Juniper Networks | Engineering Simplicity

DAY ONE: BUILDING CONTAINERS WITH cSRX, 2ND EDITION

By Andy Leung and Cindy Zhao

DAY ONE: BUILDING CONTAINERS WITH cSRX, 2ND EDITION

This *Day One* book introduces platform and container concepts and the advantages of using the cSRX to secure them. The authors present the Docker platform, explain how it is different from virtual machine platforms, review its security, and then show how to spin up containers on the Docker platform on different operating systems. The book then provides multiple step-by-step lab guides that show readers how to implement Juniper's cSRX on a server with Ubuntu OS, and how it can protect other containers on the same server. But the book doesn't stop there, it continues by showing how to deploy the cSRX in Kubernetes, how to integrate the cSRX with Juniper's networking management system, Contrail, and how the cSRX can be deployed as a containerized function block to provide advanced security policies, NAT, UTM, and IPS for a Kubernetes network.

"Containers are probably one of the hottest topics in the industry today as more applications are deployed across a vast array of platforms. The authors draw upon their extensive field experience to demonstrate the basics of Docker and containers to show how an application, in this case Juniper's firewall software, can be deployed as a container."

Allan Young, Senior Director, APAC Systems Engineering, Juniper Networks

"As cloud adoption increases combined with technologies such as containers, securing these environments is critical to ensure success of the applications using them. In this new book, Andy and Cindy share their extensive field and hands-on experience to help you secure container workloads and gain visibility into the applications that run in these containers. The book walks the reader from concepts to implementation within a short span of time, yet it is packed with ample code and practical examples. Kudos to the team!"

Mitesh Dalal, Product Management Director, NGFW and Enterprise, Juniper Networks

IT'S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN ABOUT:

- What the cSRX firewall can do and how to deploy it in a production environment.
- How to use Docker and create applications using containers.
- How to deploy the cSRX to protect container applications.

ISBN 978-1941441985

Day One: Building Containers with cSRX, Second Edition

by Andy Leung and Cindy Zhao

JUNIPER NETWORKS

Published by Juniper Networks Books
Authors: Andy Leung, Cindy Zhao
Technical Reviewers: Mithun Hebbar, Jacky Tsui, Xingming Wang & Bajan Anu
Editor in Chief: Patrick Ames
Copyeditor: Nancy Koerbel

Printed in the USA by Vervante Corporation

Version History: v1 2nd Editon, Nov. 2020
2 3 4 5 6 7 8 9 10

http://www.juniper.net/dayone

About the Authors
Andy Leung is an APAC Systems Engineering Director based in Hong Kong at Juniper Networks. He has over 20 years of experience in computer networking and security. He worked for Sun Microsystems, IBM, and Netscreen Technologies before joining Juniper. He is a CISSP.

Cindy Zhao is an APAC Systems Engineer based in Hong Kong at Juniper Networks. She has over 5 years' experience in automation and security. She is JNCIP-SEC and JNCIS-Cloud.

Authors' Acknowledgments
The authors would like to thank Patrick Ames for the opportunity to write this book, and for his guidance on writing for the Day One series. Also, we would like to thank our manager, Allan Young, who encourages innovation and supports us taking on new projects like working on this book. Great thanks to Raymond Lam and Nancy Yu for providing demo ideas, and Laurent Paumelle, Phil Chen, and Grace Li for sharing their knowledge and experience on cSRX. Also our appreciation to Mithun Hebbar, Jacky Tsui, Xingming Wang, and Bajan Anu for their detailed technical reviews. Andy would also like to thank his family for their support, especially his wife Yoyo and daughter Sophie.

Welcome to Day One

This book is part of the *Day One* library, produced and published by Juniper Networks Books. *Day One* books cover the Junos OS and Juniper Networks network-administration with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow. You can obtain the books from various sources:

- Download a free PDF edition at http://www.juniper.net/dayone
- PDF books are available on the Juniper app: Junos Genius
- Purchase the paper edition at Vervante Corporation (www.vervante.com) for between $15-$40, depending on page length

Key cSRX Resources

Readers can download the configuration examples in this book from its GitHub repository: https://github.com/csrx-dayone .

The Juniper TechLibrary supports the cSRX with its excellent documentation. This book is not a substitute for that body of work, so you should take the time to review the documentation here: https://www.juniper.net/documentation/product/en_US/csrx.

The Juniper Networks product page for the cSRX, with data sheets, specs, and overviews can be found here: https://www.juniper.net/us/en/products-services/security/srx-series/csrx/.

What You Need to Know Before Reading This Book

Before reading this book, you should be familiar with the basics of computers and networking, and have some concept of virtualization. No prior knowledge of using Docker is necessary, but the book discusses existing threats of using the Docker hypervisor and how to mitigate those threats using the cSRX firewall. You should also be familiar with general troubleshooting techniques for ISP networks running the Junos OS.

What You Will Learn by Reading This Book

- What the cSRX firewall can do and how to deploy it in a production environment.
- How to use Docker and create applications using containers.
- How to deploy the cSRX to protect container applications.

About This Book

NOTE Example source code in this book has been uploaded to a GitHub cSRX DayOne Book account repository, and users can download the code according to instructions provided in the demo examples.

This book has six chapters:

- Chapter 1 introduces platform and container concepts, and the advantages of the cSRX.
- Chapter 2 mainly presents the Docker platform, how it is different from virtual machine platforms, and its security. It includes a hands-on description about how to install Docker on different operating systems, how to set up networks, and how to spin up containers on the Docker platform.
- Chapters 3 and 4 provide multiple step-by-step lab guides that show users how to implement Juniper's containerized SRX on a server with Ubuntu OS, and how it can protect other containers on the same server. Chapter 4 contains some specific techniques, including how to configure the cSRX at launch and how to install IDP on the cSRX.
- Chapter 5 introduces how to deploy the cSRX in Kubernetes, a container-orchestration system for automating application deployment, scaling, and management. Integrating with Juniper's networking management system Contrail, the cSRX can be deployed as a containerized function block to provide advanced security policies, NAT, UTM, and IPS for a Kubernetes networkAlso it demonstrates the host-based firewall feature of Contrail with the cSRX.
- Chapter 6 demonstrates how to implement the cSRX in a Kubernetes environment without deploying Contrail. With the lightweight Multus plugin, cSRX can provide workload protection in DevOps environments.
- Chapter 7 offers some troubleshooting tips and considerations.
- The Appendix has a table comparing the cSRX to the vSRX, plus a few references and links for more support and information.

Chapter 1

Container Technology

One of the pressing problems facing enterprises today is the sheer amount of applications to develop and deploy. The types of applications can range from web applications, web frontends, database querying, system analytics, and on and on. And, these applications are also made up of different components including libraries, microservices, cloud services, and more.

Another current trend is that applications are now being deployed to more places. Traditionally, applications would be hosted and running on a standalone server, either inside the enterprise or hosted in a data center. Nowadays, developers have been asked to deploy their applications to multiple premises, including DCs, private cloud, public cloud, edge devices, and even user devices.

To reiterate: there are now more applications to develop and more places to deploy. So the challenge now is how can we package these applications more efficiently so that they can be shipped to different platforms and deployed seamlessly? Container technology was introduced to solve this problem.

Container Overview

This new logical packaging method is called *container* because of its analogy to a shipping container. Think about how goods are transported around the world today – no matter whether it's furniture, food, cars, or equipment – they are put into shipping containers as cargo. These containers are then loaded on a cargo ship and they travel to another part of the world. Upon arrival, they will be loaded onto trucks and delivered to a destination. The advantage of the container is that people don't need to create a separate packaging method for a car, a couch, or a coffee maker. Containers are now a standard way to move different types of goods around the world.

The shipping container concept readily applies to software containers. The *Docker* container is a technology for developers to package their programs and transport them to different platforms and support them to execute in different environments.

Consider the traditional way to develop a program and deploy the software to different platforms, which could be Windows, or different flavors of Linux. After the developer creates an application, they need to write a build script depending on the target platform, and also a custom deployment script for the infrastructure. There is a lot work to customize the build and deploy processes to make the software work on different platforms.

Using container technology, developers are now able to abstract the software away from the infrastructure. There will be a container platform running on the target platform, which can be on different operating systems, or even a cloud infrastructure. The container software, which has already packaged all the necessary components to execute the application, including all the programming code, binaries, and dependencies, will be seamlessly deployed on a container platform which allows the application to behave in the same way on various operating systems.

Virtual Machines Versus Containers

In the past ten to fifteen years, virtual machine (VM) technology has become very popular. There are also different hypervisors for VM like VMware, KVM, and VirtualBox, to name a few. Many people like to describe Docker as a lightweight VM, and while the comparison allows an easy understanding of the container technology, it is not entirely true.

From the official Docker's website, there is a good analogy that says if VM is like a house, then Docker (container) is like an apartment. For example, if you own a house (VM), you will be protected from unwanted guests. A house would include a bedroom, kitchen, and bathroom. You will be having your own infrastructure like electrical systems, heating, and plumbing independently. While you live there, you can manage these utilities for yourself.

Meanwhile, if you live in an apartment (container), you will also enjoy the protection from unwanted guests, and it could be an apartment with different rooms or a studio. However, the tenants need to share the common infrastructure inside the same building. And the facilities mentioned above—like electricity, gas, and plumbing—would need to be shared among all the apartments. You can see this difference in Figure 1.1.

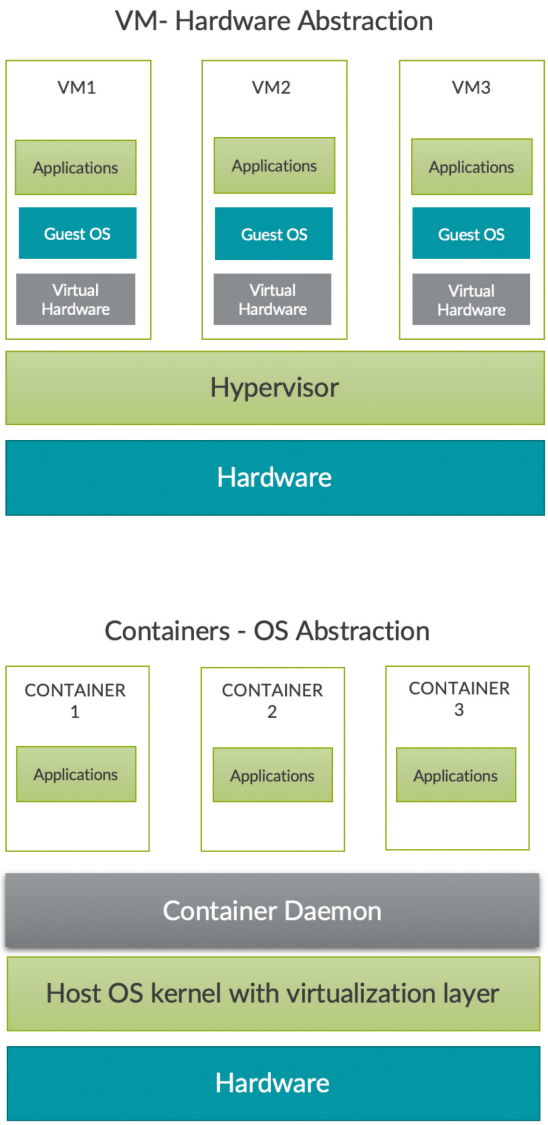

Figure 1.1 *Virtual Machines Versus Containers*

Technically, you can view both VM and Docker as different ways to package an application that can easily be transferred and migrated to another system. A VM would include the application, its libraries, and dependencies, as well as the underlying operating system and networking stack. It can record and store the stateful information where the application is running. To run a VM, the system would need to install a hypervisor layer, which can be running on a bare metal server, likeVMware's ESXi. Or, the hypervisor can also run like an application on top of an operating system, like VMware workstation, or VirtualBox.

On the other hand, a container only needs to package the application's code, binary, and dependencies. There is no operating system or networking stack inside a container. To run a container application, the system would need to support the Docker daemon, which is available in most of the common operating systems or cloud platforms today.

There are a few major differences between a VM and container. First, the size of the container image is smaller than a VM, since the container doesn't need to package the OS and networking components. Second, the launch time of a container application will be shorter as it doesn't need to wait for the all the OS components to boot up. Docker provides developers an efficient solution to package up the application and distribute it to different platforms. However, for developers who want to have a clearer separation of resources, authority, and rights, VM proves to be a better solution for that separation.

Docker's Built-in Security

Similar to today's hypervisors, like VMware or KVM, Docker has built-in security to protect container development and runtime. Docker is made up of layers of containerized stacks that have their own namespaces. Docker also enables developers to build security in their container software and integrate it directly into its software supply chain. There is a chain of custody to maintain the integrity of container images. Developers can also apply governance and control when a container is deployed in production.

A container or a container image is a layered root file system that is packaged and distributed. There is no core part of the operation system like a kernel or networking stack. Within a container, there are layers of a file system that make up an application. When a container image is broken up into separated container layers, those layers are read-only. Also, the original layer that is distributed on the host system is locked and read-only.

There is a read-write layer at the very top of the container image so the user can mount an external file system for data storage. However, every time the user starts a container image, it will start up in exactly the same way. Application code is stored in the read-only layer so it won't be affected by changes at the top layer. If there is any new modification to the application, it will be stored at the top,

temporary layer, which will be thrown away when a new container is started. In summary, application code will only be stored on the read-only application layer, which will not be permitted to change.

The Docker runtime, core engine, and containers are secure by default. There are two main concepts: *namespace* and *control group*. The namespace concept is similar to what it is in Linux OS today, which controls what a process can see and access. It lives inside the operating systems, and isolates what a container can see. The container file system is not likely to see things from the hosts unless they're mounted, because they are in separated namespaces. This applies to the networking stack, user, and user permission as well. On the other hand, a control group isolates what kind of resources the container can use. For example, the amount of system memory it can use.

A container is by default *hardened*, which limits what kind of system calls it can make. There is a whitelist of system calls, while a majority of the malicious system calls have been disabled. It is also viable to apply system control to limit what a container can do on the host file system. For example, mounting a file system into the container, and what it can connect to. This way the processes running in the container will be much safer.

Today, developers can also sign the container with its private key before publishing the image or sharing it. So users can verify the digital signature before downloading the container image. Administrators in the enterprise can also lock down a group of publishers, sources, or users. These procedures all help users to access and move the container images around securely. However, Docker's default security doesn't extend to networking and applications: this is where the cSRX comes into the picture.

The cSRX Overview

A firewall is an integral part of network and application security when implementing an enterprise or data center network. In the virtual environment, there are firewall solutions to protect applications running in hypervisors like Hyper-V, KVM, and VMware. And in cloud environments like AWS, Azure, or Google Cloud, deploying Juniper's vSRX firewall can protect cloud-native applications. Meanwhile, the cSRX is a purpose-built container firewall that protects Docker containerized environments with advanced security services, including content security, intrusion prevention systems (IPS), application security (AppSecure), and unified threat management (UTM).

Using cSRX containerized firewalls, you can gain additional visibility into securing applications that run in your containers, like application statistics, bandwidth, and other usage information. The cSRX can work with IP addresses and enterprise directories like Active Directory, and you can use it to deploy security policies that limit access to specific applications.

There are popular applications like Apache, Wordpress, SQL, and many more, that have introduced vulnerabilities upon release. These vulnerabilities could be exploited the same way even though these applications have become containerized. There is an advantage to applying the cSRX's IPS to protect applications that aren't easily patched. To control and protect applications from east-west traffic, the cSRX supports micro-segmentation with security policies from Layer 3 to Layer 7, limiting traffic between the different container segments.

In enterprise multicloud environments, the cSRX pairs with Contrail Enterprise or Multicloud Enabler to deliver micro segmentation, encryption, and security at Layers 3 to 7 for cloud-native applications.

For service providers, the cSRX can be deployed as part of a service function chain or on a per-application basis with Contrail Edge Cloud, and integrated with container orchestrators like Kubernetes. These deployments enable features such as Network Address Translation (NAT), firewall, and IPS on a single cSRX or through multiple cSRX instantiations to improve the scale and agility of cloud network environments. The cSRX can be deployed in both private and public clouds to secure mission critical, cloud-native applications from known and unknown threats in container environments.

Some of the key benefits of the cSRX in a containerized private or public cloud multi-tenant environment include:

- Stateful firewall protection at the tenant edge.
- Faster deployment of containerized firewall services into new sites.
- A small footprint and minimum resource reservation requirements, so the cSRX can easily scale to keep up with peak customer demand.
- Significantly higher density without requiring resource reservation on the host than what is offered by VM-based firewall solutions.
- Flexibility to run on a bare-metal Linux server, a VM, or Juniper Contrail.
    - In the Contrail Networking cloud platform, the cSRX can be used to provide differentiated Layer 3 through 7 security services for multiple tenants as part of a service chain.
    - With the Contrail Orchestrator, the cSRX can be deployed as a large-scale security service.
- Application security features (including IPS and AppSecure).
- UTM content security features (including antispam, Sophos Antivirus, web filtering, and content filtering).
- Authentication and integrated user firewall features.

Chapter 2

Setting Up the Demo Environment

This chapter will show you how to install Docker on different operating systems, how to set up docker networks, and how to spin up containers on the Docker platform. Let's make some containers.

Docker Installation and Verification

MacOS System Requirements

Your Mac hardware must be a 2010 or newer model, with Intel's hardware support for memory management unit (MMU) virtualization, including Extended Page Tables (EPT) and Unrestricted Mode. You can check to see if your machine has this support by running the following command in the terminal:

```
$ sysctl kern.hv_support
kern.hv_support: 1
```

MacOS El Capitan 10.11 and newer macOS releases are supported. At a minimum, Docker for Mac requires macOS Yosemite 10.10.3 or newer, with the caveat that going forward 10.10.x is a use-at-your-own risk proposition. It's recommended to upgrade to the latest version of macOS.

You need at least 4GB of RAM.

NOTE Do not install VirtualBox prior to version 4.3.30 (it is incompatible with Docker for Mac). If you have a newer version of VirtualBox installed, that's fine.

Download and Install Docker for MacOS

Docker has prepackaged all necessary parts including the Docker Engine, Docker CLI client, Docker Compose, Docker Machine, and Kitematic in one Mac installer (.dmg file) and you can download it from the Docker official website here: https://docs.docker.com/docker-for-mac/install/. Once downloaded you'll get the Docker Desktop shown in Figure 2.1.

Figure 2.1 *Download Docker Desktop*

NOTE In order to download the Docker Desktop installation file, you have to get a DockerHub account. Registration of the DockerHub account is free with a valid email address.

Okay, once downloaded, install Docker Desktop:

1. Open the Docker.dmg and do "Drag and Drop" to copy contents to applications. A new Docker icon will appear in the Launchpad.

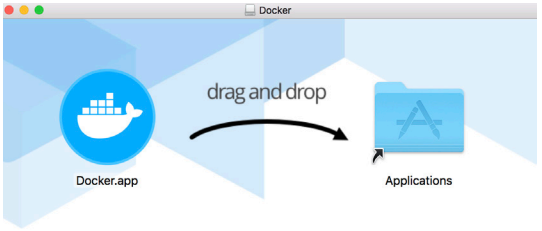

Figure 2.2 *Copy Docker.app to MacOS Applications*

2. Open Launchpad and run the new Docker icon. You'll get a confirmation message. Choose "Open." Another message says: "Docker Desktop needs privileged access." Click "OK" and enter the password for your macOS.

3. You'll see a new whale with container's animation on your macOS status bar. Docker service is starting. When the animation stops, Docker Desktop has launched.

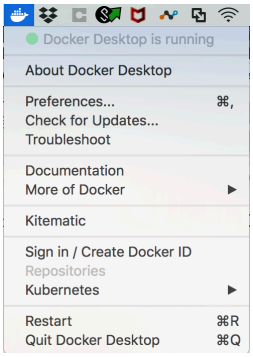

Figure 2.3 *Docker Is Running on MacOS*

Windows 10 System Requirements

You need Windows 10, 64bit: Pro, Enterprise, or Education (1607 Anniversary Update, Build 14393 or later).

Virtualization should be enabled in BIOS. Typically, virtualization is enabled by default. This information can be checked by opening the Task Manager and clicking on the Performance Tab (Figure 2.4).

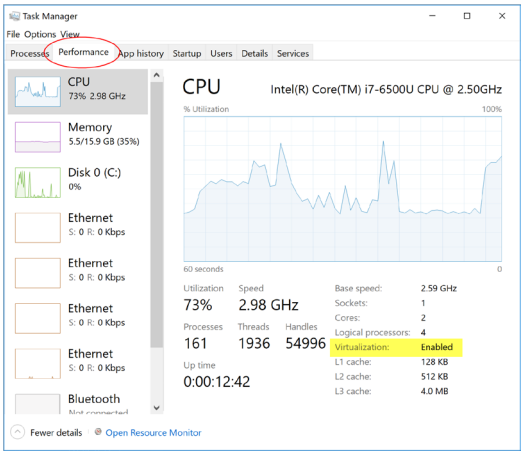

Figure 2.4 *Checking to See if Virtualization is Enabled*

The CPU SLAT-capable feature needs to be enabled. A tool called Coreinfo, which helps check if this feature is available, can be downloaded at https://docs.microsoft.com/en-us/sysinternals/downloads/coreinfo. Run the command prompt as administrator then execute "coreinfo.exe -v", and the SLAT support information will show up.

And you'll need at least 4GB of RAM.

Download and Install Docker for Windows

Docker has prepackaged all necessary parts including Docker Engine, Docker CLI client, Docker Compose, Docker Machine, and Kitematic in one Windows installer (.exe file) and you can download it from the Docker official website through https://docs.docker.com/docker-for-windows/install/.

NOTE In order to download the Docker Desktop installation file, you need to log in to your Dockerhub account. Registration of a new Dockerhub account is free with a valid email address.

