JUNIPER NETWORKS | Engineering Simplicity

cSRX Container Firewall Deployment Guide for Private and Public Cloud Platforms

Published
2024-10-17

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

cSRX Container Firewall Deployment Guide for Private and Public Cloud Platforms

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at https://support.juniper.net/support/eula/. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

About This Guide

cSRX Container Firewall is the containerized form of the Juniper Networks next-generation firewall. It is positioned for use in a containerized or cloud environment where it can protect and secure east-west and north-south traffic. This guide provides you details on deployment of cSRX Container Firewall on various private and public cloud platforms.

This guide also includes basic cSRX Container Firewall configuration and management procedures.

After completing the installation, management, and basic configuration procedures covered in this guide, refer to the Junos OS documentation for information about further security feature configuration.

1

PART

Introduction to cSRX Container Firewall

Overview

SUMMARY

In this topic you learn about cSRX Container Firewall and its benefits.

IN THIS SECTION

cSRX Container Firewall

The Containerized SRX (cSRX) Container Firewall is a containerized version of the Juniper Networks ® SRX Series Firewall built based on a Docker container, delivering agile, elastic, and cost-saving security services. Integrated into many networking services, the cSRX virtual security solution provides advanced security services, including AppSecure, and Content Security in the form of a container.

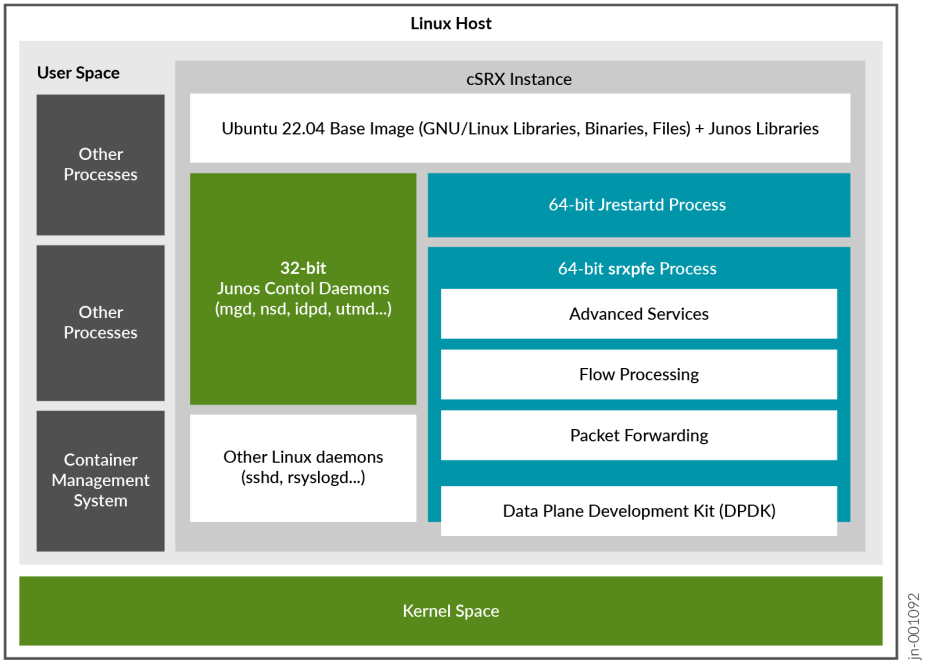
The use of a Docker container substantially reduces the overhead as each container shares the Linux host's OS kernel. Regardless of the number of containers a Linux server hosts, only one OS instance can be in use.

With its small footprint and Docker as a container management system, the cSRX enables deployment of agile, high-density security services.

The cSRX runs on Linux bare-metal server as the hosting platform for the Docker container environment. The cSRX package comprises all the dependent processes (or daemons) and libraries to support the different Linux host distribution methods (Ubuntu, Red Hat Enterprise Linux, or CentOS).

The cSRX runs on Linux bare-metal server as the hosting platform for the Docker container environment. The cSRX package comprises all the dependent processes (or daemons) and libraries to support the different Linux host distribution methods (Ubuntu, Red Hat Enterprise Linux, or CentOS).

Figure 1: cSRX Container Firewall Architecture



When the cSRX is active, several processes (or daemons) inside the docker container launch automatically. Some daemons support Linux features, providing the same service that they provide when running on a Linux host (for example, sshd, rsyslogd, and monit). You can port and compile other daemons from Junos OS to perform configuration and control jobs for security service (for example, mgd, nsd, Content Security, IDP, and AppID). The SRX PFE is the data plane daemon that receives and sends packets from the revenue ports of a cSRX. The cSRX uses SRX PFE for Layer 2 to Layer 3 forwarding functions and for Layer 4 through Layer 7 network security services.

The cSRX solution provides the following capabilities:

- Layer 7 security services such as firewall, intrusion prevention system (IPS), and AppSecure
- Automated service provisioning and orchestration
- Distributed and multitenant traffic securing
- Centralized management with Junos Space® Security Director, including dynamic policy and address update, remote log collections, and security events monitoring
- Scalable security services with small footprints

For more information on building containers with docker, see Day One: Building Containers with cSRX

Benefits of cSRX Container Firewall

The cSRX has many benefits that demonstrate its value in securing containerized workloads and ensuring robust protection against cybersecurity threats in dynamic container environments.

- **Efficient resource utilization**–Avoids the need for separate guest OS instances that significantly reduces memory and CPU usage, allowing more applications to run on the same hardware.
- **Content Security and threat prevention**–Offers robust protection against a wide array of network threats, enhancing the overall security posture of the environment with integrated Layer 7 security services such as firewall, intrusion prevention system (IPS), and AppSecure.
- **Enhanced security and isolation**–Provides a secure environment where multiple applications can run independently, reducing the risk of interference and security breaches.
- **Simplified dependency management**–Different containers with conflicting dependencies run concurrently on the same host, streamlining application management.
- **Optimized for High-Density Environments**–With small footprint and efficient resource utilization enables higher density deployments, which is particularly advantageous for environments with limited resources. Also, provides security services deployment without significant hardware investments.
- **Rapid deployment and upgrades**–Faster spin-up time compared to traditional virtual machines, enabling quick deployment and seamless upgrades of applications.
- **Cost savings**–Optimized resource usage translates to reduced hardware and energy costs, making container virtualization a cost-effective solution for running multiple applications.
- **Scalability and flexibility**–Rapid scale up and down makes cSRX highly suitable for dynamic environments, including public, private, and hybrid clouds.

Use Cases

With the cSRX, extending security to workloads running in containers is just another benefit provided by Juniper Connected Security that safeguards users, applications, and cloud workloads to all connection points throughout the network.

- You can apply the cSRX in use cases such as microsegmentation that provides threat detection for east-west traffic within a Kubernetes cluster.
- You can deploy cSRX as an application protection gateway for north-south traffic; this controls the applications that are allowed to interact with the apps running in the container.

- The cSRX offers easy, flexible, and scalable deployment options. These options address various customer use cases such as application protection, microsegmentation, and secure IoT deployments as an edge gateway through a Docker container management solution.
- The cSRX supports Software-defined networking (SDN) through Contrail® Enterprise Multicloud, OpenContrail, and other third-party solutions. The cSRX also integrates with other next-generation cloud orchestration tools such as Kubernetes.
- You can configure and manage the cSRX centrally through Security Director from the CLI with the same Junos OS syntax or using Network Configuration Protocol (NETCONF). Like other Juniper firewalls, the cSRX follows zero-trust principles, where traffic is not allowed to pass through unless explicitly permitted by a configured policy.

Container Overview

A container provides an OS-level virtualization approach for an application and associated dependencies that allow the application to run on a specific platform. Containers are not VMs, rather they are isolated virtual environments with dedicated CPU, memory, I/O, and networking.

A container image is a lightweight, standalone, executable package of a piece of software that includes everything required to run it: code, runtime, system tools, system libraries, settings, and so on. Also, because of the light weight of the containers, a server can host many more container instances than that by virtual machines (VMs), yielding tremendous improvements in utilization.

The main features of containers are:

- Includes all dependencies for an application, multiple containers with conflicting dependencies can run on the same Linux distribution.
- Use the host OS Linux kernel features, such as groups and namespace isolation, to allow multiple containers to run in isolation on the same Linux host OS.
- An application in a container can have a small memory footprint because the container does not require a guest OS, which is required with VMs, because it shares the kernel of its Linux host's OS.
- Have a high spin-up speed and can take much less time to boot up as compared to VMs. This enables you to install, run, and upgrade applications quickly and efficiently.

License for cSRX Container Firewall

The cSRX software features require a license to activate the feature. To understand more about cSRX licenses, see Supported Features on cSRX, Juniper Agile Licensing Guide, and Managing cSRX Licenses.

Requirements for cSRX Container Firewall

IN THIS SECTION

This section presents an overview of requirements for deploying a cSRX Container Firewall instance and the Junos OS feature support on cSRX.

cSRX DPDK driver supports the following NICs

Supported SRX Series Firewall Features on cSRX Container Firewall

Table 1 on page 6 provides a high-level summary of the feature categories supported on cSRX and any feature considerations.

To determine the Junos OS features supported on cSRX, use the Juniper Networks Feature Explorer, a Web-based application that helps you to explore and compare Junos OS feature information to find the right software release and hardware platform for your network. See Feature Explorer.

Table 1: SRX Series Firewall Features Supported on cSRX Container Firewall

Feature	Considerations
Application Firewall (AppFW)	Application Firewall Overview

Table 1: SRX Series Firewall Features Supported on cSRX Container Firewall *(Continued)*

Feature	Considerations
Application Identification (AppID)	Understanding Application Identification Techniques
Application Tracking (AppTrack)	Understanding AppTrack
Basic firewall policy	Understanding Security Basics
Brute force attack mitigation	Intrusion Detection and Prevention User Guide
Central management	CLI only. No J-Web support.
DDoS protection	DoS Attack Overview
DoS protection	DoS Attack Overview
Interfaces	A cSRX container supports 17 interfaces: • 1 Out-of-band management Interface (eth0) • 16 In-band interfaces (ge-0/0/0 to ge-0/0/15). Network Interfaces
Intrusion Detection and Prevention (IDP)	For SRX Series Firewall IPS configuration details, see: Understanding Intrusion Detection and Prevention for SRX Series Firewall
IPv4 and IPv6	Understanding IPv4 Addressing Understanding IPv6 Address Space
Jumbo frames	Understanding Jumbo Frames Support for Ethernet Interfaces
Malformed packet protection	Understanding IDS Screens for Network Attack Protection

Table 1: SRX Series Firewall Features Supported on cSRX Container Firewall *(Continued)*

Feature	Considerations
Network Address Translation (NAT)	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 For SRX Series Firewall NAT configuration details, see: Introduction to NAT
Routing	Basic Layer 3 forwarding with VLANs. Layer 2 through 3 forwarding functions: secure-wire forwarding or static routing forwarding
SYN cookie protection	Understanding SYN Cookie Protection
System Logs and Real-Time Logs	Starting in Junos OS Release 20.1R1, you can monitor traffic using system logs and RTlogs.

Table 1: SRX Series Firewall Features Supported on cSRX Container Firewall *(Continued)*

Feature	Considerations
User Firewall	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 For SRX Series Firewall user firewall configuration details, see: Overview of Integrated User Firewall
Content Security	Includes support for all Content Security functionality on the cSRX platform, such as: • Antispam • Sophos Antivirus • Web filtering • Content filtering For SRX Series Firewall Content Security configuration details, see: Unified Threat Management Overview For SRX Series Firewall Content Security antispam configuration details, see: Antispam Filtering Overview
Zones and zone-based IP spoofing	Understanding IP Spoofing
ATP Cloud	**Juniper Advanced Threat Prevention Cloud (ATP Cloud)**
SSL Proxy	**SSL Proxy**

Table 1: SRX Series Firewall Features Supported on cSRX Container Firewall *(Continued)*

Feature	Considerations
Security Intelligence (SecIntel), Domain Name System (DNS), and ETI	Security Intelligence Overview Understanding and Configuring DNS Security Director
Juniper Identity Management Service (JIMS)	Juniper Identity Management Service User Guide

Table 2: IKE and IPsec features

Feature		**Supported on cSRX**
IKE Features	Pre-shared key	Yes
	Certificate authentication	Yes
	IKEv1 (main mode/aggressive mode)	No
	IKEv2	Yes
	Route-based VPN	Yes
	Site-to-site VPN	Yes
	Auto VPN	Yes
	Dynamic endpoint VPN	Yes
	Point-to-point tunnel interfaces	Yes

Table 2: IKE and IPsec features *(Continued)*

Feature		Supported on cSRX
	Point-to-multipoint tunnel interfaces	No
	Numbered tunnel interfaces	No
	Unnumbered tunnel interface	Yes
	Hub-and-spoke scenario for site-to-site VPNs	Yes
	Unicast static and dynamic (RIP, OSPF, BGP) routing overt st0 interface	No
	Virtual router	No
	IKED crash recovery	Yes
	Chassis Cluster	No
	HA Link Encryption	No
	Local address selection	Yes
	Loopback address termination	No
	DNS name as IKE gateway address	Yes
	NAT-Traversal (NAT-T) for IPv4 IKE peers	Yes
	Dead Peer Detection (DPD)	Yes

Table 2: IKE and IPsec features *(Continued)*

Feature		Supported on cSRX
	Generic proposals and policies for IPv4 and IPv6	Yes
	General IKE ID	Yes
	Single proxy ID pairs	No
	Multiple traffic selector pairs	Yes
	Dual-stack (parallel IPv4 and IPv6 tunnels) over a single physical interface	Yes
	Authentication Algorithms - md5, sha1, sha-256, sha-384, sha-512	Yes
	Encryption Algorithms - des-cbc, 3des-cbc, aes-128-cbc, aes-128-gcm, aes-192-cbc, aes-256-cbc, aes-256-gcm	Yes
	IKE Proposal Sets - basic, compatible, standard, prime-128, prime-256, suiteb-gcm-128, suiteb-gcm-256	Yes
	DH groups - 1,2,5,14,15,16,19,20,21,24	Yes
	Local Identity - distinguished-name, hostname, ipv4/v6 address, user-at-hostname, key-id	Yes

Table 2: IKE and IPsec features *(Continued)*

Feature		Supported on cSRX
	Remote Identity - distinguished-name, hostname, ipv4/v6 address, user-at-hostname, key-id	Yes
	IKE Reauthentication (initiator and responder)	Yes
	Configuration payload	No
	EAP	No
	Remote Access – NCP/Licensing	No
	Tunnel establishment - immediately, on-traffic, responder-only and responder-only-no-rekey mode	Yes
	Distribution-Profile	No
	Tunnel re-distribution	No
	IKEv2 Fragmentation	Yes
	SNMP MIB	No
	Statistics, logs, per-tunnel debugging	Yes
	IKE termination on lo0 interface	No
IPsec and Dataplane Features	ESP and AH tunnel modes	Yes

Table 2: IKE and IPsec features *(Continued)*

Feature		Supported on cSRX
	Extended sequence number	Yes
	Lifetime of IKE or IPsec SA, in seconds	Yes
	Encryption Algorithms – des-cbc, 3des-cbc, aes-128-cbc, aes-192-cbc, aes-256-cbc, aes-gcm-128, aes-gcm-256 Yes	Yes
	Authentication-algorithm - hmac-sha1-96, hmac-md5-96, hmac-sha-256-128, hmac-sha-384, hmac-sha-512	Yes
	Don't Fragment bit	Yes
	IPv6 extension headers	Yes
	IPsec fragmentation and reassembly	Yes
	Session affinity	No
	Power mode IPsec	Yes
	Configurable anti-replay window	Yes
	DSCP Copy	Yes
	Configurable delay installation of rekeyed outbound SAs	Yes

Table 2: IKE and IPsec features *(Continued)*

Feature		Supported on cSRX
	Cos on st0	No

SRX Series Firewall Features Not Supported on cSRX Container Firewall

Table 3 on page 15 lists SRX Series Firewall features that are not applicable in a containerized environment, that are not currently supported, or that have qualified support on cSRX.

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall

SRX Series Firewall Feature	cSRX Container Firewall Notes
Application Layer Gateways	Avaya H.323
Authentication with IC Series Devices	Layer 2 enforcement in UAC deployments **NOTE**: UAC-IDP and UAC-Content Security also are not supported.
Class of Service	High-priority queue on SPC
	Tunnels
Data Plane Security Log Messages (Stream Mode)	TLS protocol
Diagnostics Tools	Flow monitoring cflowd version 9
	Ping Ethernet (CFM)
	Traceroute Ethernet (CFM)
DNS Proxy	Dynamic DNS

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall *(Continued)*

SRX Series Firewall Feature	cSRX Container Firewall Notes
Ethernet Link Aggregation	LACP in standalone or chassis cluster mode
	Layer 3 LAG on routed ports
	Static LAG in standalone or chassis cluster mode
Ethernet Link Fault Management	**Physical interface (encapsulations)**
	`ethernet-ccc` `ethernet-tcc`
	`extended-vlan-ccc` `extended-vlan-tcc`
	Interface family
	`ccc`, `tcc`
	`ethernet-switching`
Flow-Based and Packet-Based Processing	End-to-end packet debugging
	Network processor bundling
	Services offloading
Interfaces	Aggregated Ethernet interface
	IEEE 802.1X dynamic VLAN assignment
	IEEE 802.1X MAC bypass

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall *(Continued)*

SRX Series Firewall Feature	cSRX Container Firewall Notes
	IEEE 802.1X port-based authentication control with multisupplicant support
	Interleaving using MLFR
	PoE
	PPP interface
	PPPoE-based radio-to-router protocol
	PPPoE interface
	Promiscuous mode on interfaces
VPNs	Acadia - Clientless VPN
	DVPN
	Multicast for AutoVPN
IPv6 Support	DS-Lite concentrator (also known as AFTR)
	DS-Lite initiator (also known as B4)
Log File Formats for System (Control Plane) Logs	Binary format (binary)
	WELF
Miscellaneous	AppQoS

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall *(Continued)*

SRX Series Firewall Feature	cSRX Container Firewall Notes
	Chassis cluster
	GPRS
	Hardware acceleration
	High availability
	J-Web
	Logical systems
	MPLS
	Outbound SSH
	Remote instance access
	RESTCONF
	SNMP
	Spotlight Secure integration
	USB modem
	Wireless LAN
MPLS	CCC and TCC
	Layer 2 VPNs for Ethernet connections

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall *(Continued)*

SRX Series Firewall Feature	cSRX Container Firewall Notes
Network Address Translation	Maximize persistent NAT bindings
Packet Capture	Packet capture **NOTE**: Only supported on physical interfaces and tunnel interfaces, such as *gr*, *ip*, and *st0*. Packet capture is not supported on a redundant Ethernet interface (*reth*).
Routing	BGP extensions for IPv6
	BGP Flowspec
	BGP route reflector
	Bidirectional Forwarding Detection (BFD) for BGP
	CRTP
Switching	Layer 3 Q-in-Q VLAN tagging

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall *(Continued)*

SRX Series Firewall Feature	cSRX Container Firewall Notes
Unsupported System Logs and Real-Time log functions	cSRX does not support all the log functions supported on other SRX Series Firewalls or vSRX Virtual Firewall instances due to limited CPU power and disk capacity. Unsupported system logs and real-time log functions on cSRX are: • The binary log • On box logs (the LLMD daemon is not ported.) • On box reports (the LLMD daemon is not ported.) • TLS is not supported for sending stream mode security log to remote log server. • LSYS and Tenant related functions.
Transparent Mode	Content Security
Content Security	Express AV
	Kaspersky AV
Upgrading and Rebooting	Autorecovery
	Boot instance configuration
	Boot instance recovery
	Dual-root partitioning
	OS rollback
User Interfaces	NSM

Table 3: SRX Series Firewall Features Not Supported on cSRX Container Firewall *(Continued)*

SRX Series Firewall Feature	cSRX Container Firewall Notes
	SRC application
	Junos Space Virtual Director
Multinode High Availability	Not supported

Supported NIcs and Interfaces on cSRX Container Firewall

Table 4: NIC and Interface Support on cSRX

NICs and Interfaces	Specification	Release Introduced
cSRX DPDK driver supports the following NICs	SR-IOV over Intel 82599 series	Junos OS Release 23.2R1
	SR-IOV over Intel X710/XL710	
	PCI pass though over Intel 82599 series	
	PCI pass though over Intel X710/ XL710 series	
cSRX poll mode supports the following interface types	Kernel bridge interfaces	

Configure cSRX Using Junos OS CLI

This section provides basic CLI configurations that can be used for configuring cSRX Container Firewall containers. For more details see, Introducing the Junos OS Command-Line Interface.

