JUNIPER NETWORKS | Engineering Simplicity

DAY ONE: AMAZON WEB SERVICES WITH vSRX COOKBOOK

Extend your existing network and security practices into the AWS environment without compromising performance or compliance.

By Chirag Patel, Ali Bidabadi, Charlie Chang-Hyun Kim, Lionel Ruggeri, Peter Cucinell, Scott Sneddon, Tony Boerema, Adam Grochowski

DAY ONE: AMAZON WEB SERVICES WITH vSRX COOKBOOK

By leveraging the Juniper Networks' offerings available in the AWS Marketplace, enterprise architects can provide network connectivity and security policy enforcement that exactly mirrors what they do in their private clouds. They can extend their existing network and security operational practices into the AWS environment, without compromising performance or compliance. And as their operational practices evolve to a fully automated NetSecDevOps model, they can leverage the rich Junos® automation framework as well as the tight integration that Juniper Networks has built with the native AWS automation tools.

Day One: Amazon Web Services with vSRX Cookbook provides a hands-on introduction to public cloud building using things you already know, like Junos and vSRX. Follow along on AWS as you build the basics and then build out your network services.

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

- IProvision Junos on AWS
- Launch a Juniper instance on AWS
- Launch Juniper instances using Ansible Playbook
- Set up AWS for CloudFormation templates
- Launch a Juniper instance with Terraform
- Use dual vSRX/vMX spokes with on-box Python and Terraform templates
- Transit VPC deployment using CloudFormation templates
- Build a Multi-Region Full Mesh Transit VPC
- Extend Your data center
- Enable Adaptive Security with SDSN

ISBN 978-1-941441-79-4

Juniper Networks Books are focused on network reliability and efficiency. Peruse the complete library at www.juniper.net/books.

JUNIPER NETWORKS

Day One: Amazon Web Services with vSRX Cookbook

by Chirag Patel, Ali Bidabadi, Charlie Chang-Hyun Kim, Lionel Ruggeri, Peter Cucinell, Scott Sneddon, Tony Boerema, Adam Grochowski

Table of Contents

1: Build Your AWS Setup

Preface: Why Juniper on AWS? viii

Recipe 1 - Provisioning Junos on AWS 11

Recipe 2 - Launching a Juniper Instance 14

Recipe 3 - Launching Juniper Instances with Ansible Playbook 27

Recipe 4 - Setting Up AWS for CloudFormation Templates 35

Recipe 5 - Launching Juniper Instances with Terraform 42

2: Juniper and AWS Showcase

Recipe 6 - High Availability Design Using Dual Spokes with On-Box Python and Terraform Templates 52

Recipe 7 - Transit VPC Deployment Using CloudFormation Templates 60

Recipe 8 - Multi-Region Full Mesh Transit VPC with vSRX 75

Recipe 9 - Adaptive Security with Juniper SDSN 92

Recipe 10 - Securely Extending the Data Center 111

A Note About Professional Services 132

JUNIPER NETWORKS

Published by Juniper Networks Books
Authors: Chirag Patel, Ali Bidabadi, Charlie Chang-Hyun Kim, Lionel Ruggeri, Peter Cucinell, Scott Sneddon, Tony Boerema, Adam Grochowski
Technical Reviewers: Adam Grochowski, Dustin Splan
Project Manager: Julianne Ursch
Editor in Chief: Patrick Ames
Copyeditor and Proofer: Nancy Koerbel

ISBN: 978-1-941441-69-5 (print)
Printed in the USA by Vervante Corporation.
ISBN: 978-1-941441-70-1 (ebook)
Version History: v1, June, 2018
2 3 4 5 6 7 8 9 10

Feedback? Comments? Error reports? Email them to dayone@juniper.net.

Available in a variety of formats:
http://www.juniper.net/dayone

Contributors

Chirag Patel currently works as a Sr. Global Architect focusing on SDN, Cloud, and Security Solutions and has been with Juniper Networks for over 8 years. He holds a number of Juniper Certifications and a number of vendor-neutral certifications such as CISSP, LPI, Open Networking Foundation SDN, and OpenStack.

Ali Bidabadi has more than a decade of experience designing and building SDN / NFV products in public and private clouds. Ali is currently a Cloud Security Architect at Juniper Networks, where he designs and develops security solutions in the public cloud Marketplaces to address customer pain points. Prior to joining Juniper, Ali worked at Nephelo, a Cisco internal startup, where he helped design and develop an NFV-based orchestration platform for Cisco's Mobility Core Business (MCBU). He was a senior engineer at Intel Security Controller team (ISC) before joining Cisco, where he worked on the L4-7 integration of Intel Security IPS with VMware NSX, Cisco ACI, and OpenStack/ODL platforms.

Charlie Chang-Hyun Kim has 30 years of networking and automation experience. He is currently a Professional Services Consultant at Juniper Networks, where he works with cloud solutions worldwide.

Lionel Ruggeri builds public and private networks for Financial Enterprise, Government, and Service Providers as a Professional Services Solution Architect at Juniper Networks with more than 20 years of experience. He has consulted with many of Juniper's major customers and has specialized in data center / cloud and WAN transformation with a scope ranging from technology decision making, architecture, and design, to execution of network deployment and migration within complex and sensitive environments. Prior to Juniper, Lionel worked as technical lead on MPLS and Internet architecture and operations for several Service Providers. He graduated from a French engineering school specializing in Telecom & Networking.

Peter Cucinell is a Cloud Business Development Lead working cross organizationally to facilitate Juniper Networks software subscriptions sales and its growing presence in Public Cloud Marketplaces. Peter has many years of experience in Networking and Security in both technical and sales aspects of the business. His experience includes working with Web Services companies and large Service Providers.

Scott Sneddon is Senior Director and Chief Evangelist, Cloud and SDN at Juniper Networks where he spends most of his time evangelizing the benefits of Cloud adoption and Software Defined Networking. His background is in architecting large scale Cloud environments and data center networks. He was talking about SDN before SDN was cool.

(continued next page)

Tony Boerema is a professional Multicloud Services Architect who partners with executives to understand the subtleties of developing, implementing, and sustaining large complex systems. He has spent decades as a passionate customer advocate supporting cutting edge technologies with a track record of developing and leveraging highly-skilled technical teams. Tony has worked for a wide spectrum of small enterprise to top tier service providers as is currently a Practice Architect for Cloud Technologies at Juniper Networks.

Adam Grochowski has 20 years of experience working with internet and various communications providers. He is currently a Cloud Architect at Juniper Networks, where he works with organizations across the US to help them bring their SDN, NFV and virtualized datacenter plans to fruition. Prior to working at Juniper, Adam was the lead network security architect at a major metropolitan area hospital and held key engineering positions in an East Coast ISP/CLEC. He most recently worked at a large managed services provider as an architect, building managed and public multi-tenant cloud offerings.

This Cookbook's Landing Page

Depending on when you are reading this cookbook, there may be updated versions or errata that has been corrected. Check this cookbook's landing page for updates, posts by other readers, and new editions. Also, you can leave posts for the authors and editors here, on what's missing, what you would like to see, or comments and critiques:

https://forums.juniper.net/t5/Day-One-Books/Day-One-Amazon-Web-Services-with-vSRX-Cookbook/ba-p/326915.

You will need to sign into the J-Net Forums community. This cookbook's PDF is available for free download, along with all the other books in the *Day One* library.

List of AWS and Juniper Resources

Many of the best AWS resources are from Amazon's own documentation, solutions, and tutorials. The Juniper TechLibrary has a key document from the vSRX perspective. This book assumes you have basic understanding of AWS.

Resource Type	Description	Location
What is Amazon Web Services (AWS)?	Amazon Web Services (AWS) is a subsidiary of Amazon.com that provides a cloud services platform, offering compute power, database storage, content delivery and other functionality, to help businesses scale and grow.	https://aws.amazon.com
What is an EC2 instance?	Broadly, an EC2 instance is a virtualized server, which can be spun up or down dynamically, either via a console, command line, or programmatically via API. Once created, they can be personalized for various workloads. EC2 instances come in various types, which vary in number of virtual CPUs, Memory, and type of storage.	https://aws.amazon.com/ec2/
What is a VPC?	A virtual private cloud is an isolated virtual network construct within AWS, much like a VLAN in traditional networking. EC2 instances can be launched into a VPC, which provides a level of logical separation between those, and instances running in other VPCs.	https://aws.amazon.com/vpc/
Getting started with Junos and AWS.	Juniper's documents from the TechLibrary at Juniper.	https://www.juniper.net/documentation/en_US/vsrx/information-products/pathway-pages/security-vsrx-aws-guide-pwp.html
The AWS Marketplace	The AWS Marketplace is a digital catalog with thousands of software listings from independent software vendors that make it easy to find, test, buy and deploy software that runs on AWS.	https://aws.amazon.com/marketplace/
What Juniper Networks solutions are available on the AWS Marketplace?	Many of the solutions described in this guide are available on the AWS Marketplace.	https://aws.amazon.com/marketplace/seller-profile?id=9f7a8f2b-c239-4c66-b6a8-7f79e772f23c

Preface: Why Juniper on AWS?

If your enterprises are moving to the cloud, you face choices: you can opt for the agility of a public cloud, the security of a private cloud, or the elasticity of a hybrid cloud. While private clouds provide a solid foundation, public clouds can help you respond to the unexpected demands of new markets, and you get the flexibility and on-demand scalability to satisfy your evolving needs.

But not all workloads are created equal. Some are well suited to private clouds, and others work better in public clouds. One factor remains a constant regardless of where the workloads are located: your enterprise needs a high degree of security throughout the network.

Amazon Web Services (AWS) is a subsidiary of Amazon.com that provides a cloud services platform, offering compute power, database, storage, content delivery, and other functionality to help businesses scale and grow. AWS began offering their cloud services in 2006, and in the years since they have grown to become one of the largest cloud computing platforms. Their services are used by over a million customers, and are delivered from 18 different geographic regions around the world. When it comes to market share, AWS is certainly one of the more innovative cloud offerings; it has changed and influenced how Enterprise IT departments deliver services to their end users in immeasurable ways.

AWS provides a service called AWS Marketplace, where Amazon partners can offer products and services that augment and enhance AWS's own offerings. Juniper Networks offers solutions through the AWS Marketplace that allow users to connect their private clouds into and throughout the AWS cloud, and secure AWS cloud-based workloads using common tools and platforms.

Juniper Networks believes the future of cloud is multicloud. The well-positioned enterprise of the future will leverage private clouds where appropriate, and they will also leverage the AWS public cloud to take advantage of the global footprint, elasticity, and powerful ecosystem available there.

But deploying in both a private cloud and a public cloud, such as AWS, presents many challenges. The network architecture and security policy strategy that is adopted in a private data center is often difficult to represent in a public cloud, where much of the infrastructure is invisible to the cloud user. And while public cloud providers offer a wide range of network connectivity and security solutions, they often require that an enterprise rethink its strategy and retool its operational practices. That's a tall order.

By leveraging the Juniper Networks offerings available in the AWS Marketplace, Enterprise architects can provide network connectivity and security policy enforcement that exactly mirrors what they do in their private clouds. They can

extend their existing network and security operational practices into the AWS environment without compromising performance or compliance. And as their operational practices evolve to a fully automated NetSecDevOps model, they can leverage the rich Junos automation framework as well as the tight integration that Juniper Networks has built with the native AWS automation tools.

Additionally, Enterprise teams can leverage the advanced analytic and telemetry tools offered by Juniper Networks to provide visibility, optimization and management of applications, workloads, and infrastructure across their private cloud environment as well as their AWS cloud presence. Let's examine.

The Juniper solutions provided through the AWS Marketplace are:

- vSRX /vMX: Juniper Networks offers secure connectivity and carrier-grade routing that complements the flexibility, scalability, and agility of AWS. Customers can easily extend their global, on-premises environments to the cloud and manage their network infrastructure with granular visibility and control. Juniper Networks vSRX and vMX provide an easy-to-manage security and routing solution for hybrid cloud environments.
- Security Gateway/Connectivity: AWS operates on a "Shared Responsibility Model" with regard to public cloud assets. AWS is responsible for the security of any infrastructure that runs their cloud services. This is referred to as "Security of the Cloud." The consumer of those services, on the other hand, is responsible for "Security in the Cloud," or the management and security of the workloads created using those cloud services, so patching, firewalling, and etc. is the cloud consumer's job. Being able to deploy an enterprise grade firewall in the cloud to secure your workload and enable secure connectivity to the cloud is a powerful tool.
- Transit VPC: The basic building block of a data center in an AWS environment is a virtual private cloud (VPC), which essentially acts as a virtual data center in the cloud. Most AWS deployments evolve from a single VPC to multiple VPCs spread across numerous regions as the enterprise expands. The desire for data centers to be closer to end users, segmenting resources and tasks for which they are responsible, is common across most enterprises. However, with multi-VPC deployments, enabling connectivity between VPCs requires explicit peering using VPC peering modules, which are restricted to peering VPCs within a single AWS region and do not possess the ability to granularly filter or control traffic flowing between VPCs. Adding VPC peering modules to enable connectivity between VPCs can often get complicated, adding considerable management overhead. Most AWS deployments also need next-generation firewall (NGFW) services in their VPCs to protect the applications that they are hosting.

The Juniper Networks Transit VPC solutions solve the native AWS limitation for multi-VPC connectivity while providing advanced security. These solutions include:

- Transit VPC: Designed for larger deployments with AWS management roles distributed between many teams.
- Full-Mesh VPN: Designed for smaller deployments managed by a centralized DevOps team requiring fewer hops, lower latency, and easier troubleshooting.
- Multicloud connectivity: The Transit VPC solution can also be extended between multiple public cloud providers, providing a consistent network connectivity and security policy paradigm for any workload on any cloud.
- AppFormix: AppFormix is a new breed of optimization and management software platform for public, private, and hybrid clouds. This intent-driven software manages automated operations, visibility, and reporting in cloud use cases for Kubernetes and OpenStack, as well as AWS. It features machine learning-based policy and smart monitors, application and software-defined infrastructure analytics, alarms, and chargeback accounting.

The basic building block of a data center in an AWS environment is a virtual private cloud (VPC), which essentially acts as a virtual data center in the cloud. Most AWS deployments evolve from a single VPC to multiple VPCs spread across numerous regions as the enterprise expands. The desire for data centers to be closer to end users, segmenting resources and tasks for which they are responsible, is common across most enterprises.

Please note that AppFormix, as well as many more recipes about Juniper AWS solutions, the vMX, and all the late-breaking technologies and features coming from Juniper Networks, will be covered in the Second Edition of this cookbook. Check for new editions of this cookbook at: https://www.juniper.net/dayone.

Scott Sneddon, Juniper Networks, Senior Architect

Recipe 1: Provisioning Junos on AWS

by Peter Cucinell

Enterprise organizations are increasingly moving their workloads to AWS to take advantage of its ease of deployment, reasonable costs, feature sets, and the scaling benefits that come with the AWS platform.

Problem

Public clouds have issues such as migration planning, new connectivity, routing, and security requirements. How do you get the advantages of public clouds while avoiding their pitfalls?

Solution

Use Junos OS-based platforms like the vSRX and the vMX because they can provide scalable, secure protection across private, public, *and* hybrid clouds.

The Junos Amazon Machine Images (AMIs) available on AWS Marketplace offer the same features as Juniper's physical SRX Series or MX Series platforms, but in a virtualized form factor. Using vSRX and vMX for delivering services ensures that you get scale to match network demand while avoiding the traditional pitfalls of routing and security in public clouds.

Both the vSRX and vMX AMIs are available for purchase as BYOL (bring your own license) and PAYG (pay as you go) licensing models.

NOTE The vSRX PAYG images do not require any Juniper Networks licenses.

These Junos platforms can:

- Be deployed in a VPC in the AWS cloud.
- Launch vSRX or vMX as an Amazon Elastic Compute Cloud (Amazon EC2) instance in an Amazon VPC.
- Both vSRX and vMX use hardware virtual machine (HVM) virtualization. Figure 1.1 shows the vSRX VM layers.

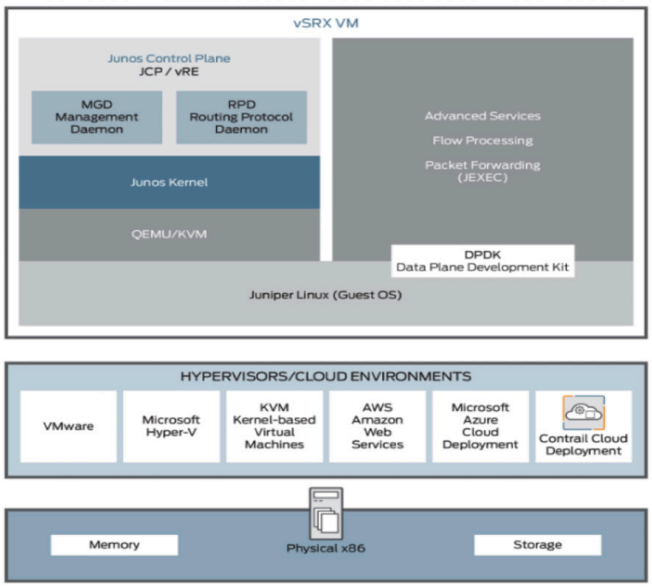

Figure 1.1 *The vSRXm VM*

In an Amazon Virtual Private Cloud (Amazon VPC), public subnets have access to the Internet gateway, but private subnets do not. Junos-based platforms require two public subnets and one or more private subnets for each individual instance group. The public subnets consist of one for the management interface (fxp0) and one for a traffic forwarding (data) interface as shown in Figure 1.2. Private subnets connected to the Junos platform ensure that all east-west traffic between applications on the private subnets and the Internet must pass through the Junos-based instance.

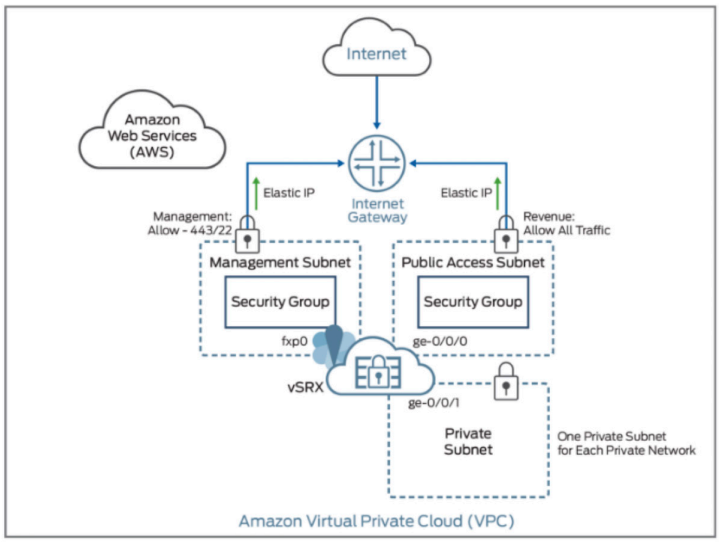

Figure 1.2 *Amazon VPC and Junos*

Once the Junos-based instance is up and running, use the elastic IP assigned to the management interface to SSH, using the key pairs into the vSRX instance:

```
ssh –i <path_to_key–pair/Your_Key_Pair.pem> root@<Elastic_IP_of_fxp0>
```

After logging into vSRX, or vMX, additional users can be configured, as well as different methods of accessing the vSRX, such as HTTPS, HTTP, etc. Other automated configuration options for Junos AMIs are also available, including CloudFormation templates for the vSRX transit VPC solution available on AWS Marketplace.

NOTE A specific CloudFormation template has been designed to use "vSRX Next-Generation Virtual Firewall" as the base AMI to create vSRX instances inside a Transit VPC. You'll find this AMI in the AWS Marketplace. More information on using CloudFormation templates in Junos can be found at: https://www.juniper.net/assets/us/en/local/pdf/implementation-guides/8010096-en.pdf.

Recipe 2: Launching a Juniper Instance

by Charlie Chang-Hyun Kim

Junos OS Used: 17.1R2

Juniper Platforms General Applicability: vMX, vEX, vSRX, AppFormix

Regardless how big or small of an AWS solution you might plan to build, the first step is knowing how to launch a virtual machine (VM), called Amazon *Elastic Compute Cloud* (Amazon EC2) in AWS lingo. This recipe starts a Juniper EC2 instance using the AWS web console.

Problem

The options available to bring up one EC2 instance are overwhelming to beginners.

NOTE For generic VMs, please refer to these AWS tutorials: https://aws.amazon.com/getting-started/tutorials/launch-a-virtual-machine/.

Solution

Launching Juniper instances is easy if you follow the steps in this recipe, and once you've learned them, bringing up an EC2 instance will seem like second nature.

The first prerequisite is that you need an AWS account, which is not a free trial. Juniper images are too big to start up with a free-trial account. (More about AWS accounts can be found in the front matter of this cookbook and on AWS itself.)

You can use your root account if you are the owner of the account, although this cookbook recommends choosing an IAM user: https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html. In this recipe, *testuser* is the IAM user under the account alias *test-aws-account*.

Open a browser and visit https://test-aws-account.signin.aws.amazon.com/console.

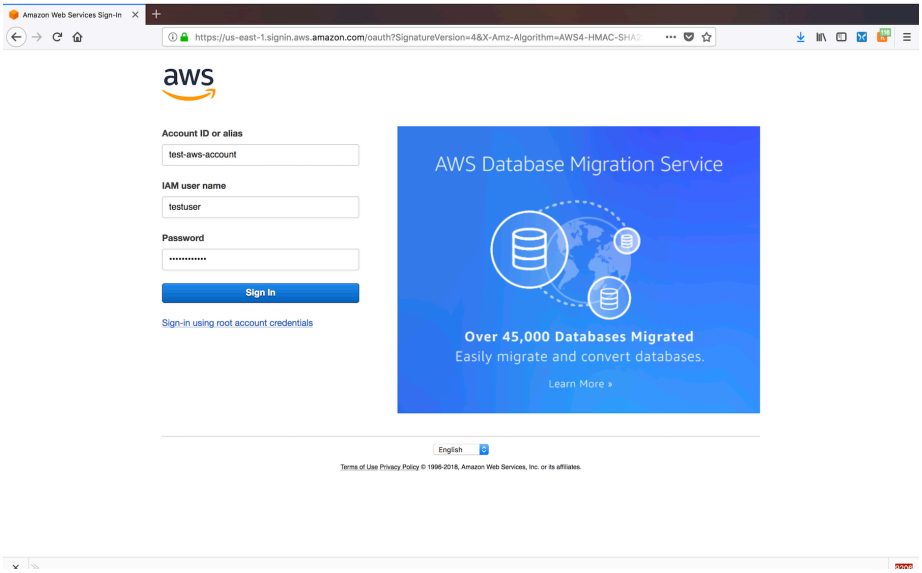

Figure 2.1 *Login Screen*

Once connected to the AWS console, create a VPC, by selecting Service > VPC as shown in Figure 2.2.

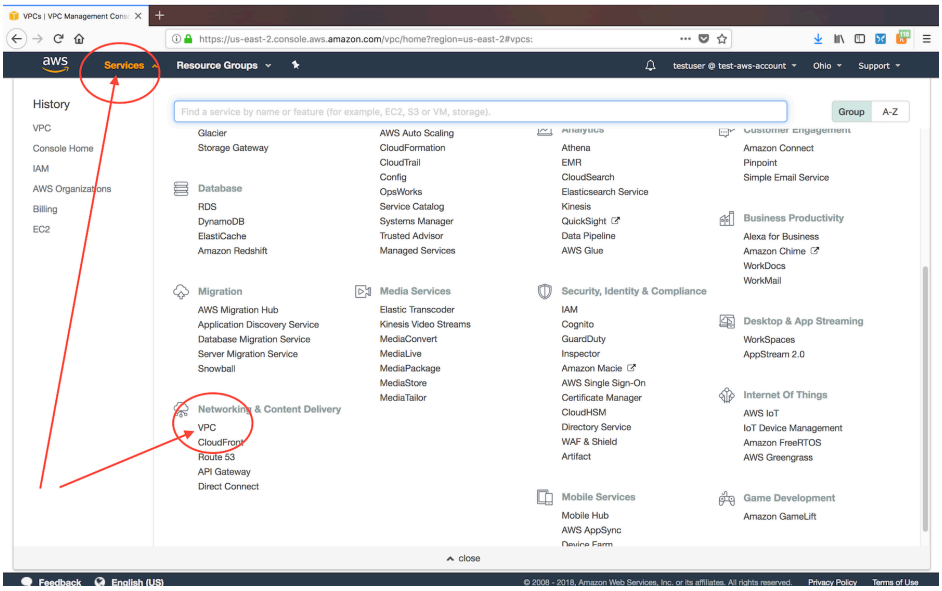

Figure 2.2 *Choose Service VPC*

Next create a VPC.

In Figure 2.3, verify the region, click over your VPCs, click Create VPC, give the VPC a name and CIDR, and then click the Yes, Create button. In this example, the VPC name is *myvpc*, and the CIDR is *10.0.0.0/16*.

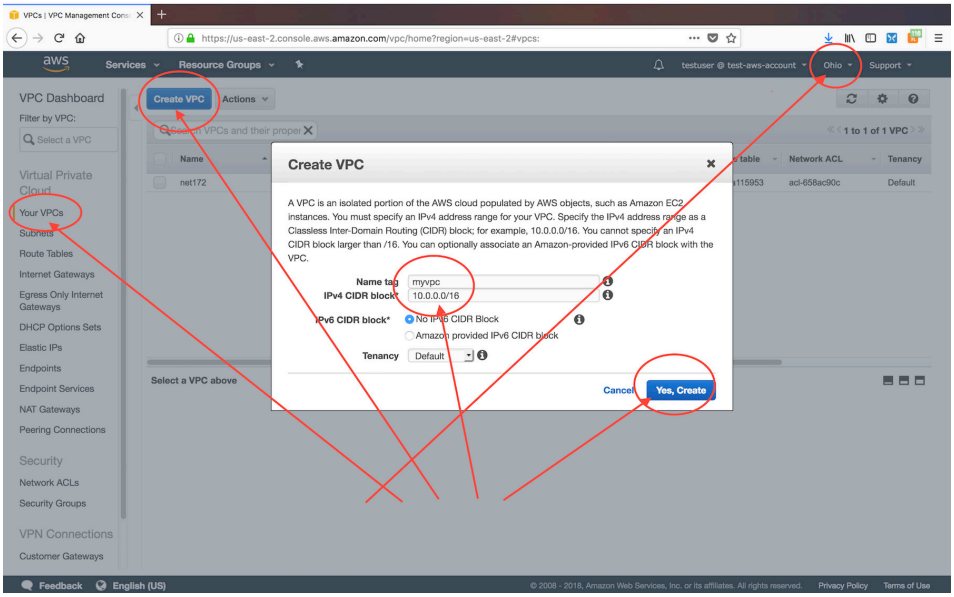

Figure 2.3 *Create a VPC*

Next create a subnet for the VPC.

Click Subnets in the left-hand VPC Dashboard, then click Create Subnet as shown in Figure 2.4 and give the subnet a name and associate to it the VPC just created. Supply the CIDR of the subnet, which is a subset of the VPC, and click Yes, Create.