After the Docker installer has been downloaded, you can run it directly. Follow the wizard to accept the license, authorize the installer, and proceed with the install. After installation has succeeded, you need to log out of Windows to complete installation. After logging back into Windows, if your computer does not have Hyper-V enabled, Docker will do it for you. All you need to do is to click Ok and restart your Windows machine. See Figure 2.5.

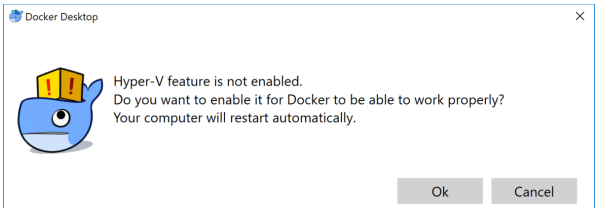

Figure 2.5 *Hyper-V Enablement*

Docker does not start automatically after installation. To start it, search for Docker, select Docker Desktop for Windows in the search results, and click it (or hit Enter). When the whale animation in the status bar stops (Figure 2.6), Docker is up and running and accessible from any terminal window.

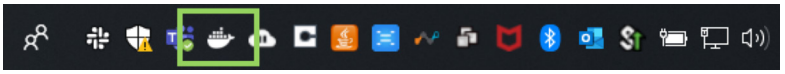

Figure 2.6 *Up and Running Docker on Windows*

Ubuntu System Requirements

To install Docker CE, you need the 64-bit version of one of these Ubuntu versions:

- Disco 19.04
- Cosmic 18.10
- Bionic 18.04 (LTS)
- Xenial 16.04 (LTS)

Docker CE is supported on x86_64 (or amd64), armhf, arm64, s390x (IBM Z), and ppc64le (IBM Power) architectures.

Install Docker on Ubuntu

To start, uninstall older versions of Docker called docker, docker.io, or docker-engine. This can be done by running:

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

Now set up the repository:

1. Update the apt package index:

```
$ sudo apt-get update
```

2. Install packages to allow apt to use a repository over HTTPS:

```
$ sudo apt-get install \
> apt-transport-https \
> ca-certificates \
> curl \
> gnupg-agent \
> software-properties-common
```

3. Add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
OK
```

Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, by searching for the last eight characters of the fingerprint:

```
$ sudo apt-key fingerprint 0EBFCD88

pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid       [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

4. Use the following command to set up the stable repository. For x86_64/amd64, you can run:

```
$ sudo add-apt-repository \
> "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
> $(lsb_release -cs) \
> stable"
```

Now install Docker CE.

1. Update the apt package index:

```
$ sudo apt-get update
```

2. Install the latest version of Docker CE and containerd:

```
$ sudo apt-get install -y docker-ce docker-ce-cli containerd.io
```

MORE? For other Linux distributions Docker installations, check Docker documentation at:

- CentOS: https://docs.docker.com/install/linux/docker-ce/centos/
- Debian: https://docs.docker.com/install/linux/docker-ce/debian/
- Fedora: https://docs.docker.com/install/linux/docker-ce/fedora/

Verify a Successful Installation

After installing Docker on any OS, the `docker -v` command in the command line can check the current docker version. Further, to ensure that docker service is running normally, run a `hello-world` container and a `Hello` message can be displayed correctly at the terminal:

```
# docker -v
Docker version 19.03.2, build 6a30dfc

# docker run hello-world
Unable to find image /hello-world:latest/ locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:b8ba256769a0ac28dd126d584e0a2011cd2877f3f76e093a7ae560f2a5301c00
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
3. The Docker daemon created a new container from that image that runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

MORE? Share images, automate workflows, and more with a free Docker ID available at: https://hub.docker.com/. For more examples and ideas, visit: https://docs.docker.com/get-started/.

To list container images:

```
# docker images
REPOSITORY        TAG              IMAGE ID          CREATED           SIZE
hello-world       latest           fce289e99eb9      8 months ago      1.84kB
```

To list running containers:

```
# docker ps -a
CONTAINER ID  IMAGE       COMMAND    CREATED       STATUS                  PORTS  ...
ca7237e2dd82  hello-world  "/hello"   2 minutes ago  Exited (0) 2 minutes ago        ...
```

To clean up, stop and remove the hello-world container using container ID:

```
# docker stop ca7237e2dd82
ca7237e2dd82
root@host:~# docker rm ca7237e2dd82
ca7237e2dd82
```

Then remove the hello-world image:

```
# docker rmi hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:b8ba256769a0ac28dd126d584e0a2011cd2877f3f76e093a7ae560f2a5301c00
Deleted: sha256:fce289e99eb9bca977dae136fbe2a82b6b7d4c372474c9235adc1741675f587e
Deleted: sha256:af0b15c8625bb1938f1d7b17081031f649fd14e6b233688eea3c5483994a66a3
```

Basic Docker Commands

Here is a list of basic Docker commands for daily operation.

To show the Docker version in details, run:

```
# docker version
```

Image Commands

To list all Docker images:

```
# docker images
```

To create an image from a Docker file:

```
# docker build [OPTIONS] PATH | URL | -
```

To create an image from a .img or a .tar file:

```
# docker load [OPTIONS] PATH
```

To remove an image:

```
# docker rmi [OPTIONS] IMAGE [IMAGE...]
```

To tag an image to a name:

```
# docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

Container-Related Commands

To create a container but not to start it:

```
# docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

To rename a Docker container:

```
# docker rename OLD_NAME NEW_NAME
```

To create a container and to run a command with it:

```
# docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

NOTE The container will stop after the command has finished command execution by default. For interactive processes (like a shell), you must use `-i -t` together in order to allocate a tty for the container process. To start a container in detached mode, option "`-d`" should be used. Detached mode means the container starts up and runs in the background.

To list all Docker containers:

```
# docker ps -a
```

To stop a container:

```
# docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

To delete a container:

```
# docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Network-Related Commands

To create a network:

```
# docker network create [OPTIONS] NETWORK
```

To list all networks:

```
# docker network ls
```

To check the details of a network:

```
# docker network inspect [OPTIONS] NETWORK [NETWORK...]
```

To connect a container to a network:

```
# docker network connect [OPTIONS] NETWORK CONTAINER
```

To disconnect a container from a network:

```
# docker network disconnect [OPTIONS] NETWORK CONTAINER
```

To remove a network:

```
# docker network rm NETWORK [NETWORK...]
```

NOTE A network cannot be removed with a container attached to it. Before removing a network, please make sure all containers attached to it have been removed.

Information Collection

To show Docker container summary info:

```
# docker inspect [OPTIONS] NAME|ID [NAME|ID...]
```

To show container run logs:

```
# docker logs [OPTIONS] CONTAINER
```

To show container port exposure and mapping:

```
# docker port CONTAINER [PRIVATE_PORT[/PROTO]]
```

Docker Networking Essentials

One of the reasons Docker containers and services are so powerful is that you can connect them to non-Docker workloads using Docker Networks. Also networks can be configured to provide complete isolation for containers, which enables building applications that work together securely.

Docker has developed a simple networking model called *container network model* (CNM), which defines how different containers can connect together, while simplifying network implementation methods. The portability comes from CNM's powerful network drivers, pluggable interfaces for the Docker Engine, Swarm, and UCP (Docker Universal Control Plane) that provide special capabilities like multi-host networking, network layer encryption, and service discovery. These are drivers that exist by default and provide core networking functionality:

- *bridge*: The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
- *host*: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. `Host` is only available for swarm services on Docker 17.06 and higher.
- *none*: For this container, disable all networking. It is usually used in conjunction with a custom network driver. `None` is not available for swarm services.
- *macvlan*: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the `macvlan` driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.
- *network plugins*: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for how to install and use a given network plugin.

Docker Networking Architecture

On Docker, users can create as many networks as needed and attach containers to them. Each network contains a gateway, and the gateway lies on the physical interface that reaches out to the Internet if your network driver is bridge. Physical interface eth0 also works as a router, looking up its route tables to forward the Docker networks' packets to the Internet. Containers on different networks cannot reach each other unless there is a container that has two IP addresses on both networks to the Internet as shown in Figure 2.7.

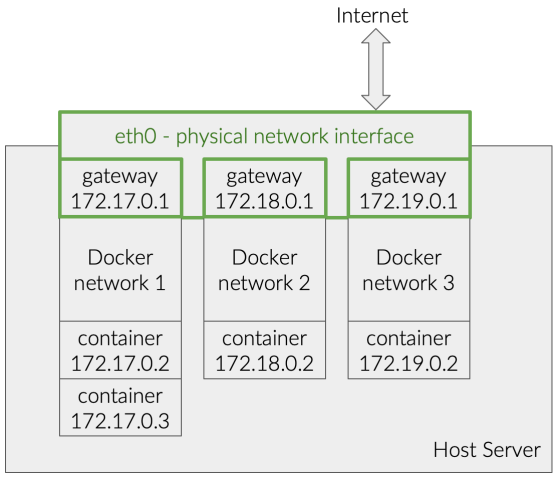

Figure 2.7 *Docker Networking Architecture*

Setting Up a Sample Container Application

After installing the Docker platform on a host, networks can be configured and containers can run on top of them. Below is an example on how to configure networks and hosts on Docker. From this point up through Chapter 4 the examples will all be running on Ubuntu 16.04.6 LTS.

First, create a sample network:

```
# docker network create --driver bridge sample-net
60e9bedb76c037884f358e36267664e3119d9cd06e5b5500cea412bbf7b8ea82
```

Verify what subnet Docker has assigned this new network to:

```
# docker network inspect sample-net | grep Subnet
                "Subnet": "172.18.0.0/16",
```

The subnet for newly created networks can also be optionally defined by adding the `--subnet` parameter:

```
# docker network create --driver bridge --subnet 172.18.0.0/16 sample-net
0e868db4838375ffaa9568f6162091d5f60bdcd86f632263f6359feb86593038
```

Then a container can run on this `sample-net`. There is an image called *Alpine*, which is a small-sized Linux. Although the newly installed Docker does not have any image yet, it will search DockerHub's online repository and download the Alpine image automatically:

```
# docker run -it --network sample-net --name alpine1 alpine:latest /bin/sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
bdf0201b3a05: Pull complete
Digest: sha256:28ef97b8686a0b5399129e9b763d5b7e5ff03576aa5580d6f4182a49c5fe1913
Status: Downloaded newer image for alpine:latest
/ #
```

The `/ #` indicates the user is able to control the shell of the Alpine container directly. The current IP address and route table of the Alpine container can be checked:

```
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
20: eth0@if21: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
   link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
   inet 172.18.0.2/16 brd 172.18.255.255 scope global eth0
      valid_lft forever preferred_lft forever
/ # ip route
default via 172.18.0.1 dev eth0
172.18.0.0/16 dev eth0 scope link  src 172.18.0.2
/ # traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 46 byte packets
172.18.0.1 (172.18.0.1)  0.015 ms  0.007 ms  0.005 ms
...
```

You can see in the output that the current alpine container's IP address is 172.18.0.2, and traffic will automatically route to its gateway 172.18.0.1 for all outgoing traffic, then reach the Internet. With another terminal window on Ubuntu Host, let's ping the Alpine container. Here's the generated response traffic:

```
# ping 172.18.0.2
PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
64 bytes from 172.18.0.2: icmp_seq=1 ttl=64 time=0.059 ms
64 bytes from 172.18.0.2: icmp_seq=2 ttl=64 time=0.063 ms
...
```

Chapter 3

Running cSRX

The cSRX Container Firewall is a containerized version of the SRX Series Services Gateway. The cSRX provides advanced security services, including content security, AppSecure, and Unified Threat Management (UTM) in a container. By using Docker container in a server, there will be fewer computing resources and less memory consumed than with a virtual SRX or physical gateway. Furthermore, cSRX is also capable of service chaining on Contrail Networking and securing users' virtual networks.

The cSRX Architecture

The cSRX is built on the Junos OS and delivers networking and security features similar to those available on the software releases for the SRX Series. As Figure 3.1 shows, the cSRX container packages all of the dependent processes (daemons) and libraries to support itself to run on Docker. Some daemons are Linux daemons for *sshd* and *rsyslogd* instances. Other daemons are imported from Junos OS that perform configuration and control jobs. Their tasks are similar to routing engines on a Junos device. *Srxpfe*, similar to the packet forward engine on an SRX, is the data plane daemon that receives and sends packets from the revenue ports of a cSRX container.

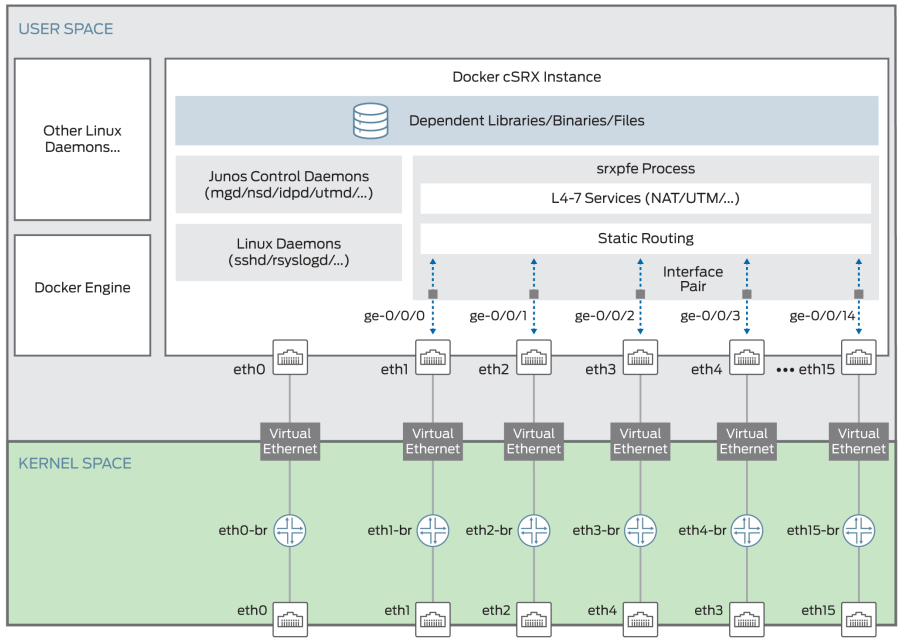

Figure 3.1 *The cSRX Architecture*

Starting from 19.2R1.8, each cSRX can be configured with up to 15 revenue interfaces: eth1, eth2, and so on, until eth15. The number of interfaces can be predefined while booting up a cSRX. Eth0, mapped to fxp0 on cSRX, is for out-of-band management. The other interfaces, eth1, eth2, etc. are mapped to ge-0/0/0 and ge-0/0/1, etc., respectively, on the cSRX, for outgoing or incoming traffic. All traffic interfaces should be connected to a docker network to make sure the cSRX is running properly.

Starting from Junos 20.2 onwards, there is only one size (Large) supported on cSRX. 2 vCPU will be used. Physical memory overhead is 4GB and number of flow sessions is 512K.

For a single cSRX (20.3R1) deployed on a Bare Metal Server, with DPDK enabled, the performance is:

- - Application Firewall with HTTP traffic throughput with 44 kB bi-directional data traffic with AppFW policy with ASC enabled is 5.4 Gbps.
- - HTTP throughput with 44 kB bi-directional data traffic with Sophos Anti-Virus, cache enabled by default is 1.2 Gbps.

- - Raw UDP throughput with IMIX having 10K sessions is 3.1 Gbps.
- - Raw UDP throughput with 1518 Byte frame size having 10K sessions is 12.3 Gbps.

The cSRX Compared with

Just as the cSRX performs network security for containers, Juniper virtualized SRX Firewall (vSRX) serves a similar function for virtual machine networks. However, they are different from each other in architecture, use cases, and functions:

1. The vSRX concentrates on integrated routing, NAT, VPN, and tends to be high performance. While the cSRX concentrates on Layer 3 to Layer 7 security, it tends to have a low footprint.
2. The minimum size vSRX requires 2 vCPU reservation. Larger vSRXs will need 5, 9, or 17 vCPU reservation, while for cSRX, there are no requirements for CPU reservation, and 2 vCPUs are necessary.
3. The SRX generally consumes more RAM. The vSRX needs 4GB memory minimum, while the cSRX needs 4GB for all implementations.
4. Bootup time for the vSRX is several minutes, while for the cSRX it's one second.
5. Disk size of a running vSRX will be above 16 GB, and the cSRX is only about 500 MB.
6. For functions, the cSRX is not capable of dynamic routing or IPsec VPN. vSRX supports both.
7. For host running vSRX, KVM, or VMWare Hypervisor should be supported. However on the cSRX, a container platform such as Docker or Kubernetes needs to be installed.

Setting Up the cSRX

The host Linux server for the cSRX requires the following:

- Linux OS CentOS 6.5 or later; or Red Hat Enterprise Linux (RHEL) 7.0 or later; or Ubuntu 14.04.2 or later
- Docker Engine 1.9 or later
- At least 2 vCPU
- At least 8 GB Memory
- At least 40 GB hard drive
- Processor type is x86_64 multicore CPU
- 1 Ethernet port

MORE? The official cSRX product documentation is located here: https://www.juniper.net/documentation/product/en_US/csrx. This page has the most current cSRX product specific information. You should always refer to the official site regarding product update and download information.

At the time this this book was published, the cSRX image could be obtained from the Juniper's standard download page. Also, a valid license is necessary for the cSRX for basic firewall functions as well as NGFW features. User could download a trial license from Juniper official website: https://www.juniper.net/us/en/dm/csrx-trial/. For acquiring licenses for production environment, please contact a Juniper representative.

Prepare the cSRX Image

Let's install the cSRX on a server running Ubuntu 16.04.6 LTS. There are only two steps for you to load the cSRX image on your machine: download the cSRX image, then import it.

Download the cSRX image.

On a browser, navigate to https://support.juniper.net/support/downloads/, and type "cSRX" in the textbox under "Find a Product". The page will update itself and show up cSRX download resource with the latest version. Img format can be imported directly. Click on the "img" link and you will be redirected to a terms and conditions page. After choosing "I Agree" and "Next" the URL for downloading this image will be given. Copy the URL.

On your Ubuntu server, run this command to get the image:

```
# wget (paste the URL)
```

To check if the file has been downloaded to your server, run:

```
# ls
junos-csrx-docker-20.3R1.8.img?SM_USER=…
```

The name of the file is very long, and we can change it to a shorter one:

```
# mv junos-csrx-docker-20.3R1.8.img?SM_USER=… cSRX.image
```

Load the image:

After the cSRX.img file has been downloaded, it is possible to save it to server's Docker image list in one step:

```
# docker load -i cSRX.img
267b1c98c501: Loading layer [==================================================>]    263MB/263MB
23f7a9961879: Loading layer [==================================================>]  14.51MB/14.51MB
0ab4de24a792: Loading layer [==================================================>]  288.2MB/288.2MB
cca4933366a1: Loading layer [==================================================>]  1.118MB/1.118MB
7c2c66920a77: Loading layer [==================================================>]   16.9kB/16.9kB
5c7ad558802e: Loading layer [==================================================>]   2.56kB/2.56kB
bc4a16173327: Loading layer [==================================================>]  1.536kB/1.536kB
3d78f6c738de: Loading layer [==================================================>]  2.048kB/2.048kB
Loaded image: csrx:20.3R1.8

# docker images
REPOSITORY TAG         IMAGE ID           CREATED            SIZE
csrx 20.3R1.8          8d6e3daa602f       3 weeks ago        553MB
```

Running the cSRX

To start the cSRX, it is necessary to create three networks in the Docker network: a management network, a left network, and a right network. For networks that are not for management, IP masquerading is recommended. If IP masquerading is not enabled, NAT function on cSRX will fail. The option `--subnet` specifies the IP range of this subnet:

```
# docker network create mgt_bridge --subnet 172.18.0.0/24
c928591e6f0d2e48834c76ab1c6ad3ed3d4c37372595af7d92f559a3e208ca22
# docker network create -o com.docker.network.bridge.enable_ip_masquerade=true \
> left_bridge --subnet 172.19.0.0/24
711f37d94fdc4c4a29eeb2696a6e342bb2de188ef63ca953245e24d5223436f6
# docker network create -o com.docker.network.bridge.enable_ip_masquerade=true \
> right_bridge --subnet 172.20.0.0/24
9fcd6c29de3c1588ab15ce111deb8560d20032d6c05d4fd8afb779ba5988f2f9
```

The next step is to run a Docker container named *csrx-example*, or any other name, and connect the `mgt_bridge` to it:

```
# docker run -d --privileged --network mgt_bridge -e CSRX_SIZE="large" \
> -e CSRX_PORT_NUM=3 -e CSRX_ROOT_PASSWORD=lab123 --name csrx-example csrx:20.3R1.8
9125e356022dc390454f31e9c78555872bf0a8c1eedabdadd73b5626d9794700
```

Here is what the parameters in this command mean:

- `-d`: Or detached mode. The cSRX will not exit unless the host Docker process ends or the Docker container is removed.
- `--privileged`: Enable access to all devices on the host as well as set some configuration in AppArmor or SELinux to allow the container nearly all the same access to the host as processes running outside containers on the host.
- `-e CSRX_SIZE="large"`: This docker container will run as a "large" flavor.
- `-e CSRX_PORT_NUM=3`: This docker container will have three ports including one management port and two revenue ports. If this variable is not specified, the cSRX will take its default value of 3.
- `-e CSRX_ROOT_PASSWORD=lab123`: Initiate the cSRX by giving its root account a password for initial SSH login. "lab123" can be replaced by any preferred password.
- `csrx:20.3R1.8`: This is the cSRX Docker image name: image tag. This can be different, depending on the result of the `docker image` command.

Now connect the Docker container with networks `left_bridge` and `right_bridge`.