To configure the cSRX container using the Junos OS CLI:

1. Launch the cSRX container. Use the `docker run` command to launch the cSRX container. You include the `mgt_bridge` management bridge to connect the cSRX to a network.

   `root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e --name=<csrx-container-name> hub.juniper.net/security/<csrx-image-name>**

   For example, to launch `csrx2` using cSRX software image `csrx:18.21R1.9` enter:

   `root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e --name=csrx2 hub.juniper.net/security/csrx:18.2R1.9**

   > **NOTE**: You must include the `--privileged` flag in the `docker run` command to enable the cSRX container to run in privileged mode.

2. Log in to the cSRX container using SSH which is accessed by cSRX exposed service port.

```
root@csrx-ubuntu3:~/csrx#ssh -p 30122 root@192.168.42.81
```

3. Start the CLI as root user.

```
root#cli
root@>
```

4. Verify the interfaces.

   `root@>` **show interfaces**

```
Physical interface: ge-0/0/1, Enabled, Physical link is Up
  Interface index: 100
  Link-level type: Ethernet, MTU: 1514
  Current address: 02:42:ac:13:00:02, Hardware address: 02:42:ac:13:00:02
Physical interface: ge-0/0/0, Enabled, Physical link is Up
  Interface index: 200
  Link-level type: Ethernet, MTU: 1514
  Current address: 02:42:ac:14:00:02, Hardware address: 02:42:ac:14:00:02
```

5. Enter configuration mode.

```
configure
[edit]
root@#
```

6. Set the root authentication password by entering a *cleartext* password, an encrypted password, or an SSH public key string (*DSA* or *RSA*).

```
[edit]
root@# set system root-authentication plain-text-password
New password: password
Retype new password: password
```

7. Configure the hostname.

```
[edit]
root@# set system host-name host-name
```

8. Configure the two traffic interfaces.

```
[edit]
root@# set interfaces ge-0/0/0 unit 0 family inet address 192.168.20.2/24
root@# set interfaces ge-0/0/1 unit 0 family inet address 192.168.10.2/24
```

9. Configure basic security zones for the public and private interfaces and bind them to traffic interfaces.

```
[edit]
root@# set security zones security-zone untrust interfaces ge-0/0/0.0
root@# set security zones security-zone trust interfaces ge-0/0/1.0
root@# set security policies default-policy permit-all
```

10. Verify the configuration.

```
[edit]
root@# commit check
configuration check succeeds
```

11. Commit the configuration to activate it on the cSRX instance.

```
[edit]
root@# commit
commit complete
```

12. (Optional) Use the `show` command to display the configuration for verification.

RELATED DOCUMENTATION

- Junos OS for SRX Series
- Introducing the Junos OS Command-Line Interface

2

PART

cSRX Container Firewall Deployment with Kubernetes

cSRX Container Firewall with Kubernetes

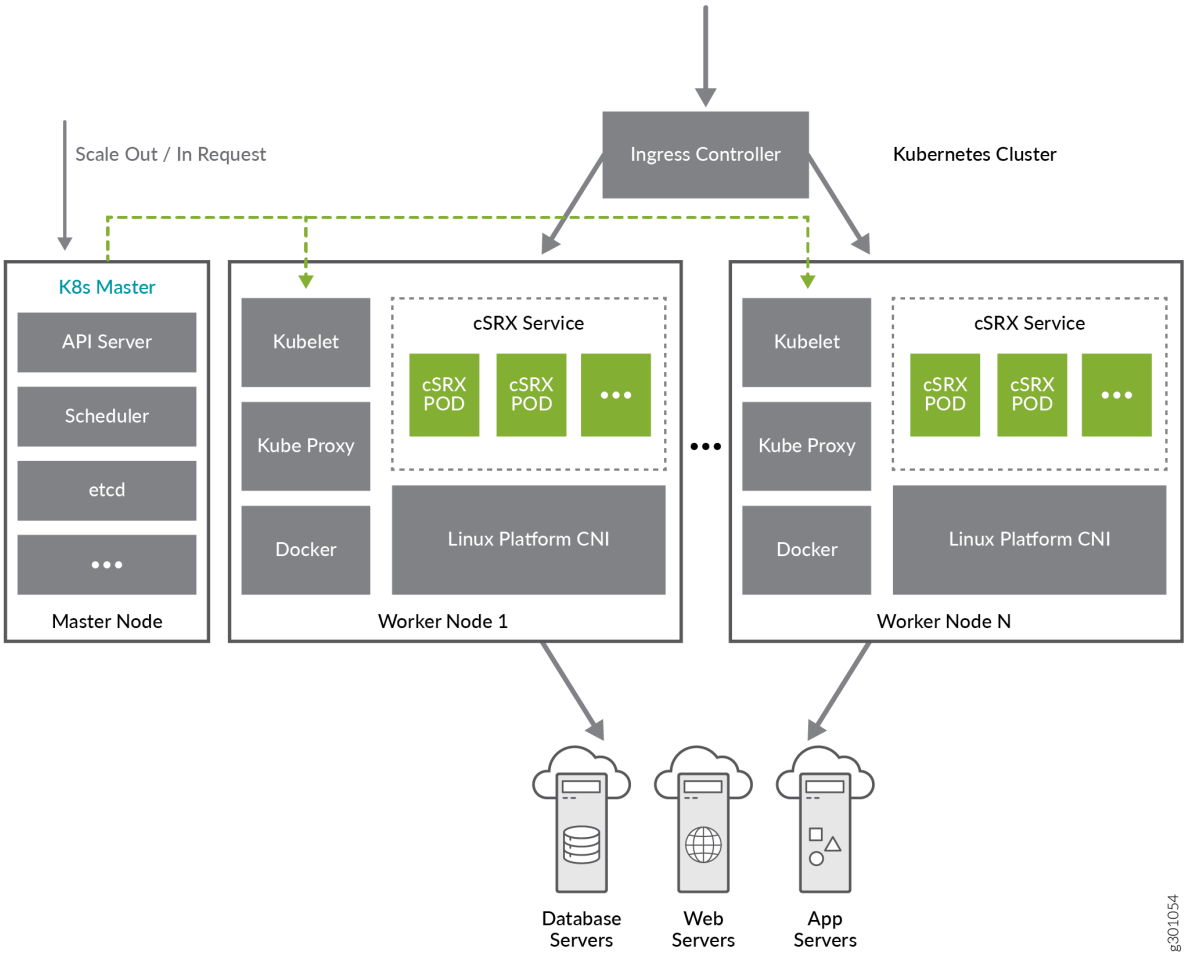
IN THIS SECTION

Overview

Containerized SRX (cSRX) Container Firewall is a virtual security solution based on Docker container to deliver agile, elastic and cost-saving security services for comprehensive L7 security protection.

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. With Kubernetes support, cSRX can scale out in a cluster running as elastic firewall service with smaller footprint when compared to virtual machines.

Figure 2: cSRX Container Firewall Service in Kubernetes on Linux



In a Kubernetes deployment, you can use Multus with both flannel and Weave Container Network Interfaces (CNIs).

To support the Kubernetes node port or the ingress controller with the cSRX, the environment variable CSRX_MGMT_PORT_REORDER allows the cSRX to use a container management interface. The Kubernetes node port or the ingress controller feature with cSRX is only supported with Flannel/Weave CNI. With CSRX_MGMT_PORT_REORDER set to yes, you can explicitly control the reconfiguration of the management port behavior. For example, you can control access to the cSRX shell or SD discovery on to the interface attached to the cSRX using Multus CNI.

For example, if you bring up cSRX with eth0, eth1, or eth2 with CSRX_MGMT_PORT_REORDER=yes, you can use eth2 as the new management interface.

NOTE: The traffic forwarding to this eth2 has to be done through the iptables rules defined explicitly by you.

Kubernetes defines a set of building objects that collectively provide mechanisms that orchestrate containerized applications across a distributed cluster of nodes, based on system resources (CPU, memory, or other custom metrics). Kubernetes masks the complexity of managing a group of containers by providing REST APIs for the required functionalities.

A node refers to a logical unit in a cluster, like a server, which can either be physical or virtual. In context of Kubernetes clusters, a node usually refers specifically to a worker node. Kubernetes nodes in a cluster are the machines that run the end user applications.

There are two type of nodes in a Kubernetes cluster, and each one runs a well-defined set of processes:

- Head node: also called primary, or primary node, it is the head and brain that does all the thinking and makes all the decisions; all of the intelligence is located here.
- Worker node: also called node, or minion, it's the hands and feet that conducts the workforce.

The nodes are controlled by the primary in most cases. The interfaces between the cluster and you is the command-line tool kubectl. It is installed as a client application, either in the same primary node or in a separate machine.

Kubernetes's objects are Pod, Service, Volume, Namespace, Replication, Controller, ReplicaSet, Deployment, StatefulSet, DaemonSet, and Job

See Junos OS Feature Supported on cSRX Container Firewall for a summary of the features supported on cSRX.

Benefits

A cSRX running in a Kubernetes cluster provides the following benefits:

- Operates services with a reduced footprint.
- Facilitates quicker scale out and sacle in of the cSRX.
- Automates management and regulation of workflow processes.

RELATED DOCUMENTATION

- What is a Container?
- Kubernetes Concepts

CHAPTER 1

Deploy and Configure cSRX in Kubernetes

IN THIS CHAPTER

Requirements for Deploying cSRX in Kubernetes

IN THIS SECTION

This section presents an overview of requirements for deploying a cSRX container on Kubernetes:

Platform and Server Requirements

Table 5 on page 31 lists the requirements for deploying a cSRX container in a Kubernetes (Primary and Worker) node.

Table 5: Primary and Worker Node Specifications

Component	Specification
Docker Engine	Docker Engine 1.9 or later installed on the same compute node as the cSRX
vCPUs	2
Memory	4 GB
Disk space	50 GB hard drive
Interfaces	16 The environment variable `CSRX_PORT_NUM` is set to=17.
Kubernetes	1.16 to 1.18

cSRX Environment Variables

IN THIS SECTION

Docker allows you to store data such as configuration settings as environment variables. At runtime, the environment variables are exposed to the application inside the container. You can set any number of parameters to take effect when the cSRX image launches. You can pass configuration settings in the YAML file or environment variables to the cSRX when it launches at boot time.

Table 6 on page 32 summarizes the list of available cSRX environment variables.

Table 6: Summary of cSRX Container Firewall Environment Variables

Environment Variable	Mandatory	Description
CSRX_AUTO_ASSIGN_IP	Optional	Automatically configure cSRX `ge-0/0/x` IP address based on IP address of cSRX container when the cSRX works in routing mode. Multus CNI is supports to create more Pod interfaces in Kubernetes. If set to `yes`, the Pod interface IP address is automatically assigned to cSRX revenue port.
CSRX_MGMT_PORT_REORDER	Optional	If set to `yes`, the last Pod interface is changed to management interface. Else, the first Pod interface is management interface.
CSRX_TCP_CKSUM_CALC	Optional	If set to `yes`, cSRX re-compute to correct TCP checksum in packets.
CSRX_LICENSE_FILE	Optional	If set, license file is loaded through ConfigMap.
CSRX_JUNOS_CONFIG	Optional	If set, initial configuration of cSRX is loaded through ConfigMap.
CSRX_SD_HOST	Optional	It is used to define Security Director (SD) server IP address or FQDN address.
CSRX_SD_USER	Optional	It is used to define Security Director server login account name.
CSRX_SD_DEVICE_IP	Optional	It is used to define cSRX management IP address, which is used by Security Director to connect to cSRX. Else it uses Port IP address.
CSRX_SD_DEVICE_PORT	Optional	It is used to define cSRX management port, which is used by Security Director to connect to cSRX. Otherwise it uses the default port number `22`.
CSRX_FORWARD_MODE	Optional	It is used in traffic forwarding mode. `"routing" | "wire"`

Table 6: Summary of cSRX Container Firewall Environment Variables *(Continued)*

Environment Variable	Mandatory	Description
CSRX_PACKET_DRIVER	Optional	It is used in Packet I/O driver. `"poll" | "interrupt"`
CSRX_CTRL_CPU	Optional	CPU mask, indicating which CPU is running the cSRX control plane daemons (such as nsd, mgd, nstraced, utmd, and so on). No CPU affinity *`hex value`*
CSRX_DATA_CPU	Optional	CPU mask, indicating which CPU is running the cSRX data plane daemon (srxpfe). No CPU affinity *`hex value`*
CSRX_ARP_TIMEOUT	Optional	ARP entry timeout value for the control plane ARP learning or response. *`decimal value`* Same as the Linux host
CSRX_NDP_TIMEOUT	Optional	NDP entry timeout value for the control plane NDP learning or response. *`decimal value`* Same as the Linux host
CSRX_PORT_NUM	Optional	Number of interfaces you need to add to container. Default is 3, maximum is 17 (which means 1 management interfaces and 16 data interfaces)

Adding License key File

You can import saved local license key file to cSRX Pod using environment variable `CSRX_LICENSE_FILE` using Kubernetes ConfigMaps.

1. Save the license key file in a text file.
2. Create ConfigMap in Kubernetes.

   root@kubernetes-master:~#**kubectl create configmap csrxconfigmap --from-file=<file path>/var/tmp/csrxlicensing**

3. Create cSRX using ConfigMaps to import the user defined configuration

```
---
deployment.spec.template.spec.containers.
    env:
      - name: CSRX_LICENSE_FILE
        value: "/var/local/config/.csrxlicense"
    volumeMounts:
    - name: lic
        mountPath: "/var/local/config"
deployment.spec.template.spec.
      volumes:
      - name: lic
    configMap:
      name: csrxconfigmap
      items:
        - key: csrxlicensing
          path: csrxlicensing
---
```

4. Run the following command to create cSRX deployment using yaml file.

   root@kubernetes-master:~#**kubectl apply -f csrx.yaml**

5. Login to cSRX pods to verify the license installed

   root@kubernetes-master:~#**kubectl exec -it csrx bash**

   root@csrx:~#**cli**

   root@csrx>**show system license**

Setting Root Password

You can set root password using Kubernetes secrets.

1. Create a generic secret in Kubernetes cSRX home namespce.

   root@kubernetes-master:~#**kubectl create secret generic csrxrootpasswd --fromliteral= CSRX_ROOT_PASSWORD=XXXXX**

2. Run the following command to verify the password is created.

   root@kubernetes-master:~#**kubectl describe secret csrxrootpasswd**

3. Run the following command to use Kubernetes Secrets to save root password in cSRX deployment yaml file.

```
---
deployment.spec.template.spec.containers.
env:
- name: CSRX_ROOT_PASSWORD
valueFrom:
secretKeyRef:
name: csrxrootpasswd
key: CSRX_ROOT_PASSWORD
---
```

4. Run the following command to create cSRX deployment using yaml file.

   root@kubernetes-master:~#**kubectl apply -f csrx.yaml**

Download cSRX Software

To download the cSRX software:

1. Download the cSRX software image from the Juniper Networks website. The filename of the downloaded cSRX software image must not be changed to continue with the installation.
2. You can either download the cSRX image file using the browser or use the URL to download the image directly on your device as in the following example:

   Run the following command to downloaded images to a local registry using `curl` command or any other http utility. The syntax for `curl` commands is:

   root@csrx-ubuntu3:~csrx# **curl -o <file destination path> <Download link url>**

root@csrx-ubuntu3:/var/tmp# **curl -o /var/tmp/images/junos-csrx-docker-20.3R1.10.img “https://cdn.juniper.net/software/csrx/20.2R1.10/junos-csrx-docker-20.3R1.10.img?SM_USER=user&__gda__=1595350694_5dbf6e62442de6bf14079d05a72464d4”**

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  160M  100  160M    0     0  1090k      0  0:02:30  0:02:30 --:--:-- 1230k
```

3. Locate the cSRX image by using the `ls` Linux shell command.

   root@csrx-ubuntu3:/var/tmp/images# **ls**

4. Load the downloaded cSRX image from the download site to the local registry using the following command.

   root@csrx-ubuntu3:/var/tmp/images# **docker image load -i /var/tmp/images/junos-csrx-docker-20.2R1.10.img**

```
e758932b9168: Loading layer [==================================================>]    263MB/
263MB
23f7a9961879: Loading layer [==================================================>]  14.51MB/
14.51MB
1e4139e6fa81: Loading layer [==================================================>]  270.3MB/
270.3MB
10334b424f86: Loading layer [==================================================>]   16.9kB/
16.9kB
202ebb2f1137: Loading layer [==================================================>]   2.56kB/
2.56kB
bc4a16173327: Loading layer [==================================================>]  1.536kB/
1.536kB
8f9a9945544a: Loading layer [==================================================>]  2.048kB/
2.048kB
Loaded image: csrx:20.2R1.10
```

5. After the cSRX image loads, confirm that it is listed in the repository of Docker images.

   root@csrx-ubuntu3:/var/tmp/images# **docker images**

```
REPOSITORY                    TAG                 IMAGE ID            CREATED
SIZE
csrx                          20.2R1.10           88597d2d4940        2 weeks ago
534MB
```

Automate Initial Configuration Load with Kubernetes ConfigMap

IN THIS SECTION

Load Initial Configuration with Kubernetes ConfigMap

ConfigMap is Kubernetes standard specification.

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. The cSRX uses ConfigMaps to load initial configuration file at cSRX container startup.

You can also add license from license key file using the steps similar to loading the initial configuration file in kubernetes.

To create cSRX ConfigMap according to cSRX initial configurations:

1. Create the cSRX.yaml file on Kubernetes-master and add the text content to deploy cSRX Pod with ConfigMap:

```
--------
apiVersion: v1kind: ConfigMap
metadata:
  name: csrx-config-map
  data:      csrx_config: | interfaces { ge-0/0/0 { unit 0; } ge-0/0/1 { unit 0; } } security
{ policies { default-policy { permit-all; } } zones { security-zone trust { host-inbound-
traffic { system-services { all; } protocols { all; } } interfaces { ge-0/0/0.0; } } security-
zone untrust { host-inbound-traffic { system-services { all; } protocols { all; } }
interfaces { ge-0/0/1.0; } } } }
```

root@kubernetes-master:~#**kubectl create -f pod_with_configmap.txt**

```
------------------
apiVersion: v1
kind: Pod
spec:
  containers:
```

```
  - name: csrx
    securityContext:
       privileged: true
    image: csrx-image:20.3
    env:
    - name: CSRX_HUGEPAGES
      value: "no"
    - name: CSRX_PACKET_DRIVER
      value: "interrupt"
    - name: CSRX_FORWARD_MODE
      value: "routing"
    volumeMounts:
    - name: disk
      mountPath: "/dev"
    - name: config
      mountPath: "/var/jail"
  volumes:
  - name: disk
    hostPath:
     path: /dev
     type: Directory
  - name: config
    configMap:
      name: csrx-config-map
      items:
      - key: csrx_config
        path: csrx_config------------------
```

2. Run the following command to create cSRX using yaml file.

   root@kubernetes-master:~#**kubectl apply -f csrx.yaml**

3. Run the following command to start cSRX in CLI mode

   root@kubernetes-master:~#**kubectl exec -it csrx bash**

   root@csrx:~#**cli**

   root@csrx#**configure**

```
Entering configuration mode
```

4. After the cSRX Pod startup, you can check the initial configuration from cSRX CLI.

root@csrx> **show**