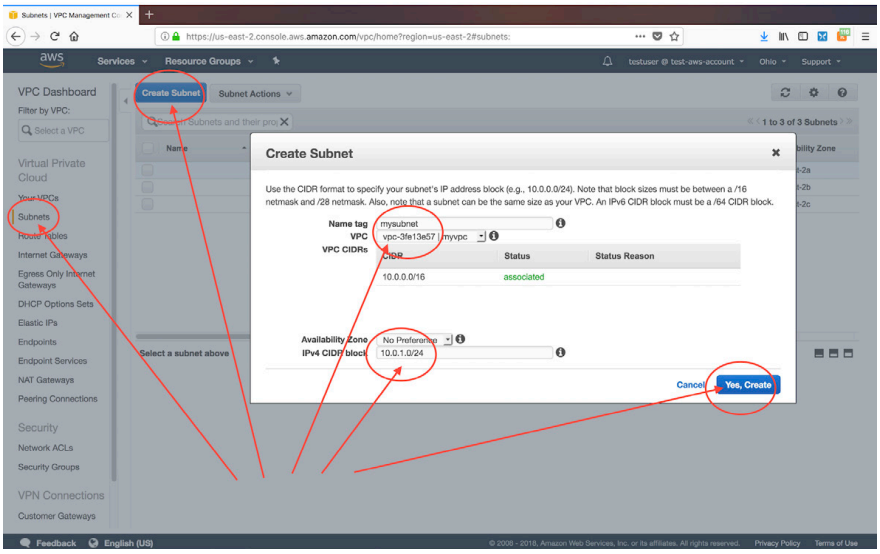

Figure 2.4 *Create a Subnet for the VPC*

Now you're going to create an Internet Gateway.

As shown in Figure 2.5, click Internet Gateways in the Dashboard, click the Create Internet Gateway, give the gateway a name tag, and click Yes, Create.

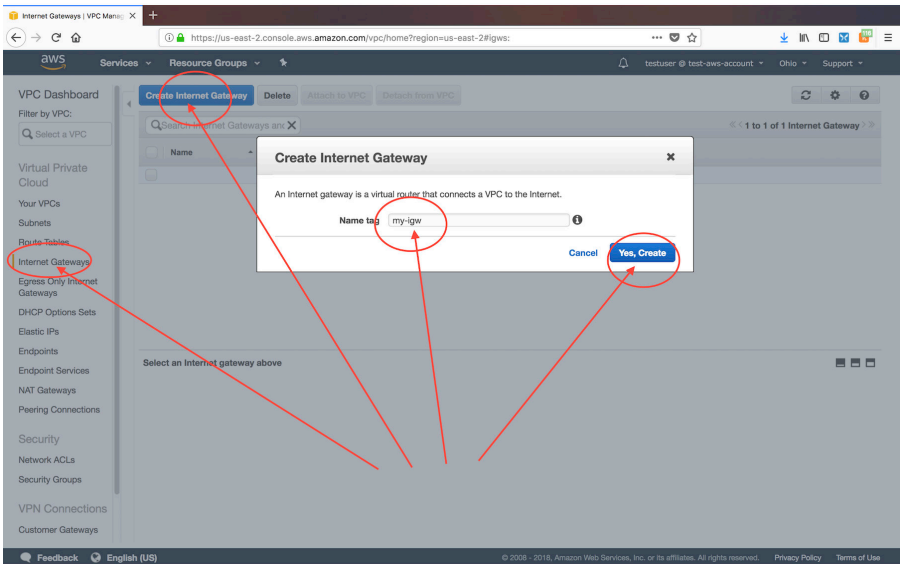

Figure 2.5 *Create an Internet Gateway*

Now let's attach the Internet gateway to your VPC. As shown in Figure 2.6, select the internet gateway just created, click Create Internet Gateway, select the VPC created above, and click Yes, Attach.

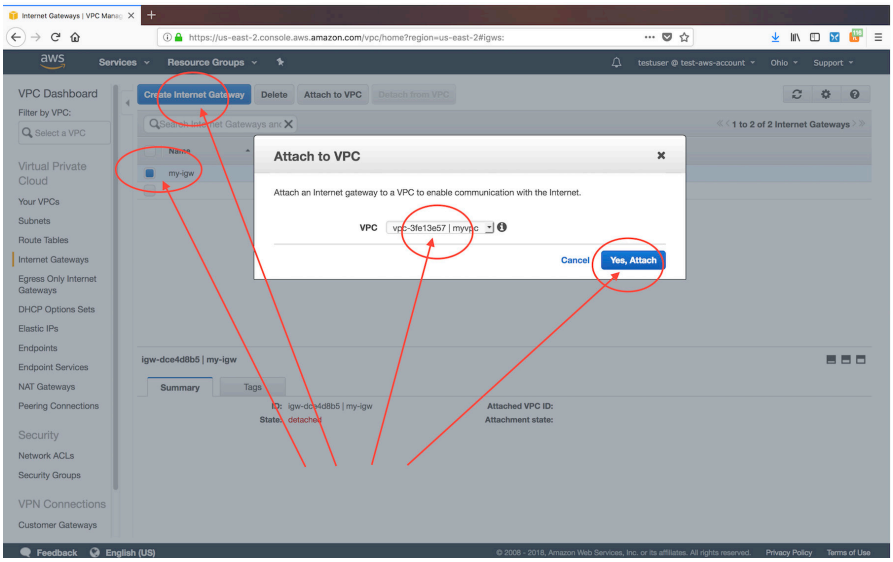

Figure 2.6 *Attach the IGW to Your VPC*

Update the route table with the internet gateway. As shown in Figure 2.7, click Route Tables in the column Dashboard, select the route table associated to your new VPC, and click the Routes tab. Click Edit.

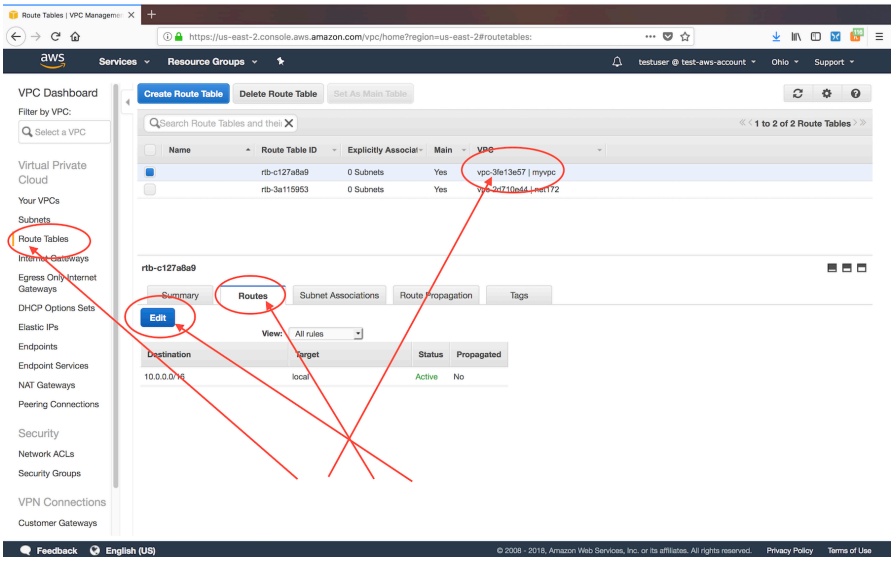

Figure 2.7 *Updating the Route Tables*

Click Add another route, add the default route, and choose the internet gateway as the Target, and click Save as shown in Figure 2.8.

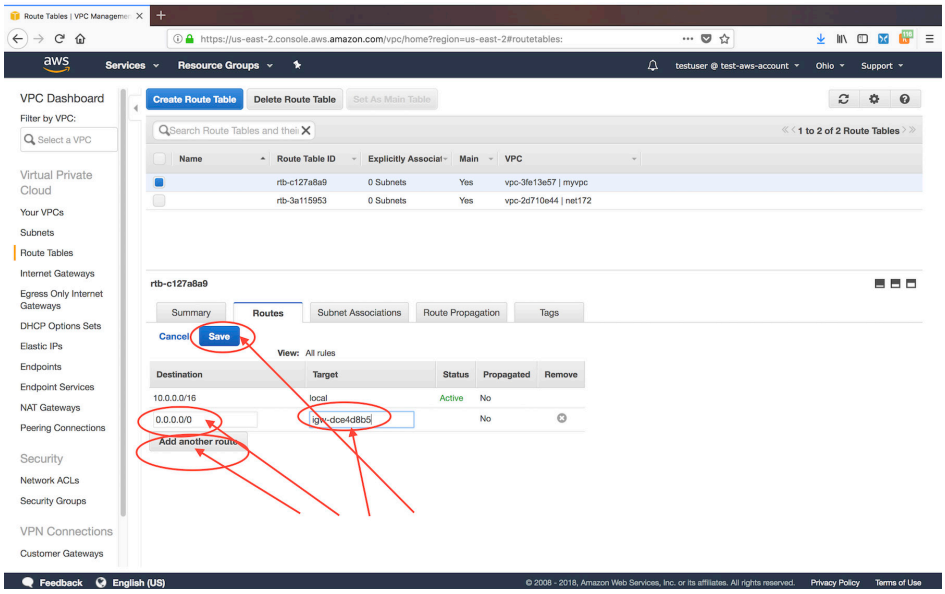

Figure 2.8 *Updating the Route Table with Default Route*

Now to create an EC2 instance, as shown in Figure 2.9, click the Services tab, and click EC2 within the Compute group.

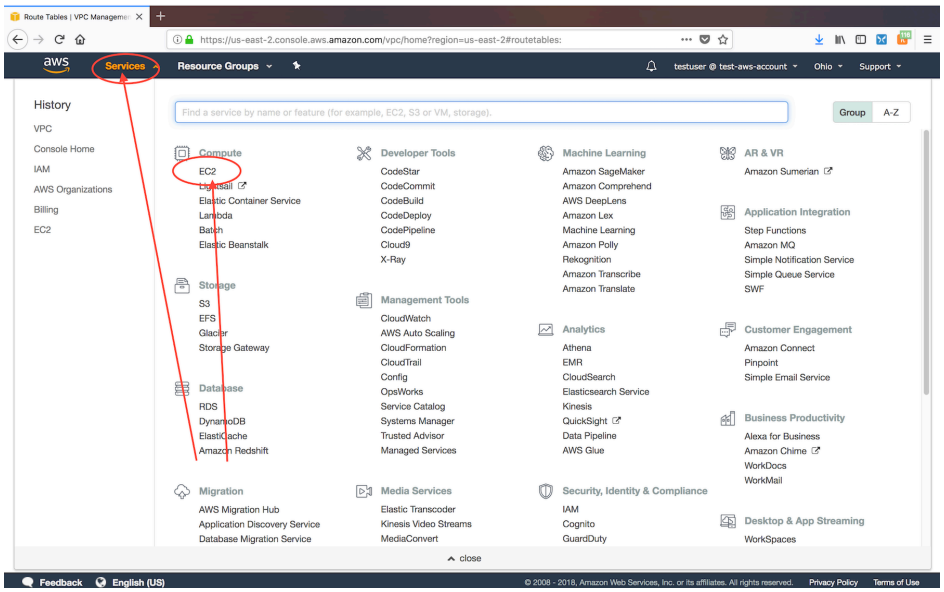

Figure 2.9 *EC2 Creation*

The Launch Instances dialog window opens. Click Instances, and click the Launch Instance button as shown in Figure 2.10.

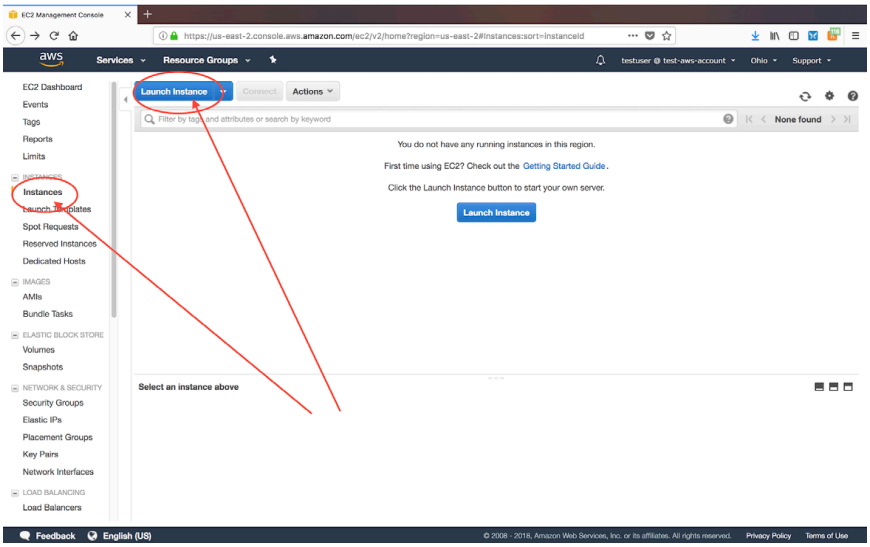

Figure 2.10 *Launch Instance*

Now let's choose an AMI for the newly launched instance. Click AWS Marketplace to select an AMI from the AWS Marketplace, give the AMI a name, in Figure 2.11 it's *vsrx*, and click on the Select button to continue the setup.

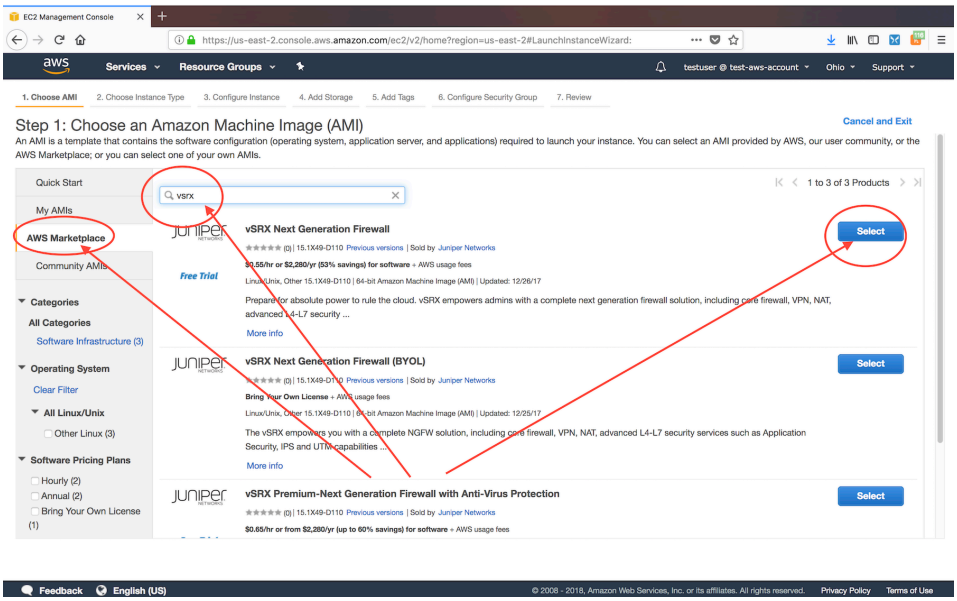

Figure 2.11 *AMI Step 1*

Now it's time to choose an Instance Type in the AMI setup wizard. Note, there may be an informational popup to guide the price. If so, click Continue.

Choose the first available type, shown here in Figure 2.12, and click Next Configure Instance Details.

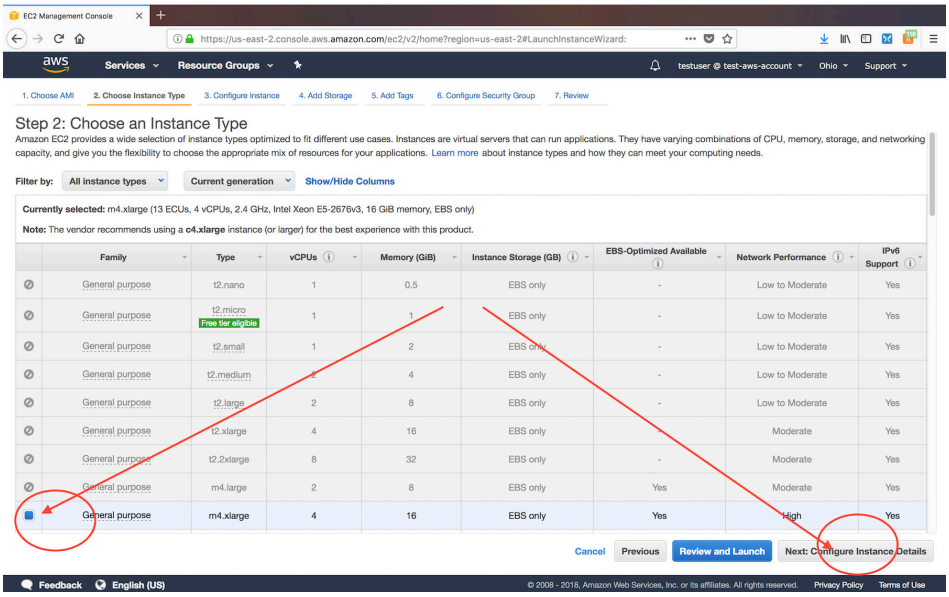

Figure 2.12 *AMI Step 2*

Now let's configure the instance details in Step 3 of the AMI wizard shown in Figure 2.13. Select the VPC you created before, enable public IP, and click Next: Add Storage.

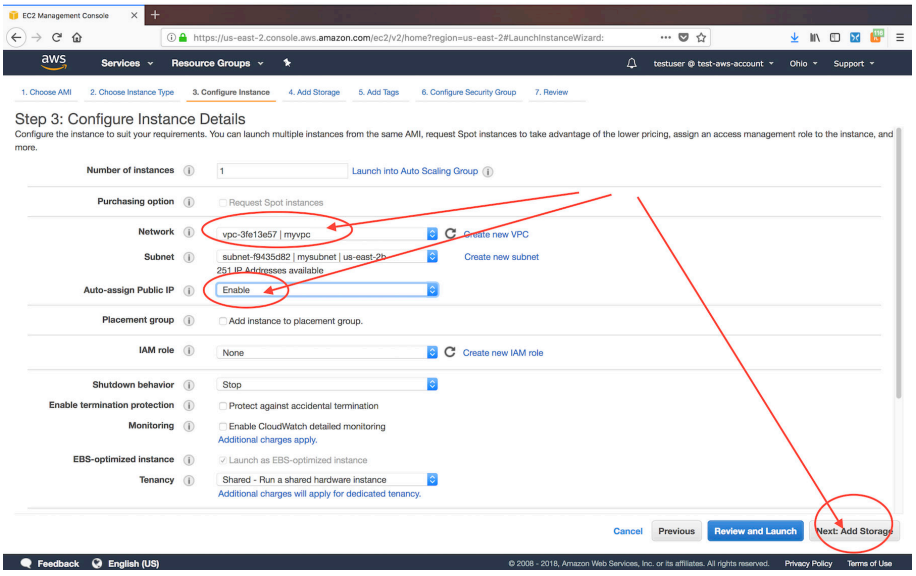

Figure 2.13 *AMI Step 3*

Step 4 of the AMI wizard is to Add Storage as shown in Figure 2.14. Take the default size and click Next: Add Tags.

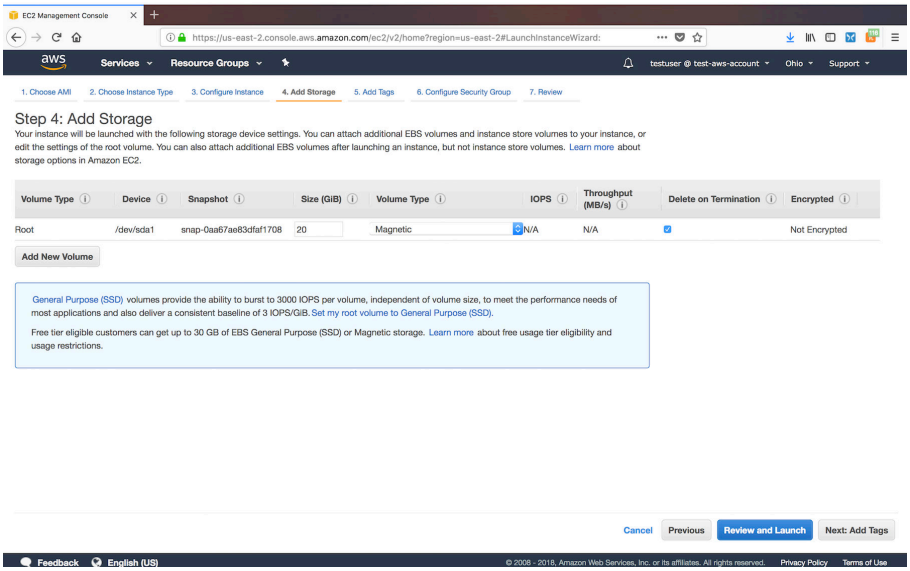

Figure 2.14 *AMI Step 4: Add Storage*

To add tags, click on the Add another tag button in Figure 2.15, add *Name* in the Key field, and provide the name of the VM in the Value field. Click the Next Configure Security Group button.

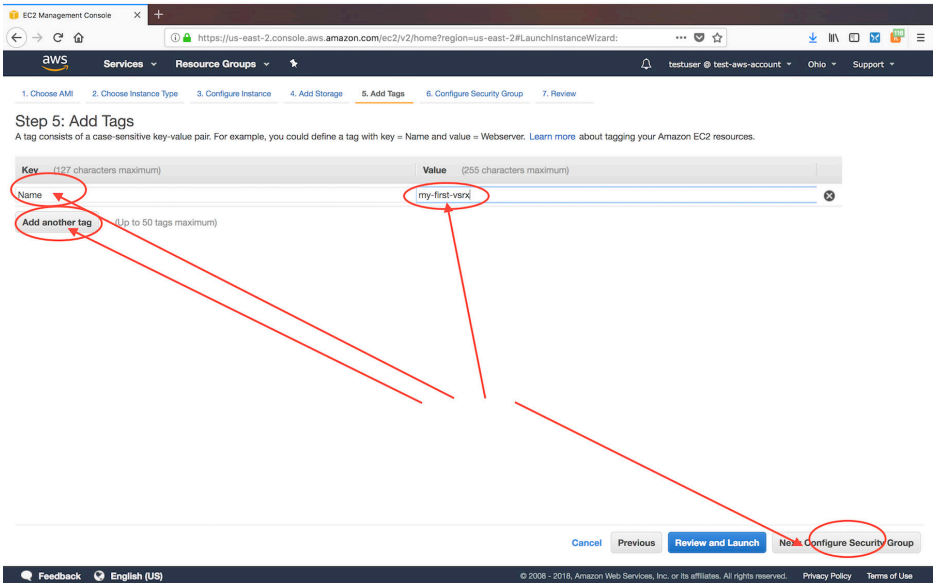

Figure 2.15 *AMI Step 5: Add Tags*

To configure the Security Group, as shown in Figure 2.16, set the Source to the address or the CIDR for you, and click Review and Launch.

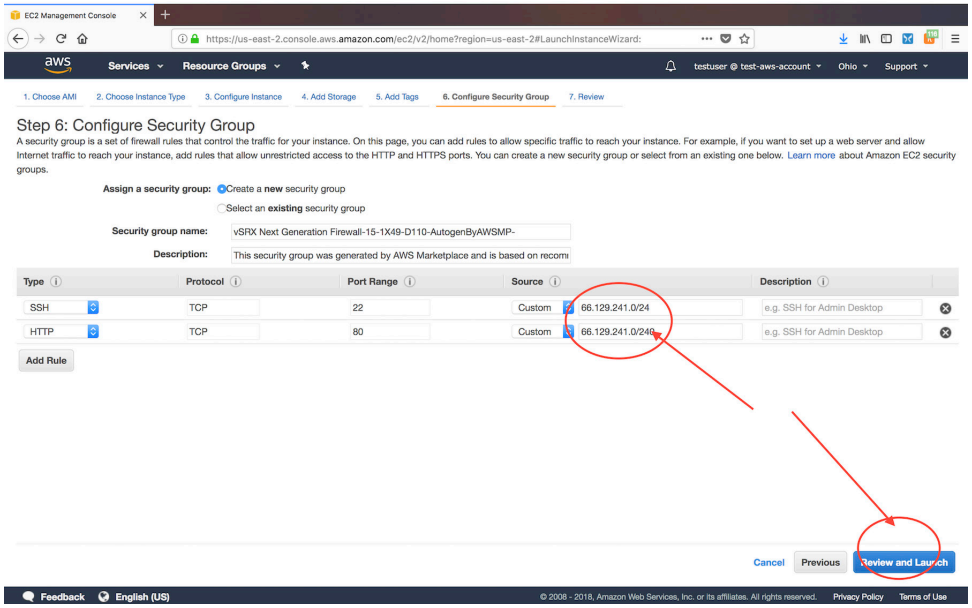

Figure 2.16 *AMI Step 6: Configure Security Group*

If you see a Boot from General Purpose (SSD) popup, take the default value and click Next.

On the Review Instance Launch screen in Figure 2.17, click the Launch button.

A Key Pair dialog box will appear. Choose to create a new key pair, give it a name, click on Download Key Pair and then keep that key pair in a safe place for you to use, and click the Launch Instances button.

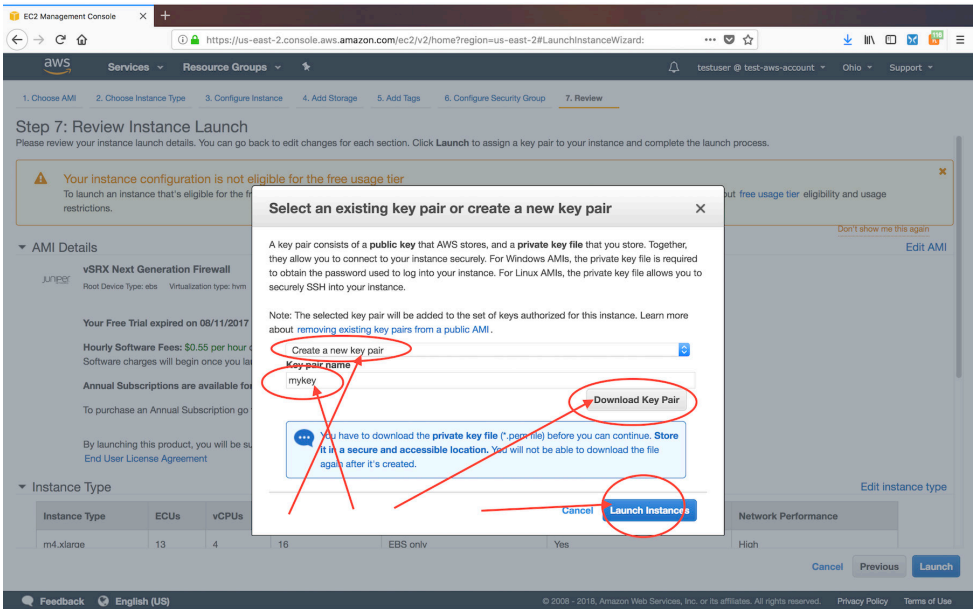

Figure 2.17 *Key Pair*

The Launch Status window opens. Click the View Instances button in the lower right-hand corner in Figure 2.18.