NOTE Here the order in which cSRX connects to one network or another matters. The first network connected to the cSRX is eth1, which is ge-0/0/0 and the second network connected to the cSRX is eth2, therefore ge-0/0/1 in the later configuration.

```
# docker network connect left_bridge csrx-example && docker network connect right_bridge csrx-example
```

To check the cSRX's IP address in the management network, run:

```
# docker network inspect mgt_bridge

[
    {
        "Name": "mgt_bridge",
        "Id": "3ce04dee8e4a73d9369fc8a581bbc34a817ccad815900f689bee1de40676098c",
        "Created": "2020-11-02T04:45:47.656493759-08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/24"
                }
            ]
        },
```

```
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "c3b2644407a9f455b5e51d5bb3f1c8a19104503df5f8d23991f7fa059e6bed29": {
                "Name": "csrx-example",
                "EndpointID": "27eac82da84ac6b61c5628dde4ccb0dddbd9784709cee6bba4135084064997df",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/24",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

The container's part of the output shows the csrx-example container has a management IP address of 172.18.0.2/24. So, let's run this next command to get the other interface's IP address for the cSRX:

```
# docker inspect csrx-example | grep IPAdd
          "SecondaryIPAddresses": null,
          "IPAddress": "",
                "IPAddress": "172.19.0.2",
                "IPAddress": "172.18.0.2",
                "IPAddress": "172.20.0.2",
```

With the three IP addresses ready, it is possible to start writing configurations for the cSRX:

```
(Host Server)# ssh root@172.18.0.2
The authenticity of host '172.18.0.2 (172.18.0.2)' can't be established.
ECDSA key fingerprint is SHA256:bsU4x9XG1NKS10dA1lWLNgPiUdWs5p+vyMEqgIQJrYM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.18.0.2' (ECDSA) to the list of known hosts.
root@172.18.0.2's password: (Please enter "lab123" as it was setup while booting cSRX)

root@86da2f3b2da4%ps -aux
...
root       442  2.6 31.1 3480996 1261680 ?     Sl   07:02   3:31 /usr/sbin/srxpfe -a -d
...
```

If the process `srxpfe` is present, the cSRX is running successfully. And it can be configured like any other SRX series firewall. Interface ge-0/0/0 and interface ge-0/0/1 use the addresses on networks `left_bridge` and `right_bridge`:

```
root@86da2f3b2da4%cli
root@86da2f3b2da4> configure
Entering configuration mode

[edit]
```

```
root@86da2f3b2da4#
set system host-name csrx-example
set system name-server 8.8.8.8
set interfaces ge-0/0/0 unit 0 family inet address 172.19.0.2/24
set interfaces ge-0/0/1 unit 0 family inet address 172.20.0.2/24
set routing-options static route 0.0.0.0/0 next-hop 172.19.0.1/24
set security nat source rule-set nat-source from zone trust
set security nat source rule-set nat-source to zone untrust
set security nat source rule-set nat-source rule nats1 match source-address 0.0.0.0/0
set security nat source rule-set nat-source rule nats1 match destination-address 0.0.0.0/0
set security nat source rule-set nat-source rule nats1 match application any
set security nat source rule-set nat-source rule nats1 then source-nat interface
set security policies from-zone trust to-zone untrust policy t2u match source-address any
set security policies from-zone trust to-zone untrust policy t2u match destination-address any
set security policies from-zone trust to-zone untrust policy t2u match application any
set security policies from-zone trust to-zone untrust policy t2u then permit
set security policies from-zone trust to-zone untrust policy t2u then log session-init
set security policies from-zone trust to-zone untrust policy t2u then log session-close
set security policies default-policy deny-all
set security zones security-zone trust interfaces ge-0/0/1.0 host-inbound-traffic system-services all
set security zones security-zone trust interfaces ge-0/0/1.0 host-inbound-traffic protocols all
set security zones security-zone untrust interfaces ge-0/0/0.0 host-inbound-traffic system-services
all
set security zones security-zone untrust interfaces ge-0/0/0.0 host-inbound-traffic protocols all
set system root-authentication plain-text-password
```

After pressing the Enter key, the system requires a new root password to be set. This password is for the user to manage the cSRX:

```
New password: (Please enter password)
Retype new password: (Please retype password)
```

One more step is to apply a license to the cSRX to activate the configurations:

```
root@csrx-example>request system license add terminal
[Type ^D at a new line to end input,
 enter blank line between each license key]
(Paste the license key and press enter to reach a blank line.)
(Then press “Ctrl + D” to apply the license.)
d402eeaa-e156-4492-abc2-a01a8b78aa05: successfully added
add license complete (no errors)

root@csrx-example> show system license
...
```

As a result, the cSRX can now reach the internet from the server gateway 172.19.0.1. Let’s verify:

```
root@csrx-example>ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=49 time=7.80 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=49 time=8.24 ms
```

To tear down the networks and containers created above, two simple steps can be implemented on the server: stop and remove the csrx container, and remove networks that don't have any containers connected to them.

```
root@csrx-example>exit
root@csrx-example#exit
(server)# docker stop csrx-example && docker rm csrx-example
(server)# docker network prune
```

Protecting Network Traffic with the cSRX

Let's create a *vulnerability scenario*, a simulation of a standard network and how the cSRX can be used to defend it, as illustrated in Figure 3.2.

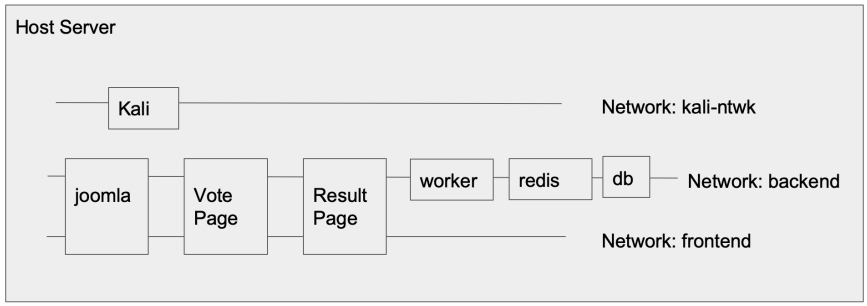

Figure 3.2 *A Vulnerability Scenario*

This example is a voting app and it has three interfaces: vote page, result page, and Joomla news page. There are three more containers in the backend: worker, redis, and database. When a user clicks to vote on the vote page, their choice will be reflected on the results page and results will be stored in database. Prepare for the Victim site in Figure 3.2 by downloading an example voting app from GitHub:

```
# apt install git -y
# git clone https://github.com/csrx-dayone/example-voting-app.git
# cd example-voting-app
```

To run this app, it is necessary to install a tool called *Docker Compose*. Docker Compose defines and runs multi-container Docker applications. Compose uses a YAML file to configure an application's services. So let's install Docker Compose following the instructions in the page: https://docs.docker.com/compose/install/.

After installing Docker Compose, it is possible to bring up the voting application with one command in the example-voting-app directory:

```
# docker-compose up
```

Docker Compose will collect necessary images and build networks for the voting app. After logs on the command line stop rolling, the user is able to see the voting app from a browser with the address http://(Host server IP address):5000, the voting results are available on http://(Host server IP address):5001, and a Joomla page, a content management system on http://(Host server IP address). The Joomla page is for releasing news and updates related to this voting site.

Now, create a network for the hacker, who is going to use a Kali-Linux to hack the voting application and run a Kali-Linux container connected to the network. If there is no Kali image in the host docker images, Docker will automatically pull it online:

```
# docker network create kali-ntwk --subnet 172.20.0.0/16
# docker run -t -d --network kali-ntwk --name kali-container \
> kalilinux/kali-linux-docker:latest
```

Users on the same Docker network can SSH to each other, whether they have ports exposed or not. In the example above, there is a container that serves as a database, and it has to connect to the backend network. When a hacker uses tools such as `Metasploit` in Kali-Linux to exploit weaknesses of `joomla` container, it will easily find vulnerabilities and sneak into the `joomla` container, then discover the database container on the same docker network, just by scanning with a tool called `nmap`. Further, it is possible for the hacker to set up port forwarding from the database container to Kali, and as a result the database can be watched and altered at any time. That's why it is necessary to set up the cSRX to protect the front-end containers, which are potential victims in this case.

MORE? If you are interested in detailed hacking procedures related to the Kali-Linux container, please refer to this whitepaper, *An Attacker Looks at Docker: Approaching Multi-Container Applications*, Black Hat 2018: https://www.blackhat.com/us-18/briefings/schedule/#an-attacker-looks-at-docker-approaching-multi-container-applications-9975.

Protection with the cSRX

To protect the Victim, let's set up the cSRX as a gateway gateway to the front-tier network for extra protection as shown in Figure 3.3.

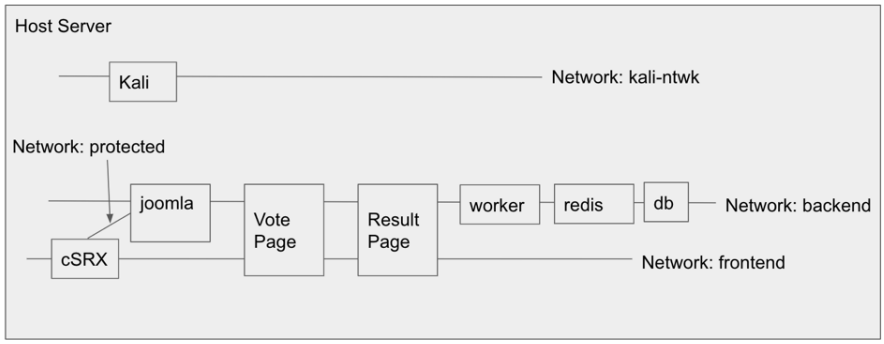

Figure 3.3 Protection Scenario

1. Tear down this setup. Then with your favorite editor change Line 35 to 36:

```
# docker-compose down
(On docker-compose.yml file, Lines 31-39)
  joomla:
    privileged: true
    image: kuthz/joomla
    container_name: joomla
    expose:
      - "80"
    networks:
      - front-tier
      - back-tier
```

2. Build up the voting site with joomla new port settings:

```
# docker-compose up
```

3. Disconnect the `joomla` container from the front-tier network:

```
# docker network disconnect example-voting-app_front-tier joomla
```

4. Create `protected-ntwk` and `mgt-ntwk`:

```
# docker network create -o com.docker.network.bridge.enable_ip_masquerade=true \
> --subnet 172.21.0.0/16 protected-ntwk
# docker network create mgt-ntwk --subnet 172.22.0.0/16
```

5. Start the cSRX, publishing the cSRX's port 3456 to host's server port 80, and connect the cSRX with `front-tier` and `protected-ntwk`:

```
# docker run -d --privileged -p 80:3456 --network mgt-ntwk -e CSRX_SIZE="large" \
> -e CSRX_ROOT_PASSWORD=lab123 --name csrx3 csrx:latest
# docker network connect example-voting-app_front-tier csrx3
# docker network connect protected-ntwk csrx3
```

6. Connect `joomla` back to `protected` network:

```
# docker network connect protected-ntwk joomla
```

7. Then apply license to the cSRX and edit the configuration on the cSRX. On every network, please check which IP addresses the cSRX and joomla containers have received, and write them into the cSRX configuration. Interface IP addresses can be different every time.

```
# ssh root@172.22.0.2
root@172.22.0.2's password: lab123
root@1d3ff156c8a1%cli

root@1d3ff156c8a1> request system license add terminal
(paste the license contents)
(Start a new line, and press ctrl + D)
...: successfully added
add license complete (no errors)
root@1d3ff156c8a1> configure
Entering configuration mode

[edit]
root@1d3ff156c8a1#
set system host-name csrx3
set system name-server 8.8.8.8
set interfaces ge-0/0/0 unit 0 family inet address 172.18.0.4/16
set interfaces ge-0/0/1 unit 0 family inet address 172.21.0.2/16
set routing-options static route 0.0.0.0/0 next-hop 172.18.0.1/32
set security nat source rule-set rs1 from zone trust
set security nat source rule-set rs1 to zone untrust
set security nat source rule-set rs1 rule r1 match source-address 0.0.0.0/0
set security nat source rule-set rs1 rule r1 match destination-address 0.0.0.0/0
set security nat source rule-set rs1 rule r1 match application any
set security nat source rule-set rs1 rule r1 then source-nat interface
set security nat destination pool joomla address 172.21.0.3/32
set security nat destination pool joomla address port 80
set security nat destination rule-set rs2 from zone untrust
set security nat destination rule-set rs2 rule r2 match destination-address 172.18.0.4/32
set security nat destination rule-set rs2 rule r2 match destination-port 3456
set security nat destination rule-set rs2 rule r2 then destination-nat pool joomla
set security policies default-policy permit-all
set security zones security-zone trust interfaces ge-0/0/1.0 host-inbound-traffic system-services all
set security zones security-zone trust interfaces ge-0/0/1.0 host-inbound-traffic protocols all
set security zones security-zone untrust interfaces ge-0/0/0.0 host-inbound-traffic system-services
all
set security zones security-zone untrust interfaces ge-0/0/0.0 host-inbound-traffic protocols all
set system root-authentication plain-text-password
```

Enter a password so you can manage the csrx3. Commit the configuration and quit the SSH process to the cSRX container.

8. On the joomla container, point its gateway to the cSRX:

```
# docker exec -it joomla /bin/bash
(joomla)# ip route delete 172.21.0.0/16
(joomla)# ip route delete 172.19.0.0/16
(joomla)# ip route delete default
```

```
(joomla)# ip route add 172.19.0.0/16 dev eth0
(joomla)# ip route add 172.21.0.0/16 dev eth2
(joomla)# ip route add default via 172.21.0.2
(joomla)# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=49 time=10.253 ms
64 bytes from 8.8.8.8: seq=1 ttl=48 time=7.600 ms
```

As you can see, traffic from the Victim container's route to the outside jumps through the cSRX, then the Ubuntu Host, and finally the Ubuntu Host's gateway to the other routers.

9. Disable tcp offload on all virtual interfaces on the host:

```
# ip link | grep UP | awk –F ':' '/veth/ {print $2}' | awk –F '@' '{print $1}' | while read line
> do
>  ethtool –K $line tx off >/dev/null
> done
```

After following these procedures, you should be able to access the Joomla page with a browser, and the address is http://(Host server IP address). Although it looks the same as before, Joomla is already under the protection of cSRX and the user is actually browsing from port 3456 of the cSRX. If a hacker tries to scan open ports of the frontend webpage, they will be scanning the cSRX with destination NAT instead. As a result, the database container is not likely to be hacked through the Joomla container.

With similar procedures, other containers in the frontend can also be protected with the cSRX.

Chapter 4

Advanced cSRX Configuration

This chapter introduces you to some advanced cSRX configurations. To replicate these in your lab, you'll need to prepare a server with the following:

- Linux OS CentOS 6.5 or later; or Red Hat Enterprise Linux (RHEL) 7.0 or later; or Ubuntu 14.04.2 or later
- Docker Engine 1.9 or later
- At least 2 CPU cores
- At least 8 GB Memory
- At least a 40 GB hard drive
- The processor type is x86_64 multicore CPU
- One Ethernet port
- A cSRX image in Docker images

For Docker installation, please refer to the section *Docker Installation and Verification* in Chapter 2. And for getting the cSRX image, please refer to *Setting Up the cSRX* in Chapter 3.

NOTE Download the configuration examples in this book from its GitHub repository: https://github.com/csrx-dayone .

Configure the cSRX at Launch

To avoid writing configurations manually into the cSRX after it boots up, here is a simplified method for bringing up the cSRX with a pre-loaded configuration. First, write a file for loading and showing configurations for the cSRX in a text file. Assume this file is under directory /home/lab:

```
# mkdir /home/lab
# vi /home/lab/confcsrx01.txt
(Or choose your favorite editor)
```

Then paste in the contents below:

```
edit
set system root-authentication encrypted-password "$6$JkGvR$LuvoEsu838iLIpuqAgm8VsHeiLEMfNFT/5ri.
F4C7SSbE10246Oo8VCR4yiLcwWFBO33Krc6rJio5fDoWHIw21"
set system name-server 8.8.8.8
set system host-name csrx01
set security zones security-zone trust interfaces ge-0/0/1.0 host-inbound-traffic system-services all
set security zones security-zone trust interfaces ge-0/0/1.0 host-inbound-traffic protocols all
set security zones security-zone untrust interfaces ge-0/0/0.0 host-inbound-traffic system-services
all
set security zones security-zone untrust interfaces ge-0/0/0.0 host-inbound-traffic protocols all
set security nat source rule-set nat-source from zone trust
set security nat source rule-set nat-source to zone untrust
set security nat source rule-set nat-source rule nats1 match source-address 0.0.0.0/0
set security nat source rule-set nat-source rule nats1 match destination-address 0.0.0.0/0
set security nat source rule-set nat-source rule nats1 then source-nat interface
set security policies from-zone trust to-zone untrust policy t2u match source-address any
set security policies from-zone trust to-zone untrust policy t2u match destination-address any
set security policies from-zone trust to-zone untrust policy t2u match application any
set security policies from-zone trust to-zone untrust policy t2u then permit
set security policies from-zone trust to-zone untrust policy t2u then log session-init session-close
set security policies default-policy deny-all
set interfaces ge-0/0/0 unit 0 family inet address 192.168.1.2/24
set interfaces ge-0/0/1 unit 0 family inet address 192.168.2.2/24
set routing-options static route 0.0.0.0/0 next-hop 192.168.2.1
commit and-quit
show configuration
show security flow status
show security flow session
```

At the same time, prepare the left and right networks:

```
# docker network create vnetwork1 --subnet 192.168.1.0/24
# docker network create vnetwork2 --subnet 192.168.2.0/24
```

Second, copy the script into the cSRX while running it. Here, the -v parameter means mount a volume. That is, while cSRX starts, it saves the contents of the server file /home/lab/confcsrx01.txt to the cSRX as /root/conf.txt:

```
# docker run -itd --privileged --cap-add=ALL --network=bridge -e CSRX_SIZE="large" -v /home/lab/
confcsrx01.txt:/root/conf.txt --name=csrx01 csrx:latest
# docker network connect vnetwork1 csrx01
# docker network connect vnetwork2 csrx01
# sleep 10s
```

Finally, run the copied script inside the cSRX after it boots up:

```
# docker exec -it csrx01 /usr/sbin/cli -f /root/conf.txt

...
root@csrx01> show configuration

## Last commit: 2019-09-23 06:09:38 UTC by root
##
## Warning: statement ignored: unsupported platform (csrx)
##
version 20190606.224121_builder.r1033375;
system {
    host-name csrx01;
    root-authentication {
        encrypted-password "$6$JkGvR$LuvoEsu838iLIpuqAgm8VsHeiLEMfNFT/5ri.
F4C7SSbE10246Oo8VCR4yiLcwWFBO33Krc6rJio5fDoWHIw21"; ## SECRET-DATA
    }
    name-server {
        8.8.8.8;
    }
}
...

root@csrx01> show security flow status

  Flow forwarding mode:
    Inet forwarding mode: none (reboot needed to change to flow based)
    Inet6 forwarding mode: flow based
    MPLS forwarding mode: none (reboot needed to change to drop)
    ISO forwarding mode: none (reboot needed to change to drop)
    Tap mode: disabled (default)
  Flow trace status
    Flow tracing status: off
  Flow session distribution
    Distribution mode: RR-based
    GTP-U distribution: Disabled
  Flow ipsec performance acceleration: off
  Flow packet ordering
    Ordering mode: Hardware
  Flow power mode IPsec: Disabled


root@csrx01> show security flow session

Total sessions: 0
```

With this method, the configuration is loaded to the cSRX while starting up with one script only. This method is beneficial when the cSRX is to be deployed automatically in a *bash script*.

Additionally, to clean up csrx01, or any other Docker container, this command can be executed:

```
# docker stop csrx01 && docker rm csrx01 && docker network prune
```

Security Director Management of the cSRX

Starting from 20.2, the cSRX can be monitored and configured on Juniper Security Director 19.3 (and later versions). No matter which container runtime is used, as long as the ports 22 and 830 of the cSRX container are exposed and can be reached by Security Director, you can watch the status of the cSRX on a web-based Security Director as well as configure it by just using the UI. Here is an example of managing the cSRX on Docker.

Step 1: Start a cSRX on a server that has Docker installed on it:

```
(server)# docker run -d --privileged --network mgt_bridge -e CSRX_SIZE="large" -e CSRX_PORT_NUM=3 -e
CSRX_ROOT_PASSWORD=lab123 --name csrx-example -p 8080:22 -p 830:830 csrx:20.2R1.10
(server)# docker network connect zleft_bridge csrx-example && docker network connect zright_bridge
csrx-example
```

Here, two more parameters are added while bringing up the cSRX: `-p 8080:22 -p 830:830`. This means that the container's port 22 will be mapped to the host server's 8080 port, and the container's port 830 is mapped to the host server's 830 port. Junos Space initially uses port 22 to SSH to the devices and add the `set system services netconf ssh` configuration to the device. Then leverage netconf ssh, which usually communicates on port 830 to deploy configurations to the device.

On Docker, when there are multiple container networks connected to a container, the port mapping automatically applies to the first container network alphabetically. When there is a connection issue running Device Discovery from Security Director, it is likely that the revenue ports are mapped on the cSRX.

Step 2: Turn off "Add SNMP configuration to device" on Junos Space. Because cSRX does not support SNMP, you need to turn it off on Junos Space for communication between Junos Space and cSRX.

Log in to Juniper Security Director (see Figure 4.1) and click on the Junos Space icon on the upper left corner to redirect to the Junos Space page. In the left column, click "Administration" and find "Applications". On the list of the right side of the page, right click on "Network Management Platform" and choose "Modify Application Settings". Then in the next screen, uncheck "Add SNMP configuration to device for fault monitoring". Save the settings.

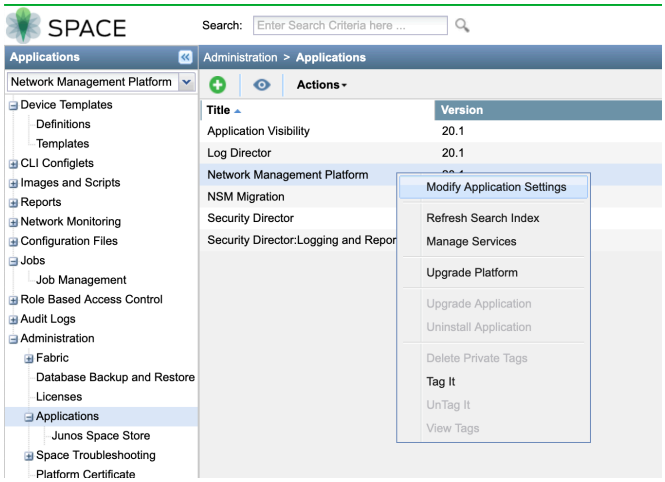

Figure 4.1 *Modify Application Settings on Junos Space*

Step 3: Run Device Discovery on Junos Space.

Create a device discovery profile in the screen "Devices/Device Discovery/Device Discovery Profiles". Create a device discovery file by entering the server's IP address in "Target Details", and "8080" in "SSH Port". Click on "Next", and uncheck "Use SNMP". In the next screen, input cSRX's authentication method and password or certificate. Until now, Junos Space has all the necessary information to discover the cSRX and a device discovery job will be created.

When the cSRX is successfully managed by Junos Space, you can choose "Security Director" from the Junos Space Applications list in the upper left corner of the screen.

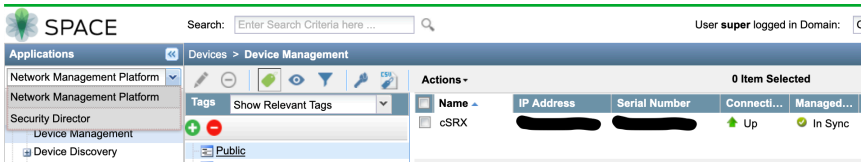

Figure 4.2 *cSRX in Up and Managed by Junos Space*

Now, you are ready to monitor and configure the cSRX on a web-based interface. Device information can be accessed on the Devices > Security Devices page of Security Director.

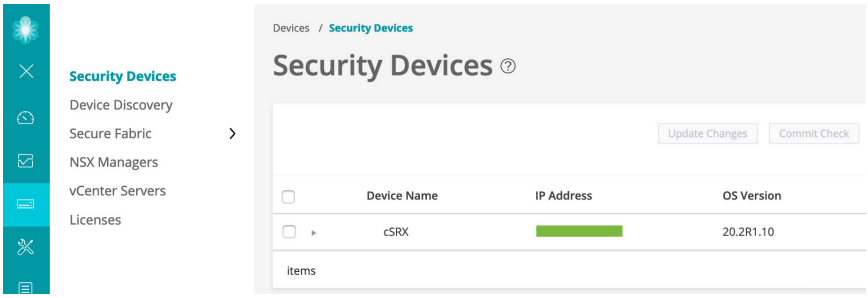

Figure 4.3 *cSRX Managed by Security Director*

NextGen Firewall Functions for the cSRX

Like other types of the Juniper SRX Series firewall, the container SRX is able to perform next generation functions despite its small size and fast bootup speed. In other words, cSRX is not only a Layer 3 stateful firewall, it is able to execute a variety of Layer 4 to Layer 7 functions, including:

- Application security features (including IPS and AppSecure)
- UTM content security features (including antispam, Sophos Antivirus, web filtering, and content filtering)
- Authentication and integrated user firewall features

These Next Generation Firewall functions now require licenses. User could download a trial license from Juniper official website: https://www.juniper.net/us/en/dm/csrx-trial/. For acquiring Juniper cSRX licenses for production environment, please contact a Juniper representative. For how to apply a license to the cSRX, please refer to Chapter 3.

In a few more paragraphs, we introduce IPS and AppSecure with configuration examples. Additionally, a new Junos unified policy, with examples, is shown. First let's get your lab set up.

IDP Security Package Download and Installation

The Junos OS Intrusion Detection and Prevention (IDP) policy enables you to selectively enforce various attack detection and prevention techniques on network traffic passing through an IDP-enabled device. It allows you to define policy rules to match a section of traffic based on a zone, network, and application, and then take active or passive preventive actions on that traffic.

After the lab *Protecting Network Traffic with the cSRX* from Chapter 3, your cSRX should be running and protecting other containers in the trust zone. Here is how to download and install IDP security package for Layer 7 protection:

```
root@csrx3> request security idp security-package download check-server
Successfully retrieved from(https://signatures.juniper.net/cgi-bin/index.cgi).

root@csrx3> request security idp security-package download
Will be processed in async mode. Check the status using the status checking CLI

root@csrx3> request security idp security-package download status
In progress:groups.xml.gz       0 % 273437 Bytes/ 0 Bytes
```

The download status will be different if the download status command is executed multiple times. Wait until the IDP security-package download status shows:

```
root@csrx3> request security idp security-package download status
Done;Successfully downloaded from(https://signatures.juniper.net/cgi-bin/index.cgi).
```

Then package installation can be performed.

```
root@csrx3> request security idp security-package install
```

Will be processed in async mode. Check the status using the status checking CLI

```
root@csrx3> request security idp security-package install status
```

The installation status will be different if the install status command is executed multiple times. When the status output shows the following, the installation is successful:

```
root@csrx3> request security idp security-package install status
Done;Attack DB update : successful - [UpdateNumber=3208,ExportDate=Tue Sep 17 14:32:15 2019
UTC,Detector=12.6.130190828]
    Updating control-plane with new detector : successful
    Updating data-plane with new attack or detector : not performed
     due to no active policy configured.
```

The CLI command below can be used to check the current IDP security package version:

```
root@csrx3> show security idp security-package-version
  Attack database version:3208(Tue Sep 17 14:32:15 2019 UTC)
  Detector version :12.6.130190828
Policy template version :N/A
```

IDP Policies Configuration

Starting with Junos OS Release 18.2R1, Unified Policies are supported on SRX Series devices, allowing granular control and enforcement of Dynamic Layer Applications within the traditional security policy. The new syntax allows different IDP policies to be applied on different security policies, where different sources, destinations, applications, or dynamic applications can be flexibly defined. This is different from previous Junos OS configurations, where all security policies fit one

IDP rule. For the old syntax, `set security idp active-policy POLICY-NAME`, the cSRX will give a warning message `## Warning: 'active-policy' is deprecated.`

Therefore, unified policies are made more flexible by giving different controls for different application packages travelling through the cSRX gateway. Here is an example of a unified IDP policy implemented on the cSRX:

```
root@csrx3> show configuration security policies
from-zone trust to-zone untrust {
    policy idp-policy {
        match {
            source-address any;
            destination-address any;
            application any;
        }
        then {
            permit {
                application-services {
                    idp-policy POLICY-NAME;
                }
            }
        }
    }
}
default-policy {
    deny-all;
}
```

Defining a particular IDP policy is the same as with old versions of Junos configuration syntax. Under the hierarchy [edit security idp idp-policy], the detailed criteria, rules, and actions are defined. Here is an example of an IDP rules definition. After `idp-policy p1` is defined, it can be used in above security policies replacing `idp-policy POLICY-NAME`:

```
root@csrx3> show configuration security idp idp-policy p1
rulebase-ips {
    rule r1 {
        match {
            attacks {
                predefined-attack-groups "DB - Critical";
            }
        }
        then {
            action {
                recommended;
            }
            notification {
                log-attacks {
                    alert;
                }
            }
        }
    }
}
```

This part of configuration defines an IDP policy, p1, to recognize all database critical attacks pre-defined in Junos attack groups. Once an attack is found, the Junos recommended action will be taken, the attack will be recorded in attack logs, and an alert will be raised in the logs.

If you would like to check details of a signature including its recommended action, use this command:

```
root@csrx3> show security idp attack detail APP:BLUECOAT-AAA-OF
Display Name: APP: Blue Coat Authentication and Authorization Agent Overflow
Severity: Major
Category: APP
Recommended: false
Recommended Action: Drop
Type: signature
Direction: CTS
False Positives: unknown
Service: TCP/16102
Shellcode: no
Flow: control
Context: packet
Negate: false
Regex: B8C3052F083AAC5C3C51D869FCA364C5
TimeBinding:
           Scope: none
           Count: 1
Hidden Pattern: True
Pattern: Protected
```

Application Firewall Policies

After downloading and installing the IDP security package, application identification is also enabled. In operation mode, it can be seen here:

```
root@csrx3> show services application-identification status

Application Identification
 Status                     Enabled
...
```

Like other SRX Series firewalls, with application identification the cSRX is able to implement unified policies for precise application control. Take the configuration below as an example:

```
root@csrx3> show configuration security policies
from-zone trust to-zone untrust {
    policy application-control {
        match {
            source-address any;
            destination-address any;
            application any;
            dynamic-application junos:TWITTER;
        }
        then {
            deny;
```

```
            log {
                session-init;
                session-close;
            }
            count;
        }
    }
    policy t2u {
        match {
            source-address any;
            destination-address any;
            application any;
            dynamic-application any;
        }
        then {
            permit;
            log {
                session-init;
                session-close;
            }
        }
    }
}
default-policy {
    deny-all;
}
```

This configuration allows all types of traffic, from zone trust to zone untrust, except for traffic for browsing and tweeting on Twitter. This is really helpful when cSRX is implemented in a DevOps environment, where you want to stop some application-related traffic on either testing or production containers, but you don't want to make changes on the container itself. Blocking traffic to the application service is an ideal method.

NOTE For unified policies containing dynamic application, when a new dynamic application policy is inserted before a policy that does not contain dynamic application, the order of security policies execution on the cSRX will not be from top to bottom. Actually, the policy that does not contain dynamic application will be evaluated first in a new Junos unified policy system. It's suggested that if one rule contains dynamic application, all the other rules should add dynamic application configurations as well (`"dynamic-application any"`). Only when all the policies are consistent, will they be evaluated in top to bottom order.

Secure Wire on the cSRX

Traditional firewall deployment requires setting up a routing table in Layer 3 networks or switching look up (transparent mode). Secure wire is a feature in the SRX-series that supports traffic forwarding from one interface to another, unchanged. In this case, the cSRX can be deployed in the path of network traffic

without a change to the routing table or reconfiguration of neighborhood devices. As long as the traffic is permitted by a security policy, a packet arriving on the ingress interface is forwarded unchanged out of the egress interface. Return traffic is also forwarded unchanged. This feature does not support NAT or IPsec VPN. Advanced security features like AppSecure, and IDP are supported.

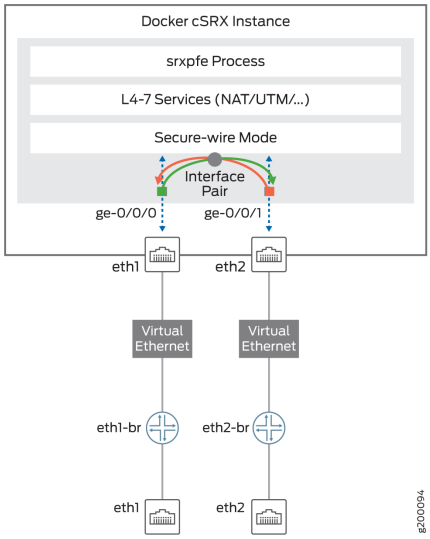

Figure 4.4 *Secure Wire on the cSRX*

To launch the cSRX in secure-wire mode, use this environment variable CSRX_FORWARD_MODE="wire":

```
# docker run -d --privileged --network mgt_bridge -e CSRX_SIZE="large" \
> -e CSRX_PORT_NUM=3 -e CSRX_FORWARD_MODE="wire" -e CSRX_ROOT_PASSWORD=lab123 \
> --name csrx3 csrx:latest
```

The environment variable CSRX_FORWARD_MODE="wire" defines the cSRX to boot up in secure wire mode. If CSRX_FORWARD_MODE is not defined, cSRX will launch taking its default value CSRX_FORWARD_MODE="routing". The CLI to configure secure wire is as follows:

```
set interfaces ge-0/0/0 unit 0
set interfaces ge-0/0/1 unit 0
set security policies default-policy permit-all
set security zones security-zone untrust interfaces ge-0/0/0.0
set security zones security-zone trust interfaces ge-0/0/1.0
```

MORE? For more information, please refer to this Juniper TechLibrary doc: https://www.juniper.net/documentation/en_US/junos/topics/concept/layer-2-secure-wire-understanding.html.

Chapter 5

Deploying the cSRX in an SDN Environment

The cSRX on Contrail

Juniper Networks Contrail Networking is an open, standards-based, software-defined networking (SDN) platform that delivers network virtualization and service automation for federated cloud networks. It provides self-service provisioning, improves network troubleshooting and diagnostics, and enables service chaining for dynamic application environments across enterprise virtual private cloud (VPC), managed Infrastructure as a Service (IaaS), and Networks Functions Virtualization (NFV) use cases.

Contrail can connect multiple orchestration stacks; for example, Kubernetes, Mesos/SMACK, OpenShift, OpenStack, and VMware and provide advanced network management for these platforms. Contrail is a versatile network manager. It is capable of IPAM, DHCP, DNS, floating IPs, EVPN fabric control, ECMP, stateful and Level 7 load balancing, virtual and physical service chaining, BGPaaS, virtual routing gateways, IPv6, and more for overlay networks. All functions are accessible by the REST API.

Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation. Kubernetes uses Container Network Interface (CNI) as an interface between network providers (Contrail) and Kubernetes networking.

By default, the Kubernetes networking model is fairly simple, "*pods on a node can communicate with all pods on all nodes without NAT.*" With Contrail Networking as a networking plugin to Kubernetes, different networks can be created and separated. Also with Contrail Networking security policies, security rules can be defined and applied to Kubernetes networks. What's more, adding cSRX in the network will make the traffic control more versatile, by adding Layer 4 to Layer 7 security policies to the pod networks.

As an orchestration system for containerized workloads, Kubernetes is able to schedule containers to run on a cluster of machines. It is possible to run multiple containers on one machine or multiple hosts. Kubernetes can manage the state of these containers including:

- Starting and stopping containers on a specific node. A node means a host, either physical or virtual, in the Kubernetes cluster.
- Monitoring and restarting a container when it goes down.
- Migrating containers from one node to another.

Kubernetes is the tool that is capable of managing the containers while building enterprise applications, instead of manually running a few docker containers on one host. Kubernetes clusters can start with one node that can replicate itself up to thousands of nodes. There are also other popular Docker orchestrators on the market such as Docker Swarm, which is part of the Docker Enterprise release, and Mesos, which is released by Apache.

Today, Kubernetes can be run anywhere. It can run on-premise, in a data center, on public cloud like Google cloud or AWS, and also on hybrid clouds. As an open source project backed by Google, Kubernetes has a great user community and is also a standard.

There are different tools to help users quickly spin up Kubernetes such as minikube and kops. While minikube can only support Kubernetes running on a single machine, kops supports Kubernetes clusters running on different nodes and even on cloud environments such as AWS. In this book, an Ansible Deployer is used to spin up both Kubernetes and Contrail in the environment.

What Is Service Chaining?

Network service chaining, also known as service function chainingis a capability that uses software-defined networking (SDN) capabilities to create a service of connected network services (such as Level 3-7 like firewalls, network address translation [NAT], intrusion protection, etc.) and connects them in a virtual chain.

In short, service chaining creates dynamic network functions such as service VMs or service containers, then assembles all the VMs or containers into a logical flow. Here, the cSRX is capable of performing as such a service container.

The benefits of service chaining are:

- Enabling automated provisioning of network applications that may have different characteristics, and
- Optimizing the use of network resources and improving application performance.

Kubernetes is a widely used tool for enterprise DevOps teams because of its ability to deploy images and scale containers up or down in an agile manner. The benefits of integrating Contrail and cSRX to it are:

- Compared with Kubernetes Network, Contrail is a more versatile tool for container networking management.
- The cSRX takes very a short time to spin up and does not consume many resources.
- Layer 3 to Layer 7 security functions are available from the cSRX.
- This implementation can be extended to any containerized DevOps environment: AWS, Google Cloud, BareMetal server, etc.

Contrail and Kubernetes Environment Setup

Contrail and Kubernetes can either be installed in multiple servers or all-in-one mode. For simplicity and demo purposes only, all-in-one mode is used in this part.

The system requirements for the server setting up the Contrail and Kubernetes environment are:

- CentOS 7.7
- 16 vCPUs
- 64 GB RAM
- 300 GB storage

NOTE Detailed installation steps are available at the Juniper TechLibrary: https://www.juniper.net/documentation/en_US/contrail20/topics/task/installation/provisioning-k8s-cluster.html.

In this section you will add all the necessary dependencies to their version on a brand new server and install Kubernetes and Contrail. The steps of installing Kubernetes and Contrail are:

1. Re-image the node to CentOS 7.7 so that the server does not have any software packages pre-installed. If some software package such as Docker was previously installed, there could be error messages caused by version incompatibilities in later steps. Also please check the host's kernel version to satisfy contrail minimum kernel version requirements from latest contrail release note. Otherwise vrouter for contrail won't be able to start successfully.

```
# uname -r
3.10.0-1127.18.2.el7.x86_64
```

2. Install the necessary tools:

```
# yum -y install epel-release git net-tools
# yum -y install python-pip wget
# pip install requests
# pip install ansible==2.7.18
```

3. Download the *contrail-ansible-deployer-(version number).tgz* Ansible Deployer application tool package onto your server from the Juniper Contrail Downloads page called Application Tools. You will need a Juniper account in order to download the package. Extract the package. The latest version at the time this book was written is 2005.1, as shown in Figure 5.1.

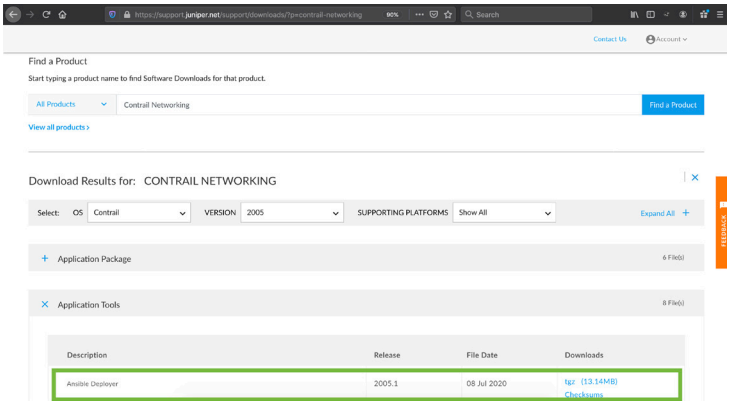

Figure 5.1 *Downloading from Contrail Page*

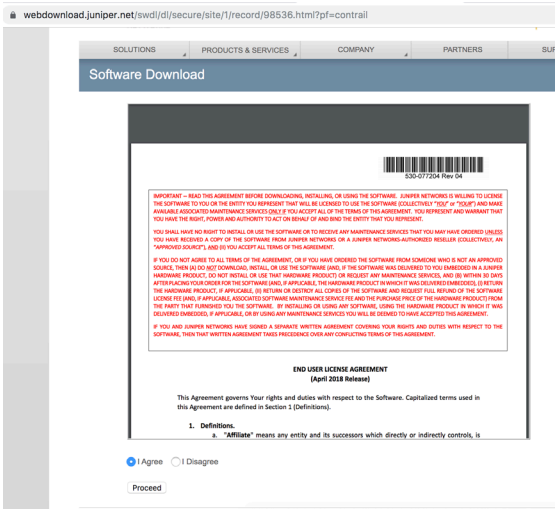

Figure 5.2 *Agree to Terms and Conditions*

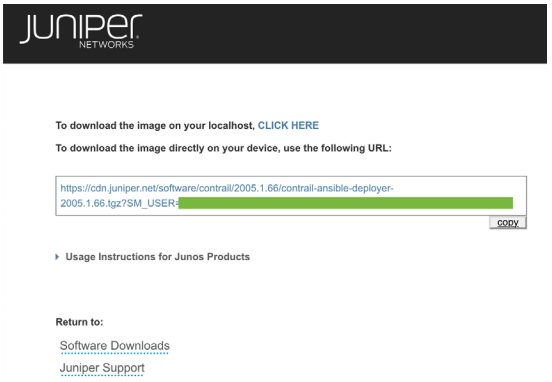

Figure 5.3 *Copy Downloaded URL*

```
(server, download ansible deployer)# wget "(paste link provided by the download page)"
(server, rename the package)# mv [long file_name] contrail-ansible-deployer-(version number).tgz
(server, extract the package)# tar xvf contrail-ansible-deployer-(version number).tgz
```

4. Navigate to the `contrail-ansible-deployer` directory:

```
#cd contrail-ansible-deployer
```

5. Edit the config/instances.yaml file with your favorite editor and enter the necessary values. Below is an example of a config/instances.yaml file for an all-in-one Kubernetes Contrail install. Please use your server's SSH password, hostname, and IP address accordingly.

NOTE YAML files use space for indentation and tabs are not recognized. For the first layer in the hierarchy, no spaces are needed, then for the second layer two spaces are necessary, third layer four spaces, and so on. If the parameters are in the wrong hierarchy, because the number of spaces in the file is not accurate, error messages may appear in later steps!