```
## Last changed: 2019-10-18 01:53:36 UTC
version "20190926.093332_rbu-builder.r1057567 [rbu-builder]";
interfaces {
    ge-0/0/0 {
        unit 0 {
            family inet {
                address 20.0.0.11/24;
            }
        }
    }
    ge-0/0/1 {
        unit 0 {
            family inet {
                address 30.0.0.11/24;
            }
        }
    }
}
security {
    policies {
        default-policy {
            permit-all;
        }
    }
    zones {
        security-zone trust {
            host-inbound-traffic {
                system-services {
                    all;
                }
                protocols {
                    all;
                }
            }
            interfaces {
                ge-0/0/0.0;
            }
        }
        security-zone untrust {
            host-inbound-traffic {
```

```
                system-services {
                    all;
                }
                protocols {
                    all;
                }
            }
            interfaces {
                ge-0/0/1.0;
            }
        }
    }
}
```

cSRX Pods With External Network

IN THIS SECTION

Know About cSRX Pods with External Network

You can connect cSRX Container Firewall with external network with two additional interfaces. Both interfaces are attached into srxpfe and handled by FLOW.

cSRX can leverage Linux native CNI to connect to external network.

cSRX use Multus plugin to support multiple interfaces connect to the external network. Applications which monitor network traffic are directly connected to the physical network. You can use the `macvlan` network driver to assign a MAC address to each container's virtual network interface, making it appear to be a physical network interface directly connected to the physical network. In this case, you need to designate a physical interface on your Docker host to use for the `macvlan`, as well as the subnet and gateway of the `macvlan`. You can even isolate your macvlan networks using different physical network interfaces.

Connect cSRX to External Network

`macvlan` functions like a switch that is already connected to the host interface. A host interface gets enslaved with the virtual interfaces sharing the physical device but having distinct MAC addresses. Since each macvlan interface has its own MAC address, it makes it easy to use with existing DHCP servers already present on the network.

To connect cSRX with external network using `macvlan`:

Figure 3: Connecting cSRX Container Firewall to External Network with Macvlan Plugin

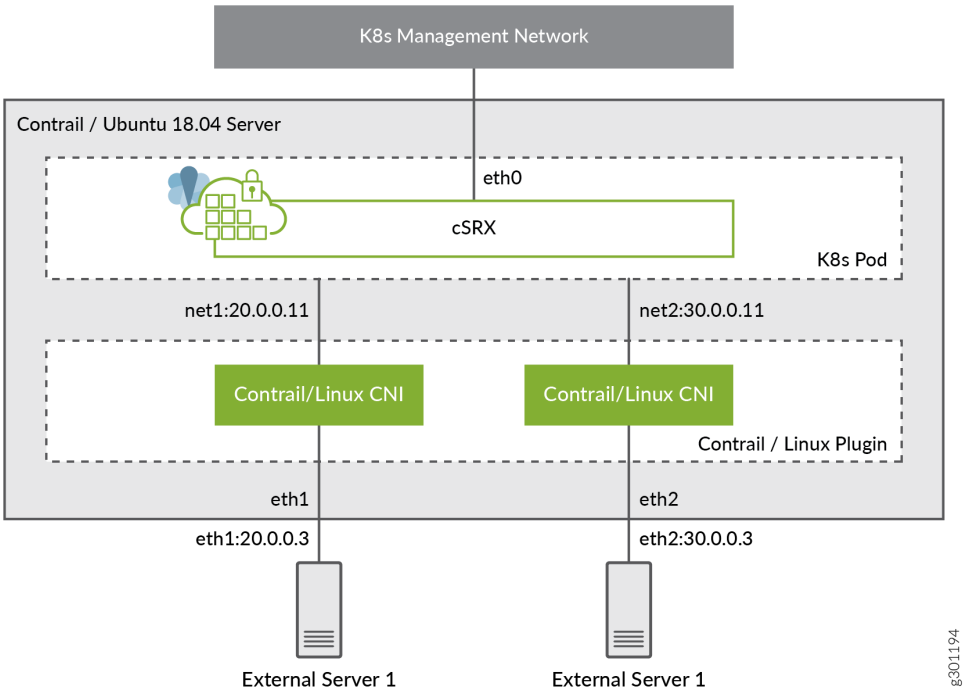
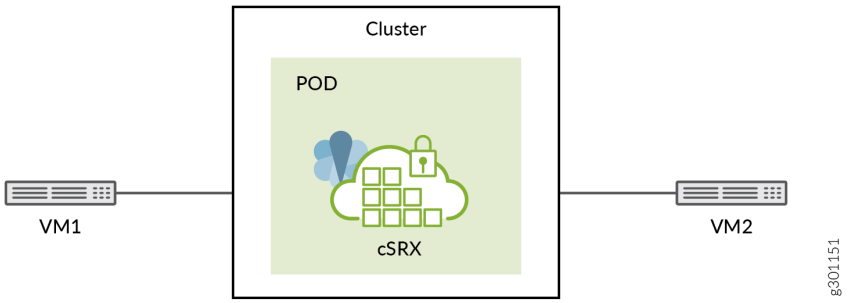


Figure 4: cSRX Container Firewall in External Network



1. Create the network-conf-1.yaml file and add the text content.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth1",
      "mode": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "20.0.0.10/24",
                "gateway": "20.0.0.2"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" },
            { "dst": "30.0.0.0/24", "gw": "20.0.0.11" }
        ]
      }
    }'
```

2. Create the network-conf-1-1.yaml file and add the text content. .

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-1-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth1",
      "mode": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "20.0.0.11/24",
                "gateway": "20.0.0.2"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
```

3. Create the network-conf-2-1.yaml and add the text content. .

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-2-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth2",
      "mode": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
            {
```

```
                "address": "30.0.0.11/24",
                "gateway": "30.0.0.2"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
```

4. Create the network-conf-2.yaml file and add the text content.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-2
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth2",
      "mode": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "30.0.0.10/24",
                "gateway": "30.0.0.2"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" },
            { "dst": "20.0.0.0/24", "gw": "30.0.0.11" }
        ]
      }
    }'
```

5. Create the cSRX.yaml file and add the text content.

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: csrx
  annotations:
    k8s.v1.cni.cncf.io/networks: network-conf-1@eth1,network-conf-1-1@eth2
spec:
  containers:
  - name: csrx
    securityContext:
       privileged: true
    image: csrx-images:20.2
    env:
    - name: CSRX_HUGEPAGES
      value: "no"
    - name: CSRX_PACKET_DRIVER
      value: "interrupt"
    - name: CSRX_FORWARD_MODE
      value: "routing"
    volumeMounts:
    - name: disk
      mountPath: "/dev"
  volumes:
  - name: disk
    hostPath:
     path: /dev
     type: Directory
```

Configure Nodeport Service for cSRX Pods

You can deploy cSRX with Nodeport service type. All the traffic is forwarded to worker node by Kubernetes in the external network.

To create a NodePort service:

1. Create the cSRX Pod yaml file and expose it as service on NodePort.

```
------------------apiVersion: v1
kind: Service
metadata:
  name: csrx1
spec:
  selector:
    app: csrx1
  ports:
    - name: ssh
```

```
      port: 22
      nodePort: 30122
  type: NodePort
---
```

2. To access cSRX:

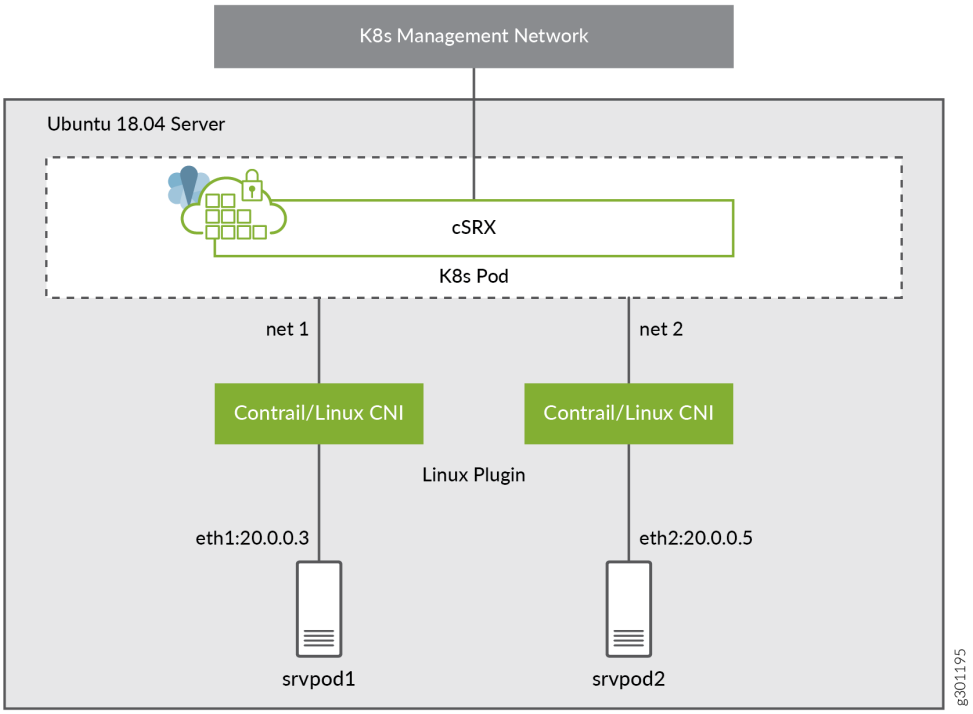
```
root@kubernetes-master:~#ssh -p 30122 root@192.168.42.81
```

cSRX Pods With Internal Network

With `bridge` plugin, all containers on the same host are plugged into a bridge (virtual switch) that resides in the host network name space. The containers receive one end of the veth pair with the other end connected to the bridge. An IP address is only assigned to one end of the veth pair in the container. The bridge itself can also be assigned an IP address, turning it into a gateway for the containers. Alternatively, the bridge can function in L2 mode and must be bridged to the host network interface (if other than container-to-container communication on the same host is desired). The network configuration specifies the name of the bridge to be used.

To connect cSRX with external network using `bridge`:

Figure 5: Connecting cSRX Container Firewall to Internal Network with Bridge Plugin



1. Create the network-conf-1-1.yaml file and add the text content.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-1-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "bridge",
      "bridge": "south-bridge",
      "promiscMode": true,
      "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "20.0.0.20/24",
                "gateway": "20.0.0.1"
            }
```

```
        ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
```

2. Create the network-conf-2-1.yaml file and add the text content.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-2-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "bridge",
      "bridge": "north-bridge",
      "promiscMode": true,
      "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "20.0.0.30/24",
                "gateway": "20.0.0.1"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
```

3. Create the srv-pod-1.yaml file and add the text content.

```
apiVersion: v1
kind: Pod
metadata:
  name: srv-pod-1
  annotations:
    k8s.v1.cni.cncf.io/networks: network-conf-1-1@north0
spec:
```

```
  containers:
  - name: srv-pod-1
    securityContext:
      privileged: true
    image: docker.io/centos/tools:latest
    command:
    - /sbin/init
```

4. Create the cSRX.yaml file and add the text content.

```
apiVersion: v1
kind: Pod
metadata:
  name: csrx
  annotations:
    k8s.v1.cni.cncf.io/networks: network-conf-1-1@eth1,network-conf-2-1@eth2
spec:
  containers:
  - name: csrx
    securityContext:
       privileged: true
    image: csrx-images:20.2
    env:
    - name: CSRX_HUGEPAGES
      value: "no"
    - name: CSRX_PACKET_DRIVER
      value: "interrupt"
    - name: CSRX_FORWARD_MODE
      value: "wire"
    volumeMounts:
    - name: disk
      mountPath: "/dev"
  volumes:
  - name: disk
    hostPath:
     path: /dev
     type: Directory
```

5. Create the srv-pod-3.yaml file and add the text content.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: srv-pod-3
  annotations:
    k8s.v1.cni.cncf.io/networks: network-conf-2-1@north0
spec:
  containers:
  - name: srv-pod-3
    image: docker.io/centos/tools:latest
    command:
    - /sbin/init
```

cSRX Deployment in Kubernetes

IN THIS SECTION

Install cSRX in Kubernetes Linux Server

Prerequisites

Following are the prerequisites required for installing cSRX Container Firewall on one primary node and 'n' number of worker nodes. Before you begin the installation:

- Install kubeadm tool on both primary and worker nodes to create a cluster. See Install Kubeadm
- Install and configure Docker on Linux host platform to implement the Linux container environment, see Install Docker for installation instructions on the supported Linux host operating systems.
- Verify the system requirement specifications for the Linux server to deploy the cSRX Container Firewall, see "Requirements for Deploying cSRX in Kubernetes" on page 30.
- Download cSRX Container Firewall software, see "Download cSRX Software" on page 35.

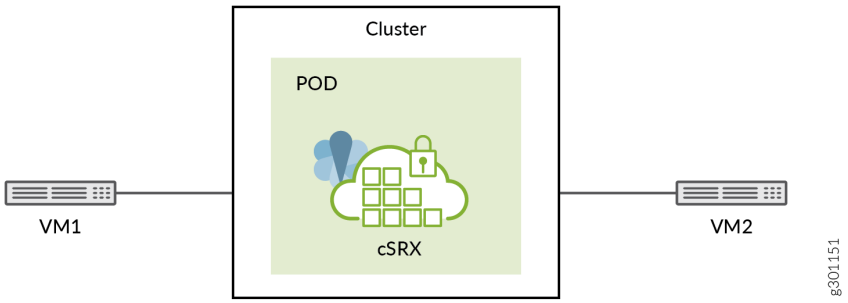
Deploy cSRX Pods in Kubernetes Linux Server

You can create cSRX Container Firewall as a Pod in routing mode and secure-wire mode to send traffic from one virtual machine to another virtual machine. You can define multiple virtual networks and connect cSRX Container Firewall interfaces to those virtual networks.

The network attachment definition is created with plugin `ipam` type as `host-local` which allocates IPv4 and IPv6 addresses out of a specified address range to ensure the uniqueness of IP addresses on a single host. The `ipam` type as `static` assigns IPv4 and IPv6 addresses statically to container.

To deploy cSRX Container Firewall with Kubernetes:

Figure 6: Deploying cSRX Container Firewall



1. Create network attachment definition for cSRX Container Firewall-eth1, cSRX Container Firewall-eth2 with `type: bridge` . For details on `type: bridge` and `type: macvlan` networks, see "cSRX Pods With External Network" on page 40.

```
------------------
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "bridge",
      "bridge": "br-1",
      "isDefaultGateway": true,
      "promiscMode": true,
      "ipam": {
        "type": "host-local",
```

```
        "ranges": [
                        [
                           {
                               "subnet": "10.10.0.0/16",
                               "rangeStart": "10.10.1.20",
                               "rangeEnd": "10.10.3.50"
                           }
                        ]
              ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
--------------------
```

```
------------------
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-1-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "bridge",
      "bridge": "br-2",
      "isDefaultGateway": true,
      "promiscMode": true,
      "ipam": {
        "type": "host-local",
        "ranges": [
                        [
                           {
                               "subnet": "55.0.0.0/16",
                               "rangeStart": "55.0.0.11",
                               "rangeEnd": "55.0.0.21"
                           }
                        ]
              ],
        "routes": [
            { "dst": "0.0.0.0/0" }
```

```
        ]
      }
    }'  ------------------
```

To create network interfaces with `type: macvlan`.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-1-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth1",
      "mode": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "20.0.0.11/24",
                "gateway": "20.0.0.2"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
```

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: network-conf-2-1
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth2",
      "mode": "bridge",
      "ipam": {
```

```
        "type": "static",
        "addresses": [
            {
                "address": "30.0.0.11/24",
                "gateway": "30.0.0.2"
            }
        ],
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
      }
    }'
```

2. Create the cSRX Container Firewall-deployment.yaml file on Kubernetes-master using `kind: Deployment`. cSRX Container Firewall as `kind: Deployment` is used to create ReplicaSet, Scaling, Rollout, Rollback in Kubernetes in this topic.

```
------------------
apiVersion: apps/v1
kind: Deployment
metadata:
  name: csrx-deployment
  labels:
    app: firewall
spec:
  replicas: 5
  selector:
    matchLabels:
      app: firewall
  template:
metadata:
  labels:
        app: firewall
      annotations:
        k8s.v1.cni.cncf.io/networks:
        network-conf-1@eth1, network-conf-1-1@eth2
spec:
  containers:
  - name: csrx
        securityContext:
 privileged: true
    image: csrx-images:20.2
```

```
     env:
    - name: CSRX_SIZE
      value: "large"
    - name: CSRX_HUGEPAGES
      value: "no"
    - name: CSRX_PACKET_DRIVER
      value: "interrupt"
    - name: CSRX_FORWARD_MODE
      value: "routing"
    volumeMounts:
    - name: disk
      mountPath: "/dev"
  volumes:
  - name: disk
    hostPath:
      path: /dev
      type: Directory
------------------
```

3. View the cSRX Container Firewall deployment:

   root@kubernetes-master:~#**kubectl get deployment csrx-deployment**

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
csrx-deployment       5/5     5            5           119m
```

Upgrade cSRX Image Using Deployment Rollout

You can upgrade the cSRX Container Firewall software image using Kubernetes Deployment rollout.

1. Run the following command to upgrade cSRX Container Firewall image using Kubernetes Deployment name in the cSRX Container Firewall Pod:

   root@kubernetes-master:~#**kubectl set image deployment csrx-deployment csrx=<new-csrx-image>**

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
csrx-deployment       5/5     5            5           119m
```

2. Run the following command to monitor rollout status:

   root@kubernetes-master:~#**kubectl rollout history deployment csrx-deployment**

root@kubernetes-master:~#**kubectl rollout status -w deployment csrx-deployment**

```
Waiting for deployment "csrx-deployment" rollout to finish: 1 old replicas are pending
termination...