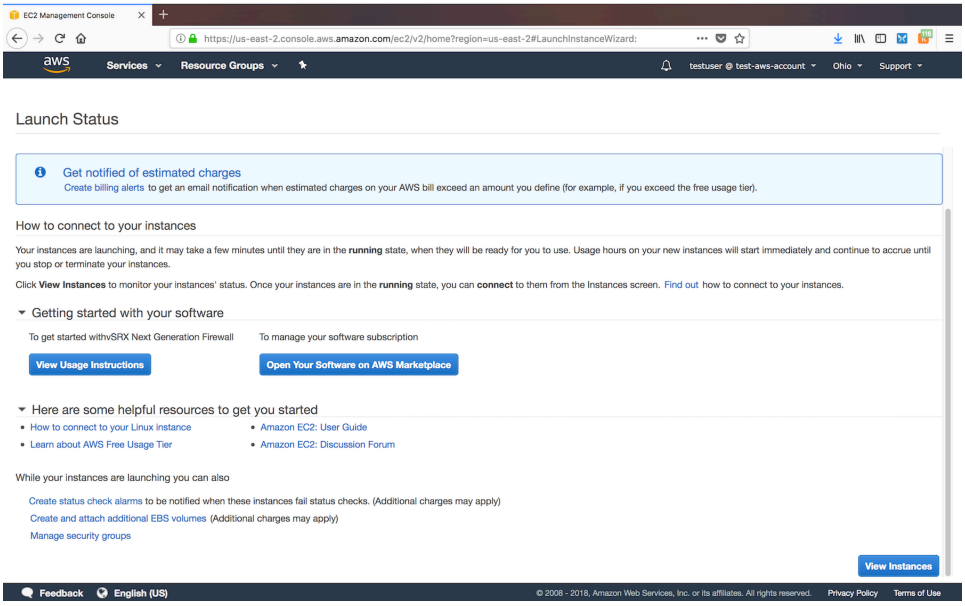

Figure 2.18 *Launch Status*

To use the new instance, you may need to wait for the instance to come up. From the left-hand column hierarchy, choose Instances, as shown in Figure 2.19. From here you can monitor the Instance State, Status Checks, and Alarm Status. When these three have a status of: *running*, *2/2 checks*, and *None*, then you may access the instance using SSH.

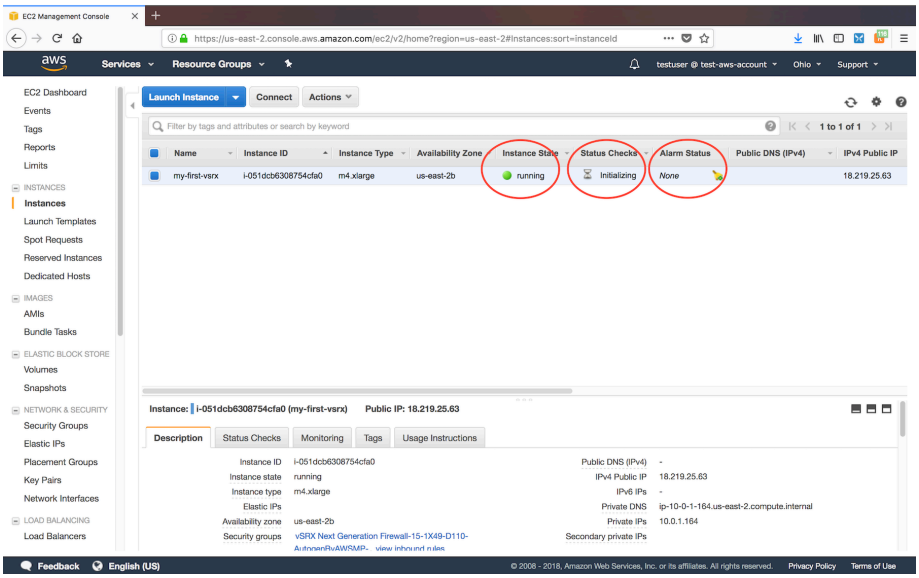

Figure 2.19 *Launch Status Indicators*

To connect via SSH, click the Connect button in the Instances window shown near the top of Figure 2.19, and you will see the SSH instruction. Set the key file to be readable only to the owner by the command `chmod 400 mykey.pem`, and initiate the SSH connection using the command `ssh -i "mykey.pem" root@18.219.25.63`.

To validate the Junos device:

```
ckim-mbp:~ ckim$ cd Downloads/
ckim-mbp:Downloads ckim$ chmod 400 mykey.pem
ckim-mbp:Downloads ckim$ ssh -i "mykey.pem" root@18.219.25.63
The authenticity of host '18.219.25.63 (18.219.25.63)' can't be established.
ECDSA key fingerprint is SHA256:NaJT0Lo3Zk985tkZUK1epX5p4MR+nIXCOgSKfalGycs.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '18.219.25.63' (ECDSA) to the list of known hosts.
--- JUNOS 15.1X49-D110.4 built 2017-09-08 03:40:37 UTC
root@% cli
root> show chassis hardware
Hardware inventory:
Item           Version  Part number  Serial number     Description
Chassis                                7F8BDEFD9FEC     VSRX
CB 0
Routing Engine 0         BUILTIN      BUILTIN          VSRX-S
FPC 0          REV 07    611-049549   RL3714040884     FPC
  PIC 0                  BUILTIN      BUILTIN          VSRX DPDK GE

root>
```

Once you have had a taste of your first instance, terminate the instance to avoid unintentional cost as shown in Figure 2.20. With the target instance selected, click on the Actions tab, select Instance State, and then select Terminate, and then confirm the decision in the window.

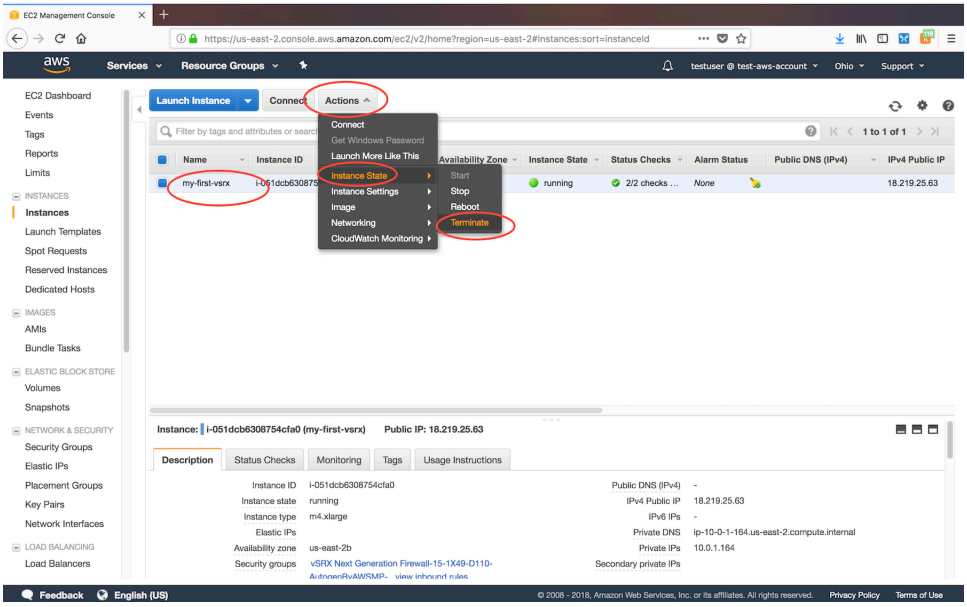

Figure 2.20 *Launch Status Indicators*

The Instance State will be changed to *Terminated*, stay as such for a while, then eventually disappear.

Discussion

This recipe is an extremely simplified version to avoid overwhelming beginners. Each step has options and choices that should be considered based on the purpose of your overall AWS solution.

In the following recipes, you will see many of these options including automation methodology. Enjoy.

Recipe 3: Launching Juniper Instances with Ansible Playbook

by Charlie Chang-Hyun Kim

- Ansible Used: 2.5.0
- Ubuntu Version Used: 16.4
- Junos OS Used: 17.4R1
- Juniper Platforms General Applicability: vMX, vSRX, AppFormix

In this recipe, you will learn how to use Ansible, one of the best orchestration tools, to bring up two vSRX EC2s. Ansible and other automation or orchestration tools use the access key ID and its secret access key instead of the AWS console username and password. You can use Ansible in various platforms including Linux, Mac, Windows, and others. In this recipe, you will see the instructions in Ubuntu 16.04 environment, which is one of the Linux distros.

Problem

In Recipe 2 you learned how to launch a Juniper EC2 using the AWS console. While AWS has made that relatively easy to do and understand, it would be error prone and time consuming if you needed to bring up tens or hundreds of EC2s and other resources, or if you needed to repeat the process often enough to make it repetitive.

Solution

Learn how to launch Juniper EC2 instances with Ansible and automate the process. This is one of great recipes in this cookbook because it uses Junos, Ansible, and AWS all at the same time. Let's start right away and assume you're at the AWS console with everything working and logged in.

First, let's get the access key ID and its access secret key. From the AWS console, click the Services tab, and click IAM as shown in Figure 3.1.

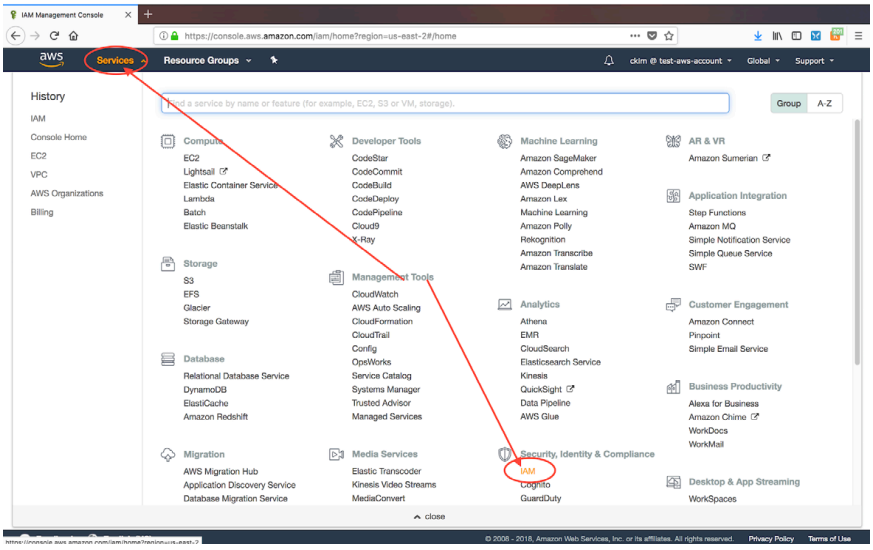

Figure 3.1 *Step One to Get Your Access Key ID*

Click Users in the left-hand sidebar, and click *testuser* as shown in Figure 3.2.

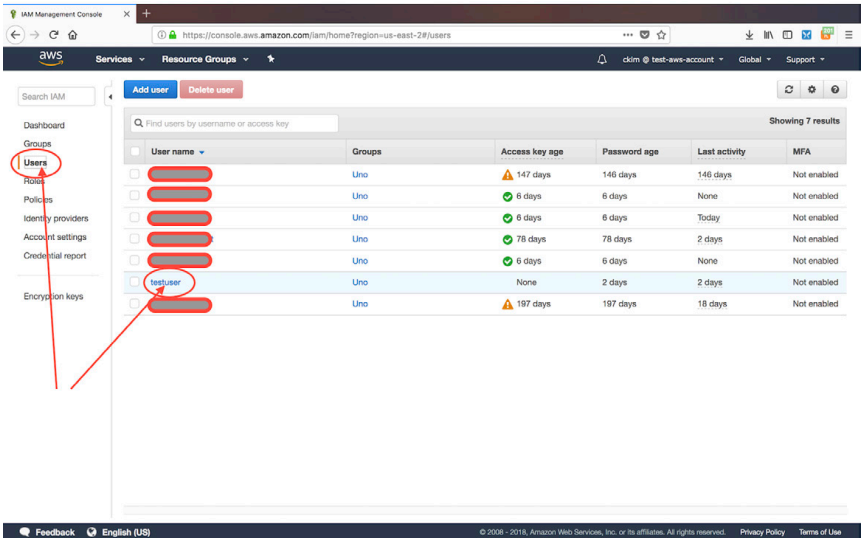

Figure 3.2 *Step Two to Get Your Access Key ID*

Now click on the Security credentials tab, and then on the Create access key button as shown in Figure 3.3.

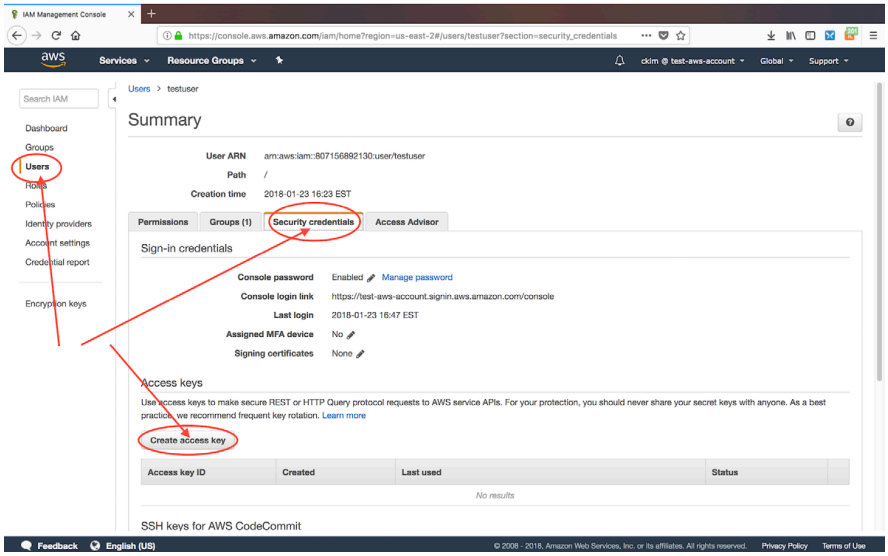

Figure 3.3 *Step Three to Get Your Access Key ID*

Now click on the Download .csv file button and save the file. Remember the filename as the file contains the same Access key ID and Secret access key. Click the Close button when done.

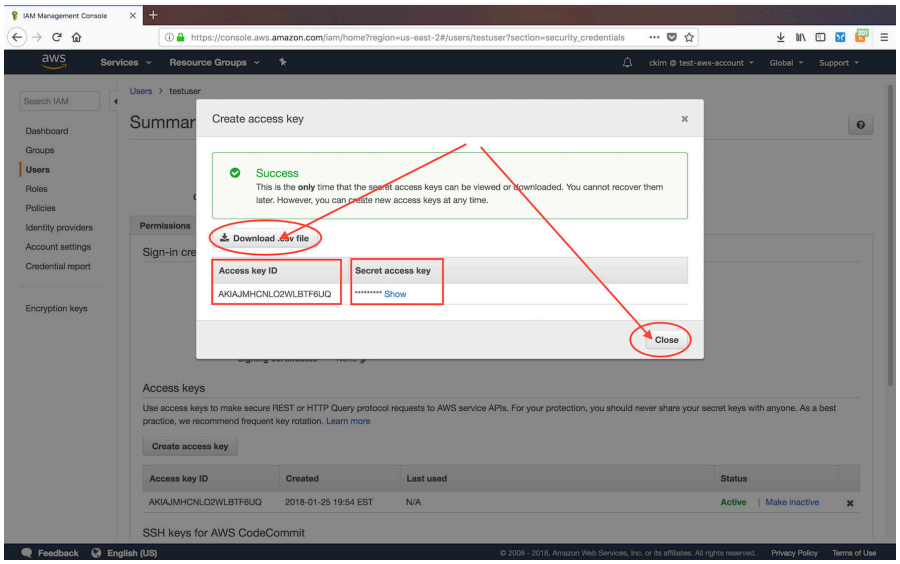

Figure 3.4 *Step Three to Get Your Access Key ID*

Okay, the access key ID and secret access key are ready, and the rest of this recipe will guide you to create and set up a client to run Ansible on Ubuntu 16.

For your convenience, a CloudFormation template has been prepared at https://raw.githubusercontent.com/kimcharli/dayone-aws-terraform/master/cf-for-ubuntu-ec2.yml. This template was modified from https://s3-us-west-2.amazonaws.com/CloudFormation-templates-us-west-2/EIP_With_Association.template just enough to load Ubuntu 16 AMI from region us-west-2. Please download *cf-for-ubuntu-ec2.yml* and save.

For added security, it is recommended to allow SSH only from your source address. One of the ways to get it is from http://icanhazip.com/. Save your source IP address.

By default, cloud instances prefer public keys instead of passwords. From the Services tab on the AWS console, follow the path EC2 > Key Pairs, and then click Create Key Pair. Give it a name, such as *day-one*, and download it. The file name is *day-one.pem*.

Now, back to the AWS console, select CloudFormation from Services, and then click Create Stack. From the Select Template dialogue box, click Browse, under the Choose a template section, and then select the file *cf-for-ubuntu-ec2.yml* file you saved earlier. Click Next, and give the Stack a name, '*day-one-stack*', and leave the Parameters as they are, but update the SSH Location with *your source IP address* with /32, as in *1.2.3.4/32*. Click Next, and set the 'Name' as the Key, and '*day-one-ubuntu*' as the Value. Click Next, review the data, and click Create.

Okay, while waiting for the Status Checks to become 2/2 for the EC2 day-one-ubuntu, adjust the attribute of the *day-one.pem* such that only you may read. The equivalent `*nix` command is '*chmod 600 day-one.pem*'. Once the EC2 Status Checks becomes 2/2, you can get connection information by clicking on the Connect button over the list, and logging in to the EC2:

```
ckim-mbp:Downloads ckim$ ssh -i "day-one.pem" ubuntu@ec2-35-162-18-126.us-west-2.compute.amazonaws.
com
The authenticity of host 'ec2-35-162-18-126.us-west-2.compute.amazonaws.com (35.162.18.126)' can't be
established.
ECDSA key fingerprint is SHA256:LNX+tIlSk2/xF0PMr5qXP4rJm4tHtzLlMmqIFLRVpE8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-35-162-18-126.us-west-2.compute.amazonaws.com,35.162.18.126' (ECDSA)
to the list of known hosts.
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-1052-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.
```

```
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-20-165:~$
```

Now install the packages and configure the AWS environment:

```
ubuntu@ip-172-31-20-165:~$ sudo apt update

ubuntu@ip-172-31-20-165:~$ sudo apt -y install awscli python-pip unzip

ubuntu@ip-172-31-20-165:~$ sudo pip install ansible==2.5.0 boto3 boto

ubuntu@ip-172-31-20-165:~$ sudo pip install --upgrade pip

ubuntu@ip-172-31-20-165:~$ ansible --version
ansible 2.5.0
  config file = None
  configured module search path = [u'/home/ubuntu/.ansible/plugins/modules', u'/usr/share/ansible/
plugins/modules']
  ansible python module location = /home/ubuntu/.local/lib/python2.7/site-packages/ansible
  executable location = /home/ubuntu/.local/bin/ansible
  python version = 2.7.12 (default, Dec  4 2017, 14:50:18) [GCC 5.4.0 20160609]
ubuntu@ip-172-31-20-165:~$

ubuntu@ip-172-31-20-165:~$ aws configure
AWS Access Key ID [None]: AAAAAAAAAAAAAAAAAAAA
AWS Secret Access Key [None]: SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
Default region name [None]: us-west-2
Default output format [None]: json
ubuntu@ip-172-31-20-165:~$

ubuntu@ip-172-31-20-165:~$ aws ec2 describe-instances --region us-west-2 --query 'Reservations[*].
Instances[*].Tags[?Key==`Name`].{Name:Value}'
[
    [
        [
            {
                "Name": "day-one-ubuntu"
            }
        ]
    ]
]
ubuntu@ip-172-31-20-165:~$
```

Now copy over the *day-one.pem* file for EC2 to be able to access the vSRX being launched:

```
ckim-mbp:Downloads ckim$ scp -i day-one.pem day-one.pem ubuntu@35.162.18.126:
day-one.pem
100% 1696    17.0KB/s   00:00
ckim-mbp:Downloads ckim$ ssh -i day-one.pem ubuntu@35.162.18.126 chmod 600 day-one.pem
ckim-mbp:Downloads ckim$
```

Download the prepared Ansible example, and run it:

```
ubuntu@ip-172-31-20-165:~$ wget https://github.com/kimcharli/dayone-aws-ansible/archive/master.zip
--2018-04-03 04:05:39--  https://github.com/kimcharli/dayone-aws-ansible/archive/master.zip
Resolving github.com (github.com)... 192.30.255.112, 192.30.255.113
Connecting to github.com (github.com)|192.30.255.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/kimcharli/dayone-aws-ansible/zip/master [following]
--2018-04-03 04:05:39--  https://codeload.github.com/kimcharli/dayone-aws-ansible/zip/master
Resolving codeload.github.com (codeload.github.com)... 192.30.255.121, 192.30.255.120
Connecting to codeload.github.com (codeload.github.com)|192.30.255.121|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: ‚Äòmaster.zip‚Äô

master.zip                              [ <=>                                                ]
7.55K  --.-KB/s    in 0s

2018-04-03 04:05:40 (95.8 MB/s) - ‚Äòmaster.zip‚Äô saved [7730]

ubuntu@ip-172-31-20-165:~$
ubuntu@ip-172-31-20-165:~$ unzip master.zip
Archive:  master.zip
7e293ffb7c045032a1acf4592e74950d1a7009a1
   creating: dayone-aws-ansible-master/
  inflating: dayone-aws-ansible-master/README.md
   creating: dayone-aws-ansible-master/dayone-ansible/
  inflating: dayone-aws-ansible-master/dayone-ansible/ansible.cfg
   creating: dayone-aws-ansible-master/dayone-ansible/roles/
   creating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/
   creating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/security_group/
   creating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/security_group/tasks/
  inflating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/security_group/tasks/main.yml
   creating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/subnet/
   creating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/subnet/tasks/
  inflating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/subnet/tasks/main.yml
   creating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/tasks/
  inflating: dayone-aws-ansible-master/dayone-ansible/roles/do_vpc/tasks/main.yml
  inflating: dayone-aws-ansible-master/dayone-ansible/site.yml
   creating: dayone-aws-ansible-master/inventory/
   creating: dayone-aws-ansible-master/inventory/group_vars/
  inflating: dayone-aws-ansible-master/inventory/group_vars/all.yml
 extracting: dayone-aws-ansible-master/inventory/hosts
ubuntu@ip-172-31-20-165:~$ cd dayone-aws-ansible-master/dayone-ansible/
ubuntu@ip-172-31-20-165:~/dayone-aws-ansible-master/dayone-ansible$ ansible-playbook -i ../inventory/
hosts site.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost
does not match 'all'
```

```
PLAY [localhost] ***************************************************************************************
*********************************************************

TASK [include_role] ************************************************************************************
*********************************************************

TASK [do_vpc : Find VPC] *******************************************************************************
*********************************************************
ok: [localhost]

TASK [do_vpc : Register vpc if present] ****************************************************************
*********************************************************
ok: [localhost]

…

TASK [do_vpc : Create Route Table in VPC tvpc1] ********************************************************
*********************************************************
changed: [localhost]

PLAY RECAP *********************************************************************************************
*********************************************************
localhost              : ok=36   changed=13   unreachable=0    failed=0

ubuntu@ip-172-31-20-165:~/dayone-aws-ansible-master/dayone-ansible$
```

This Ansible playbook usually completes without failure within a minute. However, the vSRX instance will take several minutes or so to complete its bootup. Once it passes the 2/2 checks, you may access and test its availability:

```
ckim-mbp:Downloads ckim$ ssh -i "day-one.pem" ec2-user@ec2-34-208-121-121.us-west-2.compute.
amazonaws.com
The authenticity of host 'ec2-34-208-121-121.us-west-2.compute.amazonaws.com (34.208.121.121)' can't
be established.
ECDSA key fingerprint is SHA256:iXV/jPbeKkSWFajE0h6FYizSlSeGegdG+h+6Dx2PAy8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-34-208-121-121.us-west-2.compute.amazonaws.com,34.208.121.121'
(ECDSA) to the list of known hosts.
--- JUNOS 17.4R1-S1.9 Kernel 64-bit  JNPR-11.0-20180127.fdc8dfc_buil
ec2-user> show interfaces terse ge*
Interface            Admin Link Proto    Local              Remote
ge-0/0/0             up    up

ec2-user> exit

Connection to ec2-34-208-121-121.us-west-2.compute.amazonaws.com closed.
ckim-mbp:Downloads ckim$
```

NOTE This playbook doesn't include a destroy script. Both vSRX EC2 instances, elastic IP, and VPC, should be deleted and terminated manually.

Discussion

Bringing up a Junos device on AWS – such as the vSRX, for example – is a two-step process: the first step is to configure and bring up the vSRX instance in the AWS environment. The second step is to configure the vSRX as a networking device. This recipe demonstrates the first step.

While Ansible can bring up the VPC, subnet, and most of AWS resources, configuring them is not its strength. A better example would be CloudFormation or Terraform, as Ansible keeps its states and coordinates the resources and references, in the same way that Python handles lower level tasks for developers. The example in this recipe is partially idempotent, and demands much more effort than the two mentioned above.

While CloudFormation allows you to automate configuration of AWS resources, however, Ansible helps when you need to make specific changes to the configuration of your instances themselves.

And there's more on CloudFormation in Recipe 4.

Recipe 4: Setting Up AWS for CloudFormation Templates

by Ali Bidabadi

This recipe describes the creation of a CloudFormation template using the AWS CloudFormation service. It also outlines the steps needed to deploy a CloudFormation template that provisions a vSRX Next Generation Firewall.

Problem

How do you deploy a vSRX using the AWS CloudFormation service?

Solution

The solution is to use *CloudFormation Templates* to deploy a Juniper vSRX NGFW appliance in AWS, but let's first present an overview of the AWS CloudFormation service.

AWS CloudFormation Overview

AWS CloudFormation is an AWS service that enables users to automate deployment of AWS resources into their AWS accounts. It is a free service that helps model and set up AWS resources. It also gives users an easy way to create and manage a collection of related AWS resources.

AWS CloudFormation has two parts: *stacks* and *templates*.

Resources are managed as a single unit called a *stack*. You can create, update, and delete a collection of resources by creating, updating, and deleting stacks. All the resources in a stack are defined by the stack's AWS CloudFormation template.

The *template* is a JSON (or YAML) file that describes resources that are needed to run the application. For instance, a template may declare that the application requires an EC2 instance, a S3 bucket policy, and an AWS Identity and Access Management (IAM) policy.

Once a template is submitted to the AWS CloudFormation service, it creates all the necessary resources in the user's account by making underlying service calls to AWS. It then builds a running instance of the template, putting data flows and dependencies in the right order. This running instance is called a stack.

An important advantage of AWS CloudFormation is that it allows developers to automate service deployment in a simple way. There is no additional charge for AWS CloudFormation. Users only pay for the AWS resources that are required to run their application.

AWS CloudFormation Template Anatomy

As mentioned earlier in this recipe, a template is a declaration of the AWS resources that make up a stack. The template is stored as a text file whose format complies with the JSON or YAML standard.

CloudFormation templates consist of several components. Some components are required while others are optional. Let's take a look at each one of them:

Format Version (optional): Specifies the AWS CloudFormation template that the template conforms to. The template format version is not the same as the API version and can change independent of the API version.

Description (optional): A text string that describes the template. This section must always follow the template format version section.

Metadata (optional): Another optional section that provides additional information about the template.

Parameters (optional): Specifies values that can be passed in to a template at runtime. You can refer to parameters in the Resources and Outputs sections of the template.

Mappings (optional): A mapping of keys and associated values that can be used to specify conditional parameter values, similar to a lookup table. You can match a key to a corresponding value by using the `Fn::FindInMap intrinsic` function in the Resources and Outputs sections.

Conditions (optional): Defines conditions that control whether certain resources are created or whether certain resource properties are assigned a value during stack creation or update. For example, you could conditionally create a resource that depends on whether the stack is for a production or test environment.