```
global_configuration:
  CONTAINER_REGISTRY: opencontrailnightly
provider_config:
  bms:
    ssh_pwd: <Password>
    ssh_user: root
    ssh_public_key: /root/.ssh/id_rsa.pub #optional if use password
    ssh_private_key: /root/.ssh/id_rsa #optional if use password
    domainsuffix: local
instances:
  <Server Hostname>:
    provider: bms
    roles:
      config_database:
      config:
      control:
      analytics_database:
      analytics:
      webui:
      k8s_master:
      kubemanager:
      vrouter:
      k8s_node:
    ip: <BMS IP>
contrail_configuration:
  CONTAINER_REGISTRY: opencontrailnightly
  CONTRAIL_VERSION: master-latest
  KUBERNETES_CLUSTER_PROJECT: {}
```

6. Turn off the `swap` functionality on all nodes. This command does not have any output:

```
# swapoff -a
```

7. Configure the nodes by running the Ansible playbook. If the playbook result shows zero failures, the step is successful:

```
# ansible-playbook -e orchestrator=kubernetes -i inventory/ \
> playbooks/configure_instances.yml

PLAY [Create container host group] *********************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

...

PLAY RECAP *********************************************************************
172.27.60.160            : ok=38   changed=5    unreachable=0    failed=0    skipped=46   rescued=0
ignored=1
localhost                : ok=52   changed=2    unreachable=0    failed=0    skipped=64   rescued=0
ignored=0
```

8. Install Kubernetes and Contrail by running the playbook. If the playbook result shows zero failures, the step is successful:

```
# ansible-playbook -e orchestrator=kubernetes -i inventory/ \
> playbooks/install_k8s.yml

...

PLAY RECAP *********************************************************************
172.27.60.160              : ok=31   changed=14   unreachable=0    failed=0    skipped=24   rescued=0
ignored=1
localhost                  : ok=61   changed=3    unreachable=0    failed=0    skipped=51   rescued=0
ignored=0

# ansible-playbook -e orchestrator=kubernetes -i inventory/ \
> playbooks/install_contrail.yml

...

PLAY RECAP *********************************************************************
172.27.60.160              : ok=103  changed=50   unreachable=0    failed=0    skipped=52   rescued=0
ignored=7
localhost                  : ok=64   changed=3    unreachable=0    failed=0    skipped=51   rescued=0
ignored=0
```

Verification

Verify that Kubernetes and Contrail have both been successfully installed:

- Contrail UI should be accessible from web browser, address https://<ServerIPAddress>:8143. The web browser may give a warning because the site uses a self-generated certificate. Please ignore the warning message and proceed to the site, or add an exception for this site. The login default username is “admin” and the password is “contrail123”.
- On the server command line, `kubectl` is able to show outputs:

```
# kubectl version

Client Version: version.Info{Major:"1", Minor:"14", GitVersion:"v1.14.8", GitCommit:"211047e9a1922595
eaa3a1127ed365e9299a6c23", GitTreeState:"clean", BuildDate:"2019-10-15T12:11:03Z",
GoVersion:"go1.12.10", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"14", GitVersion:"v1.14.8", GitCommit:"211047e9a1922595
eaa3a1127ed365e9299a6c23", GitTreeState:"clean", BuildDate:"2019-10-15T12:02:12Z",
GoVersion:"go1.12.10", Compiler:"gc", Platform:"linux/amd64"}
```

Set Up Service Chaining Using cSRX

After the system is installed with Contrail and Kubernetes, it's time to build a service chain. Let's start with the simple topology shown in Figure 5.4. The arrows in the topology indicate traffic flow.

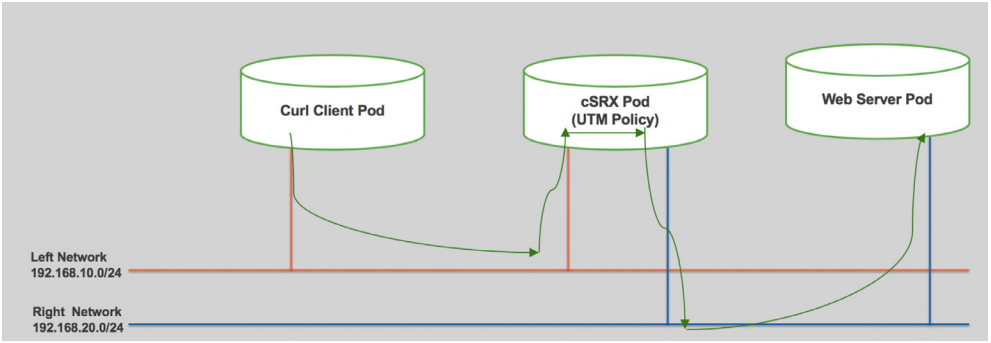

Figure 5.4 *Service Chain Simple Topology*

As you can see there are three containers in the network simulating an enterprise DevOps environment. The left pod is a container that can do curl, trying to reach the right pod, which serves as a web server on another network. There are some hidden pages on the web server pod that the DevOps team requires that the left network should not have access to, however.

This scenario is where the cSRX service chain can be applied. By using UTM policy the cSRX is able to redirect the traffic and hide the page from the curl client pod.

Set Up the Topology

The first step to create this topology is to build up the left network and the right network. In any directory of the server, clone the repository form GitHub and there is a file called network-left.yaml:

```
# git clone https://github.com/csrx-dayone/service-chaining.git
# cd service-chaining
# ls
csrx.yaml  left.yaml  network-left.yaml  network-object-model.yaml  network-right.yaml  right.yaml
```

In the YAML file network-left.yaml, it defines that the network to be created with CIDR is 192.168.10.0/24. Then, in the same directory, create this network by running this YAML file:

```
# kubectl create –f network–left.yaml
networkattachmentdefinition.k8s.cni.cncf.io/network–left created
```

With the same method, the network-right can be created by running the network-right.yaml file in the same directory, too. The left network and the right network have different CIDR:

```
# kubectl create –f network–right.yaml
networkattachmentdefinition.k8s.cni.cncf.io/network–right created
```

After creating the two networks, create the curl container called left.

In this left.yaml file, the item type pod is used. A Kubernetes pod is a group of containers that are deployed together on the same host. In this example, one pod contains only one container. Also this YAML file defines the container in this pod using the image tutum/curl, which is expected to be pulled from DockerHub. This pod is attached to network k8s-network-left-pod-network. Run this command and create the left pod:

```
# kubectl create –f left.yaml
Pod/left created
```

Similarly, the YAML file below can be used to create a web server pod to create some webpages in the pod. The two webpages are called `blacklist` and `whitelist`, and their contents can be self-defined in `/root/whitelist.html` and `/root/blacklist.html`.

Run this command and create the right pod:

```
# kubectl create –f right.yaml
pod/right created
```

Create the cSRX

Because the cSRX is not available on public container image libraries such as DockerHub, it is necessary that you download it from the Juniper download website to the Contrail compute node, then load it into Docker images:

```
#docker load –i junos–csrx–docker–20.3R1.8.img
```

Next, there is an example cSRX.yaml to create the container cSRX. Under the image name, imagePullPolicy: IfNotPresent means that kubernetes will skip pulling an image from DockerHub if it's already loaded into compute node's Docker image list.

This file also defines the image name tag of the cSRX pod. Also this pod is connected with network-left and network-right. The cSRX pod can be started with this command:

```
# kubectl create –f csrx.yaml
pod/csrx created
```

To check if the cSRX is running successfully, use the `kubectl describe pod csrx` command, and then find the `started container` message. Also the cSRX information shown from the output of this command, you can see two IP addresses of cSRX from the `Annotations` section:

```
# kubectl describe pod csrx
Name:               csrx
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               localhost.localdomain/172.27.60.142
Labels:             <none>
Annotations:        k8s.v1.cni.cncf.io/network-status:
              [
                   {
                   "ips": "192.168.10.251",
                   "mac": "02:81:4f:12:fc:c9",
                   name": "network-left"
                   },
...

Events:
   Type    Reason     Age   From                              Message
  ----    ------  m   ----  ----                               -------
  Normal  Scheduled  28m   default-scheduler                 Successfully assigned default/
csrx to localhost.localdomain
  Normal  Pulled     28m   kubelet, localhost.localdomain    Container image
"csrx:20.3R1.8" already present on machine
  Normal  Created    28m   kubelet, localhost.localdomain    Created container
  Normal  Started    28m   kubelet, localhost.localdomain    Started container
```

What's more, the IP addresses of the left and right containers are necessary for further steps. Similarly, executing `kubectl describe pod right` and `kubectl describe pod left` will show the IP addresses of the right container in `Annotations`. In this case, the IP address of the right container is 192.168.20.252 and the IP address of the left container is 192.168.10.252. This will be used in the next step. Up through now, the IP addresses of this topology are shown in Figure 5.5.

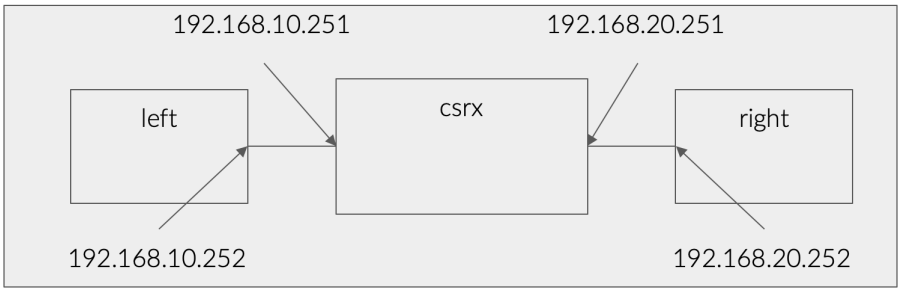

Figure 5.5 IP Addresses for Service Chaining Topology

Configure the cSRX

From the `kubectl describe pod csrx` command you are able to identify the IP addresses assigned to the cSRX from network-left and network-right. Using the `kubectl exec -it` command, it's possible to execute the cSRX bash and load configuration (the bold configurations could be different depending on your environment).

NOTE Interface ge-0/0/0 is mapped to eth1 while ge-0/0/1 is mapped to eth2 in the cSRX architecture. Also, in configuration line 3, the `black-url` value uses the right container IP address:

```
# kubectl exec -it csrx /bin/bash
root@csrx:/# cli
root@csrx> configure
Entering configuration mode

[edit]
root@csrx#

set interfaces ge-0/0/0 unit 0 family inet address 192.168.10.251/24
set interfaces ge-0/0/1 unit 0 family inet address 192.168.20.251/24
set security utm custom-objects url-pattern black-url value http://192.168.20.252/blacklist.html
set security utm custom-objects custom-url-category black-category value black-url
set security utm feature-profile web-filtering type juniper-local
set security utm feature-profile web-filtering juniper-local profile localprofile default permit
set security utm feature-profile web-filtering juniper-local profile localprofile category black-
category action block
set security utm feature-profile web-filtering juniper-local profile localprofile custom-block-message
"Access to this site is not permitted."
set security utm utm-policy my-utm-policy web-filtering http-profile localprofile
set security policies from-zone trust to-zone untrust policy http match source-address any
set security policies from-zone trust to-zone untrust policy http match destination-address any
set security policies from-zone trust to-zone untrust policy http match application junos-http
set security policies from-zone trust to-zone untrust policy http then permit application-services
utm-policy my-utm-policy
```

```
set security policies default-policy permit-all
set security zones security-zone trust interfaces ge-0/0/0.0 host-inbound-traffic system-services all
set security zones security-zone untrust interfaces ge-0/0/1.0 host-inbound-traffic system-services
all
set system root-authentication plain-text-password

(enter password twice)
csrx# commit
[edit]
root@csrx# exit
Exiting configuration mode

root@csrx> exit

root@csrx:/# exit
exit
```

Set Up Contrail

Next, open a browser and open Contrail using *https://<IP address of host>:8143*. Since the site generates its own SSL certificate, usually a warning message shows `Your connection is not Private`. Please ignore this warning and continue, or, add this site as an exception. The Contrail login interface appears: the default username is admin and the password is contrail123. Leave the domain empty, and then the Contrail main interface can be seen. The latest version (2005) of open source Contrail is called Tungsten Fabric.

Figure 5. 6 *Opening Contrail Page*

Choose Monitor > Networking > Networks > default-domain > k8s-default > k8s-network-left-pod-network, and two instances will show up in the plot on the webpage. Hovering the mouse on the instances will show which pod it is, and its interfaces, as shown in Figure 5.7.

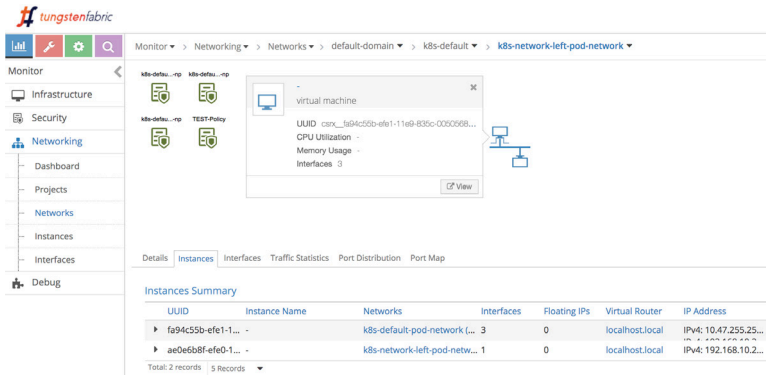

Figure 5.7 *Two Instances on Left Network*

There are four steps in making a service chain in Contrail:

1. Create a service template.
2. Using the service template, create a service instance.
3. Apply the service instance into a security policy.
4. Attach the security policy to the networks it applies to.

Step 1 is to create a service template. Navigate to the Configure > Services > Service Templates page and click on the + button to create a service template.

In the window that pops up in Figure 5.8:

- Give the service template a name.
- The Version is “v2”.
- Virtualization Type is “Virtual Machine”.
- Service Mode is “In-Network”.
- Service Type is “Firewall”.
- Then add two interfaces and choose “left” and “right” separately.
- Save the template.

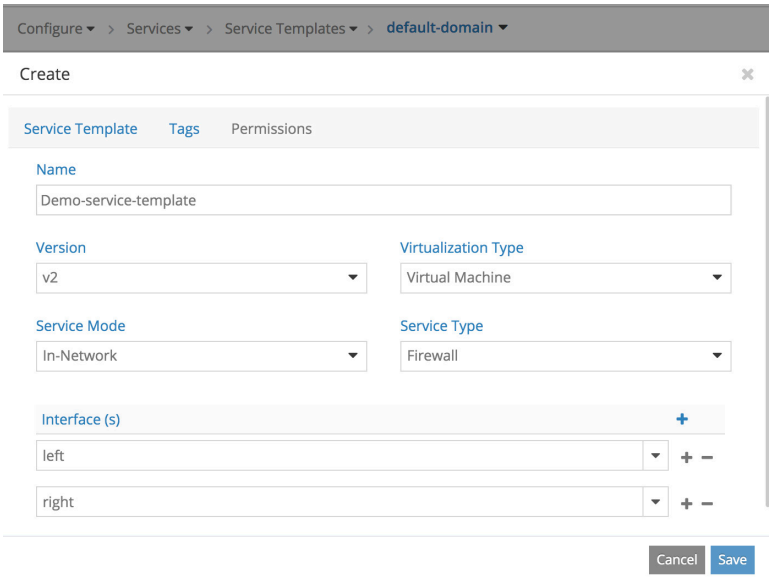

Figure 5.8 *Service Template*

To create a service instance for Step 2, navigate to the Configure > Services > Service Instances page. Here the domain will be automatically chosen to be k8s-default". Click on the + button.

In the next window, as shown in Figure 5.9:

- Give the service instance a name.
- Then choose the service template you just created as a Service Template in Figure 5.8.
- Map the Interface Type to the Virtual Network Name. Left to left-pod-network, and right to right-pod-network
- Subsequently, click on the triangle next to Port Tuples to extend it for more options. Again, map the left and right interface of the cSRX with the VM network.
- Click on Save.

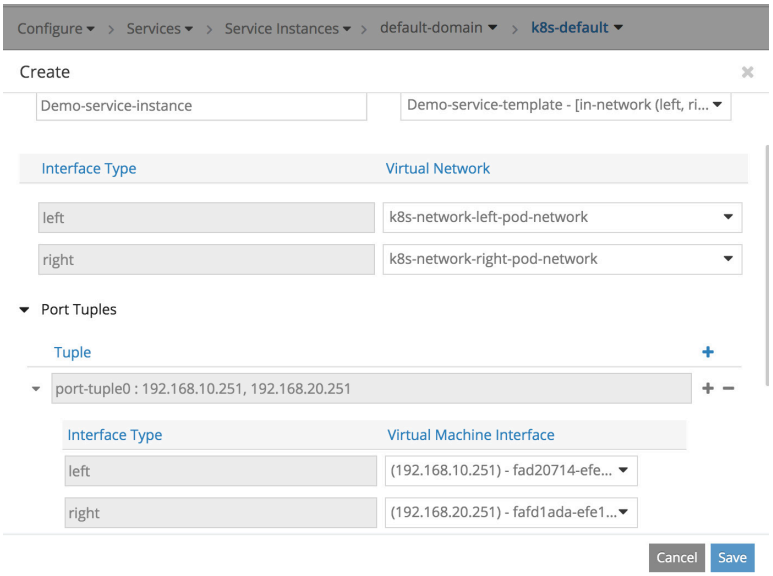

Figure 5.9 *Mapping the Left and Right Interface*

Step 3 is to create a networking policy. Navigate to Configure > Networking > Policies. Here the domain will automatically be chosen as "k8s-default". Click on the + button to create a networking policy.

In the window that pops up, shown in Figure 5.10:

- Give the policy a name, and click on the "+" sign in Policy Rule(s).
- In the "Source" and "Destination" dropdown list, expand "Network" and scroll to find *left-network* and *right-network* respectively.
- Click on "Services", then add the service instance created in Step 2.
- Save the security policy.

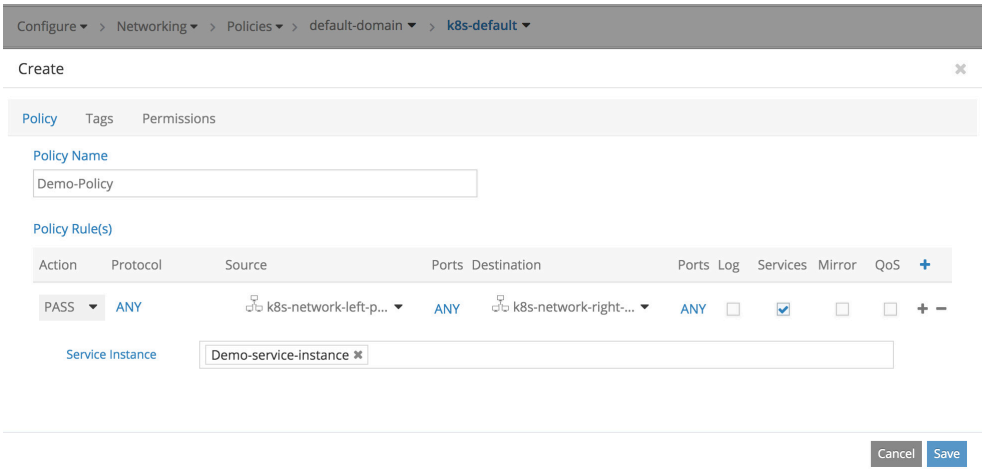

Figure 5.10 *Create the Policy*

Finally, to complete Step 4, navigate to Configure > Networking > Networks. Same as above, the domain is k8s-default. See Figure 5.11.

- For left network, press the gear on the end of the line to edit network.
- Add the security policy created in Step 3, and save the network.
- Repeat the same procedure for the right network.

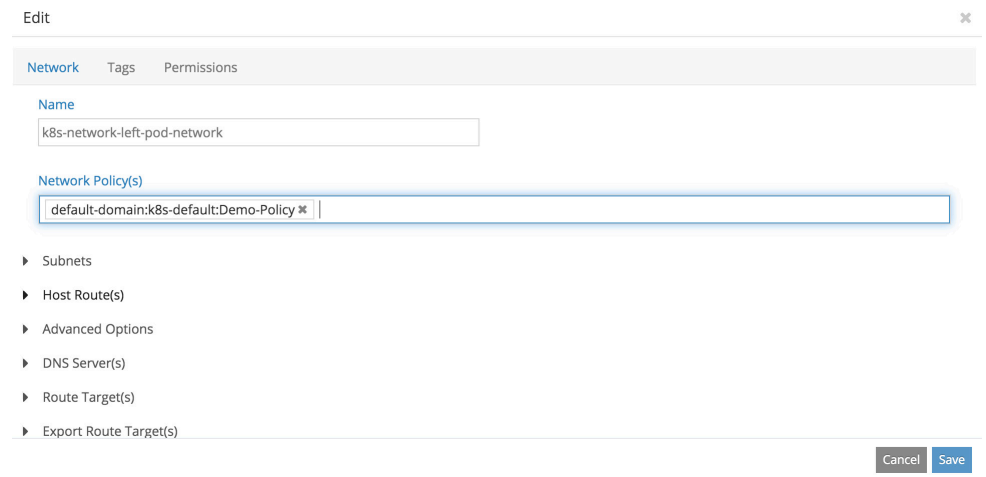

Figure 5.11 *Add Security Policy*

A service chain has been successfully created. Traffic flowing between the left and right networks should always pass by the cSRX and the packets will be examined by the cSRX policies. To verify this, open two terminals on the server. One executing on the left pod and the other executing on the cSRX to observe the traffic.

Verification

To verify that service chaining is working correctly, open two terminal windows: one is for the left container while the other one is for the cSRX:

```
(Window 1: To open left pod from CentOS host)# kubectl exec -it left /bin/bash
(Window 2: To open csrx pod from CentOS host)# kubectl exec -it csrx /bin/bash
(left pod) # ping 192.168.20.252
PING 192.168.20.252 (192.168.20.252) 56(84) bytes of data.
64 bytes from 192.168.20.252: icmp_seq=1 ttl=63 time=2.09 ms
64 bytes from 192.168.20.252: icmp_seq=2 ttl=63 time=0.504 ms
64 bytes from 192.168.20.252: icmp_seq=3 ttl=63 time=0.514 ms
...(Please keep pinging and do not stop it.)
(cSRX) # cli
(cSRX) > show security flow session
Session ID: 4115, Policy name: default-policy-logical-system-00/2, Timeout: 2, Valid
  In: 192.168.10.252/64 --> 192.168.20.252/85;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
  Out: 192.168.20.252/85 --> 192.168.10.252/64;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,