Waiting for deployment "csrx-deployment" rollout to finish: 1 old replicas are pending
termination...
deployment "csrx-deployment" successfully rolled out
```

You can verify the upgraded image version by logging into the newly created cSRX Container Firewall Pods.

cSRX Image Rollback

The cSRX Container Firewall image can be rolled back to previous version using Kubernetes Deployment rollout components.

1. Rollack cSRX Container Firewall image using Kubernetes Deployment `rollout undo`:

   root@kubernetes-master:~#**kubectl rollout history deployment csrx-deploy**
2. Rollback to previous Deployment.

   root@kubernetes-master:~#**kubectl rollout undo deployment csrx-deploy**
3. Rollback to a specified version.

   root@kubernetes-master:~#**kubectl rollout undo deployment csrx-deploy --to-version=2**
4. Monitor the old cSRX Container Firewall Pods are terminated and new cSRX Container Firewall Pods are created.

   root@kubernetes-master:~#**kubectl rollout history deployment csrx-deploy**

   root@kubernetes-master:~#**kubectl rollout status -w deployment csrx-deploy**

   You can verify the image version that has been rolled back by logging into the newly created cSRX Container Firewall Pod.

Scale cSRX Deployment

To scale the cSRX Container Firewall deployment:

1. Ensure to have cSRX Container Firewall Pods created in `kind: deployment` running in Kubernetes cluster.

   root@kubernetes-master:~#**kubectl describe deployment csrx-deployment**
2. Scale up or down by changing the `replicas` number:

   root@kubernetes-master:~#**kubectl scale deployment csrx-deployment --replicas=2**
3. View the pods:

```
root@kubernetes-master:~#kubectl get pod
```

```
NAME                                READY   STATUS    RESTARTS   AGE
csrx-deployment-547fcf68dd-7hl7r    1/1     Running   0          8m8s
csrx-deployment-547fcf68dd-xbg4b    1/1     Running   0          35s
```

cSRX Image with Packaged Preinstalled Signatures

IN THIS SECTION

What Are Preinstalled Signatures?

To support pre-installed signatures package in cSRX Container Firewall image, a Docker file is placed in localhost repository to help user compile cSRX Container Firewall with installed signatures. With the new image, you can launch cSRX Container Firewall Pod, that protects workload immediately after container is launched.

The supported functions for signature packaging are:

- Intrusion Detection and Prevention (IDP)
- Application Identification (AppID)
- Content Security

Repackage cSRX Image with Preinstalled Signatures

- Ensure to have the cSRX Container Firewall image placed in the local repository or any other Docker registry.
- Ensure to include license file together with Docker file.

To repackage cSRX Container Firewall image with signatures:

1. Create DockerFile.

   root@host# **cat Dockerfile**

```
FROM localhost:5000/csrx
ARG CSRX_BUILD_WITH_SIG=yes
ENV CSRX_LICENSE_FILE=/var/local/.csrx_license
COPY csrx.lic $CSRX_LICENSE_FILE
RUN ["/etc/rc_build.local"]
CMD ["/etc/rc.local","init"]
```

   The `ARG CSRX_BUILD_WITH_SIG=yes` triggers for APPID and IDP signature auto installation.

   The optional `ENV CSRX_LICENSE_FILE=/var/local/.csrx_license` and `COPY csrx.lic $CSRX_LICENSE_FILE` commands are used to install owned license to cSRX Container Firewall container.

2. Repackage image to include APPID and IDP signature.

   root@host# **docker build -t localhost:5000/csrx-sig**

3. Push the image to the registry.

   root@host# **docker push localhost:5000/csrx-sig**

   The new cSRX Container Firewall image `localhost:5000/csrx-sig:latest` is ready for use.

4. Change the mode to CLI.

   root@host# **ke -it csrx-sig -- bash**

   root@csrx-sig:/# **cli**

5. View the APPID status.

   root@csrx-sig> **show services application-identification status**

```
Application Identification
 Status                           Enabled
 Sessions under app detection     0
 Max TCP session packet memory    0
 Force packet plugin              Disabled
 Force stream plugin              Disabled
 Statistics collection interval   1440 (in minutes)

Application System Cache
 Status                           Enabled
  Cache lookup security-services  Disabled
  Cache lookup miscellaneous-services Enabled
 Max Number of entries in cache   0
 Cache timeout                    3600 (in seconds)
```

```
Protocol Bundle
 Download Server                https://signatures.juniper.net/cgi-bin/index.cgi
 AutoUpdate                     Disabled

Proxy Details
 Proxy Profile                  Not Configured
Slot 1:
 Application package version    0
 Status                         Free
 PB Version                     N/A
 Engine version                 0
 Micro-App Version              0
 Sessions                       0
Rollback version details:
 Application package version    0
 PB Version                     N/A
 Engine version                 N/A
 Micro-App Version              N/A
```

6. View IDP package version.

   `root@csrx-sig>` **show security idp security-package-version**

```
Attack database version:N/A(N/A)
  Detector version :12.6.130180509
  Policy template version :N/A
```

Download Juniper Signature Pack

You can download the signature pack from the Juniper Signature Repository directly when cSRX Container Firewall doesn't have a preinstalled signature pack.

To download the signature pack from Juniper Signature Repository:

`root@host>` **request services application-identification download**

`root@host>` **request security idp security-package download**

Download Juniper Signature Pack Through Proxy Server

You can download the signature pack through a proxy server. AppIDD and IDPD processes first connects to the configured proxy server. The proxy server then communicates with the signature pack download server and provides the response to the process running on the device.

To download the signature pack through the proxy server:

1. Configure the proxy server so that the IP address of the proxy server is reachable from cSRX Container Firewall.
2. Run the following command to enter the configuration mode from the CLI.

   root@host> **configure**

   ```
   Entering configuration mode
   ```

   [edit]

   root@host#
3. Configure the proxy server profile on cSRX Container Firewall using the IP address and port of the proxy server.

   root@host#**set services proxy profile appid_sigpack_proxy protocol http host 4.0.0.1**

   root@host#**set services proxy profile appid_sigpack_proxy protocol http port 3128**
4. Attach the profile to AppID and IDP.

   root@host#**set services application-identification download proxy-profile appid_sigpack_proxy**

   root@host#**set security idp security-package proxy-profile appid_sigpack_proxy**
5. Commit the configuration.

   root@host#**commit and-quit**

   ```
   commit complete
   Exiting configuration mode
   ```
6. Download the IDP and APPID signature pack through proxy server.

   root@host>**request services application-identification download**

   root@host>**request security idp security-package download**

To verify that the download is happening through the proxy server:

1. Verify the logs in the proxy server.

   [root@srxdpi-lnx39 squid]# **cat /var/log/squid/access.log**

   ```
   1593697174.470   1168 4.0.0.254 TCP_TUNNEL/200 5994 CONNECT signatures.juniper.net:443 -
   HIER_DIRECT/66.129.242.156 -
   1593697175.704   1225 4.0.0.254 TCP_TUNNEL/200 11125 CONNECT signatures.juniper.net:443 -
   HIER_DIRECT/66.129.242.156 -
   ```

```
1593697176.950   1232 4.0.0.254 TCP_TUNNEL/200 5978 CONNECT signatures.juniper.net:443 -
HIER_DIRECT/66.129.242.156 -
1593697178.195   1236 4.0.0.254 TCP_TUNNEL/200 11188 CONNECT signatures.juniper.net:443 -
HIER_DIRECT/66.129.242.156 -
1593697198.337   1243 4.0.0.254 TCP_TUNNEL/200 6125 CONNECT signatures.juniper.net:443 -
HIER_DIRECT/66.129.242.156 -
```

In cSRX Container Firewall, the TLS protocol is used and traffic the through proxy server is encrypted.

cSRX Service with Load Balancing

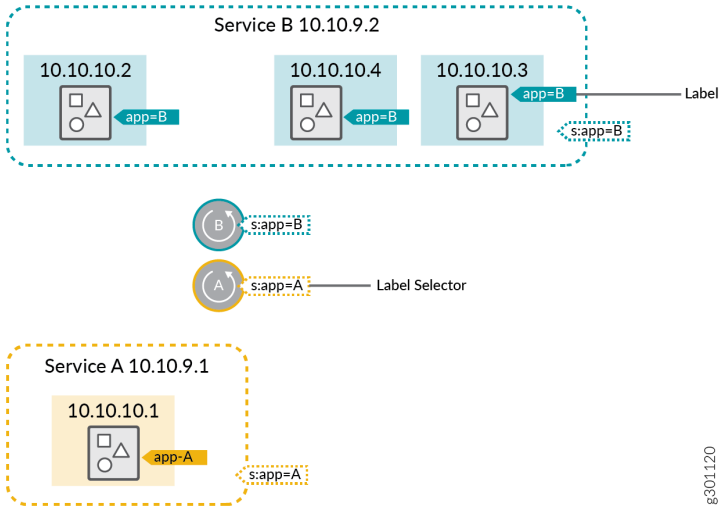
IN THIS SECTION

Know About cSRX as Kubernetes Service with Load Balancing Support

cSRX Container Firewall Pod is identified with predefined selectors and exposed with supported load balancer to distribute traffic among different cSRX Pods. The standard load balancer is ingress controller, external load balancer or cluster IP.

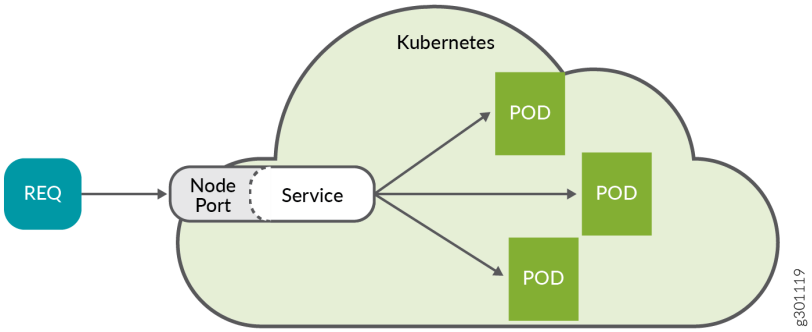
A Service enables network access to a set of Pods in Kubernetes. Services select Pods based on their labels. When a network request is made to the service, it selects all Pods in the cluster matching the service's selector, chooses one of them, and forwards the network request to it. A deployment is responsible for keeping a set of pods running.

Figure 7: Services and Labels



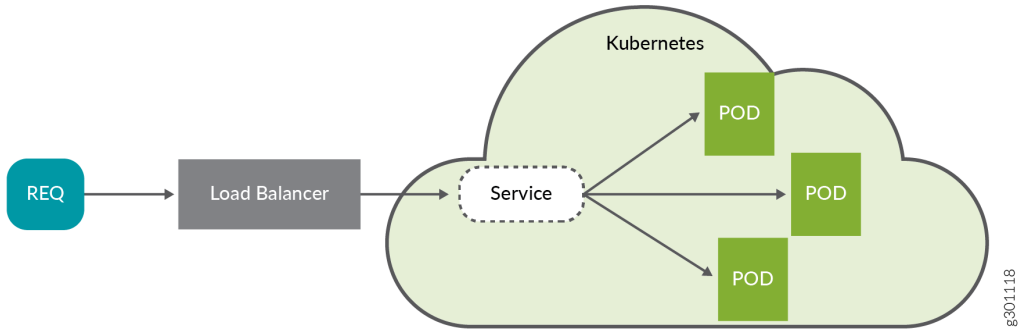
Service is to group a set of Pod endpoints into a single resource. By default, clients inside the cluster can access Pods in the Service using cluster IP address. A client sends a request to the IP address, and the request is routed to one of the Pods in the Service. The types of Services are ClusterIP (default), NodePort, LoadBalancer, and ExternalName.

Figure 8: NodePort



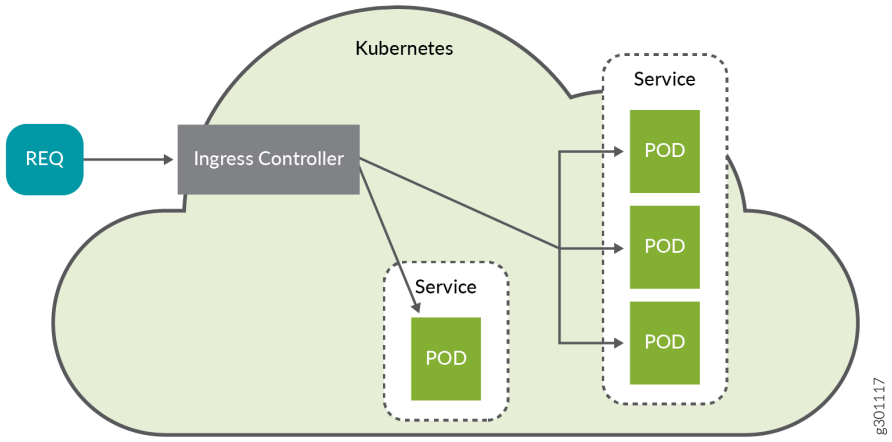
When you set a service's type to NodePort, that service starts to listen on a static port on every node in the cluster. So, you can reach the service through any node's IP address and the assigned port.

Figure 9: LoadBalancer



When you set a service's type to Load Balancer, it exposes the service externally. However, to use it, you need to have an external load balancer. The external load balancer needs to be connected to the internal Kubernetes network on one end and opened to public-facing traffic on the other in order to route incoming requests.

Figure 10: Ingress Controller



An Ingress Controller watches for new services within the cluster and dynamically creates routing rules for them. An Ingress object is an independent resource, apart from Service objects that configures external access to service's pods. You can define the Ingress, after the Service has been deployed, to connect it to external traffic. This way, you can isolate service definitions from the logic of how clients connect to them. L7 routing is one of the core features of Ingress, allowing incoming requests to be routed to the exact pods that can serve them based on HTTP characteristics such as the requested URL path. Other features include terminating TLS, using multiple domains, and load balancing traffic.

Nginx ingress controller is supported to view the traffic distribution among different cSRX Pods. For more details, see Set Up Ingress on Kubernetes Using Nginx Controller.

Configure Ingress Service for cSRX Pods

Service is used by cSRX to connect application with cSRX Pods. cSRX Service is standard Kubernetes service, in which, the load is balanced to different cSRX Pods, and the Pods are located at different work nodes. It also monitors the backend cSRX Pod and selects working cSRX Pod according to Kubernetes Pod labels. You can use YAML file to create a cSRX service.

To create a cSRX service:

1. Create the yaml file and add the following text content:

```
------------------apiVersion: v1
kind: Service
metadata:
  labels:
    app: firewall
  name: firewall
spec:
  selector:
    app:firewall
  ports:
  - name: port-1
    port: 80
    protocol: TCP
    targetPort: 80
```

2. Define routing for cSRX Pods. Ingress co-operates with Ingress controller to route outside traffic into cSRX service, then into cSRX Pods. Create a file named ingress.yaml.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: web-ingress
  namespace: default
spec:
 rules:
 - host: foo.bar
   http:
    paths:
    - path: /
```

```
        backend:
          serviceName: firewall
          servicePort: 80
```

Traffic routes to cSRX interface on `ge-0/0/0`.

3. View the cSRX service.

   `root@kubernetes-master:~#` **kubectl get svc -A**

```
NAMESPACE       NAME            TYPE          CLUSTER-IP       EXTERNAL-IP
PORT(S)                     AGE
default         csrx-service    ClusterIP     10.102.115.211   <none>        80/
TCP                         13d
default         kubernetes      ClusterIP     10.96.0.1        <none>        443/
TCP                         75d
default         nginx           NodePort      10.110.8.221     <none>        80:31454/
TCP                  18d
default         test-service    ClusterIP     10.108.236.26    <none>        80/
TCP                         11d
kube-system     kube-dns        ClusterIP     10.96.0.10       <none>        53/UDP,53/
TCP,9153/TCP       75d
```

4. View the Pod.

   `root@kubernetes-master:~#` **kubectl get pod -A**

```
NAMESPACE       NAME                                                READY   STATUS
RESTARTS   AGE
default         csrx-deployment-86f49b8dcf-7zzq9                    1/1     Running
0          11d
default         csrx-deployment-86f49b8dcf-dm6nv                    1/1     Running
0          11d
```

PART 3

cSRX Container Firewall Deployment in AWS

cSRX Deployment in AWS Using Elastic Kubernetes Service (EKS)

SUMMARY

This topic provides you an overview of cSRX Container Firewall Kubernetes orchestration in AWS Cloud using AWS Elastic Kubernetes Service (EKS).

IN THIS SECTION

cSRX with Kubernetes Orchestration in AWS

IN THIS SECTION

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. With Kubernetes support, the cSRX scales out in a cluster running as an elastic firewall service with smaller footprint when compared to virtual machines (VMs). Kubernetes groups containers that make up an application into logical units for easy management and discovery.

Kubernetes defines a set of building objects that collectively provide mechanisms that orchestrate containerized applications across a distributed cluster of nodes, based on system resources (CPU, memory, or other custom metrics). Kubernetes masks the complexity of managing a group of containers by providing REST APIs for the required functionalities.

For more information, see cSRX Container Firewall with Kubernetes.

AWS provides managed Kubernetes for services as part of their offerings. The orchestration and management of the cSRX in a Kubernetes environment using the Multus Container Network Interface (CNI) is already supported. With Kubernetes support, you can deploy, manage, and orchestrate, scale out and scale in the cSRX in a cluster that provides an elastic firewall service to application containers along with other container workloads in the AWS environment. You can deploy cSRX as Kubernetes Service or Pods.

AWS provides two orchestration services for containers: **Amazon Elastic Container Service (ECS)** and **Amazon Elastic Kubernetes Service (EKS)**.

Amazon Elastic Kubernetes Service (EKS): This is a fully managed Kubernetes service. An open source Kubernetes adaptation and fully supports the open source version. EKS is Amazon managed service that helps in running Kubernetes application on AWS cloud. EKS helps in setting up Kubernetes control plane on multiple zones providing high-availability, EKS has the capability to detect and replace unhealthy control plane instances with automated version upgrades and patches as when required. EKS is fully integrated with Elastic Container Registry (ECR) which holds container images, Identity and Access Management (IAM) roles for authentication, AWS VPC for network isolation and Elastic Load Balancing for load distribution.

You can deploy and manage cSRX in the AWS cloud using EKS orchestration for cluster management with the bring your own license (BYOL) licensing model.

Benefits

- The managed Kubernetes services reduce the dependencies on setting up and operating the Kubernetes environment.
- Automated service provisioning and orchestration
- Distributed and multitenancy traffic securing
- Scalable security services with small footprints

Amazon EKS

IN THIS SECTION

Overview

Amazon Elastic Kubernetes Service (Amazon EKS) gives you the flexibility to start, run, and scale Kubernetes applications in the AWS cloud or on-premises. Amazon EKS helps you provide highly

available and secure clusters and automates key tasks such as patching, node provisioning, and running updates.

EKS runs upstream Kubernetes and is certified Kubernetes conformant for a predictable experience. You can easily migrate any standard Kubernetes application to EKS without needing to refactor your code.

EKS makes it easy to standardize operations across environments. You can run fully managed EKS clusters on AWS. You can have an open source, proven distribution of Kubernetes wherever you want for consistent operations with Amazon EKS. You can host and operate your Kubernetes clusters on-premises and at the edge and have a consistent cluster management experience with Amazon EKS.

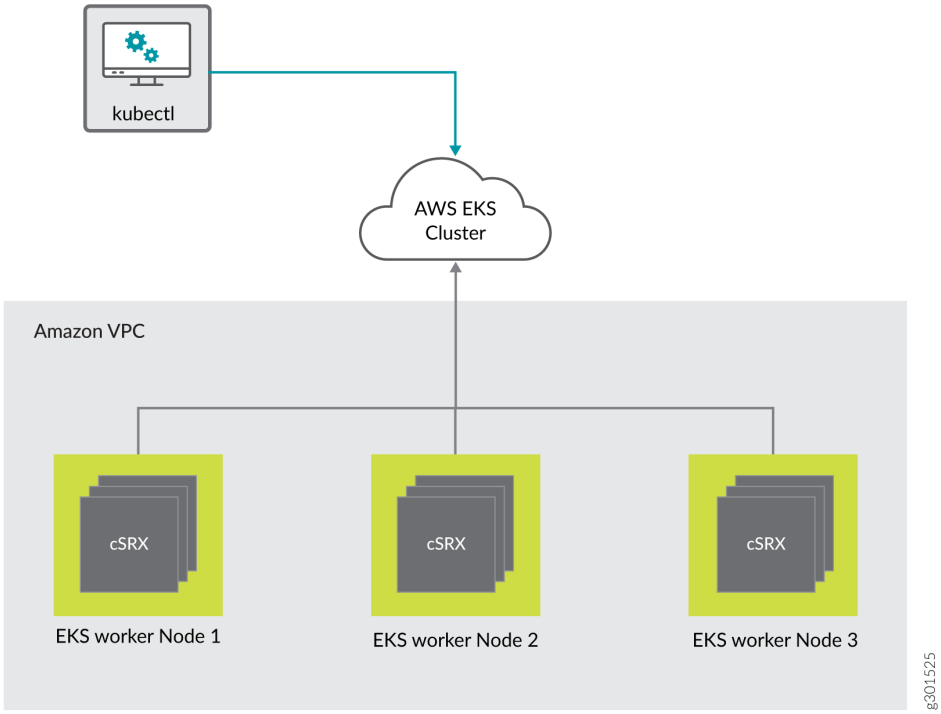
You can completely utilize the open-source Kubernetes functionality with its Elastic Kubernetes Service (EKS) on the AWS cloud. All latest Kubernetes updates are available in the EKS framework.

cSRX is supported only on EKS with EC2 instances. EKS is fully integrated with Amazon cloud watch, Autoscaling groups, AWS Identity and Access Management (IAM) and Amazon Virtual Private Cloud (VPC) enabling seamless environment to monitor and load balance the cloud application.

AWS with EKS provides a highly scalable control plane that runs on two different zones to provide high availability support. EKS is completely compatible with open-source Kubernetes, and you can easily migrate any standard Kubernetes application to EKS.

Figure 11 on page 70 illustrates AWS EKS abstraction architecture.

Figure 11: AWS EKS Abstraction Architecture



AWS proprietary Multus with flannel CNI is supported for EKS cluster deployments.

The cSRX also integrates with other next-generation cloud orchestration tools such as Kubernetes.

The cSRX adds security enforcement points where none have existed before, offering the most comprehensive network security for Kubernetes deployments.

Benefits

- Provides faster boot time.
- Supports small footprint to deliver highly agile, advanced security services in a container form factor.

  cSRX supports easy, flexible, and highly scalable deployment options covering various customer use cases, including application protection, and microsegmentation through a Docker container management solution.

  The cSRX deployed as a service in a deployment object, allows scale-up and scale down of the cSRX on demand. It functions as a firewall, protecting workloads deployed in the cluster with the configuration of rich advanced services.