Resources (required): The only required section of an AWS CloudFormation template. It specifies the stack resources and their properties, such as an Amazon Elastic Compute (Amazon EC2) instance or an Amazon Simple Storage Service (Amazon S3) bucket. You can refer to resources in the Resources and Outputs sections of the template.

Outputs (optional): Describes the values that are returned whenever you view your stack's properties. For example, you can declare an output for an Amazon S3 bucket name and then call the AWS CloudFormation describe-stacks AWS CLI command to view the name.

Okay, let's create a very simple AWS CloudFormation template. Figure 4.1 displays an AWS CloudFormation template that when submitted to the AWS CloudFormation service, creates an Amazon S3 bucket. This template has only three sections: a Resources section (which is required), and two optional sections, namely a Format Version, and Outputs.

```
{
 "AWSTemplateFormatVersion" : "2010-09-09",
  "Resources": {
   "S3Bucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
    }
   }
  },
  "Outputs": {
  "S3BucketName": {
   "Value": {"Ref": "S3Bucket"},
   "Description": "Name of S3 Bucket"
  }
 }
}
```

Figure 4.1 *Sample AWS CloudFormation Template*

Note that the Outputs section of this template returns the name of the newly created Amazon S3 bucket and makes it available to the template owner.

Deploying Juniper vSRX using the AWS CloudFormation Service

Okay, now that you have learned how to author a simple AWS CloudFormation template, let's deploy something more useful: a Juniper vSRX Next Generation Firewall appliance! Follow the steps below to achieve exactly that.

Navigate to the Juniper AWS GitHub repo homepage: https://github.com/Juniper/vSRX-AWS. There, you will find a template file called "vsrx.template". Download that file to a local folder on your computer.

Next navigate to the AWS CloudFormation service home page and upload the template that you just downloaded.

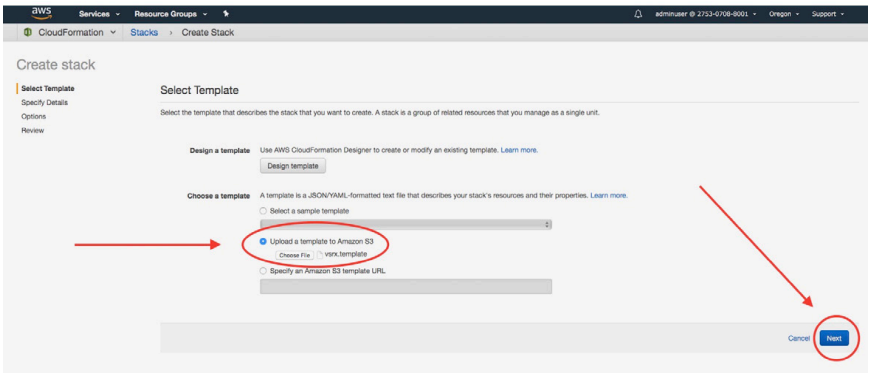

Figure 4.2 *AWS CloudFormation Home Page*

Click Next to continue deploying the stack.

It's time to specify the details of your stack as shown in Figure 4.3.

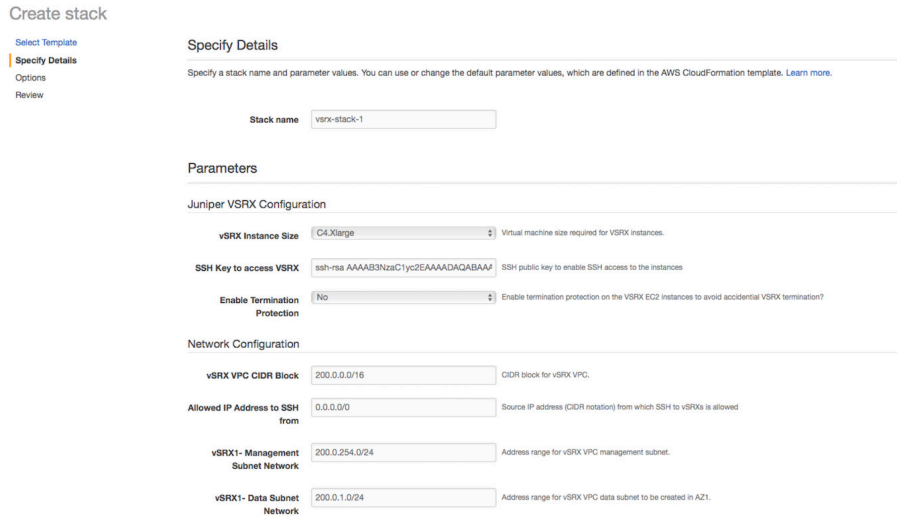

Figure 4.3 *vSRX CloudFormation Template Parameters*

Fill in the details. Note that some of the fields have been pre-populated with default values but those values can be changed. Here's a quick overview with explanations as needed:

- Specify Details: Stack name: Enter a unique name here
- Parameters: Juniper vSRX Configuration
- vSRX Instance Size: Select VM size required for vSRX instances

- SSH Key to access vSRX: SSH public key to enable SSH access to the instances. The whole content of the public key (and not the key name) needs to be copied here.
- Enable Termination Protection: (user's choice)
  - Yes
  - No (Okay for testing)
- Parameters: Network Configuration
- vSRX VPC CIDR Block
- Allowed IP Address to SSH from: Source IP address (CIDR) notation from where SSH to vSRXs is allowed.
- vSRX1: Management Subnet Network: Subnet for fxp0 for vSRX
- vSRX1: Data Subnet Network: Subnet for ge-0/0/0 for vSRX

Once the fields are filled in, click Next to be directed to the "Options" page.

On the "Options" page, in Figure 4.4, you are presented a set of Options for creation of the stack. If resource tagging is not required in your setup, you can skip this step by clicking Next, and you'll go to the Review page shown in Figure 4.5.

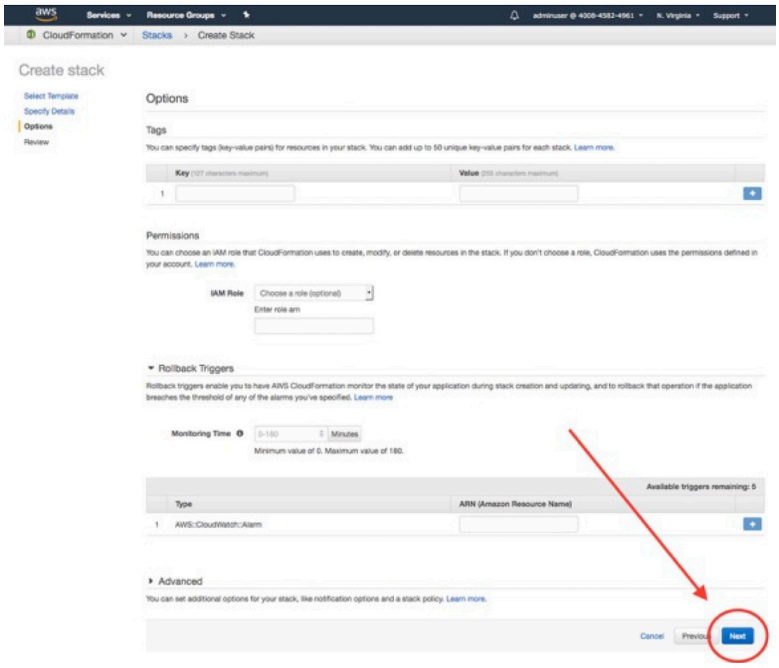

Figure 4.4 *vSRX CloudFomation Options Page*

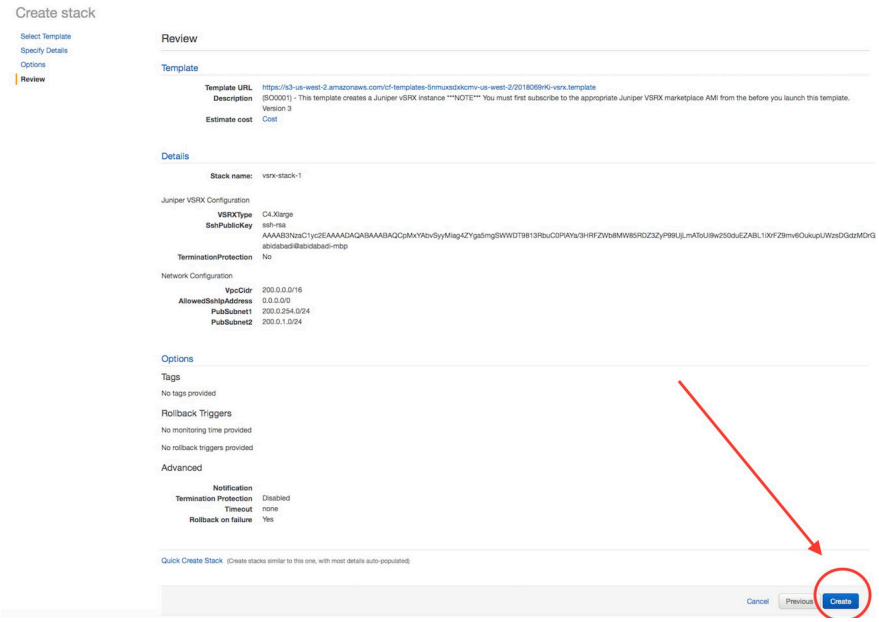

Figure 4.5 *vSRX CloudFormation Review Page*

Review the Template, Detail, and Options information. Then, as shown in Figure 4.5, click Create.

After clicking the Create button you'll follow the AWS CloudFormation stack creation process in Figure 4.6. Although at this point the vSRX has been deployed, it will take several minutes for the vSRX instance to complete its bootup.

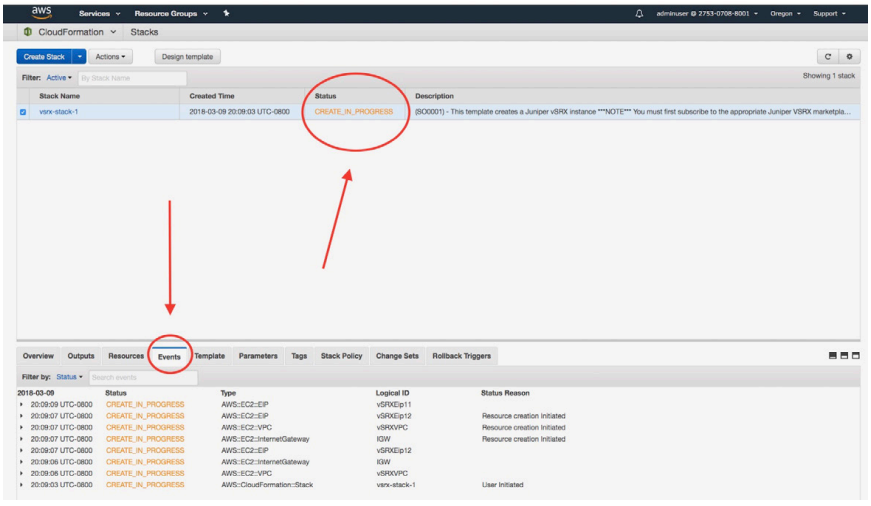

Figure 4.6 *AWS CloudFormation Stack Creation Progress*

The status will change to "CREATE_COMPLETE" only after all the related resources have been created as shown in Figure 4.7.

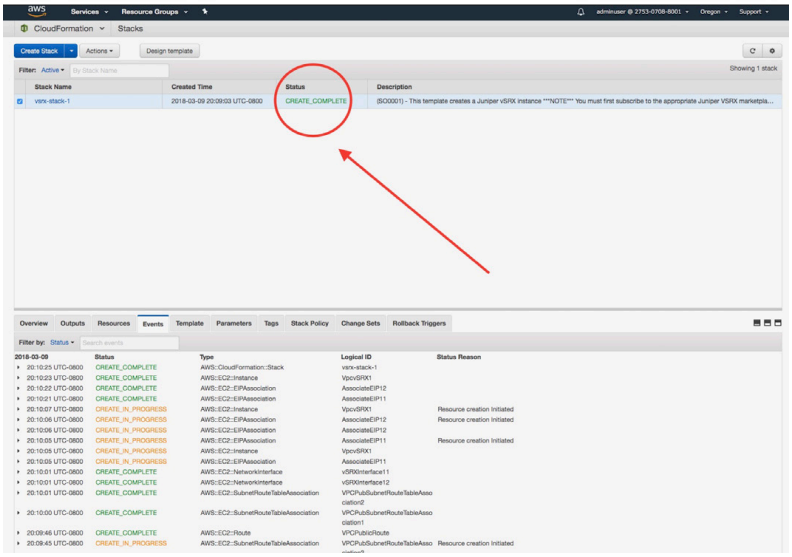

Figure 4.7 *AWS CloudFormation Stack Created*

It's a good indication when the 2/2 status checks have cleared and passed, as shown in Figure 4.8. Notice the green lights, indicating everything is up and running.

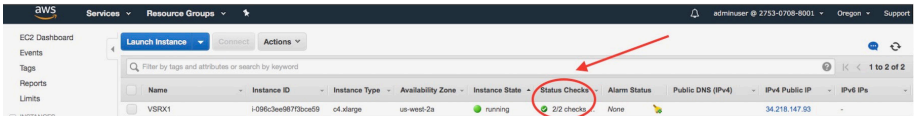

Figure 4.8 *AWS EC2 Dashboard After vSRX Deployment*

If you would like to know what other resources, in addition to the vSRX, have been created, navigate to the AWS CloudFormation service home page and click on the "Resources" tab. You will notice that a VPC, two Elastic IPs, two subnets, and a security group have been created.

Conclusion

This recipe covered an overview of the AWS CloudFormation service and discussed various components of CloudFormation templates and how a CloudFormation template can be authored. It creates a vSRX using a vSRX template from the Juniper AWS GitHub repository.

Recipe 5: Launching Juniper Instances with Terraform

by Charlie Chang-Hyun Kim

Terraform Version Used: v0.11.5

Ubuntu Version Used: 16.4

Junos OS Used: 17.4R1

Juniper Platforms General Applicability: vMX, vSRX, AppFormix

This recipe shows you how to launch a pair of vSRXs using Terraform. Terraform is a cool abstraction layer that allows your scripts to work in multiple clouds. While CloudFormation templates are in a form of simple file (or files) to upload and run by the AWS GUI, Terraform requires a client on which to run, which, in a sense, is similar to Ansible. The first part of this recipe helps you set them up with CloudFormation, and the second part shows you the steps for successful launching of Terraform and the vSRXs.

MORE? For more about Terraform, please refer to https://www.terraform.io/ .

Problem

While CloudFormation is an AWS-native orchestration tool and specialized to AWS, Terraform is more generic and has an interface that works with multiple platforms. Most of the data scheme is quite similar to CloudFormation, yet incorporates its own syntax.

Solution

Terraform runs on various operating systems and platforms. For more detail about those, refer to https://www.terraform.io/downloads.html. This recipe guides you to install Ubuntu 16 AMI within AWS for demonstration purposes.

For convenience, a CloudFormation template has been prepared at https://raw.githubusercontent.com/kimcharli/dayone-aws-terraform/master/cf-for-ubuntu-ec2.yml. This template was modified from https://s3-us-west-2.amazonaws.com/CloudFormation-templates-us-west-2/EIP_With_Association.template just enough to load Ubuntu 16 AMI from region us-west-2. Please download *cf-for-ubuntu-ec2.yml* and save it in a place you can pull it from.

For added security, it's recommended you allow SSH only from your source address. One of the ways to get it is from http://icanhazip.com/. Save *your source IP address*.

By default, cloud instances prefer public keys instead of passwords. From the AWS console, in the Services tab, follow the path: EC2 > Key Pairs, and click on Create Key Pair. Give it a name such as "*day-one*", and download it. This recipe's file name is *day-one.pem*.

Now, go to the AWS console, select CloudFormation from Services, and click Create Stack. From the Select Template dialogue, under the Choose a template section, click on Browse, and select the file *cf-for-ubuntu-ec2.yml* file you saved earlier. Click Next, and give the Stack a name, here '*day-one-stack*', leave the parameters as they are, but update the SSH Location with *your source IP address* with a postfix /32, such as *1.2.3.4/32*. Click Next, and set the 'Name' as tags Key, and '*day-one-ubuntu*' as Value. Click Next, review the data, and click Create.

While waiting for the Status Checks to become 2/2 for the Amazon EC2 *day-one-ubuntu*, adjust the attribute of *day-one.pem* such that only you may read. The equivalent `*nix` command is `chmod 600 day-one.pem`. Once the Amazon EC2 Status Checks becomes 2/2, get the connection information by clicking on the Connect button over the list, and log in to the Amazon EC2:

```
ckim-mbp:Downloads ckim$ ssh -i "day-one.pem" ubuntu@ec2-18-236-57-126.us-west-2.compute.amazonaws.
com
The authenticity of host 'ec2-18-236-57-126.us-west-2.compute.amazonaws.com (18.236.57.126)' can't be
established.
ECDSA key fingerprint is SHA256:7vezgFq/i1msg07FiT4hTfKaTOkZiJmkaPr6NaOiQEA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-18-236-57-126.us-west-2.compute.amazonaws.com,18.236.57.126' (ECDSA)
to the list of known hosts.
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-1052-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
```

```
  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user “root”), use “sudo <command>”.
See “man sudo_root” for details.

ubuntu@ip-172-31-31-91:~$
```

Now install the packages and configure the AWS environment:

```
ubuntu@ip-172-31-31-91:~$ sudo apt update

ubuntu@ip-172-31-31-91:~$ sudo apt -y install awscli zip

ubuntu@ip-172-31-31-91:~$ aws configure
AWS Access Key ID [None]: AAAAAAAAAAAAAAAAAAAA
AWS Secret Access Key [None]: SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
Default region name [None]: us-west-2
Default output format [None]: json
ubuntu@ip-172-31-22-242:~$ aws ec2 describe-instances

ubuntu@ip-172-31-22-242:~$
```

Download the Terraform binary and install it:

```
ubuntu@ip-172-31-31-91:~$ wget https://releases.hashicorp.com/terraform/0.11.5/terraform_0.11.5_
linux_amd64.zip?_ga=2.194997421.1420097966.1522597358-410716570.1522597358
--2018-04-01 21:28:41--  https://releases.hashicorp.com/terraform/0.11.5/terraform_0.11.5_linux_
amd64.zip?_ga=2.194997421.1420097966.1522597358-410716570.1522597358
Resolving releases.hashicorp.com (releases.hashicorp.com)... 151.101.53.183, 2a04:4e42:d::439
Connecting to releases.hashicorp.com (releases.hashicorp.com)|151.101.53.183|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16479399 (16M) [application/zip]
Saving to: ‚Äòterraform_0.11.5_linux_amd64.zip?_ga=2.194997421.1420097966.1522597358-
410716570.1522597358,Äô

terraform_0.11.5_linux_amd64.zip?_ga=2.1 100%[=====================================================
==========================>]  15.72M  38.1MB/s    in 0.4s

2018-04-01 21:28:41 (38.1 MB/s) - ‚Äòterraform_0.11.5_linux_amd64.zip?_
ga=2.194997421.1420097966.1522597358-410716570.1522597358,Äô saved [16479399/16479399]

ubuntu@ip-172-31-31-91:~$ mv terraform_0.11.5_linux_amd64.zip\?_
ga\=2.194997421.1420097966.1522597358-410716570.1522597358 terraform_0.11.5_linux_amd64.zip
ubuntu@ip-172-31-31-91:~$ unzip terraform_0.11.5_linux_amd64.zip
```

```
Archive:  terraform_0.11.5_linux_amd64.zip
  inflating: terraform
ubuntu@ip-172-31-31-91:~$ sudo mv terraform /usr/local/bin/
ubuntu@ip-172-31-31-91:~$
ubuntu@ip-172-31-31-91:~$ terraform --version
Terraform v0.11.5

ubuntu@ip-172-31-31-91:~$
```

Copy over the *day-one.pem* file for Amazon EC2 to be able to access the vSRX being launched:

```
ckim-mbp:~ ckim$ scp -i Downloads/day-one.pem Downloads/day-one.pem ubuntu@18.236.57.126:
day-one.pem
100% 1696    17.3KB/s   00:00
ckim-mbp:~ ckim$
ckim-mbp:~ ckim$ ssh -i Downloads/day-one.pem ubuntu@18.236.57.126 chmod 600 day-one.pem
ckim-mbp:~ ckim$
```

Now Terraform is ready to run. Download the Terraform template, and unarchive it:

```
ubuntu@ip-172-31-31-91:~$ wget https://github.com/kimcharli/dayone-aws-terraform/archive/master.zip
--2018-04-01 21:47:20--  https://github.com/kimcharli/dayone-aws-terraform/archive/master.zip
Resolving github.com (github.com)... 192.30.255.112, 192.30.255.113
Connecting to github.com (github.com)|192.30.255.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/kimcharli/dayone-aws-terraform/zip/master [following]
--2018-04-01 21:47:20--  https://codeload.github.com/kimcharli/dayone-aws-terraform/zip/master
Resolving codeload.github.com (codeload.github.com)... 192.30.255.121, 192.30.255.120
Connecting to codeload.github.com (codeload.github.com)|192.30.255.121|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: ‚Äòmaster.zip‚Äô

master.zip                              [ <=>
]  15.77K  --.-KB/s    in 0.008s

2018-04-01 21:47:20 (2.04 MB/s) - ‚Äòmaster.zip‚Äô saved [16146]

ubuntu@ip-172-31-31-91:~$ unzip master.zip
Archive:  master.zip
7614286d1bf01209a1342c526fc871c00a6792f2
   creating: dayone-aws-terraform-master/
  inflating: dayone-aws-terraform-master/cf-for-ubuntu-ec2.yml
   creating: dayone-aws-terraform-master/terraform-template/
  inflating: dayone-aws-terraform-master/terraform-template/README.md
   creating: dayone-aws-terraform-master/terraform-template/jnpr_aws_ec2/
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_ec2/main.tf
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_ec2/outputs.tf
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_ec2/security_group.tf
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_ec2/variables.tf
   creating: dayone-aws-terraform-master/terraform-template/jnpr_aws_vpc/
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_vpc/main.tf
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_vpc/outputs.tf
```

```
  inflating: dayone-aws-terraform-master/terraform-template/jnpr_aws_vpc/variables.tf
  inflating: dayone-aws-terraform-master/terraform-template/main.tf
  inflating: dayone-aws-terraform-master/terraform-template/outputs.tf
  inflating: dayone-aws-terraform-master/terraform-template/variables.tf
ubuntu@ip-172-31-31-91:~$
```

This recipe example utilizes modules of Terraform. The first step is to get the module, and initialize it:

```
ubuntu@ip-172-31-31-91:~$
ubuntu@ip-172-31-31-91:~$ cd dayone-aws-terraform-master/terraform-template/
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform get
- module.jnpr_aws_vpc
  Getting source "jnpr_aws_vpc"
- module.jnpr_aws_ec2
  Getting source "jnpr_aws_ec2"
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform init
Initializing modules...
- module.jnpr_aws_vpc
- module.jnpr_aws_ec2

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "http" (1.0.1)...
- Downloading plugin for provider "aws" (1.13.0)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 1.13"
* provider.http: version = "~> 1.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
```

This example also utilizes workspace for multiple regions and VPC:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform workspace list
* default
```

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform workspace new tvpc1
Created and switched to workspace "tvpc1"!
```

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform workspace list
  default
* tvpc1

ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
```

The main commands are `plan`, `apply`, and `destroy`. With the `plan` command, Terraform lists all the resources it would create. This is useful in case of resource conflict and troubleshooting. In the next example, `apply` is completed within a minute. But it will take several minutes for the vSRX instance to complete its boot up:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform plan
Refreshing Terraform state in-memory prior to plan...


ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform apply


Plan: 23 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: yes


Apply complete! Resources: 23 added, 0 changed, 0 destroyed.

Outputs:

fxp0_ip = [
    10.10.128.63,
    10.10.129.145
]
ge000_ip = [
    10.10.128.105,
    10.10.129.100
]
public_ips = [
    35.164.38.22,
    34.218.196.14
]
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
```

Once the vSRX gets online, SSH to it and verify the connectivity:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ ssh -i ~/day-one.pem
ec2-user@35.164.38.22
The authenticity of host '35.164.38.22 (35.164.38.22)' can't be established.
ECDSA key fingerprint is SHA256:Vsm1q8Z1xdP8qeeJO665m2WQNa5P3xCle5HsSydQbqA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.164.38.22' (ECDSA) to the list of known hosts.
--- JUNOS 17.4R1-S1.9 Kernel 64-bit  JNPR-11.0-20180127.fdc8dfc_buil
ec2-user> show interfaces terse ge*
Interface            Admin Link Proto    Local             Remote
ge-0/0/0             up    up

ec2-user>
```

Finally, when you decide to dismantle it, use the destroy command `–force flag`, to indicate that you want to execute without verifying and confirming:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform destroy
or
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform destroy –force
```

Discussion

This Terraform example has separate folders for EC2 and VPC, but it can be implemented without folders. Folders can be implemented in Terraform modules, which are useful in cases with a bigger scope, but there is no absolute guideline on what file sizes work better with modules.

There are three types of templates in this example. The file *variables.tf* is feeding values to the *main.tf*, and then *main.tf* to *outputs.tf*. Those files can also be collapsed into one file with everything in it. However, it is good practice to separate main functions from variables.

The vSRX and vMX instances are not free, so it is recommended to terminate instances whenever you no longer need them.

The following example uses a single workspace *tvpc1*. The template can be extended by augmenting the variables with additional work spaces within the *variables.tf* file:

```
variable “aws_region” {
  type = “map”
  default = {
    tvpc1 = “us-west-2”
  }
}

variable “vpc_names” {
  type = “map”
```

```
  default = {
    tvpc1 = "transit-vpc1"
  }
}

variable "vpc_nets" {
  type = "map"
  default = {
    tvpc1 = "10.10.128.0/17"
  }
}

variable "vpc_vsrx_subnet" {
  type = "map"
  default = {
    tvpc1 = [ "10.10.128.0/24", "10.10.129.0/24" ]
  }
}

variable "vsrx_host_name" {
  type = "map"
  default = {
    tvpc1 = [ "host-transit1-1", "host-transit1-2"]
  }
}
```

This template configures multiple scenarios across multiple regions. You may replace them with a single workspace, but it would be effective to keep them all, and switch between them by workspace. For example, if you want to deploy to region us-east-1 with name *tvpc2*, you would get an image-id as shown here:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ LANG=en_us.utf8 aws ec2
describe-images --region us-east-1  --filters "Name=name,Values=junos-media-vsrx*"  --query 'Images[*].
{ID:ImageId,Name:Name}'
[
   {
      "Name": "junos-media-vsrx-x86-64-vmdisk-17.4R1-S1.9-consec-deea68f3-6505-43f8-88cf-
b4e9e99e2d89-ami-f9191f83.4",
      "ID": "ami-778c760a"
   },
   {
      "Name": "junos-media-vsrx-x86-64-vmdisk-17.4R1-S1.9-appsec-8987d154-c3e3-411e-8f38-
aecdd80e529f-ami-bd1a1cc7.4",
      "ID": "ami-91f027ec"
   },
   {
      "Name": "junos-media-vsrx-x86-64-vmdisk-17.4R1-S1.9-4d1495fd-4d1f-48d0-9ec6-b67794a58765-ami-
011f197b.4",
      "ID": "ami-eea55a93"
   }
]
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
```

Then, update:

```
variable “aws_region” {
  type = “map”
  default = {
    tvpc1 = “us-west-2”
    tvpc2 = “us-east-1”
  }
}

variable “aws_vsrx_amis” {
  type = “map”
  default = {
    us-west-2 = “ami-e42db19c”
    us-east-1 = “ami-91f027ec”
  }
}

variable “vpc_names” {
  type = “map”
  default = {
    tvpc1 = “transit-vpc1”
    tvpc2 = “transit-vpc2”
  }
}

variable “vpc_nets” {
  type = “map”
  default = {
    tvpc1 = “10.10.128.0/17”
    tvpc2 = “10.10.0.0/17”
  }
}

variable “vpc_vsrx_subnet” {
  type = “map”
  default = {
    tvpc1 = [ “10.10.128.0/24”, “10.10.129.0/24” ]
    tvpc2 = [ “10.10.0.0/24”, “10.10.0.0/24” ]
  }
}

variable “vsrx_host_name” {
  type = “map”
  default = {
    tvpc1 = [ “host-transit1-1”, “host-transit1-2”]
    tvpc1 = [ “host-transit2-1”, “host-transit2-2”]
  }
}
```

Then prepare the workspace as shown below and continue with the Terraform plan, apply, and destroy routine. Make sure to create the *key-pair* with the same name in all the participant regions:

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform workspace list
  default
* tvpc1

ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform workspace new
```

```
tvpc2
Created and switched to workspace “tvpc2”!
```

You’re now on a new, empty workspace. Workspaces isolate their state, so if you run “terraform plan,” Terraform will not see any existing state for this configuration.

```
ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$ terraform workspace list
  default
  tvpc1
* tvpc2

ubuntu@ip-172-31-31-91:~/dayone-aws-terraform-master/terraform-template$
```

Recipe 6: High Availability Design Using Dual Spokes with On-Box Python and Terraform Templates

by Tony Boerema and Charlie Chang-Hyun Kim

Junos OS Used: 17.4

Juniper Platforms General Applicability: vMX, vSRX

When utilizing Juniper's vSRX/vMX transit VPC hub-and-spoke design, the spokes can be configured with dual vSRXs or vMXs as active and standby routers using Juniper's On-Box Python scripting running in the transit hub devices.