Session ID: 4116, Policy name: default-policy-logical-system-00/2, Timeout: 2, Valid
  In: 192.168.10.252/65 --> 192.168.20.252/85;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
  Out: 192.168.20.252/85 --> 192.168.10.252/65;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
...
```

This proves that traffic from the left pod to the right pod is passing by the cSRX and has been examined by security policies. Stop pinging on the left pod and try to `curl` the blacklist URL in the cSRX configuration:

```
(left pod) # curl "http://192.168.20.252/blacklist.html"
<HTML><HEAD><TITLE>Juniper Web Filtering</TITLE></HEAD><BODY>Access to this site is not
permitted.<br>CATEGORY: black-category REASON: BY_USER_DEFINED</br></BODY></HTML>
```

This verifies that the UTM web-filtering policy in the cSRX is taking effect. The user on the left pod is blocked by the cSRX using web filtering from reaching the UTM blacklist URL. Use these commands to check UTM's status and statistics:

```
root@csrx> show security utm web-filtering status
 UTM web-filtering status:
    Server status: Juniper local URL filtering

root@csrx> show security utm web-filtering statistics
 UTM web-filtering statistics:
    Total requests:                   2
    white list hit:                   0
    Black list hit:                   0
    Custom category permit:           0
    Custom category block:            2
...
```

Host-Based Firewall on Contrail

Since release 2003, Contrail Networking provides support for a host-based firewall feature which enables the creation of next generation firewalls using cSRX devices. the host-based firewall feature makes use of the cSRX's wire mode where the firewall instance does not change the packet format but just applies Layer 7 policies on the packet. Additionally, it uses tag-based policies to steer traffic. The cSRX, on Secure-Wire Mode, will run on top of vRouter in the worker nodes on Contrail. Kubernetes will be used to orchestrate cSRX instances on compute nodes.

Host-based firewall is different from service chaining. Service chaining works only in cases of inter-virtual network traffic and not for intra-virtual network traffic. The host-based firewall feature offers next-generation firewall functions for traffic originating and ending in the same virtual network as well as in different networks.

Setup Host-Based Firewall on Contrail Environment

In this setup, a cluster of three nodes is going to be implemented. One node works as a controller node and the other two are worker nodes. System requirements for the three nodes are:

- Three servers can be physical server or virtual machines
- Unique hostname, MAC address, and product_uuid for every node
- Eight CPU cores for controller node and six vCPUs for worker nodes
- 16 GB RAM on all three nodes
- 100 GB HDD on all three nodes

This topology is displayed in Figure 5.12.

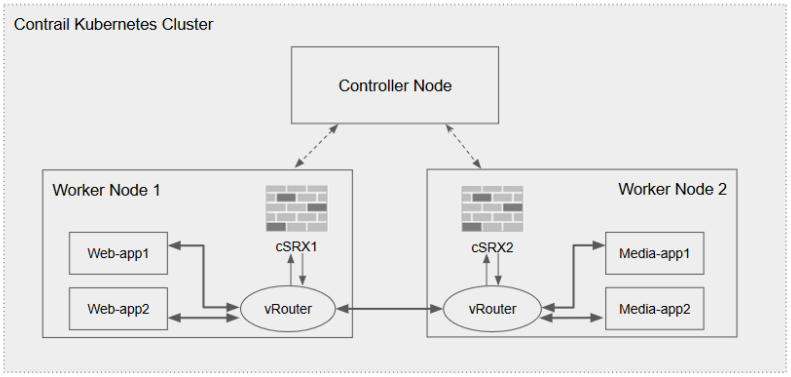

Figure 5.12 *Topology of Host-Based Firewall on Contrail*

In this topology, cSRX1 and cSRX2 sit in two worker nodes respectively. Web-app1 and Web-app2 are two pods inside Worker Node 1, and Media-app 1 and Media-app 2 are two pods inside Worker Node 2. Traffic between Web-app1 and Web-app2 will go through vrouter on Worker Node 1, therefore inspected by cSRX1. Traffic between Media-app 1 and Media-app 2 will go through vrouter on Worker Node 2, inspected by cSRX2. Traffic between the Web app and Media app will go through both vrouters, and inspected by both cSRX1 and cSRX2.

Set Up the Cluster

The cluster setup procedure is the same as the Contrail and Kubernetes Environment Setup section earlier in this chapter, and below are the working config/instances.yaml file for the three nodes. Download the ansible-deployer only to the controller node, then before running the scripts, please change the hostnames of the servers to three different names such as "controller", "workder1" and "worker2", so the Contrail ansible deployer is able to recognize different machines:

```
#config/instances.yaml
deployment:
    orchestrator: kubernetes
    deployer: contrail-ansible-deployer
provider_config:
  bms:
    ssh_pwd: <password>
    ssh_user: <username>
    ntpserver: 8.8.8.8
    domainsuffix: local
instances:
  server1:
    provider: bms
    ip: <Server IP>
    roles:
      config_database:
      config:
      control:
      analytics_database:
      analytics:
      webui:
      k8s_master:
      kubemanager:
  server2:
    provider: bms
    ip: <Server IP>
    roles:
      k8s_node:
      vrouter:
  server3:
    provider: bms
    ip: <Server IP>
    roles:
      k8s_node:
      vrouter:
global_configuration:
```

```
  CONTAINER_REGISTRY: opencontrailnightly
  REGISTRY_PRIVATE_INSECURE: True
contrail_configuration:
  CONTAINER_REGISTRY: opencontrailnightly
  CONTROLLER_NODES: <Server IP>
  CONTRAIL_VERSION: master-latest
  CLOUD_ORCHESTRATOR: kubernetes
  KUBERNETES_CLUSTER_PROJECT: {}
  CONFIG_NODEMGR__DEFAULTS__minimum_diskGB: 20
  DATABASE_NODEMGR__DEFAULTS__minimum_diskGB: 20
  CONFIG_DATABASE_NODEMGR__DEFAULTS__minimum_diskGB: 20
```

Then set up Contrail k8s environment with these three scripts:

```
(controller node) # ansible-playbook -e orchestrator=kubernetes -i inventory/ playbooks/configure_
instances.yml
(controller node) # ansible-playbook -e orchestrator=kubernetes -i inventory/ playbooks/install_k8s.
yml
(controller node) # ansible-playbook -e orchestrator=kubernetes -i inventory/ playbooks/install_
contrail.yml
```

The three nodes will be ready with Kubernetes and Contrail installed. To verify:

```
(Controller Node)# kubectl get nodes
NAME         STATUS     ROLES    AGE   VERSION
controller   NotReady   master   12m   v1.14.8
worker1      Ready      <none>   11m   v1.14.8
worker2      Ready      <none>   11m   v1.14.8
```

This output shows the Kubernetes cluster the three nodes can find each other. The controller in NotReady state is expected output because the CNI on the controller is disabled so that pods will only be scheduled on worker1 and worker 2.

Also check on Contrail web UI (https://Controller Node IP Address:8143, default login credential is admin/contrail123), both virtual routers' status should be "Up" on page Monitor > Infrastructure > Virtual Routers.

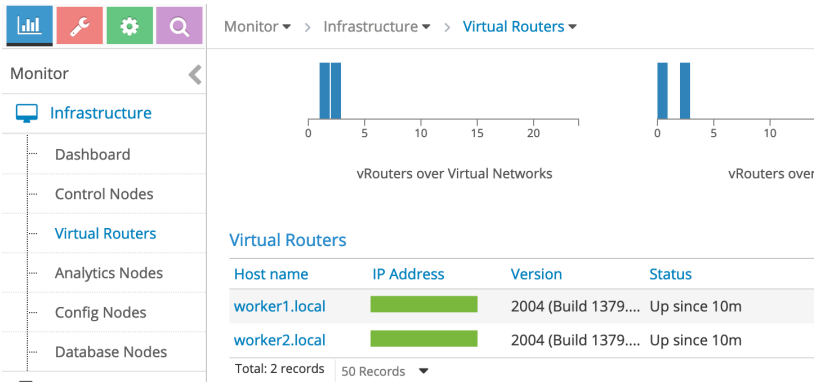

Figure 5.13 *Virtual Routers are on "Up" Status*

Here are the steps on how to create an HBF environment.

Step 1: To initiate, create a namespace. Use the YAML file in the repository to describe the properties of the namespace:

```
(Controller Node) # git clone https://github.com/csrx-dayone/contrail-kubernetes.git
(Controller Node) # cd contrail-kubernetes
(Controller Node) # cat create-namespace.yaml
#create_namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    "opencontrail.org/isolation" : "true"
    "opencontrail.org/ip_fabric_snat" : "true"
  name: hbf
```

Then create the namespace by running the `kubectl create` command:

```
(Controller Node) # kubectl create -f create_namespace.yaml
namespace/hbf created
```

To verify the namespace is created, run this command:

```
(Controller Node) # kubectl get namespaces hbf
NAME   STATUS   AGE
hbf    Active   1d
```

Step 2: Label the worker nodes for the host-based firewall (by default, host-based firewall instances run on all compute nodes). You can choose to run host-based firewall instances on specific compute nodes only by labeling them. In this topology, two worker nodes will have the cSRX connected to them to inspect intra-virtual network traffic as well as inter-virtual network traffic. Two worker nodes will be labeled:

```
(Controller Node) # kubectl label node worker1 type=hbf
node/worker1 labeled
(Controller Node) # kubectl label node worker2 type=hbf
node/worker2 labeled
```

To verify the nodes are successfully labeled, run:

```
(Controller Node) # kubectl get nodes --namespace=hbf --show-labels
NAME         STATUS     ROLES    AGE       VERSION       LABELS
Controller   NotReady    master   20d       v1.14.8       ...
worker1      Ready       <none>   20d       v1.14.8       ...,type=hbf
worker2      Ready       <none>   20d       v1.14.8       ...,type=hbf
```

Step 3: After that, create an hbs object in the previously created namespace. Use `docker exec` to access to the config_api Docker container, then clone a python file per instructions here:

```
(Controller Node) # docker exec -it config_api_1 bash
(config-api)[root@server /]$ yum install git -y
(config-api)[root@server /]$ git clone https://github.com/csrx-dayone/contrail-kubernetes.git
(config-api)[root@server /]$ cd contrail-kubernetes
```

```
(config-api)[root@server /]$ vi create_hbs.py
(Please change admin_password and api_node_ip accordingly.)
(config-api)[root@server /]$ python create_hbs.py
(config-api)[root@server /]$ exit
```

To verify this step is completed, access to the contrail web portal Setting> Config Editor, and search for "host-based-services". Click into host-based-services, project name as shown in Figure 5.14.

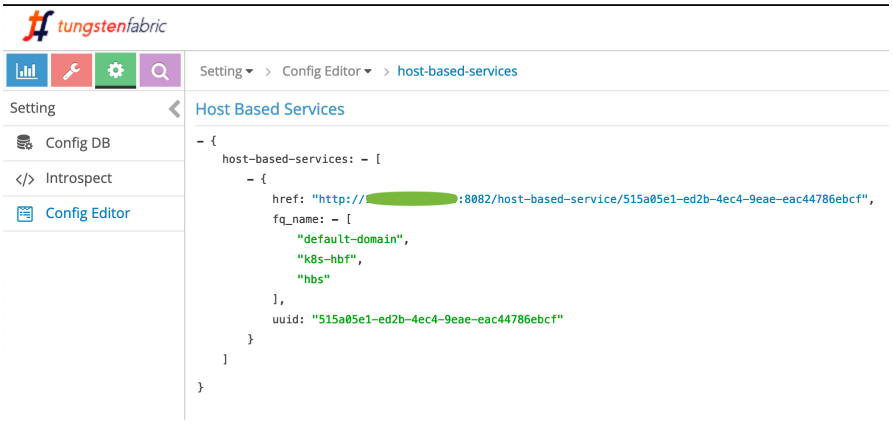

Figure 5.14 *Verify Host-based-Services Are Created*

Step 4: The next step is to create network attachment definitions, therefore Kubernetes virtual networks, and a daemonset for the host-based firewall instances. In Kubernetes, a daemonset ensures that all tagged nodes, as we did in Step 2 for worker1 and worker2, run a copy of a cSRX pod and a copy of the virtual networks. As nodes are added to the cluster, pods are added to the worker nodes. As nodes are removed from the cluster, those pods are garbage collected. Deleting a daemonset will clean up the pods it created.

The host-based firewall instance, i.e. the cSRX, has three interfaces: the traffic flows in to the left interface, firewall functions are performed on the packets, and traffic flows out of the right interface. The management interface is the default pod network. The YAML file needed in this step can be downloaded from the repository. Then use `kubectl` to create cSRX.

NOTE Please load the cSRX image in worker nodes otherwise Kubernetes will check Dockerhub for the image, which does not exist. As a result, it will lead to an error message while creating cSRX containers.

```
(Controller Node) # cd contrail-kubernetes
(Controller Node) # kubectl create -f  create_ds.yaml
networkattachmentdefinition.k8s.cni.cncf.io/left created
networkattachmentdefinition.k8s.cni.cncf.io/right created
daemonset.apps/hbf created
```

To verify the network attachment definitions, daemonset, and one cSRX are created on each node, run these commands:

```
(Controller Node) # kubectl get ds -n  hbf
NAME   DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
hbf    2         2         2       2            2           type=hbf        41s

(Controller Node) # kubectl get network-attachment-definitions -n hbf
NAME        AGE
left        48s
right       48s

(Controller Node) # kubectl get pods -n hbf -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP               NODE     NOMINATED NODE     READINESS
GATES
hbf-85gcb    1/1  Running      0       26m   10.47.255.250   worker1   <none>             <none>
hbf-p5rdh    1/1  Running      0       26m   10.47.255.249   worker2   <none>             <none>
```

Step 5: Load licenses and configurations to the two cSRX pods. This process can also be automatically deployed with ConfigMap starting from cSRX 20.2. For details please check Chapter 6.

```
(Controller Node) #  kubectl exec -it hbf-85gcb /bin/bash -n hbf
root@hbf-85gcb:/# cli
root@hbf-85gcb> (Please load configuration and license)
(Then repeat the same steps for cSRX pod hbf-p5rdh )
```

Step 6: Create workloads. Use the file create_workloads.yaml in the repository and create two web servers and two media servers in the topology. Each of the server belongs to different subnets. On the containers there are nginx and network utilities packets installed.

```
(Controller Node) # kubectl create -f  create_workloads.yaml
service/ssh created
service/ssh01 created
service/ssh02 created
service/ssh03 created
networkattachmentdefinition.k8s.cni.cncf.io/net01 created
networkattachmentdefinition.k8s.cni.cncf.io/net02 created
networkattachmentdefinition.k8s.cni.cncf.io/net03 created
networkattachmentdefinition.k8s.cni.cncf.io/net04 created
pod/web-app01 created
pod/web-app02 created
pod/media-app01 created
pod/media-app02 created
```

To verify that the services, networks, and pods are created successfully, run these commands:

```
(Controller Node) # kubectl get network-attachment-definitions -n hbf
NAME      AGE
left      35m
net01     2m35s
net02     2m35s
net03     2m35s
```

```
net04     2m35s
right      35m

(Controller Node) # kubectl get pods –n hbf
NAME            READY   STATUS    RESTARTS   AGE
hbf–85gcb       1/1     Running       0      50m
hbf– p5rdh      1/1     Running       0      50m
media–app01     1/1     Running       0      23s
media–app02     1/1     Running       0      23s
web–app01       1/1     Running       0      23s
web–app02       1/1     Running       0      23s
```

Also, on the Contrail Dashboard, networks are created and can be seen at Monitor > Networking > Networks > Default Domain > k8s-hbf. By clicking onto each of the networks, pods connected to them will show up.

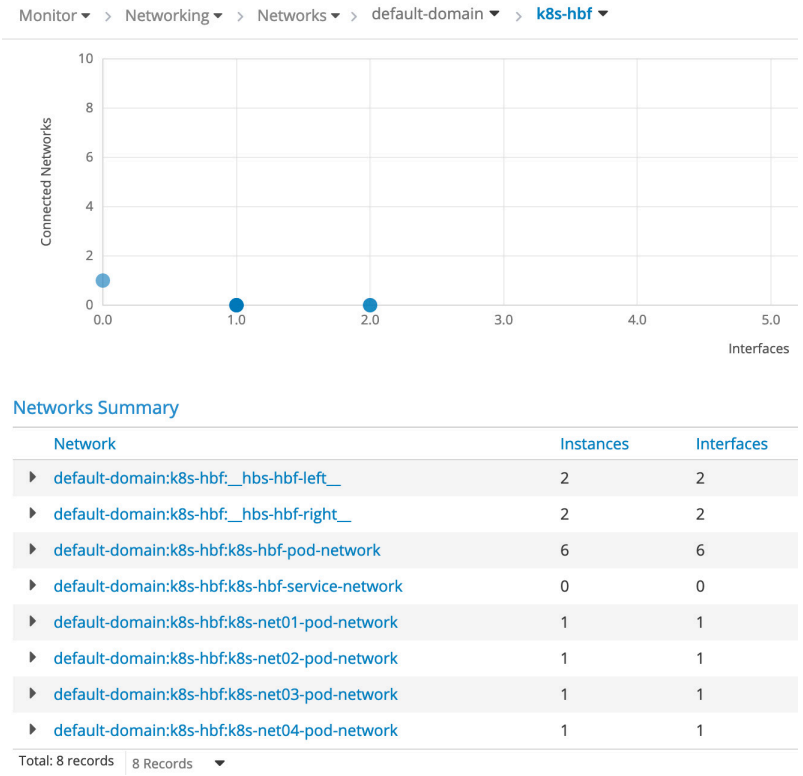

Figure 5.15 *Workload Networks are created on Contrail*

Step 7: Create network policies between the left and right interfaces through the Contrail web interface. Network policies are necessary for inter-virtual network traffic and intra-virtual network traffic.

On the Contrail web interface, navigate to Configure > Networking > Policies >

Default Domain > k8s-hbf, and create the following network policies.

1. Create Intra-Node Policy: In the policy created in this step, all traffic between k8s-net01-pod-network and k8s-net02-pod-network, also between k8s-net03-pod-network and k8s-net04-pod-network, is allowed. See Figure 5.16.

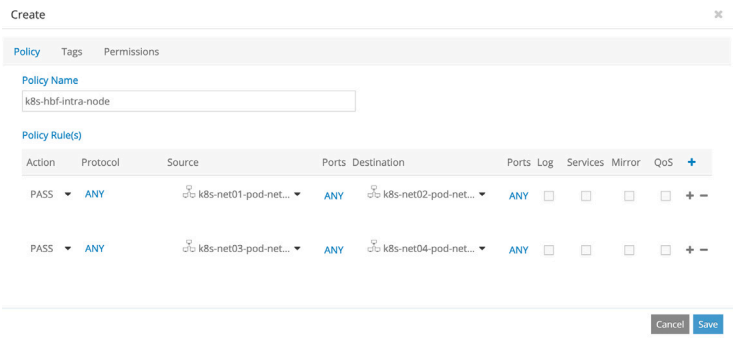

Figure 5.16 *Create Intra-Node Policy*

2. Create Inter-Node Policy: In the policy created in this step, all traffic between k8s-net01-pod-network and k8s-net03-pod-network, also between k8s-net02-pod-network and k8s-net03-pod-network is allowed. See Figure 5.17.

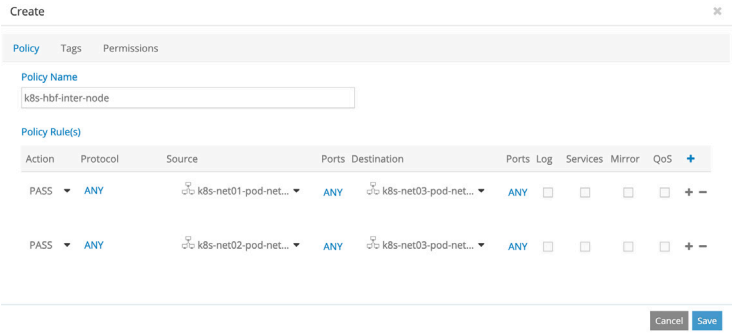

Figure 5.17 *Create Inter-Node Policy*

3. Apply the newly created security policies above to k8s-net01-pod-network, k8s-net02-pod-network, k8s-net03-pod-network and k8s-net04-pod-network in page Configure > Networking > Networks > Default Domain > k8s-hbf. The same procedures needs to be repeated for each of the network. See Figure 5.18.

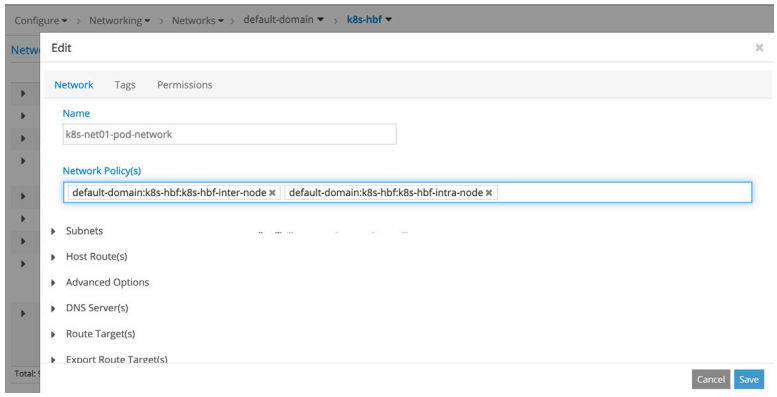

Figure 5.18 *Apply Created Policy to Networks*

Step 8: Tag the networks and enable host-based firewall.

1. On the Contrail Dashboard, navigate to Configure > Tags > Project Scoped Tags > Default Domain > k8s-hbf. Create these three tags:

- application=hbf-app
- site=node1
- site=node2

2. Associate the tags with the networks on page Configure > Networking > Networks > Default Domain > k8s-hbf. For k8s-net01-pod-network, k8s-net02-pod-network, k8s-net03-pod-network and k8s-net04-pod-network, tag them application=hbf-app, site=node1 for k8s-net01-pod-network and k8s-net02-pod-network, then site=node2 for k8s-net03-pod-network and k8s-net04-pod-network. See Figure 5.19.

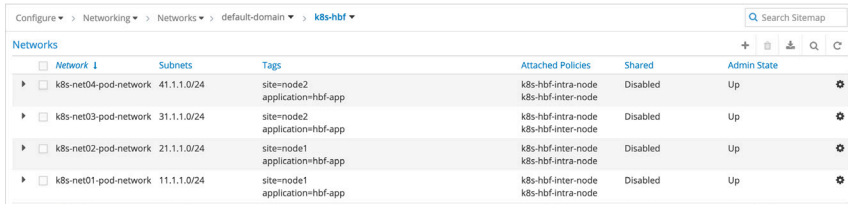

Figure 5.19 *Add Tags to Networks*

3. Navigate to Configure > Security > Project Scoped Policies > Default Domain > k8s_hbf and click on the "+" mark on top right of the page to create an application policy set.

- For the Application Policy Set to be created in this step, give it a name, for example “k8s-hbf-policy-intra”. In the application tag, choose “application=hbf-app”. Then click the Add Firewall Policy button.
- Give the firewall policy a name, for example “k8s-hbf-policy-intra”, then click Next to set rules for the policy.
- In Add Firewall Rules screen, set Action “pass”, End Point 1 “node1”, and End Point 2 “Node 1”. Click Save Policy, the policy for Node 1 intra-node policy is created.

Using similar procedures, policy set for Node 1 and Node 2 inter-node policy can be applied as well.

4. When all the policy sets are created, choose “Firewall Rules” in Project Scoped Policies, still in project k8s-hbf. The rules created in the application sets will appear. Click the policy which allows traffic from node1 to node1, then find the UUID of this rule. Copy the UUID.

5. Navigate to Setting > Config Editor > Firewall Rules. Search for firewall rules and click in. Search for UUID that was copied just now. The firewall rule details is seen. Click the Edit button on this firewall rule (Figure 5.20). Scroll down and expand the action list. There is an option “Host Based Service”. Enable it and save the firewall rule. The same option can also be applied to inter-node rule.

Figure 5.20 *Enable Host Based Service for the Firewall Rule*

To verify the host-based service for the rule is successfully enabled, two terminal can be opened to observe the traffic. Access web-app01 and cSRX on Node 1 on each of the window. Ping web-app02 from web-app01. Keep the ping and observe flow session on cSRX on Node 1:

```
(Controller Node, Window 1) # kubectl exec -it web-app01 /bin/bash -n hbf
root@web-app01:/# ping 21.1.1.252
PING 21.1.1.252 (21.1.1.252) 56(84) bytes of data.
```

```
64 bytes from 21.1.1.252: icmp_seq=1 ttl=63 time=2.57 ms
64 bytes from 21.1.1.252: icmp_seq=2 ttl=63 time=0.510 ms
64 bytes from 21.1.1.252: icmp_seq=3 ttl=63 time=0.339 ms
(Do not stop the ping)
(Controller Node, Window 2) # kubectl exec –it hbf-85gcb /bin/bash –n hbf
root@hbf-85gcb:/# cli
root@hbf-85gcb> show security flow session
Session ID: 13, Policy name: u2t/4, Timeout: 2, Valid
  In: 11.1.1.252/164 --> 21.1.1.252/13;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 98,
Out: 21.1.1.252/13 --> 11.1.1.252/164;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 98,

Session ID: 14, Policy name: u2t/4, Timeout: 2, Valid
  In: 11.1.1.252/164 --> 21.1.1.252/14;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 98,
  Out: 21.1.1.252/14 --> 11.1.1.252/164;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 98,
...
```

You can see that the traffic between web-app01 and web-app02 has been passing by cSRX and host-based service has been applied successfully. Layer 7 policies can be applied to ther cSRX, and more dynamic and effective traffic control can be accomplished between inter-virtual networks, as well as intra-virtual networks.

Chapter 6

cSRX on Native Kubernetes

Starting with version 20.3, cSRX integration with Multus CNI is supported. Multus CNI is an open source container network interface (CNI) plugin for Kubernetes that enables attaching multiple network interfaces to pods. Multus supports SRIOV, DPDK, OVS-DPDK, and VPP workloads in Kubernetes with both cloud native and NFV based applications in Kubernetes.

For cSRX on Kubernetes users, now there are two choices of CNI plugins: Juniper Contrail Networking or Intel Multus. Juniper Contrail Networking provides more powerful functions beyond basic networking management such as security policy, host-based service, and real-time monitoring; however it requires the node servers on the cluster to have large computational power and memory. Contrail is mainly used for large enterprises and datacenters for fabric management. On the other hand, Multus provides management functions for basic Kubernetes networks and the system requirements are relatively lower. Moreover, cSRX is able to support native Kubernetes features like replica sets, scaling, and ingress function. With Kubernetes and Multus, cSRX can be applied to DevOps environment flexibly for workload protection.

For detail descriptions on Kubernetes and Multus features, you can go to https://kubernetes.io and https://github.com/intel/multus-cni for more information.

Requirements for Deploying cSRX on Kubernetes

The requirements for deploying a cSRX container in a Kubernetes (Controller and Worker) node are:

- Nodes can be baremetal servers or virtual machines
- Nodes running Ubuntu 16.04+, CentOS 7 or Red Hat Enterprise Linux (RHEL) 7
- Docker Engine 1.9 or later installed on all nodes
- At least 2 vCPUs per machine
- 4 GiB or more of RAM per machine
- 50 GB hard drive per machine

Installation of Kubernetes Cluster and Multus using kubeadm

In this section a cluster of three nodes is going to be installed. The three nodes can be either virtual machines or on-prem servers. The three nodes need to reach the Internet and be able to ping each other. Hostnames of the three names need to be different. As an example, the three nodes have a clean Ubuntu 16.04 installed. Their hostnames are set to "controller", "worker1", and "worker2", and their IP addresses are 192.168.189.149, 192.168.189.150, and 192.168.189.151 respectively.

The first step is to install kubeadm: set the nodes' host files, disable swap, and install kubeadm.

1. Write three nodes' IP addresses in the three machines /etc/hosts files:

```
(Controller) # vi /etc/hosts
(Add these lines in the file)
192.168.189.149  controller
192.168.189.150  worker1
192.168.189.151  worker2
(Repeat this step for worker1 and worker2 nodes.)
```

2. After editing the three nodes' host file, install Docker for the three nodes. After the installation is complete, start the Docker service and enable it to launch every time at system boot:

```
(Controller) # apt-get install docker.io -y
(Controller) # systemctl start docker
(Controller) # systemctl enable docker
Synchronizing state of docker.service with SysV init with /lib/systemd/systemd-sysv-install...
Executing /lib/systemd/systemd-sysv-install enable docker
```

(Repeat this step for worker1 and worker2 nodes.)

3. Now disable swap on three nodes:

```
(Controller) # swapoff -a
```

(Repeat this step for worker1 and worker2 nodes.)

4. When all the preparation steps are complete, the machines are ready to have kubeadm installed:

```
(Controller) # apt install -y apt-transport-https curl
(Controller) # curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
OK
(Controller) # cd /etc/apt/
(Controller) # vi sources.list.d/kubernetes.list
(Paste the line below to the file and save the file)
deb http://apt.kubernetes.io/ kubernetes-xenial main
(Controller) # apt-get update -y
(Controller) # apt-get install -y kubeadm kubelet kubectl
```

(Repeat this step for worker1 and worker2 nodes:)

The second step is to initiate the cluster on the Controller node. All procedures described in this step are done only on the controller node. Here the "flannel" virtual network is going to be applied. Flannel is a virtual network that gives a subnet to each host for use with container runtimes. It is for distributing IP addresses in this CIDR to pods in the cluster, and the flannel virtual network's CIDR is hardcoded 10.244.0.0/16. After the cluster has been initiated, the command line will show instructions for upcoming procedures. User can either follow the command line instructions or use the below commands:

```
(Controller) # kubeadm init --pod-network-cidr=10.244.0.0/16
...
Your Kubernetes control-plane has initialized successfully!
...
```

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join 192.168.189.149:6443 --token xxx \
   --discovery-token-ca-cert-hash sha256:xxx
```

Please copy the above command in a text editor. It will be useful in later steps.

```
(Controller) # export KUBECONFIG=/etc/kubernetes/admin.conf
(Controller) # kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
Documentation/kube-flannel.yml
```

Wait for a minute and check the status of the Kubernetes cluster. It is expected to see that master node is in "Ready" state.

```
(Controller) # kubectl get nodes
NAME          STATUS   ROLES      AGE      VERSION
controller     Ready    master    1m50s    v1.19.2
```

The third step is to let worker nodes join the initiated cluster. It's the command that was just copied from the controller node, and it requires root privilege on each of the worker nodes. When the commands have completed, all three nodes should show the "Ready" state on the controller node.

```
(Worker1) # kubeadm join 192.168.189.149:6443 --token xxx \
>    --discovery-token-ca-cert-hash sha256:xxx
```

This node has joined the cluster:

- Certificate signing request was sent to apiserver and a response was received.
- The Kubelet was informed of the new secure connection details.

Run `kubectl get nodes` on the control plane to see this node join the cluster.

(Repeat the same procedure for Worker2)

```
(Controller) # kubectl get nodes
NAME          STATUS   ROLES    AGE    VERSION
controller    Ready    master   7m50s  v1.19.2
worker1       Ready    <none>   2s     v1.19.2
worker2       Ready    <none>   11s    v1.19.2
```

The forth step is to deploy Multus plugin on the cluster. This should be done on the Controller node.

```
(Controller) # git clone https://github.com/intel/multus-cni.git && cd multus-cni
(Controller) # cat ./images/multus-daemonset.yml | kubectl apply -f -
# kubectl get pods --all-namespaces | grep -i multus
kube-system   kube-multus-ds-amd64-2ddht         1/1    Running   0        20m
kube-system   kube-multus-ds-amd64-sqd2q         1/1    Running   0        20m
kube-system   kube-multus-ds-amd64-v27j8         1/1    Running   0        20m
```

When the cluster has been successfully deployed, it's time to spin up a simple network with your cSRX.

Kubernetes Concepts

For better understanding of applying cSRX on Kubernetes environment, here is some basic Kubernetes concepts for what's going to be implemented in later examples.

A *namespace* is a closed environment in Kubernetes. Namespaces are used mostly for dividing cluster resources between multiple users. If some resources are created within some specific namespace, all users will not be able to access it until they specify a namespace.

A *pod* is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers. If there are multiple containers in the same pod, they will be sharing one IP address, as well as sharing filesystem volumes.

A *deployment* provides declarative updates for pods and ReplicaSets. In the deployment specification, the image of the pod containers, their versions, and how many replicas there are, are all described. Deployment is responsible for the scaling up or scaling down of an application. When one of the replicas breaks down accidentally, a new pod with the same settings will boot up automatically. Pods in the deployment have their IP addresses released and distributed by flannel, i.e. 10.244.0.0/16, in our current setup.

A *service* exposes a group of pods to be accessed from the network. A service is responsible for forwarding internal or external requests to its corresponding deployments. A service also works as an automatic load balancer when there are multiple pods that serve the same function. The types of services are ClusterIP(default), NodePort, LoadBalancer, and ExternalName.

Figure 6.1 is an illustration of Kubernetes nodes, pods, deployments and services from Kubernetes official documentation: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/.

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. An ingress may be configured to give services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting.

NOTE Demos in this chapter were developed by Nancy Yu. Appreciations to Nancy for sharing her demo ideas and procedures.

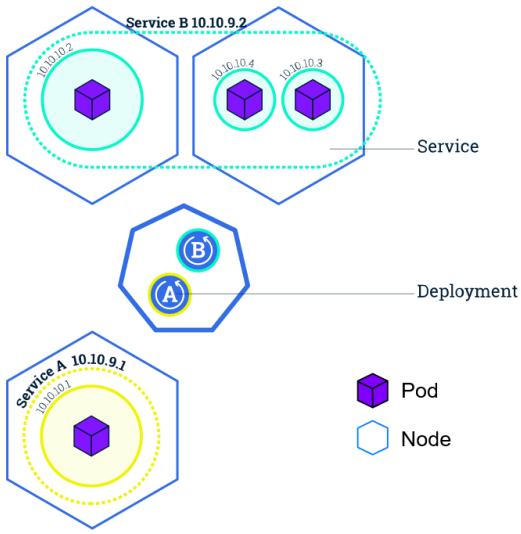

Figure 6.1 *Kubernetes node, pod, deployment and services*

Deploy cSRX in a network

The Container Network Interface (CNI) is a network plugin specification for configurating network interfaces in containers. The cSRX supports the Multus CNI plugin which enables attaching multiple network interfaces to the cSRX and pods.

To deploy cSRX in a network, you will need to define a network-attachment-definition file which is a JSON-formatted CNI configuration. To define a network attachment definition in a cluster with Multus, there is some syntax difference with defining a network attachment definition in Contrail. In the network-attachment-definition file, interface name, network name, network type, and its IPAM can be defined. If the type of network-attachment is a bridge, you can connect the interface to an existing bridge, or create one bridge if it doesn't exist. You can also assign an IP address to a bridge, turning it into a gateway for other pods. For detailed explanation of parameters in a bridge, please refer to Multus documentation: https://github.com/containernetworking/plugins/tree/master/plugins/main/bridge .

One of cSRX's features starting at Junos 20.2 is that initial configurations and license can be loaded directly with a ConfigMap. In order to notify cSRX to read ConfigMap for initial setup, you also need to send the environment variable CSRX_LICENSE_FILE and CSRX_JUNOS_CONFIG in the cSRX deployment YAML file.

Example 1: Passing Traffic Between Two Hosts with a Pre-configured cSRX

Here is an example of an east-west traffic model deploying a network consist of two nginx nodes, two bridges, and a cSRX.

To run the example in the part, you will need to obtain the cSRX image and license beforehand. Readers can download a trial license from Juniper official website: https://www.juniper.net/us/en/dm/csrxtrial/. Readers can also download the yaml files from the book's GitHub and follow the instruction below to run the exercise. The three yaml files consist of: ConfigMap, Network attachment definition, and Pod definition. You will need to paste the license under statement csrx_license in the configmap.yaml file.

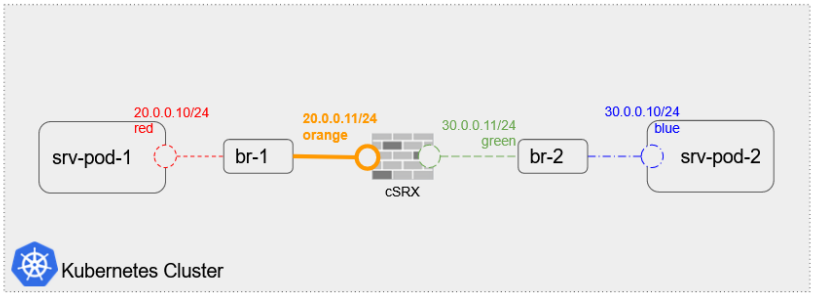

Figure 6.2 *Build a network in Kubernetes Cluster*

The first step is to clone the YAML files from the repository, and add cSRX license information in the configmap.yaml file.

```
(Controller) # git clone https://github.com/csrx-dayone/kubernetes.git
(Controller) # cd kubernetes
```

(Open configmap.yaml file with your favorite text editor and add license information in. Four spaces in front of each line.)

The second step is to create the configmap and network-attachment-definitions for the pods. There are four network-attachment-definitions, i.e. network interfaces, in the deployment, and they are marked red, orange, green and blue in the Figure 6.2. Red and orange are connected to bridge br-1, and green and blue are connected to bridge br-2.

```
(Controller) # kubectl create -f configmap.yaml
configmap/csrx-config-map created
(Controller) # kubectl create -f nad.yaml
networkattachmentdefinition.k8s.cni.cncf.io/red created
networkattachmentdefinition.k8s.cni.cncf.io/orange created
networkattachmentdefinition.k8s.cni.cncf.io/green created
networkattachmentdefinition.k8s.cni.cncf.io/blue created
```

Then tag the worker node you would like to deploy this network, and on that worker node allow forwarding traffic for the newly created bridges br-1 and br-2:

```
(Controller) # kubectl label node worker1 node=worker1
(Worker1)# iptables -A FORWARD -i br-1 -o br-1 -j ACCEPT
(Worker1)# iptables -A FORWARD -i br-2 -o br-2 -j ACCEPT
```

Finally, create pods srv-pod-1, csrx and srv-pod-2:

```
(Controller) # kubectl create -f pods.yaml
pod/srv-pod-1 created
pod/csrx created
pod/srv-pod-2 created
```

To verify that the pods have been created successfully, run:

```
(Controller) # kubectl get pods
NAME       READY   STATUS      RESTARTS    AGE
csrx       1/1     Running     0           6m
```

```
srv-pod-1    1/1         Running     0               6m
srv-pod-2    1/1         Running     0               6m
```

Log in to the Linux host and generate traffic between the two hosts:

```
(Controller) # ~# kubectl exec -it srv-pod-1 /bin/bash
(srv-pod-1) # ip a
…
inet 20.0.0.10/24
(srv-pod-1) # ping 30.0.0.10
PING 30.0.0.10 (30.0.0.10) 56(84) bytes of data.
64 bytes from 30.0.0.10: icmp_seq=1 ttl=63 time=0.816 ms
64 bytes from 30.0.0.10: icmp_seq=2 ttl=63 time=0.418 ms
```

Keep pinging srv-pod-2 from srv-pod-1, or the other way around. Log in to the cSRX firewall and verify the traffic between two Linux hosts has passed the cSRX in another terminal window:

```
(Controller) # kubectl exec -it csrx /bin/bash
root@csrx:/# cli
root@csrx> show system license
```

(System license should be applied automatically.)

```
root@csrx> show configuration
```

(Configuration should be applied automatically.)

```
root@csrx> show security flow session
Session ID: 12, Policy name: default-policy-logical-system-00/2, Timeout: 2, Valid
  In: 20.0.0.10/65 --> 30.0.0.10/1;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
  Out: 30.0.0.10/1 --> 20.0.0.10/65;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,

Session ID: 13, Policy name: default-policy-logical-system-00/2, Timeout: 2, Valid
  In: 20.0.0.10/65 --> 30.0.0.10/2;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
  Out: 30.0.0.10/2 --> 20.0.0.10/65;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,

Session ID: 14, Policy name: default-policy-logical-system-00/2, Timeout: 4, Valid
  In: 20.0.0.10/65 --> 30.0.0.10/3;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
  Out: 30.0.0.10/3 --> 20.0.0.10/65;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
Total sessions: 3
```

The network setup is successful when there are sessions showing up on the cSRX. To clean up the setup, run:

```
(Controller) # kubectl delete -f pods.yaml
pod "srv-pod-1" deleted
pod "csrx" deleted
pod "srv-pod-2" deleted
(Controller) # kubectl delete -f nad.yaml
networkattachmentdefinition.k8s.cni.cncf.io "red" deleted
networkattachmentdefinition.k8s.cni.cncf.io "orange" deleted
networkattachmentdefinition.k8s.cni.cncf.io "green" deleted
networkattachmentdefinition.k8s.cni.cncf.io "blue" deleted
(Controller) # kubectl delete -f configmap.yaml
configmap "csrx-config-map" deleted
```

Example 2: Web Server Protection

This example creates a service with the nginx web server. The cSRX will use a forwarding policy and pass the incoming traffic to the back end web server. Finally, we'll scale up the cSRX or nginx web service using the imperative kubectl scale commands.