Some deployments require highly agile and lightweight security virtual network functions (VNFs) that can scale massively. For such deployments, a VM-based VNF is not a scalable solution and requires a container-based security VNF.

- Supports network function service chains, allowing high availability as well as containerized security that scales in individual network functions as needed.
- Provides management flexibility with NETCONF and Junos Space(R) Security Director to support integration with third-party management and cloud orchestration tools such as Kubernetes. Junos Space(R)

  Also, with EKS, the latest security patches are applied to your cluster's control plane to ensure security of your cluster.

CHAPTER 2

Deploy and Manage cSRX in AWS

IN THIS CHAPTER

Deployment of cSRX in AWS Using EKS for Orchestration

SUMMARY

cSRX Container Firewall deployment on AWS can be achieved as plain docker container on EC2 instance using Amazon Elastic Kubernetes Service (Amazon EKS). The cluster management is done by Kubernetes, assisted by AWS and all Kubernetes commands work as is in case of EKS for container creation and management. This topic provides you details on how you can deploy cSRX on AWS cloud using Elastic Kubernetes Services (EKS) for Orchestration.

IN THIS SECTION

Deploy cSRX in AWS Using EKS

This topic provides you details to deploy the cSRX on AWS cloud.

1. As a prerequisite, install AWS CLI, eksctl, and kubectl packages. For more information, see Getting started with Amazon EKS.

2. Create cluster on EKS using the following CLI command:

```
# eksctl create cluster --name <cluster_name> --version 1.17 --region us-west-2 --nodegroup-
name
```

```
<node_group_name> --node-type t3.medium --nodes 2 --nodes-min 1 --nodes-max 3 --ssh-access --
ssh-public-key ~/.ssh/id_rsa.pub --managed --asg-access
```

3. Monitor the cluster status using the eksctl commands listed below:

```
# ubuntu@ip-172-31-0-168:~$ eksctl get cluster
NAME            REGION
csrx-eks-cluster        us-west-2
```

4. Verify the cluster created. Cluster with instance type of t3.medium and 2 worker nodes is created.

```
# kubectl get nodes
NAME                                          STATUS   ROLES    AGE     VERSION
ip-192-168-10-52.us-west-2.compute.internal   Ready    <none>   7d21h   v1.17.9
ip-192-168-33-89.us-west-2.compute.internal   Ready    <none>   7d21h   v1.17.9
```

5. Start a cSRX pod on the EKS cluster using the following .yaml file. Use this yaml file as reference and run the kubectl command to deploy cSRX pod. Use the cSRX image available on AWS marketplace to spawn cSRX containers.

```
# kubectl create -f csrx.yaml
```

6. Verify the deployment using the kubectl command below:

```
# kubectl get deployment csrx
 NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
 csrx5                 1/1     1            1           2m
```

Sample File for cSRX Deployment

This topic provides you sample file for deploying cSRX in AWS cloud using AWS EKS orchestration.

```
vim csrx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: csrx-byol
  labels:
    app: csrx-byol
spec:
  replicas: 2
  selector:
    matchLabels:
      app: csrx-byol
  template:
    metadata:
      name: csrx-byol
      labels:
        app: csrx-byol
      annotations:
        k8s.v1.cni.cncf.io/networks: br-51@eth1, br-52@eth2
    spec:
      serviceAccountName: csrxpod
      containers:
      - name: csrx-byol
        securityContext:
           privileged: true
        image: <csrx-image> ## replace image name with repo:tag
        ports:
          - containerPort: 80
        env:
        - name: CSRX_SIZE
          value: "large"
        - name: CSRX_HUGEPAGES
          value: "no"
        - name: CSRX_PACKET_DRIVER
          value: "interrupt"
        - name: CSRX_FORWARD_MODE
          value: "routing"
        - name: CSRX_AUTO_ASSIGN_IP
```

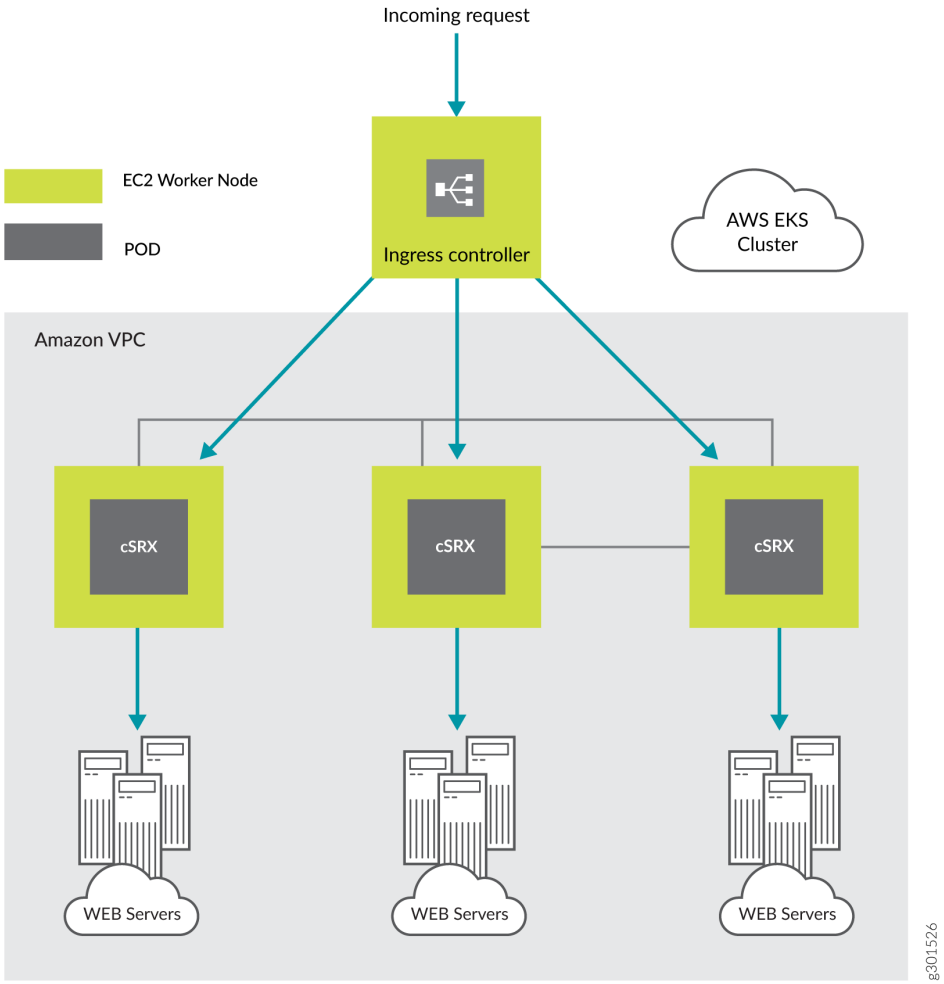
```
          value: "yes"
        - name: CSRX_MGMT_PORT_REORDER
          value: "yes"
        - name: CSRX_TCP_CKSUM_CALC
          value: "yes"
        - name: CSRX_JUNOS_CONFIG
          value: "/var/jail/csrx_config"
        - name: CSRX_LICENSE_FILE
          value: "/var/jail/.csrx_license"
        volumeMounts:
        - name: disk
          mountPath: "/dev"
        - name: config
          mountPath: "/var/jail"
      volumes:
      - name: disk
        hostPath:
          path: /dev
          type: Directory
      - name: config
        configMap:
          name: cm-byol
          items:
          - key: csrx_config
            path: csrx_config
          - key: csrx_license
            path: .csrx_license
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: csrx-byol
  name: csrx-byol
spec:
  selector:
      app: csrx-byol
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

cSRX as a Service with Ingress Controller in Amazon EKS

The cSRX Container Firewall can be deployed as a service using a Network Load Balancer with NGINX Ingress Controller on Amazon EKS. The cSRX deployed as a service in a deployment object allows you to scale up and scale down by distributing the traffic among different cSRX PODs. Also, cSRX functions as a firewall, protecting workloads deployed in the cluster with rich advanced security services.

Figure 12 on page 76 illustrates Amazon EKS ingress controller.

Figure 12: Amazon EKS Ingress Controller



To deploy the cSRX as Ingress controller on Amazon EKS:

1. Define and deploy cSRX as K8s POD or as ReplicaSet. This type of deployment is the standard K8s to define and to manage resource. Also, allows you to deploy cSRX container on specified work nodes, update or rollback based on your request.

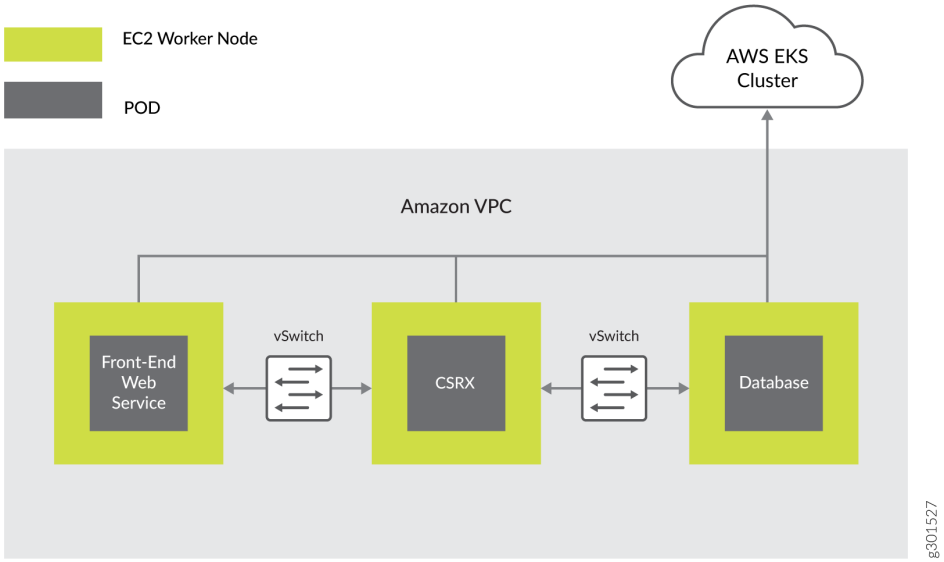
2. Use Kubectl and YAML templates to define and to deploy cSRX related resource on command line. K8s API server can process the request from other applications.
3. Expose cSRX as K8s service with load balancing. Amazon EKS supports Kubernetes Network Load Balancer (NLB) and Amazon EKS specific Application Load Balancer (ALB).
4. The cSRX POD is identified with predefined selectors and exposed with supported load balancer. The load balancer is the NGINX ingress controller and AWS NLB as external load balancer.
5. Connect cSRX container to the external network using Multus with flannel CNI. cSRX requires at least three interfaces (1 management port and 2 revenue ports).

Microsegmentation with cSRX in AWS

With micro-segmentation (East and the West firewall) application interacting in the same EKS, VPC is secured with the supported application layer security provided by cSRX Container Firewall. Multus-CNI and flannel is used to support multiple interfaces per POD for micro-segmentation. Multus-CNI and flannel leverages the Linux native CNI support of bridge and the MAC VLAN to connect to external interfaces.

Figure 13 on page 77 illustrates AWS EKS microsegmentation with cSRX in AWS.

Figure 13: AWS EKS Microsegmentation



cSRX License in AWS Marketplace

- cSRX Container Firewall is available with 60 days free trial eval license (S-cSRX-A1 SKU). The eval license in cSRX expires after 60 days.
- AWS supports Bring Your Own License (BYOL) licensing model. The BYOL license model allows you to customize your license, subscription and support to fit your needs. You can purchase BYOL from Juniper Networks or Juniper Networks authorized reseller.
- The cSRX software features require a license to activate the feature. To understand more about cSRX licenses, see
  - Supported Features on cSRX.
  - Juniper Agile Licensing Guide.
  - Flex Software License for cSRX.
- To add, delete, and manage licenses, see Managing cSRX Licenses.

4

PART

cSRX Container Firewall Deployment in Contrail Host-Based Firewall

cSRX in Contrail Host-Based Firewall

IN THIS SECTION

Containerized SRX (cSRX Container Firewall) is a virtual security solution, which is integrated into a Contrail networking as distributed host-based firewall (HBF) service. cSRX is built based on Docker container to deliver agile, elastic, and cost-saving security services.

cSRX Container Firewall on Contrail Host-Based Firewall Overview

The cSRX deploys as a single container on a Docker Engine compute node running in a Contrail cluster. The cSRX runs on a Linux bare-metal server as the hosting platform for the Docker container environment. The cSRX container packages all the dependent processes and libraries to support the different Linux host distribution methods (Ubuntu, Red Hat Enterprise Linux, or CentOS).

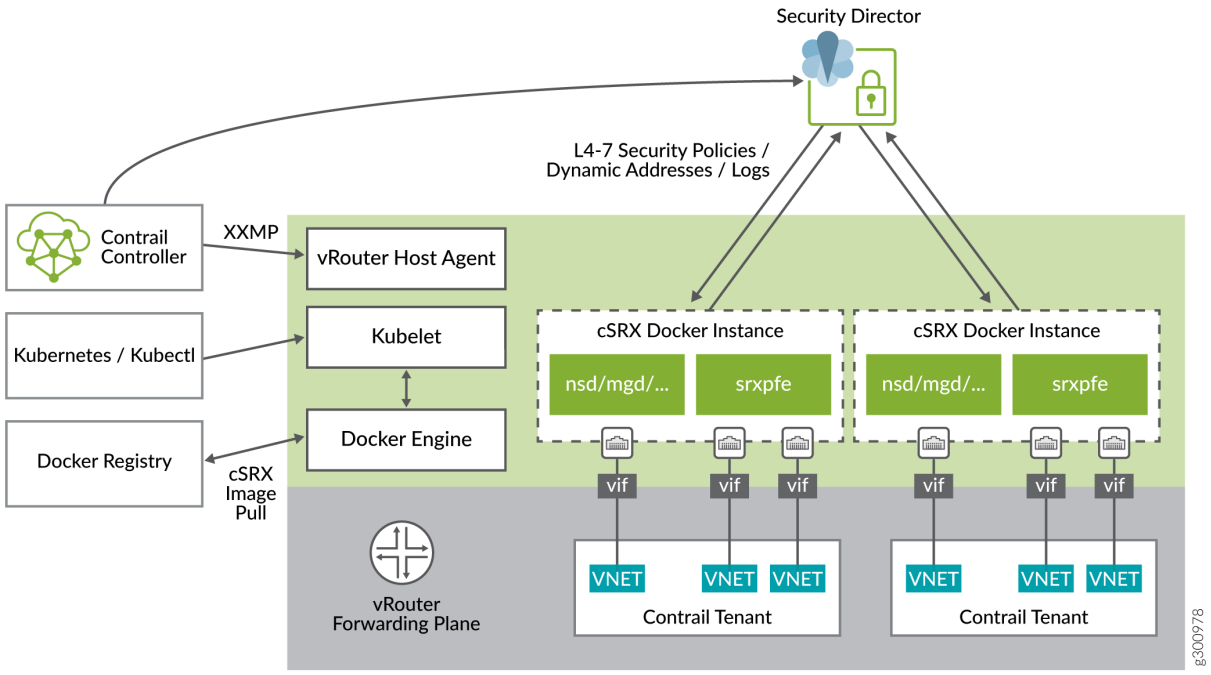
Several processes inside the Docker container launch automatically when the cSRX becomes active. Some processes support Linux features, providing the same service that they provide when running on a Linux host (for example, sshd, rsyslogd, and monit). Other processes are compiled and ported from Junos OS to perform configuration and control jobs for the security service. For example, MGD, NSD, Content Security, IDP, and AppID). srxpfe is the data plane daemon that receives and sends packets from the revenue ports of a cSRX container. cSRX uses srxpfe for Layer 2 (L2) to Layer 3 (L3) forwarding functions as well as for Layer 4 through Layer 7 network security services.

The distributed software security solution is built on top of Contrail Networking using Contrail Controller and Contrail vRouter to prevent threats in a customer's multicloud environment.

When cSRX acts as distributed firewall service on Contrail, Kubernetes is used to orchestrate cSRX instances on compute nodes. The Kubernetes API server can respond to Contrail Controller after you've configured host-based firewall (HBF) policies on the Contrail user interface. A cSRX image is pulled from the Docker registry to compute nodes after the instances are provisioned.

You can deploy the cSRX as Contrail microsegmentation–Within a Contrail environment running mixed workloads of VMs and containers, cSRX can provide security for Layer 4 through 7 traffic, managed by Security Director.

Figure 14: cSRX Container Firewall on Contrail Host-Based Firewall



This figure illustrates the integration of cSRX with the Contrail HBF using a Docker container. Contrail Security includes an integrated virtual router (vRouter) that acts as a distributed element on every host where a cSRX application is created. The vRouter enforces security at Layers 4 through 7 by monitoring traffic flows and redirecting suspicious traffic to next-generation firewalls.

After you provision the cSRX instances:

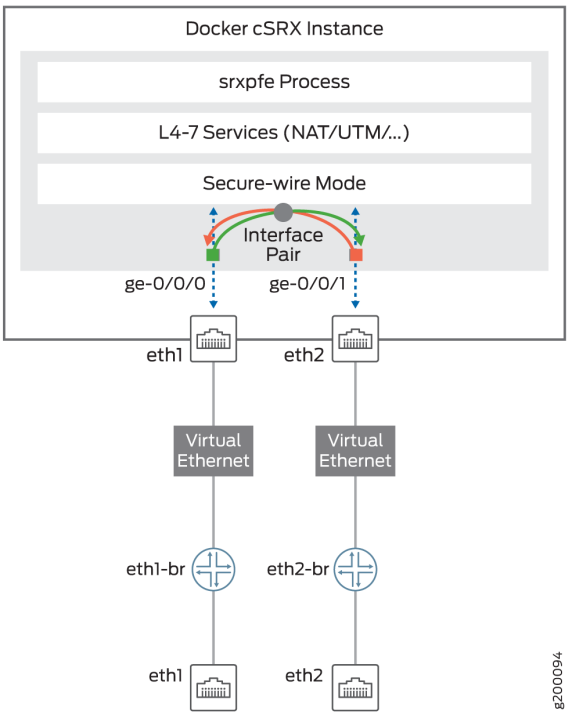
- Three vRouter interfaces (VIFs) connect the cSRX instance to the vRouter.
  - The management interface is connected to the management virtual network.
  - Two secure data interfaces are connected to the left and right virtual networks, receiving packets steered from the vRouter and sending packets to vRouter after security check.
- Security Director updates L7 security policies and dynamic addresses to the cSRX instances.
- The cSRX instances send security logs to Security Director.
- Each tenant that needs the HBF service starts a private cSRX instance on the compute node.

With Contrail Security, you can define policies and automatically distribute them across all deployments. You can also monitor and troubleshoot traffic flows inside each cSRX instance and across cSRX instances.

Contrail HBF supports the cSRX only in secure-wire mode. The secure-wire mode enables advanced security at the network edge in a multitenant virtualized environment. The cSRX provides Layer 4 through Layer 7 advanced security features such as firewall, IPS, and AppSecure. The cSRX container also provides an additional interface to manage the cSRX. When the cSRX operates in Layer 2 mode, the incoming Layer 2 frames from one interface go through Layer 4-Layer 7 processing based on the configured cSRX services. The cSRX then sends the frames out of the other interface. The cSRX container either allows the frames to pass through unaltered or drops the frames, based on the configured security policies.

Figure 15 on page 82 illustrates the cSRX operating in secure-wire mode.

Figure 15: cSRX Container Firewall in Secure-Wire Mode

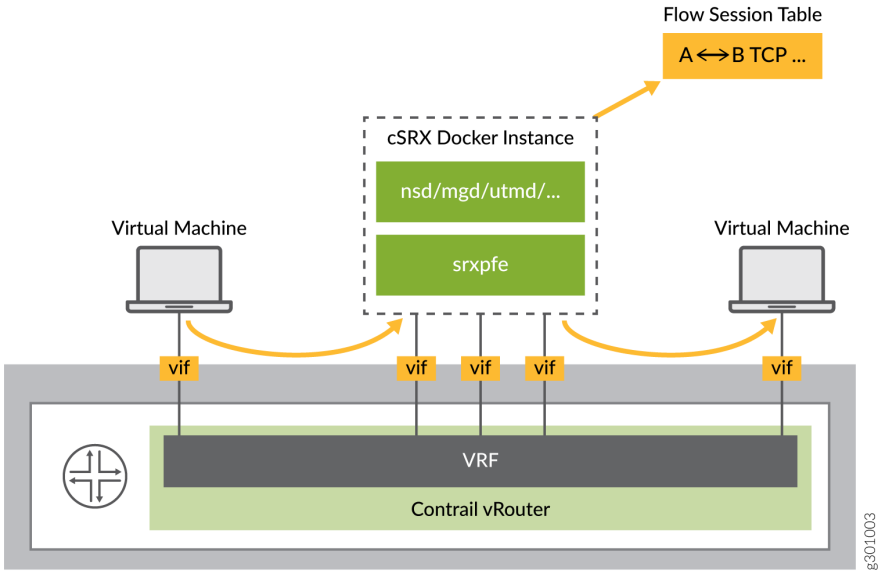


cSRX Container Firewall Deployment Modes

Secure Traffic Inside Compute Node

When the cSRX secures traffic inside a compute node, the vRouter steers the traffic that matches the HBF filter to the cSRX. Flow sessions are created for the traffic sent from the vRouter to the cSRX. After the cSRX completes the L7 security check, it sends the traffic back to the vRouter, which then forwards the traffic to the destination as shown in Figure 16 on page 83.

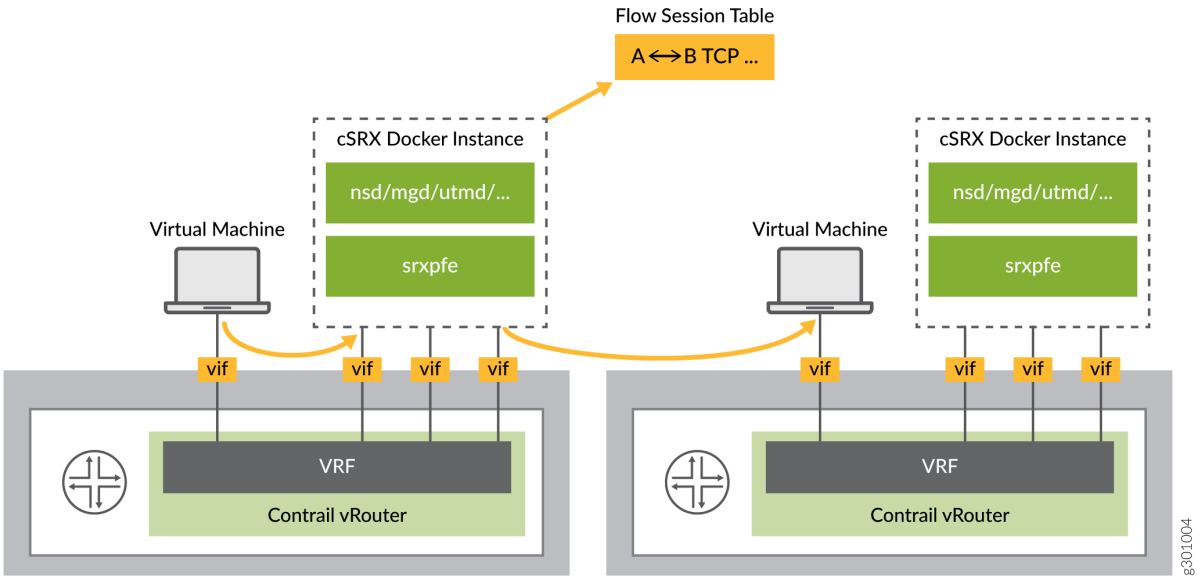
Figure 16: Secure Traffic Inside Compute Node



Secure Traffic Cross Compute Nodes

In this mode, the cSRX works in the same way as when it is securing the traffic inside the compute node. However, in this case the difference is, vRouter needs to guarantee that traffic is steered to the same cSRX instance when traffic is crossing different compute nodes. This mode ensures that the cSRX flow sessions are created and matched in the same cSRX instance in both directions.

Figure 17: Secure Traffic Cross Compute Nodes

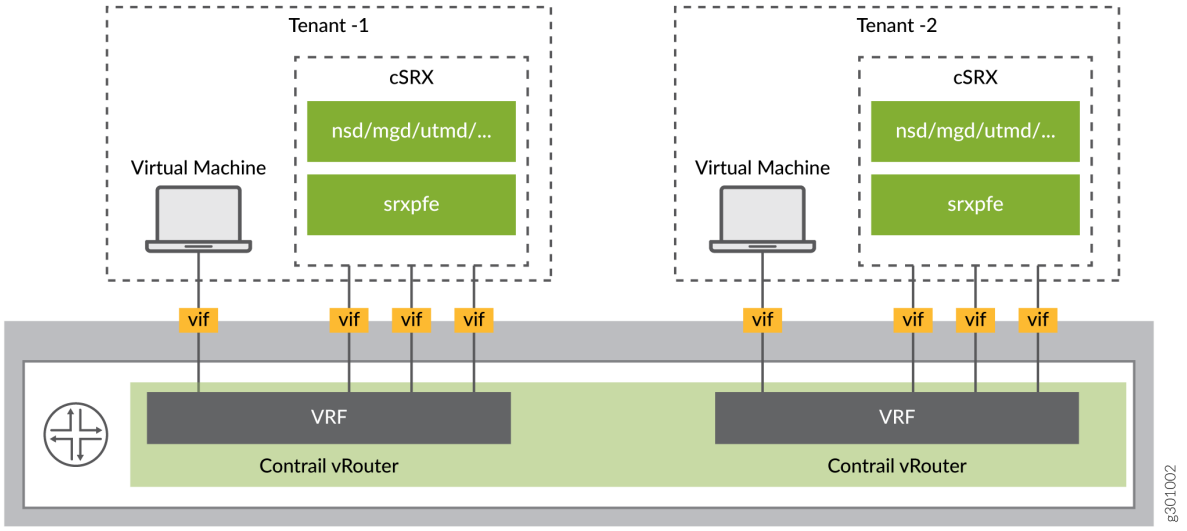


Multitenant Support

For supporting multitenancy, there is separate cSRX instance started for each tenant on same compute node.

Figure 18 on page 85 shows the multitenancy support.

Figure 18: Multitenancy Support



License for cSRX Container Firewall

You need a license to activate the cSRX software features. To understand more about cSRX software feature licenses, see cSRX Flex Software Subscription Model.

Junos OS Features Supported in cSRX for Contrail HBF

cSRX Container Firewall provides Layer 4 through 7 secure services for a Contrail HBF in a containerized environment.Table 7 on page 86 provides a high-level summary of the security features supported on cSRX.

To determine the Junos OS features supported on cSRX, use the Juniper Networks Feature Explorer, a Web-based application that helps you to explore and compare Junos OS feature information to find the right software release and hardware platform for your network. See Feature Explorer.

Table 7: Security Features Supported on cSRX Container Firewall HBF

Security Features	Considerations
Application Tracking (AppTrack)	Understanding AppTrack
Application Firewall (AppFW)	Application Firewall Overview
Application Identification (AppID)	Understanding Application Identification Techniques
Basic Firewall Policy	Understanding Security Basics
Brute force attack mitigation	
DoS/DDoS protection	DoS Attack Overview DoS Attack Overview
Intrusion Prevention System (IPS)	For SRX Series IPS configuration details, see: Understanding Intrusion Detection and Prevention for SRX Series
IPv4	Understanding IPv4 Addressing
Interfaces	Supports two revenue (ge) interfaces. Out-of-band management Interface (eth0 In-band interfaces (ge-0/0/0 to ge-0/0/1)
Jumbo Frames	Understanding Jumbo Frames Support for Ethernet Interfaces
SYN cookie protection	Understanding SYN Cookie Protection
Malformed packet protection	
Routing	Supports secure-wire mode forwarding only.

Table 7: Security Features Supported on cSRX Container Firewall HBF *(Continued)*

Security Features	Considerations
Content Security	Includes support for all Content Security functionality on the cSRX platform, such as: • Antispam • Sophos Antivirus • Web filtering • Content filtering For SRX Series Content Security configuration details, see: Unified Threat Management Overview For SRX Series Content Security antispam configuration details, see: Antispam Filtering Overview
User Firewall	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 For SRX Series user firewall configuration details, see: Overview of Integrated User Firewall
Zones and Zone based IP spoofing	Understanding IP Spoofing

Requirements to Deploy cSRX on Contrail vRouter

IN THIS SECTION

This topic discusses the requirements for integrating cSRX Container Firewall into Contrail cluster.

Contrail Requirements

Table 8 on page 88 lists the supported platforms and server requirements.

Table 8: Supported Platforms and Server Requirements

Component	Specification	Release
Contrail Networking		2005
Ubuntu		14.04 and newer
CentOS		6.5 and newer
Redhat		7.0 and newer
vCPU	2 CPU cores	
Memory	8 GB	

Table 8: Supported Platforms and Server Requirements *(Continued)*

Component	Specification	Release
Disk space	40 GB	
Network Interfaces	2 Revenue Interfaces	

cSRX Container Firewall Container Interfaces

Table 9 on page 89 lists the cSRX container interfaces.

Table 9: cSRX Container Firewall Container Interfaces

Interfaces	Purpose	Created By
eth0	Management Interface	Orchestrator
eth1	ge-0/0/0	Orchestrator
eth2	ge-0/0/1	Orchestrator
lo	Loopback	Docker Engine

cSRX Container Firewall Basic Configuration Settings

The cSRX container requires the following basic configuration settings:

- Interfaces must be bound to security zones.
- Policies must be configured between zones to permit or deny traffic.

CHAPTER 3

Deploy and Configure cSRX Container Firewall into a Contrail Network

IN THIS CHAPTER

cSRX Pod Deployment on Contrail vRouter with Kubernetes

Before you deploy the cSRX Container Firewall as an advanced security service in the Contrail Networking cloud environment, ensure that you:

- Review "Requirements to Deploy cSRX on Contrail vRouter" on page 88 for deploying a cSRX container in a compute node.

Kubernetes is enhanced to support multiple interfaces all supported by a single Contrail Container Network Interface (CNI) (Network Provider). The cSRX container can be orchestrated on compute nodes and attached to multiple virtual networks. For a single cSRX container, those virtual networks are either attached for management purposes or used to collect traffic from vRouter. A cSRX POD can be deployed with a YAML template in Kubernetes.

To deploy a cSRX POD, see Host-Based Firewalls on a compute node.

Debug cSRX Container Firewall in Contrail Network

IN THIS SECTION

Stop a cSRX Pod

By default, cSRX Container Firewall does not mount any external volumes from compute node. When a new cSRX instance is started, then that instance synchronizes the configuration from Security Director. Any syslog and security logs are posted to Security Director as well. So cSRX POD can be stopped and destroyed directly by Contrail Service Orchestration (CSO).

To stop the cSRX POD:

- Run the Docker command to stop cSRX.

  # **kubectl delete -f <csrx-yaml-file>**

  After the cSRX POD is stopped and destroyed, compute and storage resources of this cSRX POD are released.

kubectl delete -f <csrx-yaml-file>

Verify Network Name

To verify the network name:

Run the following command to check the network name:

kubectl get network-attachment-definitions -n

Verify Logs

To view and verify logs:

1. Run the following command to access the path for log details:

   # **cat /var/log/contrail/**

2. Run the following command to view the logs:

   # **kubectl describe pods -n**

PART 5

cSRX Container Firewall Deployment in Bare-Metal Linux Server

cSRX in Bare-Metal Linux Server

IN THIS SECTION

The cSRX Container Firewall is a containerized version of the SRX Series Firewall with a low memory footprint. cSRX is built on the the Junos® operating system (Junos OS) and delivers networking and security features similar to those available on the software releases for the SRX Series. cSRX provides advanced security services, including content security, AppSecure, and Content Security in a container form factor. A bare-metal Linux server uses a Docker container to allow the cSRX Container Firewall to substantially reduce overhead. This efficiency occurs because each container shares the Linux host's OS kernel. Regardless of the number of containers a Linux server hosts, only one OS instance can be in use. Also, because of the light weight of the containers, a server can host many more container instances than that by virtual machines (VMs), yielding tremendous improvements in utilization. With its small footprint and Docker as a container management system, the cSRX enables deployment of agile, high-density security service.

The cSRX enables you to quickly introduce new firewall services, customize services as per your requirements, and scale security services based on dynamic needs. The cSRX differs from VMs in several aspects. The cSRX does not require a guest OS to operate. It has a notably smaller memory footprint and is easier to migrate or download. The cSRX does not require a guest OS to operate. It has a notably smaller memory footprint and is easier to migrate or download. The boot time is reduced from several minutes with a VM-based environment to less than a few seconds with the cSRX container. The cSRX is ideal for public, private, and hybrid cloud environments.

Overview

The cSRX runs as a single container on a Linux bare-metal server which serves as the hosting platform for the Docker container environment. The cSRX container packages comprises all of the dependent processes (daemons) and libraries to support the different Linux host distribution methods (Ubuntu, Red

Hat Enterprise Linux, or CentOS). You can use standard Docker commands to manage the cSRX container.

When the cSRX becomes active, several daemons inside the Docker container launch automatically. Some daemons support Linux features, providing the same services that they provide when running on a Linux host (for example,sshd, rsyslogd, and monit). You can compile and port other daemons from Junos OS to perform configuration and control jobs for security service (for example, and so on). SRX PFE is the data-plane daemon that receives and sends packets from the revenue ports of a cSRX container. The cSRX uses srxpfe for Layer 2 to Layer 3 forwarding functions (secure-wire forwarding or static routing forwarding) as well as for Layer 4 through Layer7 network security services.

The cSRX enables advanced security at the network edge in a multitenant virtualized environment. cSRX provides Layer 4 through Layer 7 advanced security features such as firewall, IPS and AppSecure. When cSRX in Layer 2 secure wire mode, incoming Layer 2 frames from one interface go through Layer 4 through Layer 7 processing based on the configured cSRX services. cSRX then sends the frames out of the other interface.

Launch the cSRX instance in secure-wire mode using the following command:

```
root@csrx-ubuntu3:~/csrx# docker run -d --privileged --network=mgt_bridge -e
CSRX_FORWARD_MODE="wire" --name=<csrx-container-name> <csrx-image-name>
```

> **NOTE**: As part of your Docker container configuration, you must connect the cSRX container to three virtual networks: one virtual network for out-of-band management sessions and two to receive and transmit data traffic. See "Install cSRX in Bare-Metal Linux Server" on page 104.

Figure 19 on page 95 illustrates the cSRX operation in a secure-wire mode. It is an example of how a cSRX container is bridged with an external network. In this illustration, cSRX eth1 is bridged with host physical NIC eth1 and cSRX eth2 is bridged with host physical NIC eth2.

Figure 19: cSRX in Secure-Wire Mode

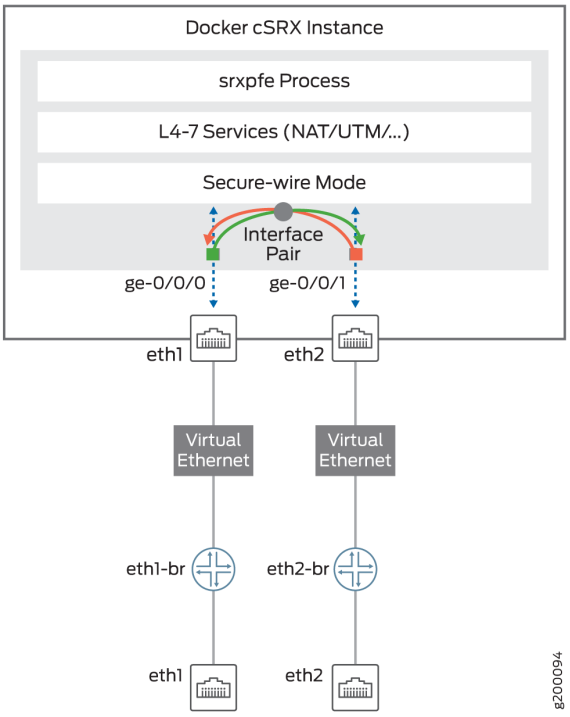
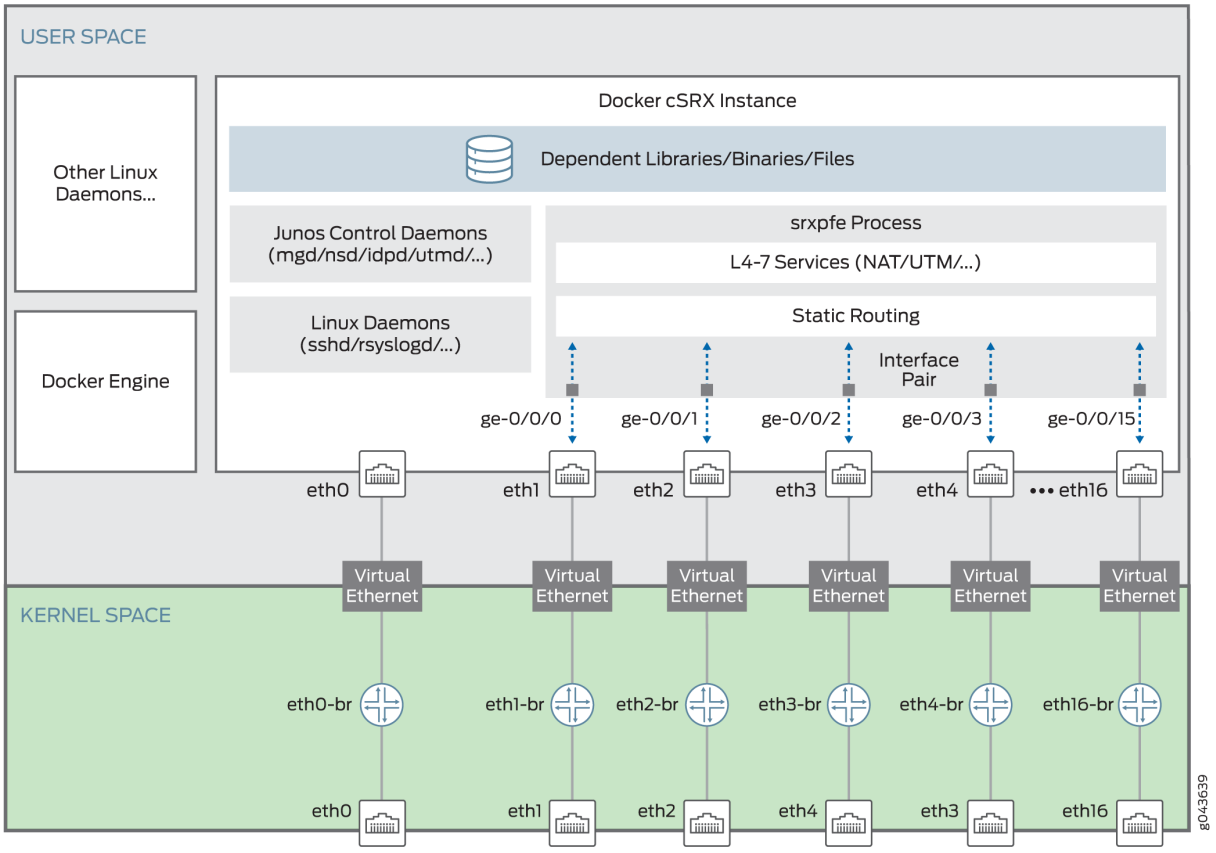


Figure 20 on page 96 illustrates the cSRX operating in routing mode.

Figure 20: cSRX Container Firewall Container in Routing Mode



Starting in Junos OS Release 19.2R1, in routing mode, with the increase in the number of supported interfaces, the mapping of ge interfaces are reordered as:

Prior to Junos OS Release 19.2R1, in routing mode, eth0 was mapped as out-of-band management interface—eth1 as ge-0/0/1 and eth2 as ge-0/0/0.

Starting in Junos OS Release 19.2R1, in routing mode, the default number of interfaces supported are 3 and the maximum number of interfaces supported are 17 (1 management interface and 16 data interfaces). With this increase in the number of interfaces supported, the mapping of ge interfaces is reordered as:

- eth0 - out-of-band management interface
- eth1 - ge-0/0/0
- eth2 - ge-0/0/1
- eth3 - ge-0/0/2
- eth4 - ge-0/0/3 and so on

cSRX Container Firewall Benefits and Uses

Some of the key benefits of cSRX Container Firewall in a containerized private or public cloud multitenant environment include:

- *Stateful firewall* protection at the tenant edge.
- Faster deployment of containerized firewall services into new sites.
- With a small footprint and minimum resource reservation requirements, the cSRX can easily scale to keep up with customers' peak demand.
- Provides 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 or Juniper Networks Contrail.
  - In the Contrail Networking cloud platform, cSRX can be used to provide differentiated Layer 4 through 7 security services for multiple tenants as part of a service chain.
  - With the Contrail orchestrator, cSRX can be deployed as a large scale security service.
- Application security features (including IPS and AppSecure).
- Content Security features (including antispam, Sophos Antivirus, web filtering, and content filtering).
- Authentication and integrated user firewall features.

> **NOTE**: While the security services features between cSRX and vSRX Virtual Firewall are similar, there are scenarios in which each product is the optimal option in your environment. For example, the cSRX does not support routing instances and protocols, switching features, MPLS LSPs and MPLS applications, chassis cluster, and software upgrade features. For environments that require routing or switching, a vSRX Virtual Firewall VM provides the best feature set. For environments focused on security services in a Docker containerized deployment, cSRX is a better fit.
>
> See No Link Title for a summary of the feature categories supported on cSRX, and also for a summary of features not supported on cSRX.

You can deploy the cSRX in the following scenarios:

- Cloud CPE–For service providers (SPs) and managed security service providers (MSSPs) where there is a large subscriber base of branch offices or residential subscribers. MSSPs can offer differentiated services to individual subscribers.

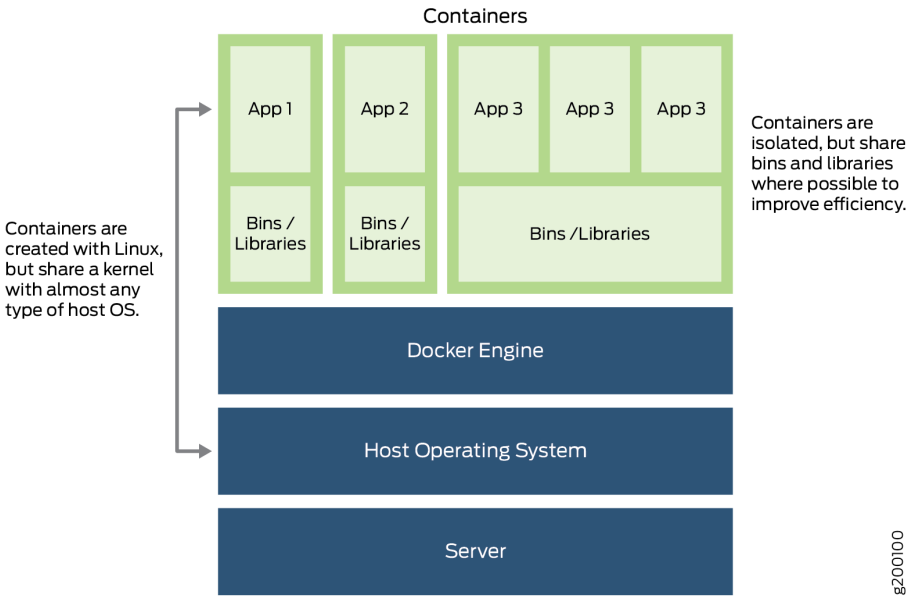
- Contrail microsegmentation–Within a Contrail environment running mixed workloads of VMs and containers, cSRX can provide security for Layer 4 through 7 traffic, managed by Security Director.
- Private clouds–cSRX can provide security services in a private cloud running containerized workloads and can include Contrail integration.

Docker Overview

Docker is an open-source software platform that simplifies the creation, management, and teardown of a virtual container that can run on any Linux server. A Docker container packages applications in "containers" making them portable among any system running the Linux OS.

Figure 21 on page 98 provides an overview of a typical Docker container environment.

Figure 21: Docker Container Environment



cSRX Container Firewall Scale-Up Performance

You can scale the performance and capacity of a cSRX Container Firewall container by increasing the allocated amount of virtual memory or the number of flow sessions. Table 10 on page 99 shows the

cSRX scale-up performance applied to a cSRX container based on its supported sizes. The default size for a cSRX container is large.

NOTE: See Changing the Size of a cSRX Container for the procedure on how to scale the performance and capacity of a cSRX container by changing the container size.

Table 10: cSRX Container Firewall Scale Up Performance

cSRX Container Firewall Size	Specification	Junos OS Release Introduced
vCPUs/Memory	2 vCPU / 4 GB RAM	Junos OS Release 23.2R1
	4 vCPU / 8 GB RAM	
	6 vCPU / 12 GB RAM	
	8 vCPU / 16 GB RAM	
	12 vCPU / 24 GB RAM	
	16 vCPU / 32 GB RAM	
	20 vCPU / 48 GB RAM	
	32 vCPU / 64 GB RAM	

RELATED DOCUMENTATION

- Docker Overview
- What is Docker?
- What is a Container?
- Get Started With Docker

Requirements for Deploying cSRX in Bare-Metal Linux Server

IN THIS SECTION

This section presents an overview of requirements for deploying a cSRX Container Firewall container on a bare-metal Linux server:

Host Requirements

Table 11 on page 100 lists the Linux host requirement specifications for deploying a cSRX container on a bare-metal Linux server.

NOTE: The cSRX can run either on a physical server or virtual machine. For scalability and availability reasons, we recommended using a physical server to deploy the cSRX container.

Table 11: Host Requirement Specifications for cSRX Container Firewall

Component	Specification	Release Introduced
Linux OS support	CentOS 6.5 or later	Junos OS Release 18.1R1
	Red Hat Enterprise Linux (RHEL) 7.0 or later	
	Ubuntu 14.04.2 or later	

Table 11: Host Requirement Specifications for cSRX Container Firewall *(Continued)*

Component	Specification	Release Introduced
Docker Engine	Docker Engine 1.9 or later installed on a Linux host	
Contrail Cloud Platform	Contrail 3.2 with OpenStack Liberty or OpenStack Mitaka	
vCPUs	2 CPU cores	
Memory	4 GB	
Disk space	40 GB hard drive	
Host processor type	x86_64 multicore CPU	
Network interface	1 Ethernet port (minimum)	

cSRX Container Firewall Basic Configuration Settings

The cSRX container requires the following basic configuration settings:

- Interfaces must be assigned IP addresses.
- Policies must be configured between zones to permit or deny traffic.

Interface Naming and Mapping

A cSRX container supports 17 interfaces:

- 1 Out-of-band management Interface (eth0)
- 16 In-band interfaces (ge-0/0/0 to ge-0/0/15).

Table 12 on page 102 lists the cSRX interface assignments with Docker.

Table 12: cSRX Container Firewall Interface Assignment

Interface Number	cSRX Interfaces	Docker Interfaces
1	eth0	eth0
2	ge-0/0/0	eth1
3	ge-0/0/1	eth2
4	ge-0/0/2	eth3
6	ge-0/0/4	eth5
7	ge-0/0/5	eth6
8	ge-0/0/6	eth7
9	ge-0/0/7	eth8
10	ge-0/0/8	eth9
11	ge-0/0/9	eth10
12	ge-0/0/10	eth11
13	ge-0/0/11	eth12
14	ge-0/0/12	eth13
15	ge-0/0/13	eth14
16	ge-0/0/14	eth15

Table 12: cSRX Container Firewall Interface Assignment *(Continued)*

Interface Number	cSRX Interfaces	Docker Interfaces
17	ge-0/0/15	eth16

CHAPTER 4

Deploy cSRX Container Firewall in Bare-Metal Linux Server

IN THIS CHAPTER

Install cSRX in Bare-Metal Linux Server

IN THIS SECTION

This section outlines the steps to install the cSRX Container Firewall container in a Linux bare-metal server environment that is running Ubuntu, Red Hat Enterprise Linux (RHEL) , or CentOS. The cSRX container is packaged in a Docker image and runs in the Docker Engine on the Linux host.

This section includes the following topics:

Before You Deploy

Before you deploy the cSRX as an advanced security service in a Linux container environment, ensure that you:

- Review "Requirements for Deploying cSRX in Bare-Metal Linux Server" on page 100 to verify the system software requirement specifications for the Linux server required to deploy the cSRX container.

- Install and configure Docker on your Linux host platform to implement the Linux container environment. Docker installation requirements vary based on the platform and the host OS (Ubuntu, Red Hat Enterprise Linux (RHEL), or CentOS). Install Docker. You can also use the script at: https://get.docker.com/ to install docker easily. You need to execute this script on shell.

  For docker installation instructions on the different supported Linux host operating systems, see:

  - **Centos/Redhat**—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/
  - **Ubuntu**—https://docs.docker.com/install/linux/docker-ce/ubuntu/

Confirm Docker Installation

Before you load the cSRX image, confirm that Docker is properly installed on the Linux host and that the Docker Engine is running.

To confirm Docker installation:

1. Confirm that Docker is installed and running on the Linux server by using the `service docker status` command.

   ```
   root@csrx-ubuntu3:~# service docker status