This recipe explains the basics of configuring and deploying a High Availability solution with Terraform templates on AWS Cloud.

Problem

Inter VPC traffic within a spoke is forwarded to the primary or active vSRX/vMX based on the default route rule associated to the subnet, but AWS route rule allows only a single next-hop for the same destination, and it doesn't have fail-safe mechanisms like redundancy, recovery, or backup. As such, in case the primary/active vSRX/vMX goes out of service due to its failure or other maintenance purpose, the route table should be updated to steer the traffic to a secondary, or standby, vSRX or vMX.

Solution

The problem can be detected and acted on with monitoring and provisioning tools, which are not part of the vSRX (or vMX). Note there are cases when it is encouraged for this to be done by the vSRX itself, but this cookbook's current ver-

sion of Junos does not have native AWS API implemented yet.

So, the first section of this recipe deals with the failure recovery mechanism itself, and the second section shows you how to provision the vSRX with such tools as Ansible.

Figure 6.1 is a high-level illustration of fault recovery interaction between components of transit and spoke VPC.

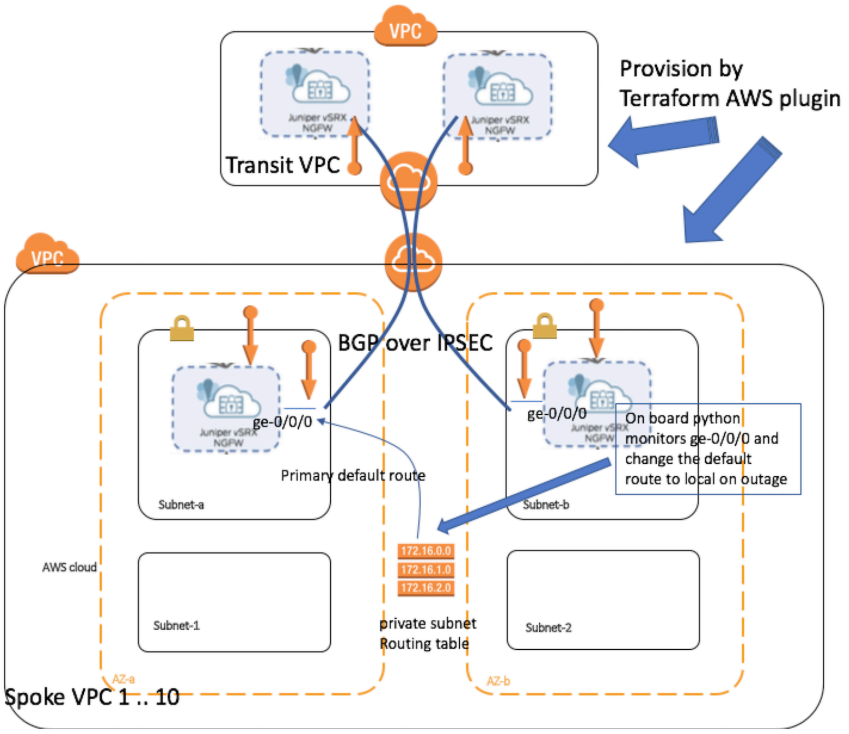

Figure 6.1 *Dual vSRX Spoke With Fault Recovery*

You can see the IPsec tunnels connect spoke VPC to transit VPC and exchange routing information using BGP. For earlier detection of the service outage, regardless of the failure of the spoke vSRX itself, or the data path between each endpoint, BFD is configured on each BGP session.

Here is the spoke vSRX `bfd-liveness-detection` configuration under the BGP protocol:

```
routing-instances {
   spoke {
      instance-type virtual-router;
      interface ge-0/0/0.0;
      interface st0.${st0_left};
      interface st0.${st0_right};
      routing-options {
         static {
```

```
            route 0.0.0.0/0 {
               next-hop ${vpc_vsrx_gw};
               preference 200;
            }
         }
         router-id ${vsrx_lo0_ip};
      }
      protocols {
         bgp {
            group transit {
               type external;
               description “Transit VPC peers”;
               preference 20;
               hold-time 30;
               export transit-export;
               peer-as ${transit_bgp_as};
               local-as ${vsrx_local_as};
               bfd-liveness-detection {
                  minimum-interval 1000;
                  multiplier 3;
                  transmit-interval {
                     minimum-interval 1000;
                  }
               }
               neighbor ${st0_left_ip_peer};
               neighbor ${st0_right_ip_peer};
            }
         }
      }
   }
}
```

And the respective transit vSRX configuration:

```
routing-instances {
   transit {
      instance-type virtual-router;
      interface ge-0/0/0.0;
      routing-options {
         generate {
            route 10.10.0.0/16;
         }
         router-id ${vsrx_lo0_ip};
         static {
            route 0.0.0.0/0 next-hop ${vpc_vsrx_gw};
         }
      }
      protocols {
         bgp {
            group spoke-peers {
               type external;
               preference 20;
               hold-time 30;
               import FILTER-DEFAULT;
               export spoke-export;
               peer-as 4400000000;
               local-as ${vsrx_local_as};
               bfd-liveness-detection {
```

```
                    minimum-interval 1000;
                    multiplier 3;
                    transmit-interval {
                       minimum-interval 1000;
                    }
                 }
              }
           }
        }
     }
}
```

With the configuration above, transit vSRX generates event BFDD_TRAP_SHOP_STATE_DOWN when packets are not received three times in a row from a spoke vSRX. The event is handled by the python JET script */var/db/scripts/jet/route_commander.py*:

```
# relevant JUNOS configuration
# set system extensions extension-service application file route_commander.py daemonize
# set system extensions extension-service application file route_commander.py username root
# set system services extension-service notification max-connections 20
# set system services extension-service notification allow-clients address 66.129.241.0/24
# set system services extension-service notification allow-clients address 127.0.0.1
# set interfaces lo0 unit 100 family inet address 127.0.0.1/32

import paho.mqtt.client as mqtt
import json
import yaml
import subprocess

mytopic = "/junos/events/syslog/BFDD_TRAP_SHOP_STATE_DOWN"
mymgd = "52.41.32.106"
myprivate_key = "/etc/ssh/ssh_host_rsa_key"
remote_user = "icenes"
remote_command= "op url /var/db/scripts/op/set-route-worker.py"
ssh_options = "-o StrictHostKeyChecking=no"

# write to syslog (/var/log/messages)
def do_syslog(msg):
    if __name__ == '__main__':
        import jcs
        jcs.syslog("172", "{} - {}".format(__file__.split('/')[-1], msg))
    else:
        print "{} - {}".format(__file__.split('/')[-1], msg)

def get_alternate(st0):
    # thefile = myfile if __name__ == "__main__" else "alternate.yaml"
    # with open(thefile, 'r') as stream:
    #     return yaml.load(stream)[st0]
    with open("/var/db/scripts/op/{}".format(st0), "r") as f:
        ip_address = f.read()
    return ip_address


def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
```

```
    client.subscribe(mytopic)

def on_message(client, userdata, msg):
    # print(msg.topic+" "+str(msg.payload))
    try:
        myevent = json.loads(msg.payload)['jet-event']
        do_syslog("{} event found with pip-interface={}".format(myevent['event-id'],
myevent['attributes']['pip-interface']))
        st0 = myevent['attributes']['pip-interface']
        alt = get_alternate(st0)
        do_syslog("alterate endpoint is {}".format(alt))
        try:
            command_in_string = "ssh -JU ice1 -i {} {} {}@{} {}".format(myprivate_key, ssh_options,
remote_user, alt,                                                    remote_command)
            do_syslog("sending command {}".format(command_in_string))
            subprocess.check_output(command_in_string.split(), stderr=subprocess.STDOUT)
        except subprocess.CalledProcessError as e:
            # do_syslog("e = {}".format(e))
            do_syslog("ERROR: {}:{}, {}".format(e.__class__.__name__, e.returncode, e.output))
            pass
        do_syslog("done")

    except (KeyError, AttributeError) as e:
        do_syslog("{}: key={}".format(e.__class__.__name__, e.message))
        pass

def main():
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message
    client.connect("127.0.0.1" if __name__ == '__main__' else mymgd, 1883, 60)

    client.loop_forever()

def test():
    '''test function to be called from Linux
      usage: python -c 'from route_commander import test; test()'
    '''
    print "{}: testing...".format(__name__)
    main()


if __name__ == '__main__':
    main()
```

Upon detection of BFD failure (`on_message()`), the script finds the standby spoke of this failing one (`get_alternate()`), and invokes the command "`op url /var/db/scripts/op/set-route-worker.py`".

The script */var/db/scripts/op/set-route-worker.py* of spoke vSRX retrieves the metadata to find its access key ID and secret access key allocated by AIM role:

```
    def get_access_keys(self):
        # role_name = urllib2.urlopen(AwsEc2.METADATA_SECURITY_CREDENTIALS_URL).read()
        role_name = get_meta_data("meta-data/iam/security-credentials/")
        do_syslog("attached role {}".format(role_name))
```

```
        # f = urllib2.urlopen("{}{}".format(AwsEc2.METADATA_SECURITY_CREDENTIALS_URL, role_name))
        j = json.loads(get_meta_data("meta-data/iam/security-credentials/{}".format(role_name)))
        # j = json.loads(f.read())
        self.AccessKeyId = j['AccessKeyId']
        self.SecretAccessKey = j['SecretAccessKey']
        self.SecurityToken = j['Token']
```

Next, it finds its VPC and ENI information from metadata for further processing:

```
def get_vpc_eni_id():
    macs = get_meta_data("meta-data/network/interfaces/macs/").split()
    for mac in macs:
        # do_syslog("interating macs {}".format(mac))
        if get_meta_data("meta-data/network/interfaces/macs/{}device-number".format(mac)) == '1':
            vpc_id = get_meta_data("meta-data/network/interfaces/macs/{}vpc-id".format(mac))
            eni_id = get_meta_data("meta-data/network/interfaces/macs/{}interface-id".format(mac))
            do_syslog("vpc_id={}, eni_id={}".format(vpc_id, eni_id))
            return (vpc_id, eni_id)
```

With the authentication keys retrieved above, script queries route tables to pick one associated with the specific tag. Here is its equivalent AWS CLI:

```
aws ec2 describe-route-tables --filters Name=vpc-id,Values=vpc-1f10397b
```

Next, it replaces the next hop of the destination 0.0.0.0/0 with the ENI of itself. Here is the equivalent AWS CLI:

```
aws ec2 create-route --destination-cidr-block "0.0.0.0/0" --network-interface-id eni-ba538395
--route-table-id rtb-553af032
```

These actions take place once the vSRX has been successfully deployed.

Deploy the vSRX

This recipe explores a Terraform template to launch a pair of vSRXs within a specific region in a specific availability zone with a default vSRX configuration.

The `cloud-init` feature helps instances to bring up initial packages and sets up in the usual images. The vSRX has its own dependencies due to its boot time. The boot time can interrupt the initial setup, so the script *wait-for-instance-ok.sh* waits for the instance to pass the AWS 2/2 checks before connecting.

Here, the initial configuration is prepared by the template, with variables:

```
data "template_file" "user_data" {
  count = 2
  template = "${file("${path.module}/files/${var.base_config_file[count.index]}")}"
  vars {
    host_name = "${var.vsrx_host_name[count.index]}"
    public_key = "${trimspace("${var.public_key}")}"
...
  }
```

```
}
```

It uses the configuration template like this:

```
#junos-config
system {
   host-name ${host_name};
   root-authentication {
      ssh-rsa "${public_key}";
   }
...
}
```

This data gets injected to `user_data` when two instances of vSRX are launched like this:

```
resource "aws_instance" "vsrx" {
  count = 2

  ami = "${var.aws_vsrx_amis[var.aws_region]}"
  instance_type = "${var.vsrx_instance_types[var.aws_region]}"
  key_name = "${aws_key_pair.mykp.key_name}"
  iam_instance_profile = "${var.instance_profile_name}"
  user_data = "${data.template_file.user_data.*.rendered[count.index]}"

  network_interface {
     device_index = 0
     network_interface_id = "${var.interfaces_fxp0_ids[count.index]}"
  }
  network_interface {
    device_index = 1
    network_interface_id = "${var.interfaces_ge000_ids[count.index]}"
  }

  tags {
    Name = "vsrx-${count.index}-${var.vpc_name}"
  }

  provisioner "local-exec" {
    command = "sh ${path.module}/files/wait-for-instance-ok.sh ${var.aws_region} ${aws_eip.
default.*.public_ip[count.index]} ${self.id} "
  }

}
```

The vSRX boot time can challenge the initial setup, so the script `wait-for-instance-ok.sh` is doing it by waiting for the instance to pass the AWS 2/2 checks before connecting:

```
MYDIR=$(dirname $0)
MYREGION=$1
MYIP=$2
MYINSTANCE=$3

AWSCMD="aws --region ${MYREGION} ec2 describe-instance-status --instance-ids ${MYINSTANCE}"

MYSTATUS=not-ok
```

```
while [ "${MYSTATUS}" != "ok" ]
do
  sleep 30
  MYSTATUS=$($AWSCMD | python -c "import json, sys; print json.load(sys.stdin)['InstanceStatuses'][0]
['InstanceStatus']['Status']")
done

ssh-keygen -f "/home/ubuntu/.ssh/known_hosts" -R ${MYIP}
scp -o StrictHostKeyChecking=no -r ${MYDIR}/op/ root@${MYIP}:/var/db/scripts/
scp -o StrictHostKeyChecking=no -r ${MYDIR}/jet/ root@${MYIP}:/var/db/scripts/
ssh -o StrictHostKeyChecking=no root@${MYIP} curl -s http://169.254.169.254/latest/user-data -o /var/
db/scripts/op/user-data
TIME_TO_COMMIT=$(ssh -o StrictHostKeyChecking=no root@${MYIP} date -v+3M +%H:%M)
ssh -o StrictHostKeyChecking=no root@${MYIP} "set PATH=/sbin:/bin:/usr/sbin:/usr/bin:/opt/sbin:/opt/
bin:/usr/local/bin ; /usr/sbin/cli -c \"edit ; load override /var/db/scripts/op/user-data ; commit at
${TIME_TO_COMMIT}; exit ; exit\" ; exit"
```

Once the instance status becomes active, those scripts in the front sections are copied to the correct folder of the vSRX. Next, it gets user data from the metadata, and loads the configuration from it.

Discussion

The AWS API has been implemented by this native python library with a narrow scope of a specific task, even though general implementation of the AWS API was not in the scope of this task.

One part of this implementation was achieved by Terraform and the other half with provisioning tools, like Ansible, Puppet, Chef, and so on.

Recipe 7: Transit VPC Deployment Using CloudFormation Templates

by Ali Bidabadi

This recipe describes the creation of a Transit VPC inside AWS using the AWS Marketplace Juniper Transit VPC solution. It also discusses how the solution can be customized to meet various needs of organizations by providing sample CloudFormation templates.

Problem

How do you deploy a Transit VPC from scratch?

Solution

The solution is to use CloudFormation templates to deploy a Transit VPC, but in order to get there let's first review the Transit VPC architecture because it's important to understand the general VPC concept.

Transit VPC Overview

Virtual Private Cloud (VPC) is an AWS offering that enables compute and storage resources to be consumed as a service from a publicly available pool of resources in a virtually private manner – that is, while the resources are part of a public cloud. The service is private in the sense that the subscriber of the service controls access to the resources.

The benefits of this type of service to the user are scalability, availability, and flexibility. Transit VPC is a specialized way of adding capability and flexibility to VPCs. The Transit VPC interconnects other VPCs acting as the hub for data flow between spoke VPCs and potentially other on-premise customer resources.

Transit VPCs are different than other VPCs because they contain the Juniper Networks vSRX Virtual Firewall. The AMI is available within the AWS Marketplace as a "Bring Your Own License" (BYOL) option or as a bundled annual or hourly usage license.

Transit VPC can be deployed in any region and availability zone that supports CloudFormation and Lambda functions. Transit VPCs can be shared between AWS accounts, and connecting spoke VPCs to the Transit VPC is automated.

VPC VGW tags trigger Lambda automation to configure the vSRX and spoke VPC VPN connections. Once the base vSRX elements are set up, the complexity of building this cloud-centric network are automated, including the management of dynamic BGP routing, organizing VPN connection policies, interface IP addressing, and inter-zone firewall security policies – all integrated and automated by the Juniper Networks Transit VPC stack.

The vSRX is the central element of the capabilities and flexibility of the Transit VPC. The connection between all cloud resources in spoke VPCs can be aggregated via the Transit VPCs to simplify and reduce overhead when connecting physical resources with cloud-based resources.

Figure 7.1 illustrates the relationship between the Transit VPC, spoke VPCs, and the data center.

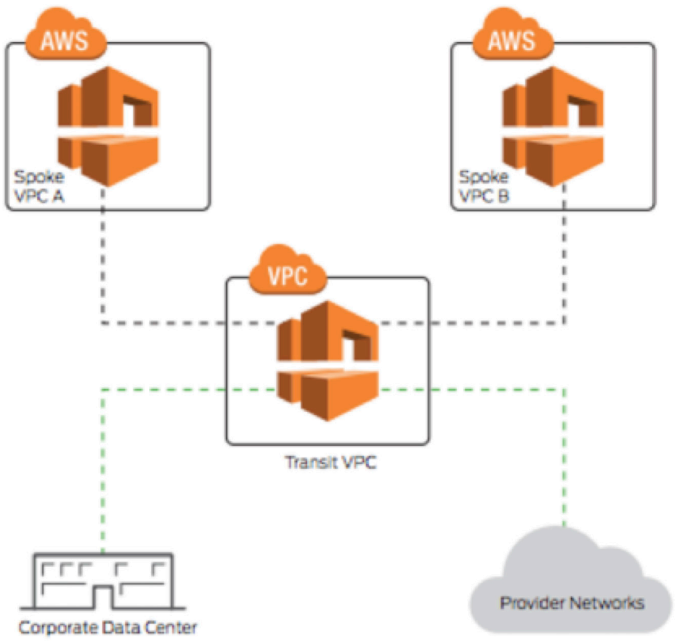

Figure 7.1 Transit VPC Overview

Transit VPC Architecture

This Transit VPC solution consists of the following four components:

- *Transit VPC:* The Transit VPC is like any other VPC except that it doesn't need a virtual private gateway (VGW) only an Internet gateway. It needs two subnets, one for the vSRX management interface and another for a VPN-connection termination interface. Transit VPC must be deployed in an available zone that supports CloudFormation, Lambda functions, S3, and KMS. Two AWS Lambda functions are critical to the Transit VPC solution: the Configurator and the Poller. Security group policy must allow vSRX management interfaces to be accessible from AWS Lambda, while vSRX VPN connection interfaces must be accessible from the spoke VPC VGWs. Transit VPCs are associated with a VGW tag (*Name:Value pair*). Spoke VPC VGWs with the specified tag will be automatically connected to the Transit VPC.
- *vSRX Virtual Next Generation Firewall:* The vSRX is a fully functional virtual appliance deployed as an Amazon EC2 instance inside the Transit VPC. The vSRX uses interface fxp0 as its management interface and ge-0/0/0 for VPN connections and inter VPC data flow. The CloudFormation template deploys two vSRX firewalls, each in different availability zones. Each vSRX uses two elastic network interfaces, one for its management interface (fxp0), and the other for its VPN-connection interface. Once CloudFormation has deployed the Transit VPC stack, vSRX management can be performed like any other Junos OS device, using SSH and interacting with the Junos OS CLI, although no manual configuration is necessary as the Transit VPC is a fully automated deployment.
- *Spoke VPC(s):* The spoke VPC is typical of any VPC use case. Classless Interdomain Routing (CIDR) is defined and subnets within the CIDR are assigned. Amazon EC2 resources deployed in the spoke VPC are connected and associated with subnets. Transit VPCs must have an internet gateway for external/public access, while spoke VPCs must have a VGW. Spoke VGWs intended to connect to a Transit VPC must have a tag configured to signal participation/inclusion with a Transit VPC. The Transit VPC VGW tag is specified in the CloudFormation stack deployment workflow. Spoke VPC VGW VPN connections use dynamic protocol BGP, which will share spoke VPC subnet information with the vSRX at the Transit VPC. The vSRX manages and control reachability information with all Transit VPC member spoke VPCs.
- *Physical data center (optional)*: The physical data center can represent any physical hardware network element. Its intention is to demonstrate connecting the Transit VPC environment with a physical network. While inter-VPC dataflow stays in the cloud, data can still be exchanged with the physical network and resources not in the cloud.

Figure 7.2 depicts the relationship between the four components of the Transit

VPC architecture.

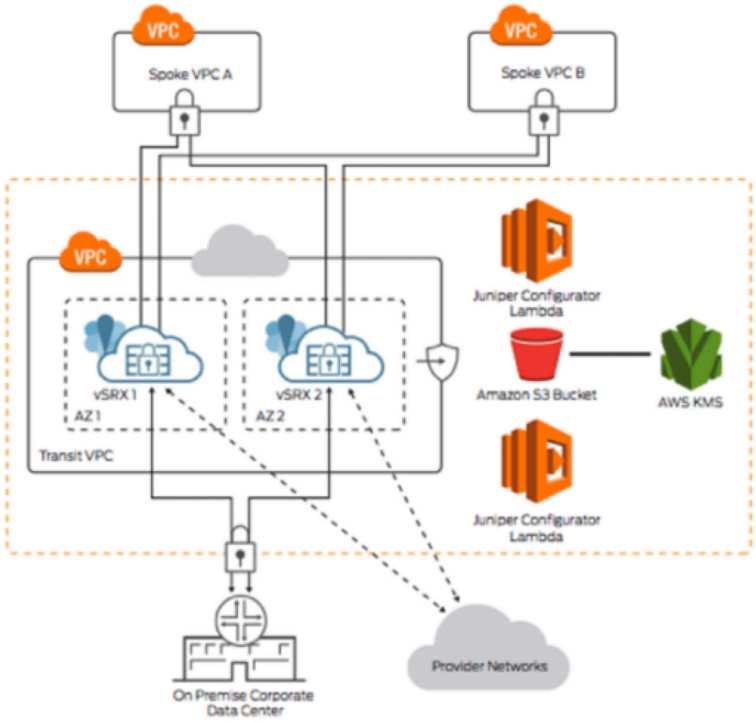

Figure 7.2 *Transit VPC Architecture*

Connecting Spoke VPCs to Transit VPC

Once the CloudFormation template successfully creates the Transit VPC and its artifacts (we'll begin in just a few more paragraphs), the Lambda automation infrastructure is in place to connect the spoke VPC to the Transit VPC. The VGW Poller and the Juniper Configurator Lambda functions work together to discover VGWs and configure VPN connections to the vSRX. As the Lambda function's VGW Poller checks VGWs, it looks for the tag specified in the CloudFormation workflow and checks VGWs in the Transit VPC accounts based on the functions that trigger *cron* settings. The default cron setting is "every 1 minute." If the Poller finds a VGW with the Transit VPCs tag "name:value" pair, the Lambda Function Configurator defines the VPN connection and builds and commits the vSRX configuration. The Lambda Poller function creates files in the S3 bucket specified during CloudFormation to signal to the configurator what to configure. All parameters and configurations associated with a VPN connection are saved in the S3 bucket as shown in Figure 7.3.

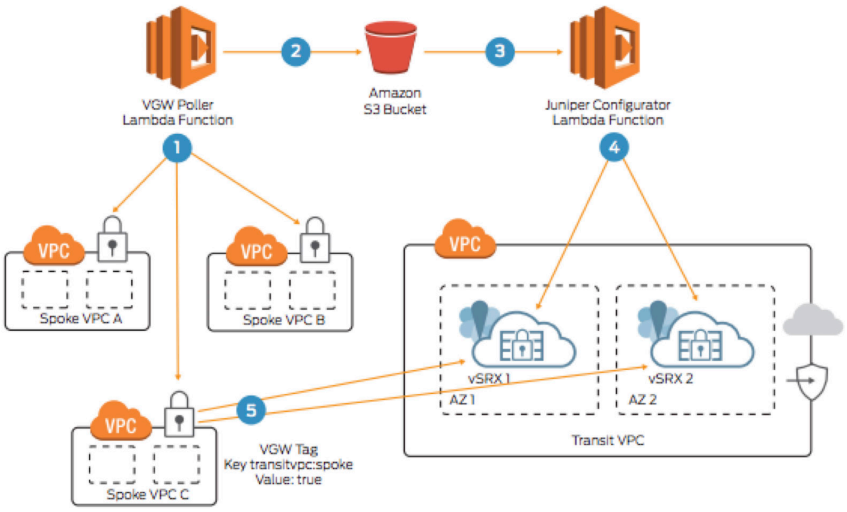

Figure 7.3 *Connecting Spoke VPCs to the Transit VPC*

In Figure 7.3 you can clearly see the VGW Poller and Configurator Lambda function interaction to automatically configure IPsec, BGP, and the security policies on the VGW and the vSRX.

The Automation Logic When Adding a Spoke VPC

Here's the automation logic in five steps:

- *Step 1:* At regular one-minute intervals, the VGW Poller Lambda function is called by a *CloudWatch* event, which scans each region of the customer's AWS account, specifically searching for virtual gateways of spoke VPCs tagged with the key and value defined during the CloudFormation stack creation. The Poller Lambda function checks for VGWs, which are not already associated with an existing VPN connection.
- *Step 2:* Once those VGWs have been identified with the correct tags applied, and without any existing VPN connections associated with them, the Lambda function creates the corresponding customer gateways, if required, for those spoke VPCs. The corresponding VPN connections to each vSRX in the Transit VPC are also created and the Elastic IP address associated with the *eth1* (ge-0/0/0) interface of the vSRX instance is targeted. The Lambda function pushes this connection information to an S3 bucket using S3 SSE-KMS keys. All data objects stored in the S3 bucket are encrypted using a solution-specific AWS KMS-managed customer master key (CMK).
- *Step 3:* A PUT event inside AWS S3 instantiates the Juniper Configurator Lambda function and parses the VPN connection information, generating the

required Junos OS configuration statements for creating new VPN connections and related security and routing policies on the vSRX instances.