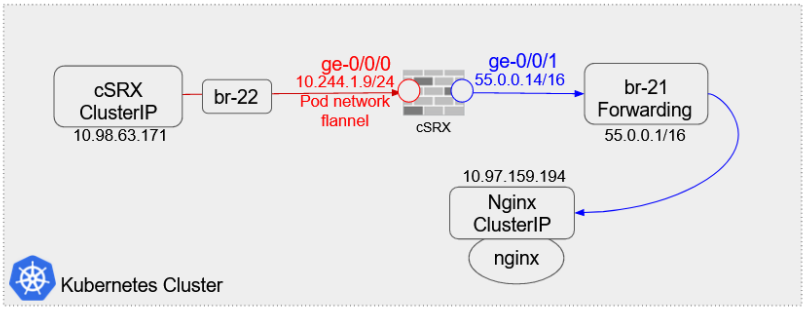

Figure 6.3 *Protect the nginx deployment on Kubernetes*

1. Create the network-attachment-definition with Multus bridges br-21 and br-22:

```
(Controller) # git clone https://github.com/csrx-dayone/k8s-demo.git
(Controller) # cd k8s-demo
(Controller) # kubectl create -f network.yaml
networkattachmentdefinition.k8s.cni.cncf.io/nw2-1 created
networkattachmentdefinition.k8s.cni.cncf.io/nw2-2 created
```

2. Create a web service with image nginx:

```
(Controller) # kubectl create -f run-my-nginx.yaml
deployment.apps/my-nginx created
# kubectl expose deployment/my-nginx
service/my-nginx exposed
```

3. Get the service ClusterIP once backend service is created:

```
(Controller) # kubectl get service
NAME         TYPE        CLUSTER-IP         EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1          <none>        443/TCP   8d
my-nginx     ClusterIP   10.97.159.194      <none>        80/TCP    9h
```

4. Modify configmap.yaml to change csrx baseline configuration to forward traffic to backend service. Replace "10.97.159.194" with your nginx service IP. Then add cSRX license information starting from Line 7. Please use four spaces indentation for each line.

```
# set routing-options static route 10.97.159.194/32 next-hop 55.0.0.1/32
# set security nat destination pool forward-pool address 10.97.159.194/32
```

5. Create the configMap which includes the cSRX license and the cSRX baseline configuration:

```
(Controller)# kubectl create -f configmap.yaml
configmap/csrx-config-map created
```

6. Create csrx deployment and service. Please make sure the environment variable CSRX_MGMT_PORT_REORDER should be set to “yes”. This is to bind the last interface as MGMT port.

```
(Controller)# kubectl create -f csrx.yaml
deployment.apps/csrx1 created
service/csrx1 created
```

7. Check the created resource status and verify that the license and configuration are applied on the cSRX. Remember which node has the cSRX located.

```
(Controller)# kubectl get all -o wide
NAME                              READY   STATUS    RESTARTS   AGE    IP             NODE …
pod/csrx1-77d874d5b9-qtqhj        1/1     Running   0          116s   10.244.1.18    worker1
pod/my-nginx-5b56ccd65f-dsn7l     1/1     Running   0          10h    10.244.2.3     worker2

NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE …
service/csrx1           ClusterIP   10.98.63.171     <none>        80/TCP     116s
service/kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP    8d
service/my-nginx        ClusterIP   10.97.159.194    <none>        80/TCP     10h

NAME                          READY   UP-TO-DATE   AVAILABLE   AGE …
deployment.apps/csrx1         1/1     1            1             116s
deployment.apps/my-nginx      1/1     1            1             10h

NAME                                        DESIRED   CURRENT   READY   AGE …
replicaset.apps/csrx1-77d874d5b9            1         1         1       116s
replicaset.apps/my-nginx-5b56ccd65f         1         1         1       10h

# kubectl exec -it csrx1-77d874d5b9-qtqhj /bin/bash
root@csrx1-77d874d5b9-qtqhj:/# cli
root@csrx1-77d874d5b9-qtqhj> show system license
(Please make sure the license has been applied successfully on the pod.)
root@csrx1-77d874d5b9-qtqhj> show configuration | display set
set version 20200819.234446.1_builder.r1131461
set interfaces ge-0/0/0 unit 0 family inet address 10.244.1.18/24
set interfaces ge-0/0/1 unit 0 family inet address 55.0.0.16/16
set routing-options static route 10.97.159.194/32 next-hop 55.0.0.1/32
set routing-options static route 0.0.0.0/0 next-hop 10.244.1.1/32
set routing-options static route 10.244.0.0/16 next-hop 10.244.1.1/32
set routing-options static route 10.244.1.0/24 next-hop 0.0.0.0/32
set security nat source rule-set s-forward from zone trust
set security nat source rule-set s-forward to zone untrust
set security nat source rule-set s-forward rule s-forward-rule match source-address 0.0.0.0/0
set security nat source rule-set s-forward rule s-forward-rule then source-nat interface
set security nat destination pool forward-pool address 10.97.159.194/32
set security nat destination pool forward-pool address port 80
set security nat destination rule-set forword from zone trust
set security nat destination rule-set forword rule forward-rule match destination-address 0.0.0.0/0
set security nat destination rule-set forword rule forward-rule match destination-port 80
set security nat destination rule-set forword rule forward-rule then destination-nat pool forward-pool
```

```
set security policies default-policy permit-all
set security zones security-zone trust host-inbound-traffic system-services all
set security zones security-zone trust host-inbound-traffic protocols all
set security zones security-zone trust interfaces ge-0/0/0.0
set security zones security-zone untrust host-inbound-traffic system-services all
set security zones security-zone untrust host-inbound-traffic protocols all
set security zones security-zone untrust interfaces ge-0/0/1.0
root@csrx1-77d874d5b9-qtqhj> exit
root@csrx1-77d874d5b9-qtqhj:/# exit
exit
(controller) #
```

8. Add iptables forward accept rules for Multus-created bridge br-21 on the node where cSRX has been implemented:

```
(Worker1) # iptables -A FORWARD -i br-21 -o br-21 -j ACCEPT
```

9. Config br-21 IP address as cSRX gateway on the node where cSRX is implemented. When br-21 was assigned an IP address, it turns into a gateway for the container cSRX according to Figure 6.3.

```
(Worker1) # ifconfig br-21 55.0.0.1/16
```

Now the setup of the cSRX in front of web server is ready. If the cSRX has license and configuration applied, the NAT function will be working as expected and you can verify it by accessing the ClusterIP of cSRX, displayed in the output of kubectl get all command:

```
(controller) # curl 10.98.63.171
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
…
```

That verifies that the contents of the web server has been forwarded to the port 80 of the cSRX successfully!

Ingress Controller

Here's a little more on the cSRX protecting web server scenario just discussed... when the ClusterIP service of the cSRX has been successfully deployed, access to the cSRX port 80 is still within the cluster. If you are outside the cluster and would like to access the web server, implementing an ingress controller is necessary.

The ingress controller needs a specific namespace, service account, cluster role bindings, configmaps, etc. These can be created by running ingress-roles.yaml in the repository:

```
(Controller) # kubectl create -f ingress-roles.yaml
namespace/ingress-nginx created
configmap/nginx-configuration created
```

```
configmap/tcp-services created
configmap/udp-services created
serviceaccount/nginx-ingress-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-role created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-role-nisa-binding created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-clusterrole-nisa-binding created
```

The next step is to create a service of Type NodePort to expose the nginx controller deployment outside the cluster:

```
(Controller) # kubectl create -f ingress-service.yaml
deployment.apps/nginx-ingress-controller created
service/ingress-nginx created
```

And the last step is to create the ingress, which binds the cSRX service with the ingress controller:

```
(Controller) # kubectl create -f ingress-csrx.yaml
ingress.networking.k8s.io/csrx-ingress created
```

To verify the ingress controller has implemented, on the controller node, list all resources created for namespace ingress-nginx. A pod, a service, a deployment and a replicaset has been created:

```
(Controller) # kubectl get all -n ingress-nginx -o wide
NAME                                READY   STATUS    RESTARTS  AGE   IP                NODE
NOMINATED NODE   READINESS GATES
pod/nginx-ingress-controller-…    1/1     Running   0         9h    192.168.189.151   worker2
<none>           <none>

NAME                    TYPE       CLUSTER-IP        EXTERNAL-IP          PORT(S)
AGE   SELECTOR
service/ingress-nginx   NodePort   10.105.19.182     <none>               80:32256/TCP,443:30557/TCP
9h    app.kubernetes.io/name=ingress-…

NAME                                       EADY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS
deployment.apps/nginx-ingress-controller   1/1    1            1           9h    nginx-ingress-
controller  …

NAME                                                 DESIRED   CURRENT   READY   AGE   CONTAINERS
replicaset.apps/nginx-ingress-controller-57c4b94788   1         1         1       9h    nginx-ingress-
controller   …
```

From this information, you can see that the ingress controller pod has been implemented on node worker2. Using a host outside of the cluster, browse on worker2's IP address , and the default nginx page will show up. This is because the service ingress-nginx has forwarded the user's access request to the cSRX service.

For the ingress controller, you can also define rules and configure DNS with the path for redirecting requests to different services. Please refer to Kubernetes official documentation for more details.

Scale Up the cSRX and Web Server

After implementing the web server workload on a Kubernetes cluster as in the topology above, the DevOps team needs to deal with the situation when there are increased number of visitors for the web server contents. If visitors keep increasing, it will cause increased CPU usage and memory consumption for both the cSRX and the web server, which will cause slow response for visitors. So we need to scale up the workloads. It can be implemented simply with two commands:

```
(Controller) # kubectl scale deploy csrx1 --replicas=3
deployment.apps/csrx1 scaled
(Controller) # kubectl scale deploy my-nginx --replicas=3
deployment.apps/my-nginx scaled
```

One extra step is to add some iptables forward accept rules for Multus-created bridge br-21, and config br-21 IP addresses as cSRX gateways for all other nodes when replicas of the cSRX have spun up. Commands can be found in Steps 8 and 9 in this chapter's section, Implementing cSRX in Front of a Webserver.

When those procedures have been completed, the cSRX service and the nginx service will be able to replicate themselves to three copies and distribute visitor hits to all pods averagely.

To see the replicas have been created successfully, run:

```
(Controller) # kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
csrx1-68b79bbb7-7vkmx      1/1    Running   0         4m25s
csrx1-68b79bbb7-hc9wx      1/1    Running   0         11m
csrx1-68b79bbb7-sv8v2      1/1    Running   0         4m25s
my-nginx-5b56ccd65f-92nz6   1/1    Running   0         4m16s
my-nginx-5b56ccd65f-lqq6f   1/1    Running   0         4m16s
my-nginx-5b56ccd65f-wd26k   1/1    Running   0         19m
```

To see that the workload has been distributed to the three pods, it can be verified by opening three windows for each replica of the cSRX. Keep visiting the cSRX pod and observe security flow sessions. Just in case one of the replica pods has been accidently terminated, another replica will start running automatically to keep the number of replica pods to three all the times. This is all handled automatically by Kubernetes deployments.

Chapter 7

Troubleshooting for cSRX

Let's verify the cSRX status on Docker and then troubleshoot a little.

Verify cSRX

To see if the cSRX has booted up normally, use these commands on the Docker host to gather information:

```
# docker ps
CONTAINER ID     IMAGE          COMMAND                CREATED
464560854dcf     csrx:latest    "/etc/rc.local init"   28 minutes ago

STATUS       PORTS          NAMES
Up 28 minutes  22/tcp,830/tcp   csrx-example
```

With `docker logs`, use these steps to boot up the cSRX. If the cSRX did not boot up normally, the output could be different. In this case, you can always try to stop the cSRX container and start again:

```
# docker logs csrx-example
Copyright (c) 1996-2019, Juniper Networks, Inc.
All rights reserved.

info: setup srxpfe config
info: setting environment variables ...
export CSRX_PACKET_DRIVER=interrupt
export CSRX_FORWARD_MODE=routing
```

```
export CSRX_CTRL_CPU=0x1
export CSRX_DATA_CPU=0x2
export CSRX_ARP_TIMEOUT=1200
export CSRX_NDP_TIMEOUT=1200
export CSRX_SIZE=large
export CSRX_HUGEPAGES=no
export CSRX_USER_DATA=no
export CSRX_PORT_NUM=3
info: creating capacity link
info: setup rsyslog config
Remote Server IP not set
info: initializing mgd ...
info: done
info: starting rsyslogd
info: starting sshd
 Already running. If you want to run multiple instances, you need to specify different pid files (use
-i option)
info: starting nstraced
info: starting mgd
info: setting initial root password ...
info: starting monit ...
```

As before, if the `srxpfe` process shows up in the container shell, the cSRX has booted up normally. Also, if the Docker `run csrx` command is called while the cSRX is not connected to trust or untrust network, the `srxpfe` process won't be running. So the best way to make sure the cSRX is running normally is to connect all its traffic interfaces to networks immediately after the cSRX starts.

In case the `srxpfe` is not initializing on the cSRX, check if the `csrx_base_cfg.xml` file is present in `/var/local/`. If it is not, create it and then launch `srxpfe` using the `/usr/sbin/srxpfe -a -d` command:

```
root@d2fd8df3597a:/# cat /var/local/csrx_base_cfg.xml
<?xml version="1.0" encoding="UTF-8"?>
<csrx mode="L3" vdev-mode="2">
  <cplane-core-mask>0x1</cplane-core-mask>
  <dpdk-core-mask>0x2</dpdk-core-mask>
  <dpdk-socket-mem>500</dpdk-socket-mem>
  <jumbo-frames-support></jumbo-frames-support>
  <arp-timeout-in-secs>1200</arp-timeout-in-secs>
  <disable-hugepages>1</disable-hugepages>
  <interfaces>
     <interface>
         <eth-dev>eth1</eth-dev>
      </interface>
     <interface>
         <eth-dev>eth2</eth-dev>
      </interface>
   </interfaces>
</csrx>
```

When the cSRX Destination NAT Is in Docker Networks

When troubleshooting a destination NAT issue, note that the issue can come from different parts of the network. First of all, the two Docker networks connected to the cSRX should support IP masquerading. This can be checked:

```
# docker network inspect NETWORK | grep masquerade
         "com.docker.network.bridge.enable_ip_masquerade": "true "
```

Say a web server is under the protection of the cSRX while implementing destination NAT, and viewers are actually viewing an open port from cSRX, for example 3456. On the other hand, the cSRX's other port is receiving information from the web service container. So while running the cSRX, the new port viewers you are seeing must be published on the host server. This can be done by adding the `-p` parameter while running the cSRX. It can be confirmed on the host:

```
# docker port csrxnat
3456/tcp -> 0.0.0.0:80
```

Second, make sure the web server container and the cSRX are connected correctly and they can ping each other. Sometimes interface misconfiguration on the cSRX will cause connection error.

Third, try to visit host-server:80 address on the browser. While trying to refresh the page, there should be some traffic generated on the cSRX. If no traffic is generated on the cSRX, then refer to the Juniper SRX documentation and check if the NAT configurations are correct:

```
User1@csrx01> show security flow session
Session ID: 445295, Policy name: u2t/5, Timeout: 298, Valid
In: 172.29.197.191/59146 --> 172.18.0.5/3456;tcp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 10, Bytes:
1036,
Out: 192.168.100.3/80 --> 172.29.197.191/59146;tcp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 9, Bytes:
8528,
```

If there's no issue with the above checkpoints, there could be some settings on the host that need to be changed. That is, disable tcp-offload on all virtual network interfaces. This can be done by using a shell script:

```
ip link | grep UP | awk -F ':' '/veth/ {print $2}' | awk -F '@' '{print $1}' | while read line
> do
>  ethtool -K $line tx off >/dev/null
> done
```

Information Collection on the cSRX

There are a few CLI `show` commands on the cSRX that you can check to gather information about its status.

To show the system version:

```
user1@csrx3> show version
Hostname: csrx3
Model: csrx
Junos: 20190622_0521_19
```

To check cSRX's syslog:

```
user1@csrx3> show log syslog
Sep 19 07:13:36 ec57da60606e rsyslogd: [origin software="rsyslogd" swVersion="7.4.4" x-pid="128.
x-info="http://www.rsyslog.com"] start
Sep 19 07:13:35 ec57da60606e rsyslogd-3003: ID for user /syslog/ could not be found or error [try
http://www.rsyslog.com/e/3003 ]
Sep 19 07:13:35 ec57da60606e rsyslogd-3003: ID for user /syslog/ could not be found or error [try
http://www.rsyslog.com/e/3003 ]
Sep 19 07:13:36 ec57da60606e rsyslogd: rsyslogd/s groupid changed to 104
Sep 19 07:13:36 ec57da60606e rsyslogd-2039: Could no open output pipe //dev/xconsole/: No such file or
directory [try http://www.rsyslog.com/e/2039 ]
Sep 19 07:13:36 ec57da60606e mgd: UI_COMMIT_COMPLETED: commit complete
Sep 19 07:13:36 ec57da60606e logger: Starting mgd
...
```

To check cSRX's IPv4 and IPv6 forwarding tables:

```
User1@csrx3> show route forwarding-table
Routing table: default.inet
Internet:
Destination       Type RtRef Next hop         Type Index    NhRef Netif
0.0.0.0          perm     0                 dscd    2517     1
20.0.0.1         perm     0 20.0.0.1         ucast    5500     1
20.0.0.2         perm     0 20.0.0.2          locl    2001     1
20.0.0.255       perm     0                  bcst    2002     1
20.0.0/24        perm     0                  rslv    2004     1
224.0.0.1        perm     0                  mcst    2515     1
224/4           perm     0                  mdsc    2516     1
30.0.0.1         perm     0 30.0.0.1         ucast    5502     1
30.0.0.2         perm     0 30.0.0.2          locl    2006     1
30.0.0.255       perm     0                  bcst    2007     1
30.0.0.3         perm     0 30.0.0.3         ucast    5501     1
30.0.0/24        perm     0                  rslv    2009     1
default          perm     0 20.0.0.1         ucast    5500     1

Routing table: default.inet6
Internet6:
Destination       Type RtRef Next hop         Type Index    NhRef Netif
::              perm     0                 dscd    2527     1
ff00::/8         perm     0                  mdsc    2526     1
ff02::1          perm     0                  mcst    2525     1
```

To check what type of traffic is going through the cSRX, please do not stop the on-going traffic and open a new window with the cSRX operational mode:

```
root@csrx3> show security flow session
Session ID: 34897, Policy name: t2u/4, Timeout: 2, Valid
  In: 30.0.0.3/3 --> 8.8.8.8/25088;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 8.8.8.8/25088 --> 20.0.0.2/20387;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,

Session ID: 34898, Policy name: t2u/4, Timeout: 2, Valid
  In: 30.0.0.3/4 --> 8.8.8.8/25088;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 8.8.8.8/25088 --> 20.0.0.2/7655;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,

Session ID: 34899, Policy name: t2u/4, Timeout: 4, Valid
  In: 30.0.0.3/5 --> 8.8.8.8/25088;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 8.8.8.8/25088 --> 20.0.0.2/27076;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
Total sessions: 3
```

If the traffic is not going through the cSRX, the above command will generate a blank output. Then it's time to troubleshoot the connection and routing issues. Table 6.1 lists some of the main security-related CLI show commands (similar to Juniper SRX Series devices).

Table 7.1 Security Show Commands for the cSRX

To show security zones	`show security zones`
To show security policies	`Show security policies`
To show destination NAT summary	`show security nat destination summary`
To show source NAT summary	`show security nat source summary`
To show security NAT rules	`show security nat static rule all`
To show AppFW status	`show services application-identification status`
To show AppFW statistics	`show services application-identification statistics applications`
To show IDP status	`show security idp status`
To show IDP statistics	`show security idp application-statistics`
To show web-filtering status	`show security utm web-filtering status`
To show web-filtering statistics	`show security utm web-filtering statistics`

If TCP traffic is not flowing through the cSRX, this may be due to an invalid checksum of the TCP packet received by the cSRX. Disable the checksum in the Linux kernel using the following script:

```
sh /root/CSRX_POCS/set_ethtool.sh

cat CSRX_POCS/set_ethtool.sh
#!/bin/bash
```

```
#ip link | grep UP | awk –F ':' '{print $2}' | sed 's/^\s//g' | grep –v "lo" | while read line
ip link | grep UP | grep veth | awk –F ':' '{print $2}' | awk –F '@' '{print $1}' | while read line
do
   veth="`ethtool –i $line | grep driver | grep veth`"
   if [ ! –z "$veth" ]
   then
      echo "ethtool –K $line tx off >/dev/null"
      ethtool –K $line tx off
   fi
```

Other useful tips to debug the traffic flow through the cSRX involve using `tcpdump` on the cSRX and checking the IP tables on the host machine.

On the cSRX, running `tcpdump` can sometimes throw an error that says a `library .so` file is not found. If this error is encountered, then move the file to `/usr/bin/`.

```
mv /usr/sbin/tcpdump /usr/bin/tcpdump
```

Then run the `tcpdump` from the `/usr/bin/` location to monitor the traffic on the interface.

Appendix

cSRX Versus vSRX Features

Here's a simple table listing the supported features on the cSRX and the vSRX.

	vSRX 3.0 (20.3R1)	cSRX(20.3R1.8)
Central management	Supported with CLI, J-Web and Security Director	CLI and Security Director supported; No J-Web support
Interfaces	10 Interfaces including management interface	16 interfaces including management interface
Basic firewall policy	Supported	Supported
Network Address Translation (NAT)	Supported	Includes support for all NAT functionality on the cSRX platform, such as: Source NAT Destination NAT Static NAT Persistent NAT and NAT64 NAT hairpinning NAT for multicast flows
SNMP	Supported	Not Supported

Routing	Supported	Basic Layer 3 forwarding with VLANs. Layer 2 through 3 forwarding functions: secure-wire forwarding or static routing forwarding
Application Firewall (AppFW)	Supported	Supported
Application Identification (AppID)	Supported	Supported
Application Tracking (AppTrack)	Supported	Supported
Unified Threat Management (UTM)	Supported	Includes support for all UTM functionality on the cSRX platform, such as: Antispam Sophos Antivirus Web Filtering Content Filtering
User Firewall	Supported	Includes support for all user firewall functionality on the cSRX platform, such as: Policy enforcement with matching source identity criteria Logging with source identity information Integrated user firewall with active directory Local authentication
Intrusion Detection and Prevention (IDP)	Supported	Supported
IPv4 and IPv6	Supported	Supported
Zones and zone-based IP spoofing	Supported	Supported
Dynamic Routing	Supported	Not Supported
IPsec VPN	Supported	Not Supported

cSRX References

Download the configuration examples in this book from its GitHub repository: https://github.com/csrx-dayone.

The Juniper Networks product page for the cSRX, with data sheets, specs, and overviews: https://www.juniper.net/us/en/products-services/security/srx-series/csrx/.

Juniper cSRX official documentation at the TechLibrary: https://www.juniper.net/documentation/product/en_US/csrx .

An Attacker Looks at Docker: Approaching Multi-Container Applications, Black Hat 2018: https://www.blackhat.com/us-18/briefings/schedule/#an-attacker-looks-at-docker-approaching-multi-container-applications-9975 .

Juniper: Overview of IDP Policy support for Unified Policies: https://www.juniper.net/documentation/en_US/junos/topics/concept/security-idp-policy-in-unified-policy.html .

Juniper Application Firewall: https://www.juniper.net/documentation/en_US/junos/topics/topic-map/security-application-firewall.html.

Contrail microservice installation with Kubernetes: https://github.com/Juniper/contrail-ansible-deployer/wiki/Contrail-microservice-installation-with-kubernetes.