   docker start/running, process 701
   ```

   You should also be able to run `docker run hello-world` and see a similar response.

   ```
   root@csrx-ubuntu3:~# docker run hello-world


   Hello from Docker!
     This message shows that your installation appears to be working correctly.
   ```

   - If Docker is not installed, see Install Docker for installation instructions.
   - If Docker is not running, see Configure and troubleshoot the Docker daemon.

2. Verify the installed Docker Engine version by using the `docker version` command.

NOTE: Ensure that Docker version 1.9.0 or later is installed on the Linux host.

```
root@csrx-ubuntu3:~# docker version

Client:

Docker version 17.05.0-ce-rc1, build 2878a85

API Version: 1.30

Go version: go1.8.3

Git commit: 02cid87

Built: Fri June 23 21:17:13 2017

OS/Arch: linux/amd64

Server:

Docker version 17.05.0-ce-rc1, build 2878a85

API Version: 1.30 (minimum version 1.12)

Go version: go1.8.3

Git commit: 02cid87

Built: Fri June 23 21:17:13 2017

OS/Arch: linux/amd64

Experimental: False
```

Load the cSRX Image

Once the Docker Engine has been installed on the host, perform the following to download and start using the cSRX image:

1. Download the cSRX software image from the Juniper Networks website. The filename of the downloaded cSRX software image must not be changed to continue with the installation.
2. You can either download the cSRX image file normally using the browser or use the URL to download the image directly on your device as in the following example:

   Run the following command to downloaded images to a local registry using `curl` command or any other http utility. The syntax for `curl` commands is:

   ```
   root@csrx-ubuntu3:~csrx# curl -o <file destination path> <Download link url>
   ```

root@csrx-ubuntu3:/var/tmp# **curl -o /var/tmp/images/junos-csrx-docker-20.2R1.10.img "https://cdn.juniper.net/software/csrx/20.2R1.10/junos-csrx-docker-20.2R1.10.img?SM_USER=user=1595350694_5dbf6e62442de6bf14079d05a72464d4"**

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  160M  100  160M    0     0  1090k      0  0:02:30  0:02:30 --:--:-- 1230k
```

3. Locate the cSRX image by using the `ls` Linux shell command.

   root@csrx-ubuntu3:/var/tmp/images# **ls**

4. Load the downloaded cSRX image to the local registry.

   root@csrx-ubuntu3:/var/tmp/images# **docker image load -i /var/tmp/images/junos-csrx-docker-20.2R1.10.img**

```
e758932b9168: Loading layer [==================================================>]    263MB/
263MB
23f7a9961879: Loading layer [==================================================>]  14.51MB/
14.51MB
1e4139e6fa81: Loading layer [==================================================>]  270.3MB/
270.3MB
10334b424f86: Loading layer [==================================================>]   16.9kB/
16.9kB
202ebb2f1137: Loading layer [==================================================>]   2.56kB/
2.56kB
bc4a16173327: Loading layer [==================================================>]  1.536kB/
1.536kB
8f9a9945544a: Loading layer [==================================================>]  2.048kB/
2.048kB
Loaded image: csrx:20.2R1.10
```

5. After the cSRX image loads, confirm that it is listed in the repository of Docker images.

   root@csrx-ubuntu3:/var/tmp/images# **docker images**

```
REPOSITORY                    TAG                 IMAGE ID            CREATED
SIZE
csrx                          20.2R1.10           88597d2d4940        2 weeks ago
534MB
```

Create Linux Bridge Network for cSRX

A Linux bridge is a virtual switch implemented as a kernel module. This Linux bridge is used within a Linux host to emulate a hardware bridge. Docker allows you to create a Linux bridge network and connect the cSRX container to this network to implement management and data processing sessions. The interfaces are created with the Linux VETH driver and are used to communicate with the Linux kernel.

This procedure describes how to create a three-bridge network for the cSRX container that includes: mgt_bridge (eth0), left_bridge (eth1), and right_bridge (eth2). The mgt_bridge is used by the cSRX for out-of-band management to accept management sessions and traffic, and the left_bridge and right_bridge are both used by the cSRX as the revenue ports to process in-band data traffic.

NOTE: Docker automatically connects the management interface (eth0) to the Linux bridge and assigns an IP address. Interfaces eth1 and eth2 are for the inband traffic. cSRX must be bound with the Linux bridge to pass traffic.

To create a three-bridge network for a cSRX in the Linux host:

1. Create the management bridge in the network.

   ```
   root@csrx-ubuntu3::~/csrx# docker network create --driver bridge mgt_bridge

   3228844986eae1d1a8d367b34b54b31b130842be072b9dcdf7da3601c95b7130
   ```

2. Create the left bridge in the network (untrusted interface (eth1)).

   ```
   root@csrx-ubuntu3::~/csrx# docker network create --driver bridge left_bridge

   f1324b0a9072c55ababbcc51d83c83658084b67513811e13829172cccbc08e5d
   ```

3. Create the right bridge in the network (trusted interface (eth2)).

   ```
   root@csrx-ubuntu3::~/csrx# docker network create --driver bridge right_bridge

   196bd039f7c2401df4c117ea684114548a3df0b9d406cf3cf8f17338fab96774
   ```

RELATED DOCUMENTATION

- Docker commands

Launch cSRX in Bare-Metal Linux Server

You are now ready to launch the cSRX Container Firewall container that is running in Docker on the Linux bare-metal server. When you start the cSRX image, you have a running container of the image. You can stop and restart the cSRX container (see "Manage cSRX in Bare-Metal Linux Server" on page 123), and the container retains all the settings and file system changes unless those changes are explicitly deleted. However, the cSRX looses anything in memory and all processes are restarted.

You have a series of cSRX environment variables that enable you to modify operating characteristics of the cSRX container when it is launched. You can modify:

- When you deploy cSRX you must enable the SSH service and SSH option for root-login. SSH service is not enabled by default.

  To enable SSH service run the `set system services ssh` command and for root user login run the `set system services ssh root-login allow` command.

- Traffic forwarding mode (static route or secure-wire)
- cSRX container size (small, medium, or large)
- Packet I/O driver (polled or interrupt)
- CPU affinity for cSRX control and data daemons
- Address Resolution Protocol (ARP) and Neighbor Discovery Protocol (NDP) entry timeout values
- Number of interfaces you need to add to container. Default is 3 and maximum is 17 (which means 1 management interfaces and 16 data interfaces).

> **NOTE**: Specification of an environment variable is not mandatory when launching the cSRX container; most environment variables have a default value as shown in "cSRX Environment Variables Overview" on page 113. You can launch the cSRX using the default environment variable settings.

To launch the cSRX container:

1. Use the `docker run` command to launch the cSRX container. You include the `mgt_bridge` management bridge to connect the cSRX to a network.

   `root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e --name=<csrx-container-name> hub.juniper.net/security/<csrx-image-name>**

   For example, to launch `csrx2` using cSRX software image `csrx:18.21R1.9` enter:

```
root@csrx-ubuntu3:~/csrx# docker run -d --privileged --network=mgt_bridge -e --name=csrx2
hub.juniper.net/security/csrx:18.2R1.9
```

NOTE: You must include the `--privileged` flag in the `docker run` command to enable the cSRX container to run in privileged mode.

2. Connect the left and right bridges to the Docker network.

```
root@csrx-ubuntu3:~/csrx# docker network connect left_bridge csrx2

root@csrx-ubuntu3:~/csrx#

root@csrx-ubuntu3:~/csrx# docker network connect right_bridge csrx2

root@csrx-ubuntu3:~/csrx#
```

3. Confirm that the three-bridge network has been created for the cSRX container.

```
root@csrx-ubuntu3:~/csrx# docker network ls

NETWORK ID NAME DRIVER SCOPE

80bea9207560 bridge bridge local

619da6736359 host host local

112ab00aab1a left_bridge bridge local

1484998f41bb mgt_bridge bridge local

daf7a5a477bd none null local

e409a4f54237 right_bridge bridge local
```

4. Confirm that the cSRX container is listed as a running Docker container.

```
root@csrx-ubuntu3:~/csrx# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

35e33e8aa4af csrx "/etc/rc.local init" 7 minutes ago Up 7 minutes 22/tcp, 830/tcp csrx2
```

5. Confirm that the cSRX container is up and running. You should see the expected Junos OS processes, such as nsd, srxpfe, and mgd.

```
root@csrx-ubuntu3:~/csrx# docker top csrx2
```

```
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                318                 305                 0
```

```
09:13              pts/1              00:00:00           bash
root               27423              27407              0
Mar30              pts/0              00:00:00           /bin/bash -e /etc/rc.local init
root               27867              27423              0
Mar30              ?                  00:08:16           /usr/sbin/rsyslogd -M/usr/lib/
rsyslog
root               27880              27423              0
Mar30              ?                  00:00:00           /usr/sbin/sshd
root               27882              27423              0
Mar30              ?                  00:00:00           /usr/sbin/nstraced
root               27907              27423              0
Mar30              ?                  00:00:08           /usr/sbin/mgd
root               27963              27423              0
Mar30              pts/0              00:34:50           /usr/bin/monit -I
root               27979              27423              0
Mar30              ?                  00:01:10           /usr/sbin/nsd
root               27989              27423              0
Mar30              ?                  00:00:02           /usr/sbin/appidd -N
root               28023              27423              0
Mar30              ?                  00:00:21           /usr/sbin/idpd -N
root               28040              27423              0
Mar30              ?                  00:09:21           /usr/sbin/wmic -N
root               28048              27423              0
Mar30              ?                  00:52:50           /usr/sbin/useridd -N
root               28126              27423              2
Mar30              ?                  1-05:21:47         /usr/sbin/srxpfe -a -d
root               28186              27423              0
Mar30              ?                  00:01:37           /usr/sbin/utmd -N
root               28348              27423              0
Mar30              ?                  00:02:44           /usr/sbin/kmd
```

6. Confirm the IP address of the management interface of the cSRX container.

```
root@csrx-ubuntu3:~/csrx# docker inspect csrx2 | grep IPAddress
```

```
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.19.0.2",
                    "IPAddress": "172.18.0.2",
                    "IPAddress": "172.20.0.2",
```

RELATED DOCUMENTATION

- Docker commands

CHAPTER 5

Configure and Manage cSRX Container Firewall in Bare-Metal Linux Server

IN THIS CHAPTER

cSRX Environment Variables Overview

Docker allows you to store data for example configuration settings, as environment variables. At runtime, the environment variables are exposed to the application inside the container. You can set any number of parameters to take effect when the cSRX Container Firewall image launches. You can set an environment variable by specifying the `docker run -e VARIABLE=VALUE ...` key.

A series of cSRX environment variables enables you to modify the characteristics of the cSRX instance when it is launched. The specification of an environment variable is not mandatory; most environment variables have a default value as shown in Table 13 on page 114. If desired, you can launch the cSRX using the default environment variable settings.