- *Step 4:* The generated Junos OS configuration is pushed to the vSRX instances over an SSH connection.
- *Step 5:* Once the Junos OS configuration is loaded on the vSRX instances and committed, the IPsec VPN tunnels come up. The establishment of IPsec VPN tunnels enables BGP peering sessions to connect with the spoke VPC's VGW, allowing routing information to propagate between spoke and Transit VPC.

Deploying Transit VPC from the AWS Marketplace

Okay, that was the solution in a nutshell. It may sound complicated, but Juniper's Transit VPC can make all this automation logic simple to deploy.

The Juniper Transit VPC is available in the AWS Marketplace in two different licensing models: Bring Your Own License (BYOL), and Pay As You Go (PAYG). Let's start there.

Navigate to the AWS Marketplace home page. Type "Juniper Transit VPC" in the search bar. Both BYOL and PAYG licensing models should be available. Select BYOL (for the purpose of this recipe) as displayed in Figure 7.4.

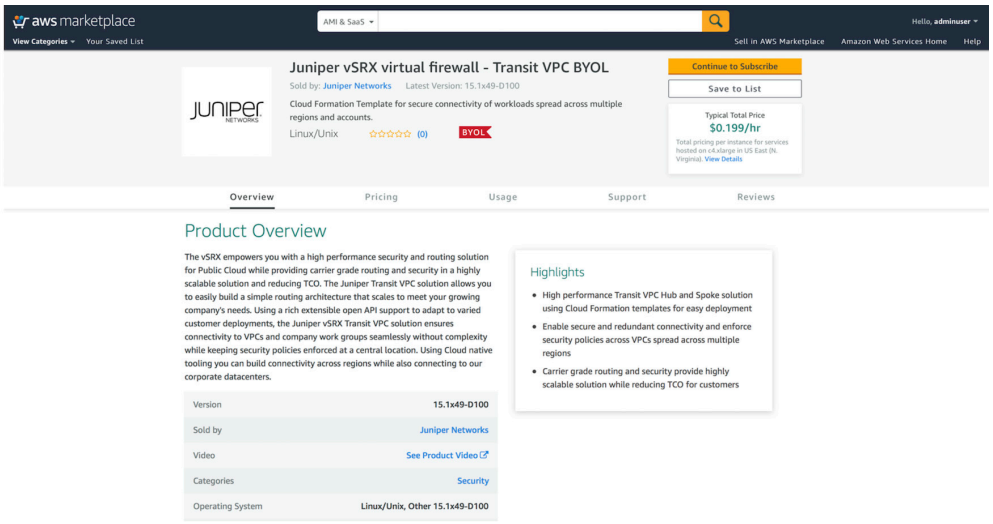

Figure 7.4 *Juniper Transit VPC Marketplace Listing*

Click on the "Continue to Subscribe" button to go to the launch page.

The prices are shown based on the region that you have selected in Figure 7.5. When you are ready, click the "Launch with CloudFormation console" button to be directed to the CloudFormation service home page.

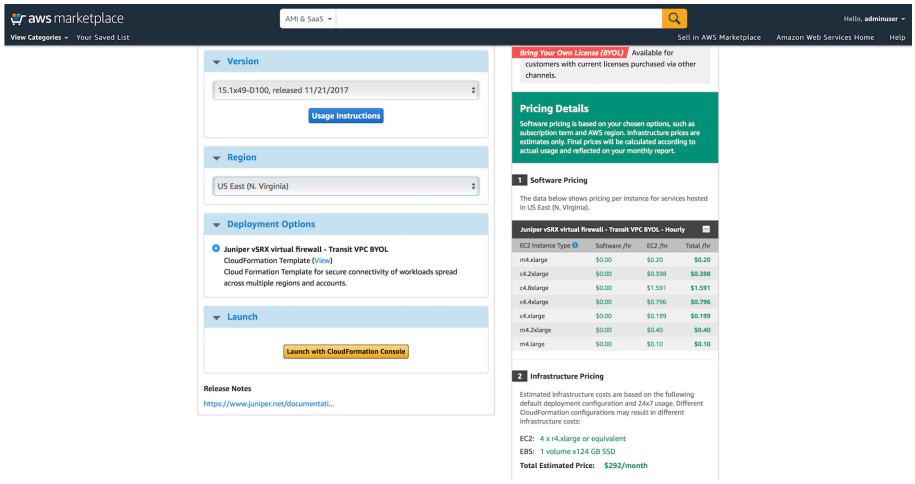

Figure 7.5 *Launch Juniper Transit VPC from the AWS Marketplace*

The CloudFormation template has already been uploaded to a S3 location and selected for you. Click Next to continue deploying the stack.

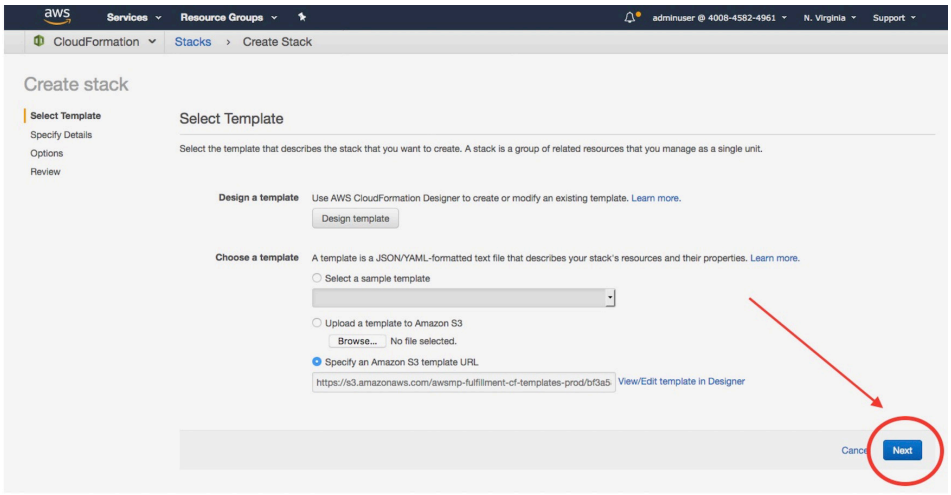

Figure 7.6 *AWS CloudFormation Home Page*

It's time to specify the details of your stack as shown in Figure 7.7.

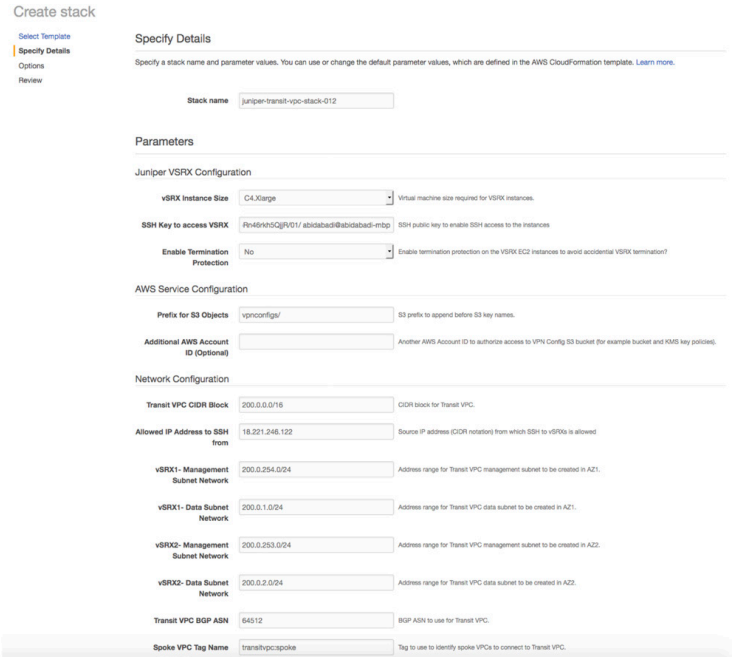

Figure 7.7 *Transit VPC CloudFormation Parameters*

And fill in the details. Note that some of the fields have been pre-populated with default values but those values can be changed. Here's a quick list with explanations as needed:

- Specify Details: Stack name: Enter a unique name here
- Parameters: Juniper vSRX Configuration vSRX Instance Size: Select VM size required for vSRX instances
- SSH Key to access vSRX: SSH public key to enable SSH access to the instances. The whole content of the public key (and not the key name) needs to be copied here
- Enable Termination Protection: (user's choice)
  - Yes
  - No (Okay for testing)
- Parameters: AWS Service Configuration Prefix for S3 Objects
- Parameters: Network Configuration Transit VPC CIDR Block Allowed IP Address to SSH from: Source IP address (CIDR) notation from where SSH to vSRXs is allowed
- vSRX1: Management Subnet Network: Subnet for fxp0 for vSRX1

- vSRX1: Data Subnet Network: Subnet for ge-0/0/0 for vSRX1
- vSRX2: Management Subnet Network: Subnet for fxp0 for vSRX2
- vSRX2: Data Subnet Network: Subnet for ge-0/0/0 for vSRX2
- Transit VPC BGP ASN: Any ASN as long as it is within the range
- Spoke VPC Tag Name: Very important. This tag name along with the tag value will be used to identify spoke VPCs to connect to the Transit VPC
- Spoke VPC Tag Value: Same as the above

Once the fields are filled in, click Next to be directed to the "Options" page.

Next you are presented a set of Options for creation of the stack. If resource tagging is not required in your setup, you can skip this step by clicking Next shown in Figure 7.8, and you'll go to the Review page as shown in Figure 7.9.

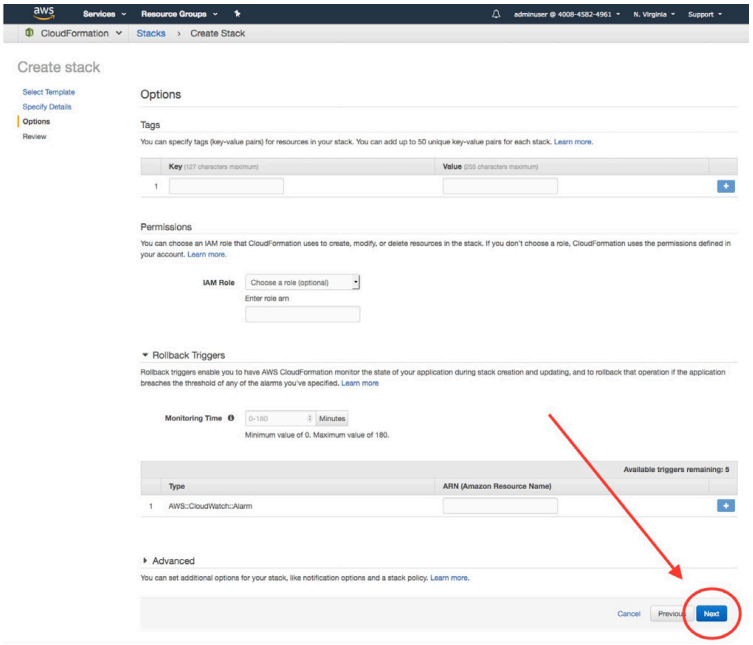

Figure 7.8 *Transit VPC CloudFormation Options Page*

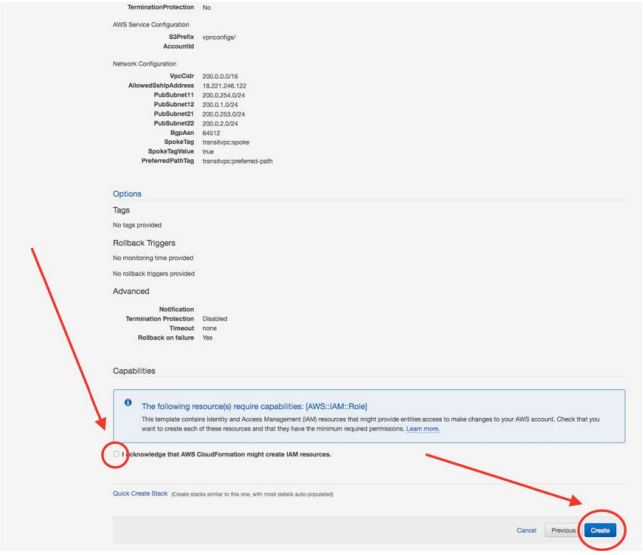

Figure 7.9 *Transit VPC CloudFormation Review Page*

Review the Template, Detail, and Options information. Then, as shown in Figure 7.9, check the box next to "*I acknowledge that AWS CloudFormation might create IAM resources*" and click Create.

After clicking the Create button you'll follow the CloudFormation stack creation process shown as beginning in Figure 7.10. This process will take several minutes to complete and no errors are expected in the logs.

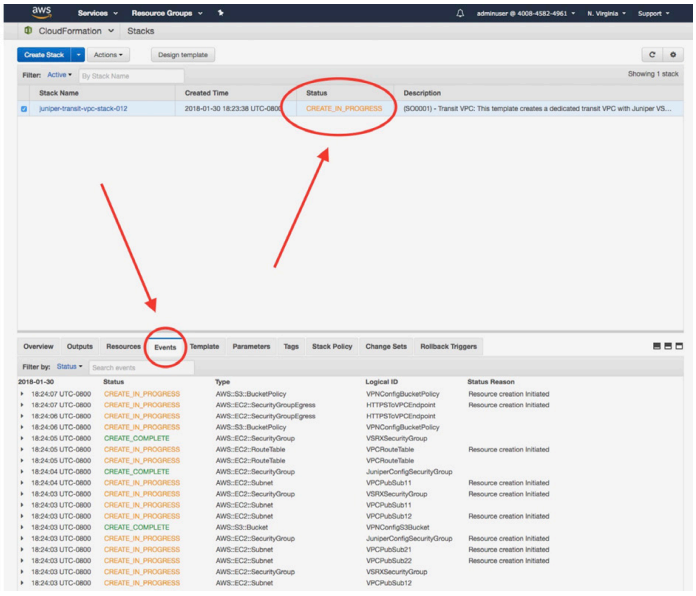

Figure 7.10 *CloudFormation Stack Creation Progress*

The status will change to "CREATE_COMPLETE" only after all the related resources have been created as shown in Figure 7.11.

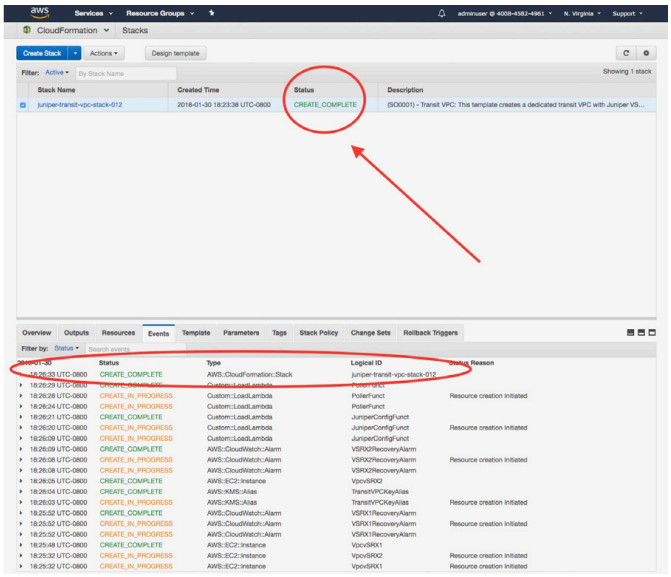

Figure 7.11 *CloudFormation Stack Created*

Although at this point the Transit VPC, including its two vSRXs, has been deployed, it will take several minutes for the vSRX instances to complete their boot up. A good indication as to when the instances are ready, as shown in Figure 7.12, is when the 2/2 status checks have cleared and passed. Notice the green lights, indicating up and running.

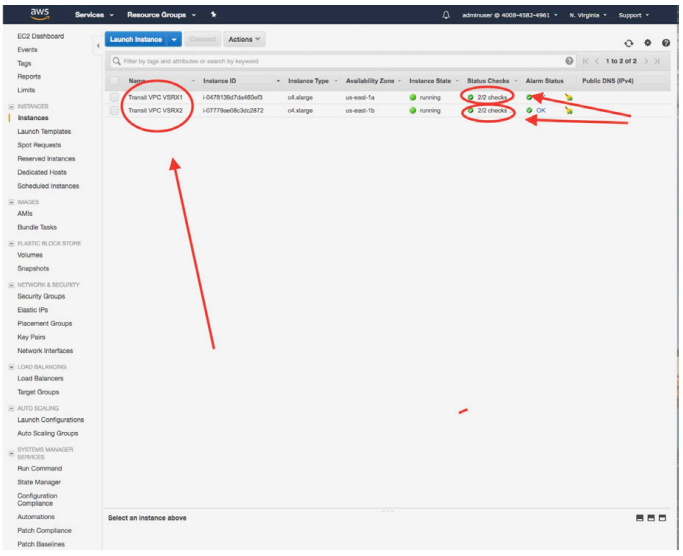

Figure 7.12 *AWS EC2 Dashboard After Transit VPC Deployment*

Connecting Spoke VPCs via the Transit VPC

As mentioned earlier in this recipe, one of the main objectives of deploying Transit VPC is to provide inter-VPC connectivity regardless of where each VPC is located. For instance, you can connect a spoke VPC in the Virginia region to a spoke VPC in the Oregon region. The steps to achieve this are as follows.

Navigate to the AWS VPC dashboard and click "Virtual Private Gateways" as shown in Figure 7.13. Make sure that you are in the region where your first spoke VPC is located.

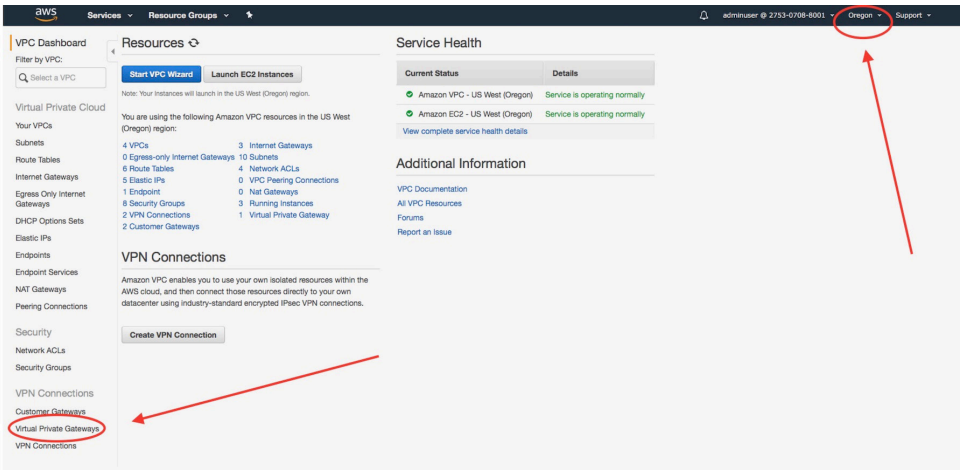

Figure 7.13 *AWS VPC Dashboard*

All VGWs in the selected region are listed as shown in Figure 7.14. Note that your selected VGW must be attached to your first spoke VPC.

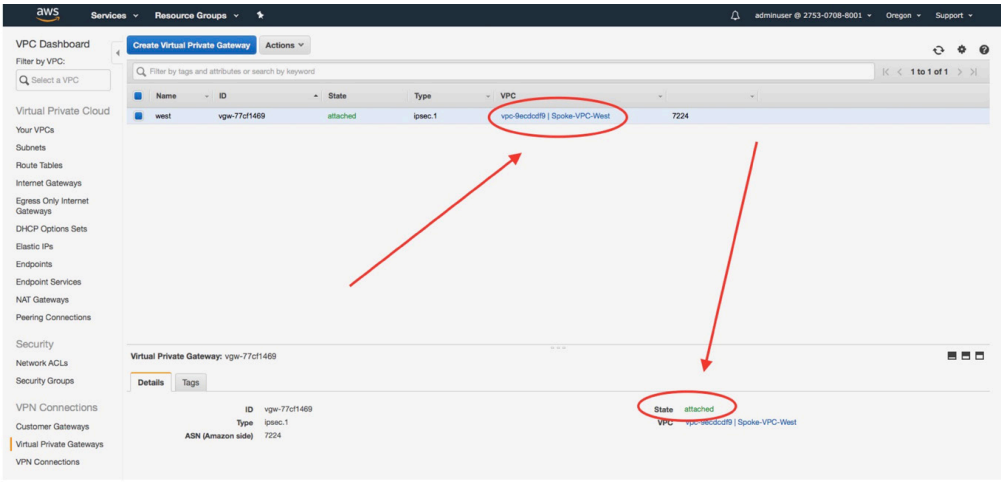

Figure 7.14 *VGW Attached to a Spoke VPC*

The next step in the process is to tag the selected VGW with the name/value pair that was defined during the stack creation in order to identify this spoke to connect to transit VPC. Figure 7.15 shows the tagging process.

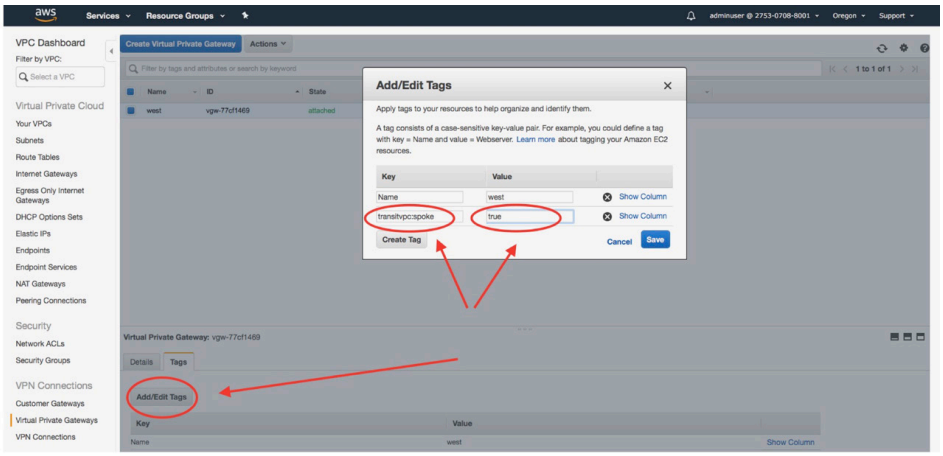

Figure 7.15 *Tagging the AWS VGW*

Once the appropriate tags have been applied to the VGW, it will take a minute or so for the VGW Lambda function to detect the tags and trigger the configurator Lambda function, which pushes the appropriate configuration into the vSRX instances. After a few minutes two VPN connections from the VGW will be established, one to vSRX1 and another one to vSRX2, as depicted in Figure 7.16. The tunnel status should be "UP" at this point, as indicated at the bottom of Figure 7.16.

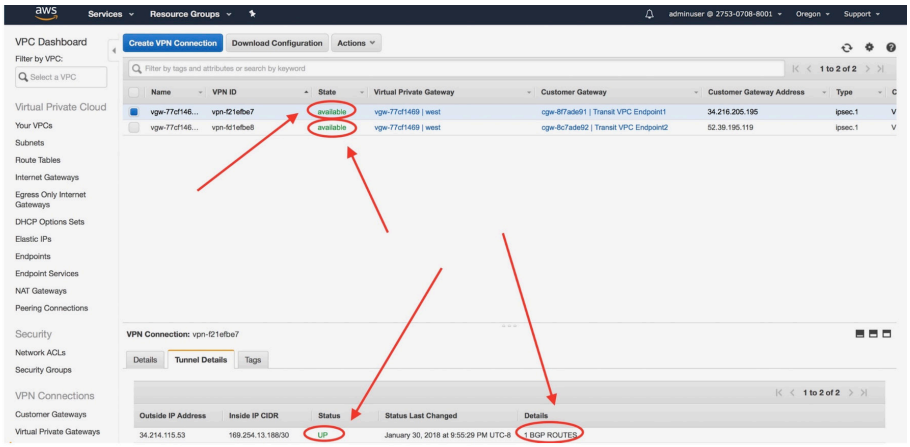

Figure 7.16 *VPN Connections Between Spoke VPC and Transit VPC*

In order for the spoke VPC routes to be successfully propagated to the transit VPC, "Route Propagation" must be enabled in the corresponding spoke VPC subnet route table. To verify, navigate to the Route Tables page and select the route table from which the routes should be propagated to the Transit VPC. Then select the "Route Propagation" tab at the bottom half of the window. Finally, check the box next to Propagate the VPG as shown in Figure 7.17.

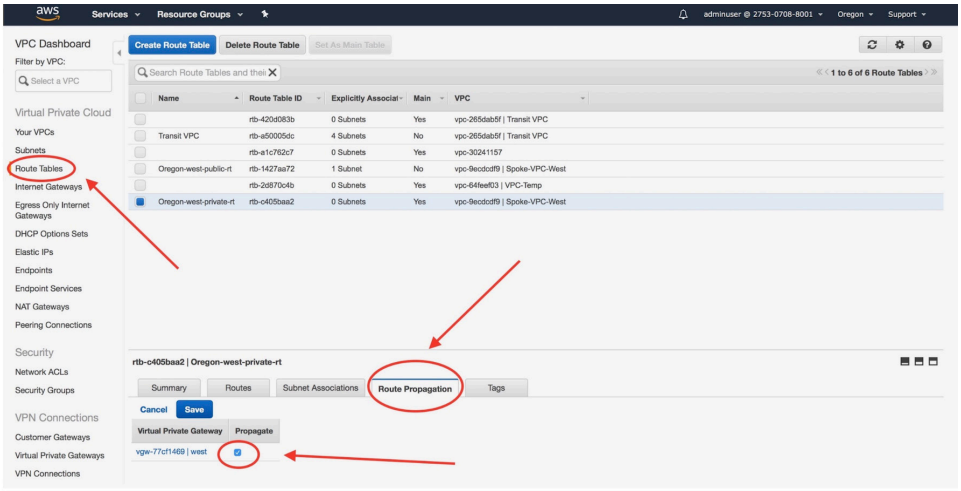

Figure 7.17 *Enabling Route Propagation in the Spoke VPC*

Repeat these steps for the other spoke VPC(s) to connect to the Transit VPC. For example, connect the other spoke VPC in the Virginia region to the Transit VPC.

After completing, you should have two spoke VPCs connected to the Transit VPC and each spoke VPC should see the other spoke VPC's routes in its route table. This is achieved by the BGP route propagation feature of the Transit VPC. Figure 7.18 displays the routes inside the Virginia spoke VPC subnet route table. Note that the first route is the local VPC CIDR, but the second route was learned through the Transit VPC BGP route discovery and propagation process.

