloud perimeter should be protected by a private cloud zone firewall (also referred to as a data center perimeter firewall [3]). This firewall protects all the applications in the private cloud and not just the modern application that we have been discussing.

Kube-proxy
Kube-proxy is a network proxy that runs on each machine of the Kubernetes cluster. Kube-proxy maintains network rules that allow network communication to Kubernetes pods from network sessions inside or outside the cluster.

Kube-proxy uses the operating system packet filtering layer if one is available. Otherwise, kube-proxy forwards the traffic itself.

Modern applications can use kube-proxy to expose a virtual IP address (called ClusterIP) for load balancing traffic between pods that provide a micro-service.

Thus, our application has two levels of protection for defense in depth (see Figure 8). The first is the protection provided by the private cloud zone firewall for user-application traffic, and the second is the protection offered by the CNI and service-mesh for intra-application traffic.

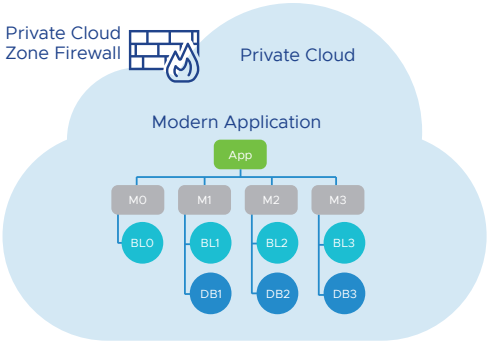

FIGURE 8: Defense in Depth

Other ways of organizing the application

This paper has described a particular way of organizing modern applications and securing them at run time. There are alternative ways of architecting and protecting the application using related organizational and security concepts. We have deliberately focused on a specific micro-service friendly network-security-centric exposition for brevity and clarity.

VMware sponsored CNI and service mesh

Several vendors and open-source projects have created CNIs and service meshes for Kubernetes. One of these vendors is VMware.

Antrea is an open-source project sponsored by VMware that implements a CNI that provides network connectivity and security for Kubernetes pods. Antrea supports network policies and encryption [4].

VMware has also created Tanzu Service Mesh (TSM) – a service mesh that provides connectivity and security for modern applications. Among other capabilities, TSM supports encryption using mTLS [5].

References

[1] Container Security. Liz Rice. O'Reilly. 2020.

[2] Internal Firewalls for Dummies. VMware Special Edition eBook. 2020.

[3] Bringing Software-Based Elasticity to High-Capacity Firewall Use Cases. VMware blog post. 2021.

[4] Antrea. antrea.io. Retrieved 2021.

[5] VMware Tanzu Service Mesh. tanzu.vmware.com/service-mesh. Retrieved 2021.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com Item No: Container Network Security Primer_Whitepaper_JR5 10/21