For example, to launch a cSRX instance in secure-wire forwarding mode using the CSRX-2CPU-4G size cSRX configuration:

`root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e CSRX_FORWARD_MODE="wire" --name=<csrx-container-name> <csrx-image-name>**

NOTE: You must include the `--privileged` flag in the `docker run` command to enable the cSRX container to run in privileged mode.

Table 13 on page 114 summarizes the list of available cSRX environment variables along with a link to the topic that outlines its usage.

Table 13: Summary of cSRX Container Firewall Environment Variables

Variable	Description	Values	Default	Topic
`CSRX_FORWARD_MODE`	Traffic forwarding mode	`"routing"` | `"wire"`	`"routing"`	"Configure Traffic Forwarding on cSRX" on page 116
`CSRX_PACKET_DRIVER`	Packet I/O driver	`"poll"` | `"dpdk"` | `"interrupt"` **NOTE**: The "interrupt" and "poll" modes are only supported for large flavor of cSRX (CSRX-2CPU-4G), otherwise only "dpdk" mode is supported for any cSRX size larger than that.	`"poll"`	Specifying the Packet I/O Driver for a cSRX Container
`CSRX_CTRL_CPU`	CPU mask, indicating which CPU is running the cSRX control plane daemons (such as nsd, mgd, nstraced, utmd, and so on)	*`hex value`*	No CPU affinity	Configuring CPU Affinity for a cSRX Container

Table 13: Summary of cSRX Container Firewall Environment Variables *(Continued)*

Variable	Description	Values	Default	Topic
CSRX_DATA_CPU	CPU mask, indicating which CPU is running the cSRX data plane daemon (srxpfe)	*hex value*	No CPU affinity	Configuring CPU Affinity for a cSRX Container
CSRX_ARP_TIMEOUT	ARP entry timeout value for the control plane ARP learning or response	*decimal value*	Same as the Linux host	"Configure Traffic Forwarding on cSRX" on page 116
CSRX_NDP_TIMEOUT	NDP entry timeout value for the control plane NDP learning or response	*decimal value*	Same as the Linux host	"Configure Traffic Forwarding on cSRX" on page 116
CSRX_PORT_NUM	Number of interfaces you need to add to the container Example: docker run -d --privileged --net=none -e CSRX_PORT_NUM=17 e CSRX_HUGEPAGES=no -e CSRX_PACKET_DRIVER=interrupt -e CSRX_FORWARD_MODE=routing --name=<cSRX-container-name> <cSRX-image-name>	Default is 3, maximum is 17 (1 management interface and 16 data interfaces)	3	

Table 13: Summary of cSRX Container Firewall Environment Variables *(Continued)*

Variable	Description	Values	Default	Topic
CSRX_HUGEPAGES	You can set this env variable to "yes" or "no" to enable or disable using hugepages in cSRX. By default, cSRX will set CSRX_HUGEPAGES to "no"	**NOTE**: This variable must be set to "yes" for any size larger than CSRX-2CPU-4G.	It is important to note that cSRX only supports 1G hugepages. For some flavors of cSRX, it is required to set CSRX_HUGEPAGES = "yes".	

Change the Size of cSRX

Based on your specific cSRX Container Firewall deployment requirements, scale requirements, and resource availability, you can scale the performance and capacity of a cSRX instance by specifying a specific size (small, middle, or large). Each cSRX size has certain characteristics and can be applicable to certain deployments. By default, the cSRX container launches using the large size configuration.

To assign a specific size for a cSRX instance, include the `CSRX_SIZE` environment variable in the `docker run` command.

For example, to launch a cSRX instance using the CSRX-2CPU-4G size configuration to scale performance and capacity:

`root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e CSRX_SIZE="CSRX-2CPU-4G" --name=<csrx-container-name> <csrx-image-name>**

Configure Traffic Forwarding on cSRX

IN THIS SECTION

You can change the traffic forwarding mode of the cSRX Container Firewall container as a means to facilitate security service provisioning when running the cSRX. For example, if you deploy a cSRX container inline of protected segments, the cSRX should be transparent to avoid changing the virtual network topology. In other deployments, the cSRX container should be able to specify the next-hop address of egress traffic. To address variations in cSRX network deployment, you can configure the traffic forwarding mode of the cSRX to operate in routing mode (static routing only) or secure-wire mode.

NOTE: The cSRX uses `routing` as the default environment variable for traffic forwarding mode.

This section includes the following topics:

Configure Routing Mode

When running the cSRX container in routing mode, the cSRX uses a static route to forward traffic for routes destined to interfaces ge-0/0/0 and ge-0/0/1. You must create a static route and specify the next-hop address.

When you start the cSRX container, you need to specify port number in the environment using the variable CSRX_PORT_NUM to define the number of interfaces you need to add to container in routing mode.

For example, to launch cSRX instance in routing mode with 17 interfaces:

`root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --net=none -e CSRX_PORT_NUM=17 CSRX_SIZE=large -e CSRX_HUGEPAGES=no -e CSRX_PACKET_DRIVER=interrupt -e CSRX_FORWARD_MODE=routing --name=<srx-container-name> <csrx-image-name>**

NOTE: The interfaces specified in the CSRX_PORT_NUM environment variable (default value is 3) must be added to a network after instantiation of the cSRX. Unless all the interfaces are added to the bridge or the macvlan networks, the PFE does not launch on the cSRX, and the ge-x/y/z interfaces remains down.

Include the `-e CSRX_FORWARD_MODE=routing` environment variable in the `docker run` command to instruct the cSRX to run in static route forwarding mode.

To configure the cSRX container to run in static routing mode:

1. Launch the cSRX container in routing forwarding mode:

   `root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e CSRX_FORWARD_MODE="routing" --name=<csrx-container-name> <csrx-image-name>**

2. Log into cSRX instance and start configuration mode.

```
root@csrx# cli
root@csrx> configure
[edit]
```

3. Configure interfaces.

   Starting from 19.2R1.8, each cSRX can be configured with up to 15 revenue inter-faces: eth1, eth2, and so on, until eth15. The number of interfaces can be predefined while booting up a cSRX. Usually, management IP on a cSRX is assigned by docker based on network settings while spinning the cSRX(--network=mgt_bridge). If you don't specify this variable, docker is going to assign IP from default docker network bridge.

   The eth0 is used by the cSRX for out-of-band management to the accept management sessions and traffic, and eth1 and eth2 are both used by the cSRX as the two revenue ports to process in-band data traffic (the ge-0/0/0 and ge-0/0/1 interfaces).

   For this example, assume that the docker default or the custom network management bridge is 172.31.21.0/24, docker assigns one IP address from this network. If your cSRX is the first container on the system, then cSRX is assigned with 172.31.21.2 and default gateway for the cSRX management plane is assigned with 172.31.21.1.

   Table 14: IP Address Assignment for Interfaces

Interface	IP Address
Management Interface eth0 (fxp0)	172.31.21.1
Default gateway for the cSRX management plane	172.31.21.2
Eth1 (ge-0/0/0)	172.19.0.2/24
Eth2 (ge-0/0/1)	172.20.0.2/24
External Server	10.10.10.0

```
root@csrx# show | display set

root@csrx# set interfaces ge-0/0/0 unit 0 family inet address 172.19.0.2/24

root@csrx# set interfaces ge-0/0/1 unit 0 family inet address 172.20.0.2/24
```

4. Configure static routes.

   Configure static route and specify next-hop address.

   root@csrx# **set routing-options static route 0.0.0.0/0 next-hop 172.19.0.2/24**

5. View the forwarding table to verify the static routes.

   root@csrx> **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      517     1
172.19.0.2            perm     0 172.19.0.10            locl     2006     1
172.19.0.10           perm     0 172.19.0.10          ucast     5501     1
1.255.255.255      perm     0                    bcst     2007     1
1/8                perm     0                    rslv     2009     1172.20.0.2
perm     0 172.20.0.2            locl     2001     1
172.20.0.10           perm     0 172.20.0.10          ucast     5500     1
2.255.255.255      perm     0                    bcst     2002     1
2/8                perm     0                    rslv     2004     1
224.0.0.1          perm     0                    mcst      515     1
224/4              perm     0                    mdsc      516     1
172.31.21.2/28         perm     0 172.20.0.10          ucast     5501     1

Routing table: default.inet6
Internet6:
Destination        Type RtRef Next hop           Type Index    NhRef Netif
::                 perm     0                    dscd      527     1
ff00::/8           perm     0                    mdsc      526     1
ff02::1            perm     0                    mcst      525     1
```

6. Specify a route for the management interface. Static routes can only configure routes destined for interfaces ge-0/0/0 and ge-0/0/1. The route destined for the management interfaces (eth0) must be added by using the Linux `route` shell command.

   root@csrx% **route add -net 10.10.10.0/24 gw 172.31.21.1**

   root@csrx% **route -n**

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         0.0.0.0         0.0.0.0         U     0      0        0 pfe_tun
```

```
172.19.0.2          0.0.0.0          255.0.0.0       U     0      0        0 tap1
172.20.0.2          0.0.0.0          255.0.0.0       U     0      0        0 tap0
172.31.21.2          1.0.0.10         255.255.255.240 UG    0      0        0 tap1
10.10.10.0      172.31.21.1     255.255.255.0   UG    0      0        0 eth0
172.21.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
```

7. If required for your network environment, you can configure an IPv6 static route for the cSRX using the `set routing-options rib inet6.0 static route` command.

```
[edit routing-options]

root@csrx# set routing-options rib inet6.0 static route 3000::0/64 next-hop 1000::10/128

[edit interfaces]

root@csrx# commit

root@csrx# show routing-options rib inet6.0

static {

route 3000::0/64 next-hop 1000::10/128;

}
```

8. Under routing mode, the control plane ARP/NDP learning/response is provided by the Linux kernel through the TAP 0 and TAP 1 interfaces created to host the traffic for eth1 and eth2 through srxpfe. You can view ARP entries by using the Linux `arp` shell command.

> **NOTE**: While there are multiple interfaces created inside the cSRX container, only two interfaces, ge-0/0/0 and ge-0/0/1, are visible in srxpfe.

```
root@csrx% arp -a


? (2.0.0.10) at 6e:81:38:41:5e:0e [ether] on tap0
? (1.0.0.10) at 96:33:66:a1:e5:03 [ether] on tap1
? (172.31.12.1) at 02:c4:39:fa:0a:0d [ether] on eth0
```

The default ARP/NDP entries timeout is set to 1200 seconds. You can adjust this value by modifying either the `ARP_TIMEOUT` or `NDP_TIMEOUT` environment variable when launching the cSRX container. For example:

root@csrx-ubuntu3:~/csrx# **docker run -d --privileged --network=mgt_bridge -e CSRX_FORWARD_MODE="routing" -e CSRX_ARP_TIMEOUT=<seconds> -e CSRX_NDP_TIMEOUT=<seconds> --name=<csrx-container-name> <csrx-image-name>**

The maximum ARP entry number is controlled by the Linux host kernel. If there are a large number of neighbors, you might need to adjust the ARP or NDP entry limitations on the Linux host. There are options in the `sysctl` command on the Linux host to adjust the ARP or NDP entry limitations.

For example, to adjust the maximum ARP entries to 4096:

sysctl -w net.ipv4.neigh.default.gc_thresh1=1024

sysctl -w net.ipv4.neigh.default.gc_thresh2=2048

sysctl -w net.ipv4.neigh.default.gc_thresh3=4096

For example, to adjust the maximum NDP entries to 4096:

sysctl -w net.ipv6.neigh.default.gc_thresh1=1024

sysctl -w net.ipv6.neigh.default.gc_thresh1=2048

sysctl -w net.ipv6.neigh.default.gc_thresh1=4096

Configure Secure-Wire Mode

When operating in secure-wire mode, all traffic that arrives on a specific interface, ge-0/0/0 or ge-0/0/1, is forwarded unchanged through the interface. This mapping of interfaces, called *secure wire*, allows the cSRX to be deployed in the path of network traffic without requiring a change to routing tables or a reconfiguration of neighboring devices. A cross-connection is set up between interface pairs ge-0/0/0 and ge-0/0/1 to steer traffic from one port to the other port based on the Interworking and Interoperability Function (IIF) as the input key.

Include the `-e CSRX_FORWARD_MODE=wire` environment variable in the `docker run` command to instruct the cSRX to run in secure-wire forwarding mode.

NOTE: When you launch the cSRX container in secure-wire mode, the cSRX instance automatically creates a default secure-wire named *csrx_sw* in the srxpfe process, and the ge-0/0/0 and ge-0/0/1 interface pair are added into the secure-wire.

Launch the cSRX instance in secure-wire mode using the following command:

root@csrx-ubuntu3:~/csrx# **docker run -d --privileged --network=mgt_bridge -e CSRX_FORWARD_MODE="wire" --name=<csrx-container-name> <csrx-image-name>**

Configure CPU Affinity on cSRX

A cSRX Container Firewall instance requires two CPU cores in the Linux server. To help schedule the Linux server tasks and adjust performance of the cSRX running on a Linux host, you can launch the cSRX container and assign its control and data processes (or daemons) to a specific CPU. In a cSRX container, srxpfe is the data plane daemon and all other daemons (such as nsd, mgd, nstraced, utmd, and so on) are control plane daemons.

CPU affinity ensures that the cSRX control and data plane daemons are pinned to a specific physical CPU, which can improve the cSRX container performance by using the CPU cache efficiently. By default, there is not a defined CPU affinity for the cSRX control and data plane daemons; the CPU on which the control and data plane daemons run depends on Linux kernel scheduling.

To assign cSRX container control and data daemons to a specific CPU, include the environment variables `CSRX_CTRL_CPU` and `CSRX_DATA_CPU` in the `docker run` command.

For example, to configure the cSRX container to launch the control plane daemons on CPU 1 and the data plane daemon on CPU 2:

`root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e CSRX_CTRL_CPU="0x1" -e CSRX_DATA_CPU="0x2" --name=<csrx-container-name> <csrx-image-name>**

Enable Persistent Log File Storage to a Linux Host Directory

In a cSRX Container Firewall container, log files are stored in the /var/log directory. By default, if there are no external volumes mounted for the /var/log directory, the log files are maintained only for this cSRX Firewall container. If, in future, the cSRX container is deleted, those log files are lost. You can enable persistent log file storage to a Linux host directory as a means to directly mount a directory from a Linux host to the cSRX container when the cSRX is launched.

To configure the cSRX container to enable persistent log file storage to a Linux host directory, use the following command.

`root@csrx-ubuntu3:~/csrx#` **docker run -d --privileged --network=mgt_bridge -e CSRX_FORWARD_MODE="routing" -e CSRX_PACKET_DRIVER="poll" –e CSRX_CTRL_CPU="0x1" –e CSRX_DATA_CPU="0x6" -v <path-log-directory-on-host>:/var/log --name=<csrx-container-name> <csrx-image-name>**

Manage cSRX in Bare-Metal Linux Server

IN THIS SECTION

This section outlines basic Docker commands that you can use with a running cSRX Container Firewall container. It includes the following topics:

Pause or Resume Processes Within cSRX

You can suspend or resume all processes within one or more cSRX containers. On Linux, this task is performed using the `cgroups freezer` process.

To pause and restart a cSRX container:

1. Use the `docker pause` command to suspend all processes in a cSRX container.

```
hostOS# docker pause <csrx-container-name>
```

2. Use the `docker unpause` command to resume all processes in the cSRX container.

```
hostOS# docker unpause <csrx-container-name>
```

View Processes on a Running cSRX Container

Use the `docker exec` command to view the details of the processes (applications, services and status) running on a cSRX container.

```
hostOS# docker exec <csrx-container-name> ps aux
```

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  18048  1648 pts/8    Ss   May15   0:00 /bin/bash -e /etc/rc.local init
root        78  0.0  0.0 260072   968 ?        Ssl  May15   0:09 /usr/sbin/rsyslogd -M/usr/lib/
```

```
rsyslog
root        97  0.0  0.0  61376  1304 ?        Ss   May15   0:00 /usr/sbin/sshd
root       118  0.0  0.0 108552  1304 ?        Sl   May15  34:12 /usr/bin/monit
root       124  0.0  0.0 723392  1516 ?        Ss   May15   0:00 /usr/sbin/nstraced
root       133  0.0  0.0 734084  4388 ?        Ss   May15   1:18 /usr/sbin/nsd
root       135  0.0  0.0   4440   644 ?        S    May15   0:00 /bin/sh /etc/init.d/appidd start
root       141  0.0  0.2 752132 21184 ?        Sl   May15   0:02 /usr/sbin/appidd -N &
root       147  0.0  0.0   4440   652 ?        S    May15   0:00 /bin/sh /etc/init.d/idpd start
root       153  0.0  0.0 730520  2768 ?        S    May15   0:25 /usr/sbin/idpd -N &
root       170  0.0  0.1 1001088 12528 ?       Sl   May15  29:22 /usr/sbin/useridd -N
root       211  0.0  0.0 728448  2104 ?        Ss   May15   0:07 /usr/sbin/mgd
root       222  3.5  1.8 3943936 152920 ?      Sl   May15 1416:22 /usr/sbin/srxpfe -a -d
root       250  0.0  0.0   4440   648 ?        S    May15   0:00 /bin/sh /etc/init.d/utmd start
root       256  0.0  0.0 725092  3880 ?        S    May15   1:36 /usr/sbin/utmd -N &
root       267  0.0  0.0 731556  2472 ?        Ss   May15   2:39 /usr/sbin/kmd
root       301  0.0  0.0  18160  1916 pts/8    S+   May15   0:00 /bin/bash
root       324  0.0  0.0 853708  3324 ?        Sl   May15   6:13 /usr/sbin/wmic -N
```

Remove a cSRX Container or Image

To remove a cSRX container or image:

NOTE: You must first stop and remove a cSRX container before you can remove a cSRX image.

1. Use the `docker stop` command to stop the cSRX container.

```
hostOS# docker stop <csrx-container-name>
```

2. Use the `docker rm` command to remove the cSRX container.

```
hostOS# docker rm <csrx-container-name>
```

NOTE: Include `--force` to force the removal of a running cSRX container.

3. Use the `docker rmi` command to remove one or more cSRX images from the Docker Engine.

NOTE: Include `--force` to force the removal a cSRX image.

```
hostOS# docker rmi <csrx-container-name>
```

SEE ALSO

- Docker Engine User Guide
- Docker commands

cSRX Configuration and Management Tools

IN THIS SECTION

Understanding the Junos OS CLI and Junos Scripts

The Junos operating system command-line interface (Junos OS CLI) is a Juniper Networks specific command shell that runs on top of a UNIX-based operating system kernel.

Built into Junos OS, Junos script automation is an onboard toolset available on all Junos OS platforms, including routers, switches, and security instances.

You can use the Junos OS CLI and the Junos OS scripts to configure, manage, administer, and troubleshoot the cSRX Container Firewall container.

Understanding cSRX Container Firewall with Contrail and Openstack Orchestration

The cSRX Container Firewall Container Firewall can provide security services in a software-defined networking (SDN) environment. Juniper Networks Contrail is an open, standards-based software-defined networking (SDN) platform that delivers network *virtualization* and service automation for federated cloud networks. You use the Contrail Cloud Platform with open cloud orchestration systems

such as OpenStack or CloudStack to instantiate instances of cSRX Container Firewall in a containerized environment. Contrail Cloud Platform automates the orchestration of compute, storage, and networking resources to create and scale open, intelligent, and reliable OpenStack clouds that seamlessly merge and hybridize through highly intelligent secure networks.

cSRX Container Firewall can be deployed as a dedicated firewall compute node in a Contrail Cloud platform environment to provide differentiated Layer 4 through 7 security services for multiple tenants as part of a service chain in the Contrail cloud platform. In the Contrail networking environment, you can deploy the cSRX Container Firewall container as a large-scale security service in a multicloud environment, and configure the cSRX Container Firewall to steer traffic from a vRouter with vRouter interface (VIF). Traffic and health statistics are monitored by the Contrail service orchestrator.

See cSRX Guide for Contrail for details on using cSRX Container Firewall with Juniper Networks Contrail.

RELATED DOCUMENTATION

- Introducing the Junos OS Command-Line Interface
- Contrail Networks
- Mastering Junos Automation Programming