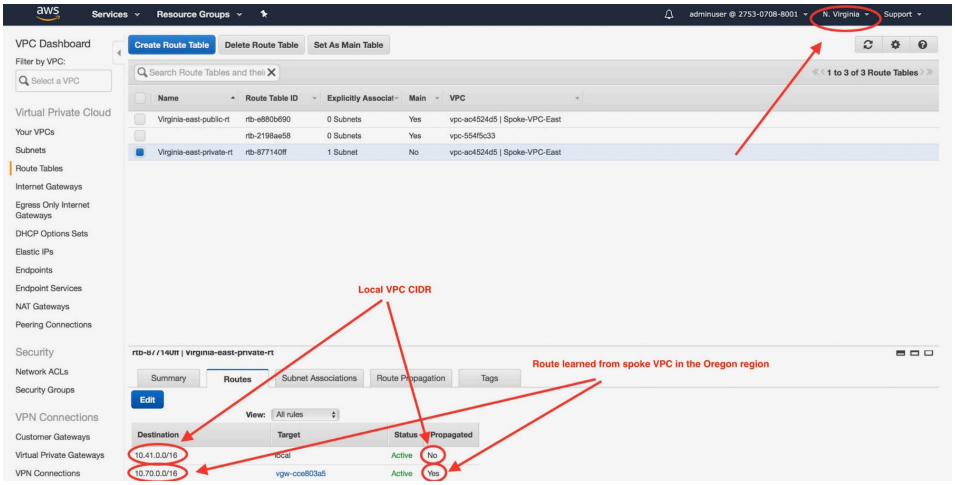

Figure 7.18 *Transit VPC Route Propagation*

Deploying Transit VPC from the Juniper vSRX Github

Although deploying Transit VPC from the AWS Marketplace is the recommended provisioning model, some users may need to customize the CloudFormation templates as well as the Lambda functions to satisfy their specific requirements. To help those users, Juniper has made all of its Transit VPC automation code open source and has published the code in the following github repo: https://github.com/Juniper/vSRX-AWS.

CloudFormation templates can be downloaded and modified. Then they can be uploaded to the AWS CloudFormation service similar to how it was described earlier in this recipe.

Conclusion

This recipe covered the Juniper Transit VPC architecture as well as its deployment walkthrough, and how spoke VPCs across AWS regions and accounts can connect via the Juniper Transit VPC with a zero touch, fully automated deployment. Finally, it created a Transit VPC using the Juniper AWS github templates.

Recipe 8: Multi-Region Full Mesh Transit VPC with vSRX

by Lionel Ruggeri and Tony Boerema

Junos OS Used: 15.1X49-D80

Juniper Platforms General Applicability: vMX, vSRX

There are several reasons why you might want to build a multi-region architecture in AWS:

- It improves overall performance by distributing content closer to end users.
- It builds high availability and disaster recovery solutions by leveraging multiple AWS *Availability Zones* and regions.
- And it meets local regulations and compliance requirements concerning certain types of data and services.

MORE? The AWS global infrastructure continues to grow by offering new capacity and access, new Availability Zones, and new data centers in different geographic regions. For the latest additions to the AWS global Infrastructure see: https://aws.amazon.com/about-aws/global-infrastructure/. And for AWS regional capabilities see: https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/.

Problem

Once the decision is made to invest, deploy, and run services across multiple AWS regions, you need a reliable and scalable virtual transport network to interconnect multiple VPCs that reside in different AWS regions without compromising security.

Solution

The solution is to use the Juniper components in AWS to:

- Build an AWS transit peering VPC to connect multiple, geographically diverse VPCs and remote networks.
- Build a secure IPsec tunnel to the vSRX entities in the transit VPC hub.
- Build an overlay BGP routing scheme to route the traffic between regions.
- Automate the tasks in order to build a fully-meshed Transit VPC solution and add/connect new regions to existing networks. As the number of nodes continues to increase, this become a serious scaling and operational problem that can find an answer in an Ansible Playbook framework.

Figure 8.1 illustrates the reference architecture used to illustrate the concept of building a multi-region full mesh transit VPC. A transit VPC is a central hub for traffic flowing between VPCs and/or a remote network. It is typically comprised of routing virtual nodes, logically connected with secure tunnels across AWS and running overlay routing protocols to exchange VPC leaf destination prefixes.

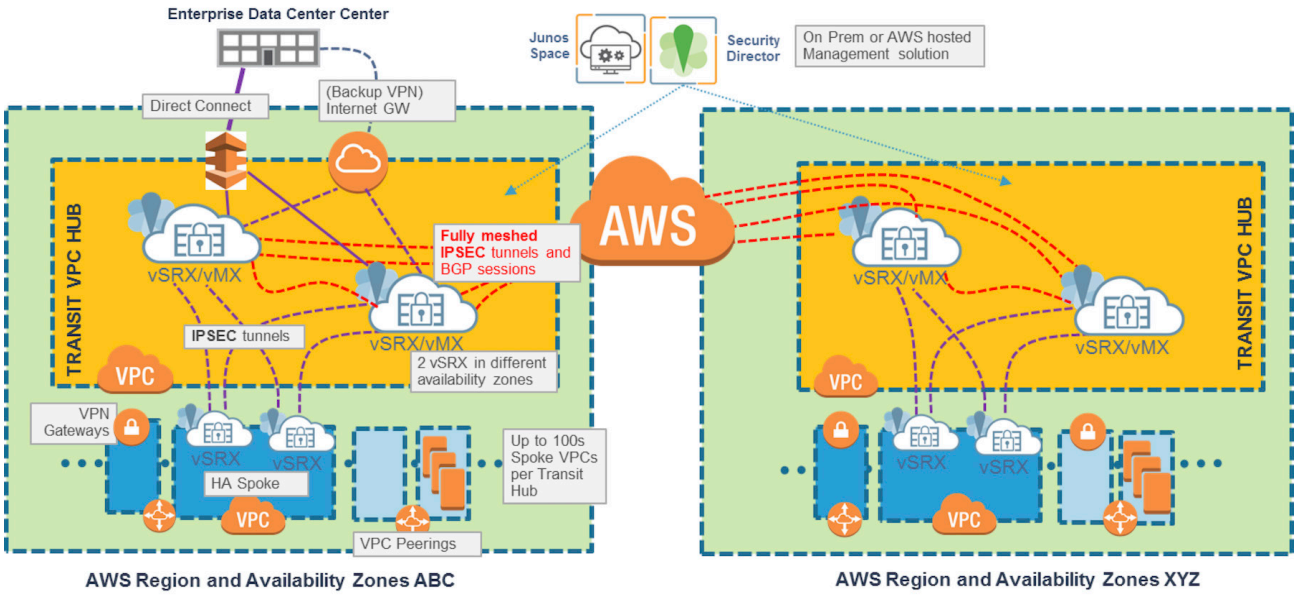

Figure 8.1 *Multi-region Transit VCP Reference Architecture*

The solution has two main parts:

- Issue Junos CLI commands, in jinja2 format, to build IPsec tunnels and BGP overlays for routing.

- Implement Ansible playbooks to automate the on-demand creation of fully meshed tunnels between the vSRX in the inventory (an example of a variable table is provided). As an example, Figure 8.2 illustrates a few of the tasks accomplished by the Ansible playbooks and roles:

NOTE The Ansible script was authored by Dusten Splan of Juniper's Center of Excellence, who graciously provided it for this recipe.

Figure 8.2 illustrates the topology created by the Ansible script and the low-level design convention defined in the playbook and variable tables.

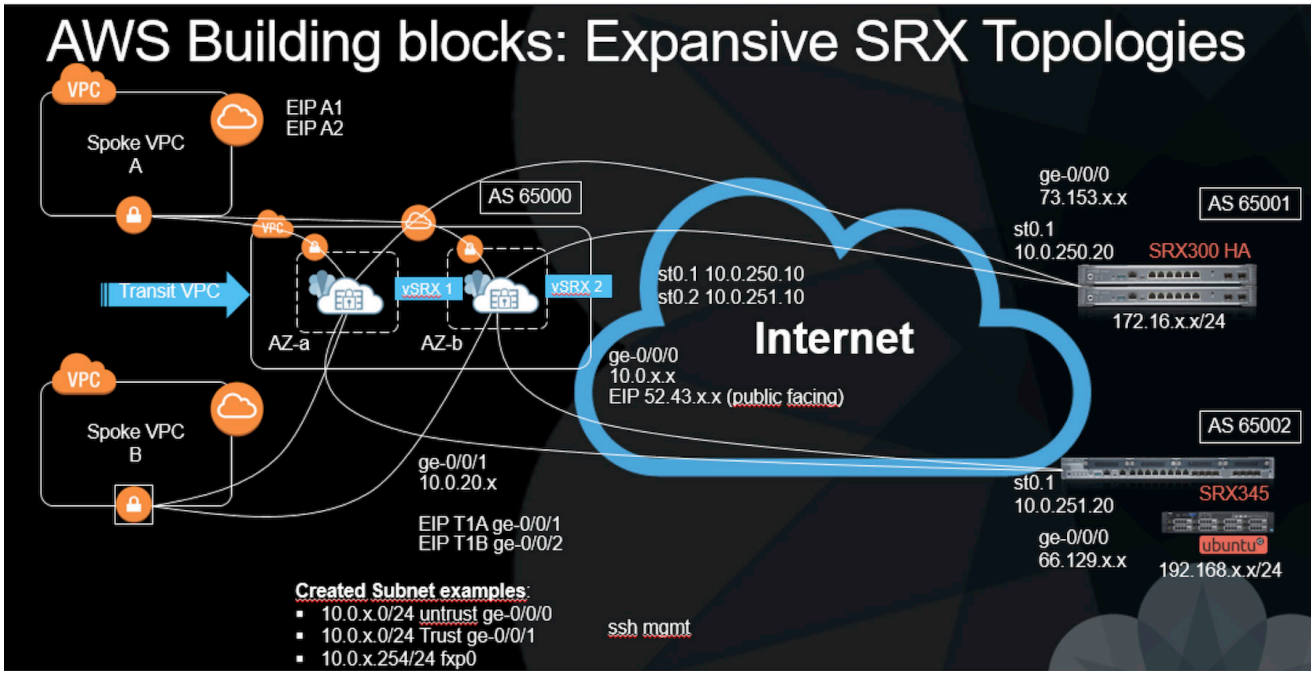

Figure 8.2 *Transit VCP Topology*

Prior to running the script, you need to ensure that the following prerequisites are met:

- You have set up your AWS account and you are getting API tokens.
- You created an IAM user and obtained a token with the right permissions.
- You have a license for BYOL vSRX in AWS.
- You have installed the software on a local machine to enable the automation.
- You have the latest Junos Ansible Galaxy modules(1.4.1/2.0) that support SSH keys.

VSRX Configuration j2 Templates

Use the following example for a route-based VPN configuration on the vSRX.

1. First, define a new unit number for every route-based tunnel under the virtual st0 tunnel interface. Each vSRX uses two IPsec tunnels for redundancy:

```
set interfaces st0.{{ st0_unit }} family inet address {{ st0_interface_1_ip }}/30
set interfaces  st0.{{ st0_unit }} family inet mtu 1436

set interfaces st0.{{ st0_unit_2 }} family inet address {{ st0_interface_2_ip }}/30
set interfaces st0.{{ st0_unit_2 }} family inet mtu 1436
```

2. Define an IKE proposal. All tunnels can use the same proposal, but it needs to be managed on either end:

```
set security ike proposal ike-prop-{{ vpn_id }}-1 authentication-method pre-shared-keys
set security ike proposal ike-prop-{{ vpn_id }}-1 authentication-algorithm sha1
set security ike proposal ike-prop-{{ vpn_id }}-1 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-{{ vpn_id }}-1 lifetime-seconds 28800
set security ike proposal ike-prop-{{ vpn_id }}-1 dh-group group2

set security ike proposal ike-prop-{{ vpn_id }}-2 authentication-method pre-shared-keys
set security ike proposal ike-prop-{{ vpn_id }}-2 authentication-algorithm sha1
set security ike proposal ike-prop-{{ vpn_id }}-2 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-{{ vpn_id }}-2 lifetime-seconds 28800
set security ike proposal ike-prop-{{ vpn_id }}-2 dh-group group2
```

3. Define an IKE policy that references the proposal and the authentication method:

```
set security ike policy ike-pol-{{ vpn_id }}-1 mode main
set security ike policy ike-pol-{{ vpn_id }}-1 proposals ike-prop-{{ vpn_id }}-1
set security ike policy ike-pol-{{ vpn_id }}-1 pre-shared-key ascii-text {{ pre_shared_key_1 }}

set security ike policy ike-pol-{{ vpn_id }}-2 mode main
set security ike policy ike-pol-{{ vpn_id }}-2 proposals ike-prop-{{ vpn_id }}-2
set security ike policy ike-pol-{{ vpn_id }}-2 pre-shared-key ascii-text {{ pre_shared_key_2 }}
```

4. Define the IKE gateway referencing the IKE policy:

```
set security ike gateway gw-{{ vpn_id }}-1 ike-policy ike-pol-{{ vpn_id }}-1
set security ike gateway gw-{{ vpn_id }}-1 external-interface {{ vSRX_interface }}
set security ike gateway gw-{{ vpn_id }}-1 address {{ aws_publicip_1 }}
set security ike gateway gw-{{ vpn_id }}-1 no-nat-traversal
set security ike gateway gw-{{ vpn_id }}-1 dead-peer-detection interval 10 threshold 3

set security ike gateway gw-{{ vpn_id }}-2 ike-policy ike-pol-{{ vpn_id }}-2
set security ike gateway gw-{{ vpn_id }}-2 external-interface {{ vSRX_interface }}
set security ike gateway gw-{{ vpn_id }}-2 address {{ aws_publicip_2 }}
set security ike gateway gw-{{ vpn_id }}-2 no-nat-traversal
set security ike gateway gw-{{ vpn_id }}-2 dead-peer-detection interval 10 threshold 3
```

5. Define the IPsec VPN proposal. Again, you can use the same proposal for all tunnels, but it needs to match on either end:

```
set security ipsec proposal ipsec-prop-{{ vpn_id }}-1 protocol esp
set security ipsec proposal ipsec-prop-{{ vpn_id }}-1 authentication-algorithm hmac-sha1-96
```

```
set security ipsec proposal ipsec-prop-{{ vpn_id }}-1 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-{{ vpn_id }}-1 lifetime-seconds 3600

set security ipsec proposal ipsec-prop-{{ vpn_id }}-2 protocol esp
set security ipsec proposal ipsec-prop-{{ vpn_id }}-2 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-{{ vpn_id }}-2 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-{{ vpn_id }}-2 lifetime-seconds 3600
```

6. Define the IPsec policy referencing the proposal:

```
set security ipsec policy ipsec-pol-{{ vpn_id }}-1 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-{{ vpn_id }}-1 proposals ipsec-prop-{{ vpn_id }}-1

set security ipsec policy ipsec-pol-{{ vpn_id }}-2 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-{{ vpn_id }}-2 proposals ipsec-prop-{{ vpn_id }}-2
```

7. Define the IPsec VPN rule, referencing the policy:

```
set security ipsec vpn {{ vpn_id }}-1 ike gateway gw-{{ vpn_id }}-1
set security ipsec vpn {{ vpn_id }}-1 ike ipsec-policy ipsec-pol-{{ vpn_id }}-1
set security ipsec vpn {{ vpn_id }}-1 df-bit clear
set security ipsec vpn {{ vpn_id }}-1 bind-interface st0.{{ st0_unit }}

set security ipsec vpn {{ vpn_id }}-2 ike gateway gw-{{ vpn_id }}-2
set security ipsec vpn {{ vpn_id }}-2 ike ipsec-policy ipsec-pol-{{ vpn_id }}-2
set security ipsec vpn {{ vpn_id }}-2 df-bit clear
set security ipsec vpn {{ vpn_id }}-2 bind-interface  st0.{{ st0_unit_2 }}
```

8. Attach units of st0 tunnel interface to security zones:

```
set security zones security-zone trust interfaces st0.{{ st0_unit }}
set security zones security-zone trust interfaces st0.{{ st0_unit_2 }}
```

9. Attach units of st0 tunnel interface to virtual router instance for best practice:

```
set routing-instances AWS-data interface st0.{{ st0_unit }}
set routing-instances AWS-data interface st0.{{ st0_unit_2 }}
```

10. Establish BGP sessions across the IPsec tunnels:

```
set routing-instances AWS-data protocols bgp group ebgp type external
set routing-instances AWS-data protocols bgp group ebgp as-override

set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_1_peer }} peer-as
7224
set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_1_peer }} hold-time
30
set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_1_peer }} local-as
{{ ansible_asn }}
set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_1_peer }} import
AWS-policy


set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_2_peer }} peer-as
7224
set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_2_peer }} hold-time
30
set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_2_peer }} local-as
```

```
{{ ansible_asn }}
set routing-instances AWS-data protocols bgp group ebgp neighbor {{ st0_interface_2_peer }} import
AWS-policy
```

11. Define the route into the tunnel interface to push traffic out of the public AWS interfaces:

```
set routing-instances AWS-data routing-options static route {{ aws_publicip_1 }}/32 next-hop {{ ge_gw
}}
set routing-instances AWS-data routing-options static route {{ aws_publicip_2 }}/32 next-hop {{ ge_gw
}}
```

Ansible Playbook and Variables to Build VPC and Turn Up IPsec

Run the `create_ipsec.yml` playbook.

Let's examine what each part of the playbook does.

1. First the playbook pushes NTP configuration snippets to the vSRXs to sync-up time with the AWS cloud:

```
- name: Junos Push NTP Config / Get Used ST0 interfaces
  hosts: vSRX
  connection: local
  roles:
    - Juniper.junos
  gather_facts: no
  tasks:
    - name: Create NTP config
      template: src=./roles/ipsec/templates/ntp_config.j2 dest=./output/{{ inventory_hostname }}_ntp_
config.set mode=600

    - name: Pushing NTP Sync
      junos_install_config:
        user:         "{{ ansible_ssh_user }}"
        host:         "{{ ansible_ssh_host }}"
        port:         "{{ ansible_ssh_port }}"
        ssh_private_key_file: "{{ inventory_dir }}/{{ ansible_ssh_private_key_file }}"
        file:        "./output/{{ inventory_hostname }}_ntp_config.set"
```

2. Then it connects to the vSRX host to determine which st0 interfaces are in use and exports the output in an XML file:

```
    - name: Get ST0 Interfaces In-Use
      junos_cli:
        user:         "{{ ansible_ssh_user }}"
        host:         "{{ ansible_ssh_host }}"
        port:         "{{ ansible_ssh_port }}"
        ssh_private_key_file: "{{ inventory_dir }}/{{ ansible_ssh_private_key_file }}"
        cli:          "show interfaces st0"
        format:       "xml"
        dest:         "{{ inventory_dir }}/output/{{ inventory_hostname }}_test.output"
```

3. Then it determines the next st0 interface unit that can be used to create the new tunnels:

```
  - name: Read interfaces output into the interfaces fact
    set_fact:
      interfaces:    "{{ lookup('file', '{{ inventory_dir }}/output/{{ inventory_hostname }}_test.
output') | replace('\n', '') }}"

  - name: Getting Unit ID
    xml:
      xmlstring:     "{{ interfaces }}"
      xpath: /interface-information/physical-interface/logical-interface/name
      content: text
    ignore_errors: yes
    register: unit

  - name: Getting st0 unit ID
    set_fact:
      st0_unit:      "{{ unit.matches|default([]) | map(attribute='name') | list }}"

  - name: Get Max Unit ID for ST0 interfaces
    set_fact:
      st0_unit:      "{{ st0_unit | max }}"
    when: st0_unit != []

  - set_fact:
      st0_unit:      "{{ st0_unit.split('.')[1]|int + 1  }}"
    when: st0_unit != []

  - set_fact:
      st0_unit:      "10"
    when: st0_unit == []
```

4. Now it determines which IP address are already in use:

```
  - name: Getting XML st0 Used IP addresses
    xml:
      xmlstring:     "{{ interfaces }}"
      xpath: /interface-information/physical-interface/logical-interface/address-family/interface-
address/ifa-local
      content: text
    ignore_errors: yes
    register: st0_addresses

  - name: Getting Used st0 ip addresses
    set_fact:
      st0_ip:       "{{ st0_addresses.matches|default([]) | map(attribute='ifa-local') | list }}"

  - name: Dump IP List
    debug:
      var=st0_ip

  - name: Setting Empty vpn_id
    set_fact:
      vpn_id:       ""
```

5. The section is calling a role `create_vpn.yml` **in** order to get the VPC setting information from AWS, create VPC VPGs, and then attach a VPN tunnel from the AWS spoke VPC to the vSRX transit (the .yml file is provided in the next recipe section – but realize it's outside of the transit scope):

```
   - name: Calling create VPN for "{{ vpc_name_A }}"
     include: ./roles/ipsec/tasks/create_vpn.yml name={{ vpc_name_A }} cgw_name={{ inventory_hostname
}}_{{ vpc_interface_A }} vsrx_name={{ inventory_hostname }} tun_interface={{ vpc_interface_A }} az={{
ansible_az }} vpn_id={{ vpn_id }}
     until: st0_interface_1_ip not in st0_ip and st0_interface_2_ip not in st0_ip

   - name: Setting vSRX_interface
     set_fact:
       vSRX_interface: "ge-0/0/0.0"
     when: vpc_interface_A == "ge-000"

   - name: Setting vSRX_interface
     set_fact:
       vSRX_interface: "ge-0/0/1.0"
     when: vpc_interface_A == "ge-001"

   - name: Setting vSRX_interface
     set_fact:
       vSRX_interface: "ge-0/0/2.0"
     when: vpc_interface_A == "ge-002"

   - name: Set st0_unit_2
     set_fact:
       st0_unit_2:    "{{ st0_unit|int + 1 }}"
```

6. The script checks the output information message about the settings – vSRX interfaces, static routes, and public next hops to force traffic out of interfaces:

```
   - name: AWS VPN Connection info
     debug:
       msg:
         - "ST Unit - {{ st0_unit }}"
         - "ST Unit - {{ st0_unit_2 }}"
         - "vSRX Interface - {{ vSRX_interface }}"
         - "Static Route - {{ ge_gw }}"
         - "VPN ID - {{ vpn_id }}"
         - "Peer 1 - {{ st0_interface_1_peer }} {{ st0_interface_1_ip }} PSK 1 - {{ pre_shared_key_1 }}
AWS PubIP - {{ aws_publicip_1 }}"
         - "Peer 2 - {{ st0_interface_2_peer }} {{ st0_interface_2_ip }} PSK 1 - {{ pre_shared_key_2 }}
AWS PubIP - {{ aws_publicip_2 }}"
```

7. Now, the script generates the vSRX configuration-based j2 template provided earlier:

```
- name: Merge IpSec Template with Vars
     template: src=./roles/ipsec/templates/ipsec_template.j2 dest=./output/{{ inventory_hostname
}}_{{ vpc_name_A }}_ipsec.set mode=600
```

8. Then pushes the resulting configurations to the vSRX:

```
  - name: Install IpSec Config
    junos_install_config:
      user:         "{{ ansible_ssh_user }}"
      host:         "{{ ansible_ssh_host }}"
      port:         "{{ ansible_ssh_port }}"
      ssh_private_key_file: "{{ ansible_ssh_private_key_file }}"
      file:         "./output/{{ inventory_hostname }}_{{ vpc_name_A }}_ipsec.set"
```

9. And repeats the process for the B-side or backup tunnel:

```
- name: Get ST0 Interfaces In-Use
    junos_cli:
      user:         "{{ ansible_ssh_user }}"
      host:         "{{ ansible_ssh_host }}"
      port:         "{{ ansible_ssh_port }}"
      ssh_private_key_file: "{{ inventory_dir }}/{{ ansible_ssh_private_key_file }}"
      cli:          "show interfaces st0"
      format:       "xml"
      dest:         "{{ inventory_dir }}/output/{{ inventory_hostname }}_test.output"

  - name: Read interfaces output into the interfaces fact
    set_fact:
      interfaces:    "{{ lookup('file', '{{ inventory_dir }}/output/{{ inventory_hostname }}_test.
output') | replace('\n', '') }}"

  - name: Getting Unit ID
    xml:
      xmlstring:     "{{ interfaces }}"
      xpath: /interface-information/physical-interface/logical-interface/name
      content: text
    ignore_errors: yes
    register: unit

  - name: Getting st0 unit ID
    set_fact:
      st0_unit:      "{{ unit.matches|default([]) | map(attribute='name') | list }}"

  - name: Get Max Unit ID for ST0 interfaces
    set_fact:
      st0_unit:      "{{ st0_unit | max }}"
    when: st0_unit != []

  - set_fact:
      st0_unit:      "{{ st0_unit.split('.')[1]|int + 1  }}"
    when: st0_unit != []

  - set_fact:
      st0_unit:      "10"
    when: st0_unit == []
  - name: Getting st0 Used IP addresses
    xml:
      xmlstring:     "{{ interfaces }}"
      xpath: /interface-information/physical-interface/logical-interface/address-family/interface-
address/ifa-local
      content: text
```

```
    ignore_errors: yes
    register: st0_addresses

  - name: Getting Used st0 ip addresses
    set_fact:
      st0_ip:       "{{ st0_addresses.matches|default([]) | map(attribute='ifa-local') | list }}"

  - name: Dump IP List
    debug:
      var=st0_ip
  - name: Setting Empty vpn_id
    set_fact:
      vpn_id:       ""

  - name: Calling create VPN for "{{ vpc_name_B }}"
    include: ./roles/ipsec/tasks/create_vpn.yml name={{ vpc_name_B }} cgw_name={{ inventory_hostname
}}_{{ vpc_interface_B }} vsrx_name={{ inventory_hostname }} tun_interface={{ vpc_interface_B }} az={{
ansible_az }} vpn_id={{ vpn_id }}
    until: st0_interface_1_ip not in st0_ip and st0_interface_2_ip not in st0_ip

  - name: Setting vSRX_interface
    set_fact:
      vSRX_interface: "ge-0/0/0.0"
    when: vpc_interface_B == "ge-000"

  - name: Setting vSRX_interface
    set_fact:
      vSRX_interface: "ge-0/0/1.0"
    when: vpc_interface_B == "ge-001"

  - name: Setting vSRX_interface
    set_fact:
      vSRX_interface: "ge-0/0/2.0"
    when: vpc_interface_B == "ge-002"

  - name: Set st0_unit_2
    set_fact:
      st0_unit_2:    "{{ st0_unit|int + 1 }}"

  - name: AWS VPN Connection info
    debug:
      msg:
        - "ST Unit - {{ st0_unit }}"
        - "ST Unit - {{ st0_unit_2 }}"
        - "vSRX Interface - {{ vSRX_interface }}"
        - "Static Route - {{ ge_gw }}"
        - "VPN ID - {{ vpn_id }}"
        - "Peer 1 - {{ st0_interface_1_peer }} {{ st0_interface_1_ip }} PSK 1 - {{ pre_shared_key_1 }}
AWS PubIP - {{ aws_publicip_1 }}"
        - "Peer 2 - {{ st0_interface_2_peer }} {{ st0_interface_2_ip }} PSK 1 - {{ pre_shared_key_2 }}
AWS PubIP - {{ aws_publicip_2 }}"

  - name: Merge IpSec Template with Vars
    template: src=./roles/ipsec/templates/ipsec_template.j2 dest=./output/{{ inventory_hostname
}}_{{ vpc_name_B }}_ipsec.set mode=600

  - name: Install IpSec Config
    junos_install_config:
```

```
      user:         "{{ ansible_ssh_user }}"
      host:         "{{ ansible_ssh_host }}"
      port:         "{{ ansible_ssh_port }}"
      ssh_private_key_file: "{{ ansible_ssh_private_key_file }}"
      file:         "./output/{{ inventory_hostname }}_{{ vpc_name_B }}_ipsec.set"
```

And here is *vars.yml*, the variable setting file to use while running the *create_ipsec.yml* playbook:

```
---

aws_access_key: ""
aws_secret_key: ""

#SSH keys AWS
vpn_keyname:          "jnpr_aws"

# Owner Name
owner:               "dsplan"

# Special
used_for:            "development"

# AWS Environment Tag
env:                "transit-vpc-demo"

# Default AWS Region
aws_region:          "us-east-1"

# VPC A Information
vpc_name_A:          "VPC_A"
vpc_cidr_A:          "10.1.0.0/16"
vpc_subnet_A:
  - { az:            "a",
      subnet:        "10.1.1.0/24",
      interface:      "generic" }
  - { az:            "b",
      subnet:        "10.1.2.0/24",
      interface:      "generic" }
#  - { az:            "c",
#      subnet:        "10.1.3.0/24",
#      interface:      "generic" }
#  - { az:            "d",
#      subnet:        "10.1.4.0/24",
#      interface:      "generic" }
#  - { az:            "e",
#      subnet:        "10.1.5.0/24",
#      interface:      "generic" }
vpc_interface_A:      "ge-001"

# VPC B Information
vpc_name_B:          "VPC_B"
vpc_cidr_B:          "10.10.0.0/16"
vpc_subnet_B:
  - { az:            "a",
      subnet:        "10.10.1.0/24",
      interface:      "generic" }
```

```
  - { az:            "b",
      subnet:        "10.10.2.0/24",
      interface:     "generic" }
#   - { az:          "c",
#       subnet:      "10.10.3.0/24",
#       interface:   "generic" }
#   - { az:          "d",
#       subnet:      "10.10.4.0/24",
#       interface:   "generic" }
#   - { az:          "e",
#       subnet:      "10.10.5.0/24",
#       interface:   "generic" }
vpc_interface_B:     "ge-002"

# VPC Transit Information
vpc_name_T:          "VPC_T"
vpc_cidr_T:          "10.100.0.0/16"
vpc_subnet_T:
  - { az:            "a",
      subnet:        "10.100.10.0/28",
      interface:     "fxp0" }
  - { az:            "a",
      subnet:        "10.100.10.16/28",
      interface:     "ge-000" }
  - { az:            "a",
      subnet:        "10.100.10.32/28",
      interface:     "ge-001" }
  - { az:            "a",
      subnet:        "10.100.10.48/28",
      interface:     "ge-002" }
  - { az:            "a",
      subnet:        "10.100.10.64/28",
      interface:     "ge-003" }
  - { az:            "a",
      subnet:        "10.100.10.80/28",
      interface:     "ge-004" }
  - { az:            "a",
      subnet:        "10.100.10.96/28",
      interface:     "ge-005" }
  - { az:            "a",
      subnet:        "10.100.10.112/28",
      interface:     "ge-006" }
  - { az:            "b",
      subnet:        "10.100.20.0/28",
      interface:     "fxp0" }
  - { az:            "b",
      subnet:        "10.100.20.16/28",
      interface:     "ge-000" }
  - { az:            "b",
      subnet:        "10.100.20.32/28",
      interface:     "ge-001" }
  - { az:            "b",
      subnet:        "10.100.20.48/28",
      interface:     "ge-002" }
  - { az:            "b",
      subnet:        "10.100.20.64/28",
      interface:     "ge-003" }
  - { az:            "b",
```

```
    subnet:         "10.100.20.80/28",
    interface:      "ge-004" }
  - { az:           "b",
    subnet:         "10.100.20.96/28",
    interface:      "ge-005" }
  - { az:           "b",
    subnet:         "10.100.20.112/28",
    interface:      "ge-006" }
  - { az:           "a",
    subnet:         "10.100.1.0/24",
    interface:      "generic" }
  - { az:           "b",
    subnet:         "10.100.2.0/24",
    interface:      "generic" }
#  - { az:          "c",
#     subnet:       "10.100.3.0/24",
#     interface:    "generic" }
#  - { az:          "d",
#     subnet:       "10.100.4.0/24",
#     interface:    "generic" }
#  - { az:          "e",
#     subnet:       "10.100.5.0/24",
#     interface:    "generic" }


# VPC A HOST
host_name_A:        "Testing_Host_VPC_A"
host_name_A_az:     "a"

# VPC B HOST
host_name_B:        "Testing_Host_VPC_B"
host_name_B_az:     "a"

# vSRX 1
vsrx_1:             "vSRX_A"
vsrx_1_asn:         "65010"
vsrx_1_az:          "a"

# vSRX 2
vsrx_2:             "vSRX_B"
vsrx_2_asn:         "65020"
vsrx_2_az:          "b"

# Which Host AMI do we want to use?
host_ami_base:      "amzn-ami-hvm-*"
host_ami_owner:     "amazon"
host_ami_arch:      "x86_64"

# Which vSRX AMI do we want to use? This is BYOL
vsrx_ami_base:      "media-vsrx-vmdisk-15.1X49-D80.4-4*"
vsrx_ami_owner:     "juniper"
vsrx_ami_arch:      "x86_64"

# Instance Type  #4xlarger - 8 interfaces
vsrx_ec2_type:      "c4.xlarge"  # 4 interfaces
host_ec2_type:      "t2.nano"

# Instance Default Volume Config
```

```
volumes:
  - device_name: /dev/xvda
    device_type: gp2
    volume_size: 20
    delete_on_termination: true
volumes_srx:
  - device_name: /dev/sda1
    device_type: gp2
    volume_size: 20
    delete_on_termination: true

useraccesskeys: “access_key” #for {{ new_username }}

#Cleanup script
instanceids: “i-*”
```

Discussion

MORE? If you need either a refresher or a tutorial on using Ansible and Junos, see the excellent *Day One* books by Sean Sawtell, *Day One: Automating Junos with Ansible* at: https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/automating-junos-ansible/.

This recipe can help you build a secure and reliable multi-region full mesh Transit VPC architecture based on Juniper Networks vSRX and vMX. But there are several key design challenges you must consider before running applications or services at global scale across AWS:

- Hosting decisions that reach into multiple geographic regions of upper-layer key components such as database, application backends and frontends, and data caching.
- Latency-based routing and health checks to achieve an active-active setup that can fail over between regions in case of an issue.
- Database synchronization, throughput capabilities, costs, etc.

The create_vpn.yml Role

```
- name: Dump IP List
  debug:
    var=vpn_id

- name: Setup AWS CLI (1/3)
  shell: >
    aws configure set aws_access_key_id “{{ aws_access_key }}”
```

```
- name: Setup AWS CLI (2/3)
  shell: >
    aws configure set aws_secret_access_key “{{ aws_secret_key }}”

- name: Setup AWS CLI (3/3)
  shell: >
    aws configure set region {{ aws_region }}

- name: Get Customer Gateway
  shell:
    aws ec2 describe-customer-gateways --filter “Name=state,Values=available”
  register: cgw_id_raw

- set_fact:
    cgw_id_raw:    “{{ cgw_id_raw.stdout | from_json }}”

#- name: Dumping cgw
#  debug:
#    var=cgw_id_raw

- name: Getting VGW for {{ name }}_vgw
  ec2_vpc_vgw_facts:
    region:           “{{ aws_region }}”
    aws_access_key:    “{{ aws_access_key }}”
    aws_secret_key:    “{{ aws_secret_key }}”
    filters:
      “tag:Environment”: “{{ env }}”
      “tag:Name”: “{{ name }}_vgw”
  register: vpc_vgw

- name: Set Fact VGW
  set_fact:
    vgw_id:           “{{ vpc_vgw.virtual_gateways[0].vpn_gateway_id }}”

- set_fact:
    vsrx_cgw_id:      “{{ item.0.CustomerGatewayId }}”
    ignore_errors: yes
  with_subelements:
    - “{{ cgw_id_raw.CustomerGateways }}”
    - Tags
    - flags:
  when:
    - item.1.Key == “Name”
    - item.1.Value == cgw_name

- set_fact:
    ge_gw:            “{{ item.subnet|ipaddr(1)|ipaddr(‘address’) }}”
  with_items:
    - “{{ vpc_subnet_T }}”
  when:
    - item.az == az
    - item.interface == tun_interface

- name: Delete VPN Connection if vpn_id is Set
  shell:
    aws ec2 delete-vpn-connection --vpn-connection-id {{ vpn_id }}
  when:
```

```
    - vpn_id != ""

- name: Creating VPN Connection
  shell:
    aws ec2 create-vpn-connection --type ipsec.1 \
    --customer-gateway-id {{ vsrx_cgw_id }} \
    --vpn-gateway-id {{ vgw_id }}
  register: vpn_connection

- name: Set Facts
  set_fact:
    temp:            "{{ vpn_connection.stdout | from_json }}"

- name: Setting Fact for vpn_id
  set_fact:
    vpn_id:          "{{ temp.VpnConnection.VpnConnectionId }}"

- name: Configure Tags on VPN Connection
  ec2_tag:
    region:          "{{ aws_region }}"
    aws_access_key:   "{{ aws_access_key }}"
    aws_secret_key:   "{{ aws_secret_key }}"
    resource:        "{{ vpn_id }}"
    state:           present
    tags:
      Name:          "{{ vsrx_name }}-{{ name }}"
      Environment:    "{{ env }}"
      Use:           "{{ used_for }}"
      Owner:         "{{ owner }}"

- name: Getting from XML pre_shared_key
  xml:
    xmlstring: "{{ temp.VpnConnection.CustomerGatewayConfiguration }}"
    xpath: /vpn_connection/ipsec_tunnel/ike/pre_shared_key
    content: text
  register: Output

- name: Set Fact for pre_shared_key_1
  set_fact:
    pre_shared_key_1:  "{{ Output.matches[0].pre_shared_key }}"

- name: Set Fact for pre_shared_key_2
  set_fact:
    pre_shared_key_2:  "{{ Output.matches[1].pre_shared_key }}"

- name: Getting from XML tunnel_inside_address s0.x side
  xml:
    xmlstring: "{{ temp.VpnConnection.CustomerGatewayConfiguration }}"
    xpath: /vpn_connection/ipsec_tunnel/customer_gateway/tunnel_inside_address/ip_address
    content: text
  register: Output

- name: Set Fact for st0.x addresses
  set_fact:
    st0_interface_1_ip: "{{ Output.matches[0].ip_address }}"

- name: Set Fact for st0.x addresses
  set_fact:
```

```
    st0_interface_2_ip: “{{ Output.matches[1].ip_address }}”

- name: Trying xml
  xml:
    xmlstring: “{{ temp.VpnConnection.CustomerGatewayConfiguration }}”
    xpath: /vpn_connection/ipsec_tunnel/vpn_gateway/tunnel_outside_address/ip_address
    content: text
  register: Output

- name: Set Fact for aws_publicip_1
  set_fact:
    aws_publicip_1:    “{{ Output.matches[0].ip_address }}”

- name: Set Fact for aws_publicip_2
  set_fact:
    aws_publicip_2:    “{{ Output.matches[1].ip_address }}”

- name: Trying xml
  xml:
    xmlstring: “{{ temp.VpnConnection.CustomerGatewayConfiguration }}”
    xpath: /vpn_connection/ipsec_tunnel/vpn_gateway/tunnel_inside_address/ip_address
    content: text
  register: Output

- name: Set Fact for st0_interface_1_peer
  set_fact:
    st0_interface_1_peer:   “{{ Output.matches[0].ip_address }}”

- name: Set Fact for st0_interface_2_peer
  set_fact:
    st0_interface_2_peer:   “{{ Output.matches[1].ip_address }}”
```

Recipe 9: Adaptive Security with Juniper SDSN

by Ali Bidabadi

Junos Space version used: 17.2r1 and later

Juniper Security Director version used: 17.2r2 and later

Policy Enforcer version used: 17.2r2 and later

This recipe begins with an introduction to Juniper SDSN and its integration with the Policy Enforcer AWS Connector (or connector). It then describes how AWS workloads can be protected adaptively and how infected hosts can be remediated, using the metadata-based policies and threat remediation capabilities of the Policy Enforcer AWS connector.

Problem

How can organizations extend their security posture to their AWS workloads?

Solution

The solution is to deploy Juniper Software Defined Secure Network (SDSN) and the Policy Enforcer AWS Connector. This recipe will guide you in that deployment but first let's briefly review Juniper SDSN.

SDSN Overview

Juniper Networks Software-Defined Secure Network (SDSN) provides end-to-end network visibility allowing enterprises to secure their entire network, both physical and virtual, using threat detection and policy enforcement. SDSN solutions can automate and centrally manage security in a multi-vendor and multi-cloud environment.

The SDSN solution is comprised of:

- *A threat detection engine*: Cloud-based Sky ATP detects known and unknown malware. Known threats are detected using feed information from a variety of sources, including command control server and GeoIP. Unknown threats are identified using various methods such as sandboxing, machine learning, and threat deception.
- *Centralized policy management*: Junos Space Security Director, which also manages SRX Series devices, provides the management interface for the SDSN solution called Policy Enforcer. Policy Enforcer communicates with Juniper Networks devices and third-party devices across the network, globally enforcing security policies and consolidating threat intelligence from different sources. With monitoring capabilities, it can also act as a sensor, providing visibility for intra- and inter-network communications.
- *Expansive policy enforcement*: In a multi-vendor enterprise, SDSN enforces security across Juniper Networks devices, cloud-based solutions, and third-party devices. By communicating with all enforcement points, SDNS can quickly block or quarantine threats, preventing the spread of bilateral attacks within the network.
- *User intent-based policies:* Create policies according to logical business structures such as users, user groups, geographical locations, sites, tenants, applications, or threat risks. This allows network devices (switches, routers, firewalls, and other security devices) to share information, resources, and when threats are detected, remediation actions within the network.

Adaptive Security in Multi-Cloud Environment

In traditional networks security administrators are responsible for configuration and operation of all security devices. Moving from designing the firewall rules to provisioning them to the security devices can take weeks and sometimes months to fully complete. In a multi-cloud environment where workloads and services can be launched in a matter of minutes, the weeks-long cycle to provision security rules is simply unacceptable. In other words, security administrators cannot become the bottleneck in the application launch phase.

Juniper SDSN's metadata-based policy model allows for pre-created security policies to be dynamically enforced for *all* existing and new applications as they are launched in the cloud. Figure 9.1 depicts a common three-tier application deployment in an enterprise multi-cloud environment.

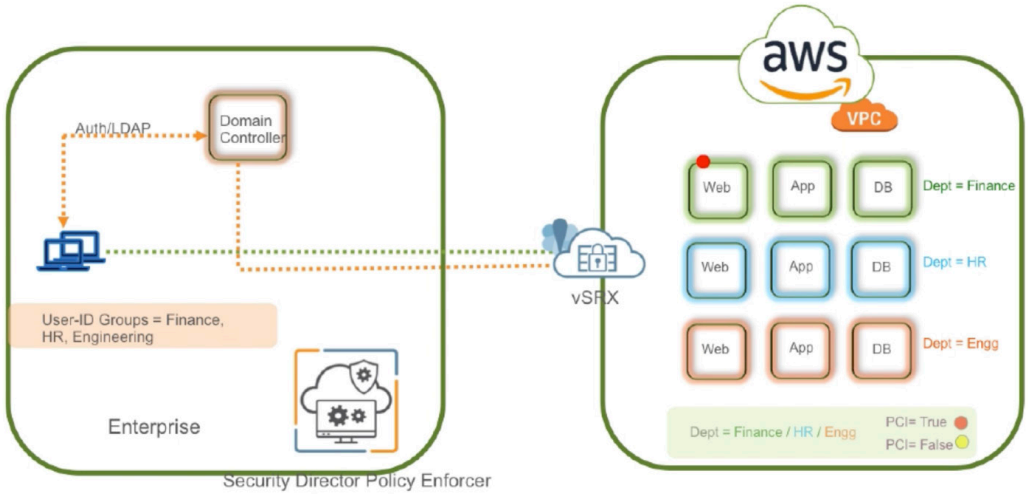

Figure 9.1 *Metadata-based Multi-cloud Deployment*

You can see that this deployment has AWS workloads that have been applied to certain metadata tags. These tags help associate workloads with different policies. For example, in the deployment shown in Figure 9.1, the "Dept" tag has been used to identify workloads that belong to the HR, Engineering, and Finance departments. The Security Director Policy Enforcer enforces the user-intent based policies by pushing the corresponding firewall rules to the vSRX. Juniper SDSN Dynamic Address Group ensures that when new workloads are added to the user's cloud deployment, the necessary changes are instantly and automatically applied to the perimeter firewall.

Figure 9.1 also shows that user-ID groups have been defined and sent to the firewall that will indicate the mapping between the users and their respective organizations/departments.

AWS PE Connector: A Step-by-Step Guide

Okay, those are the benefits of Juniper SDSN and the fundamentals of metadata-based policies. Now let's deploy a Policy Enforcer AWS Connector to see how security policies in action can be enforced on an AWS deployment.

IMPORTANT In addition to having an AWS account, in order to be able to follow the steps and install the Policy Enforcer AWS connector, you will need the following software installed:

- Junos Space 17.2r1 and later releases
- Juniper Security Director 17.2r2 and later releases

- Policy Enforcer 17.2r2 and later releases
- vSRX in AWS VPC

First, let's look at a sample AWS deployment in a VPC in the US-West-2 region (Oregon) that is comprised of a vSRX Next Generation Firewall (NGFW) and some workloads. Figure 9.2, shows the deployment with three workloads: one web server, one app server, and one database server.

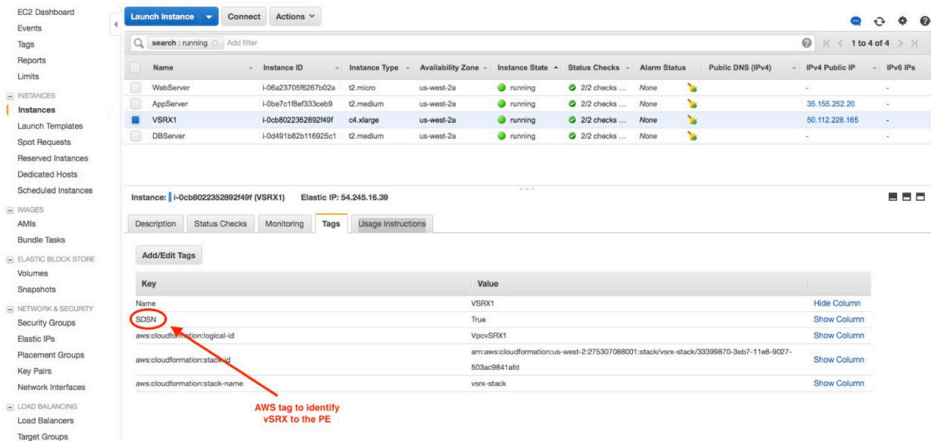

Figure 9.2 *Workloads in Sample AWS Deployment*

You can see in Figure 9.2 that the vSRX has a tag called "SDSN," which will be used to identify the firewall in the Policy Enforcer. Note that each workload can have multiple tags (metadata). For example, in our deployment, the web server has "PCI=False", and "Department=Finance" key/value tags. This is shown in Figure 9.3.

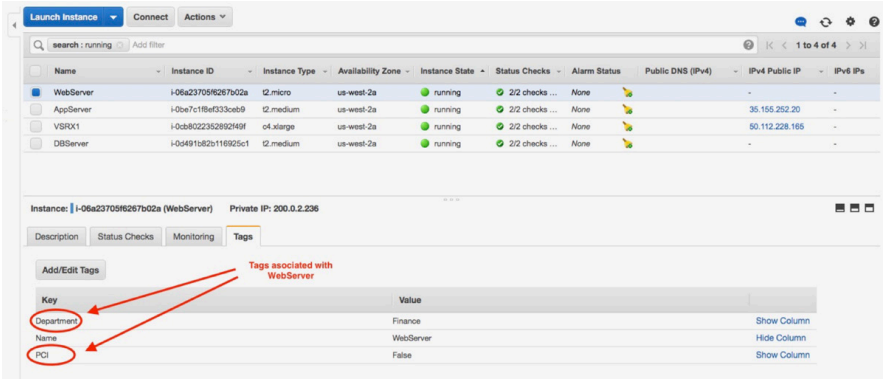

Figure 9. 3 *AWS Tags (Metadata)*

Okay, let's jump right in. The following steps provide complete provisioning of the Policy Enforcer AWS connector:

First, from the Junos Space Security Director homepage, navigate to this path: Administration > Policy Enforcer > Settings. Verify that your PE is integrated and managed by your Security Director as shown in Figure 9.4.

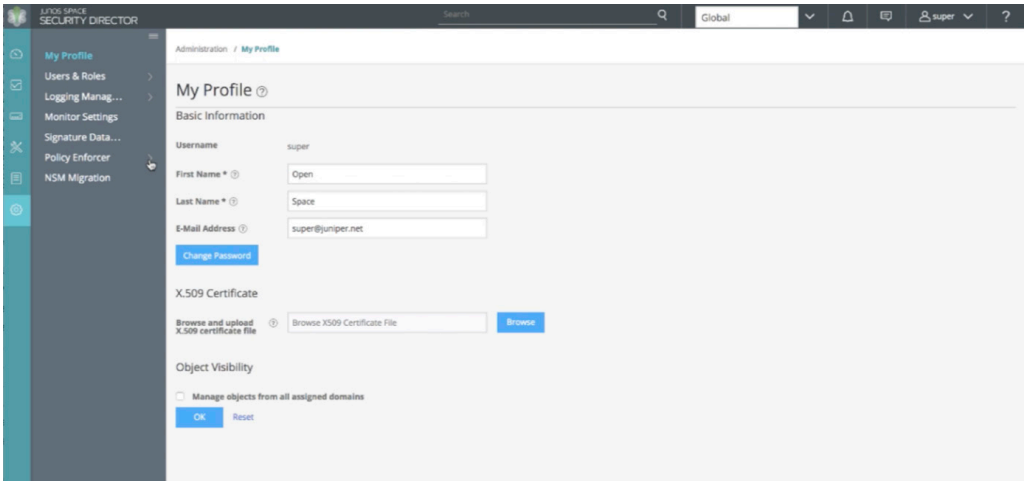

Figure 9.4 *Policy Enforcer Settings Page*

Click on "Policy Enforcer" in the sidebar and click on "Connectors". Here you can add one or more Policy Enforcer connectors by clicking at "+" sign located at the top left-hand corner of the page. A new process with three steps will open where you choose and define the connector type. Make sure "Amazon Web Services" is selected as shown in Figure 9.5.

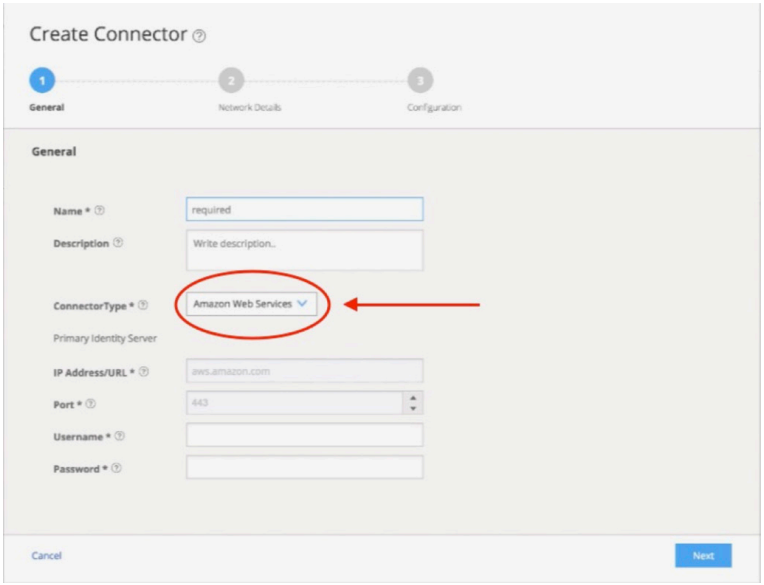

Figure 9.5 *Policy Enforcer Connector Type*

The other information that needs to be entered in Figure 9.5 is:

- Name: Choose a name for your connector. This is mandatory.
- Description: Optionally describe your AWS connector.
- Username: Enter your AWS access key ID here.
- Password: AWS secret access key has to be entered in this field.

Click Next to be directed to step 2 of the Create Connector process, the "Networks Details" step.

Select the region in which your deployment is located. Also select all the VPCs that you would like to adaptively protect as shown in Figure 9.6.

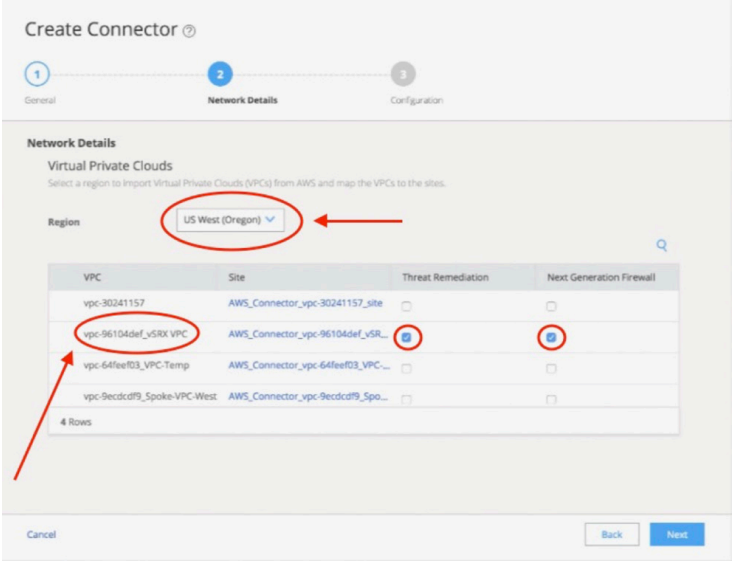

Figure 9.6 *AWS Connector Networks Details Page*

Note that for each VPC, two options are available: "Threat Remediation" and "Next Generation Firewall." The latter addresses the meta-data based policy portion. The threat remediation option should be selected only if you would like your infected hosts to be quarantined. We will discuss this later in this recipe.

Click Next to be taken to Step 3 of the process, the Configuration page. On the top part of this page, you can choose all the tags (metadata) that need to be imported for later use. In this example, both "Department" and "PCI" tags are selected as shown in Figure 9.7.

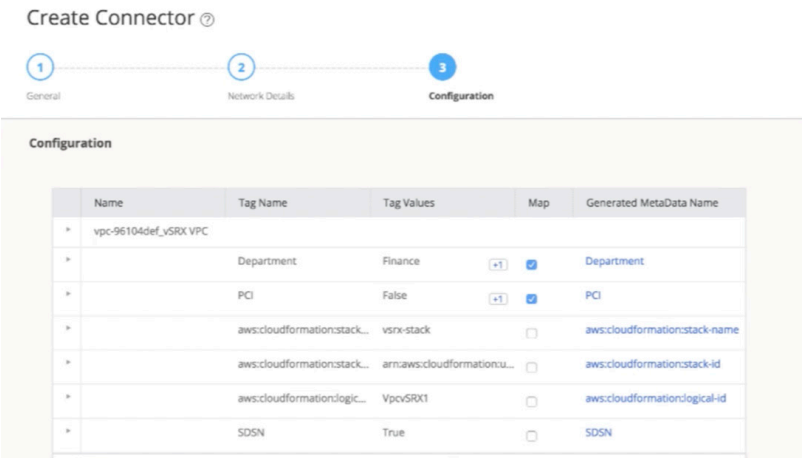

Figure 9.7 *Tags Imported to Policy Enforcer AWS Connector*

It's time to specify the details of your AWS connector in the bottom portion of the Configuration page as shown in Figure 9.8.

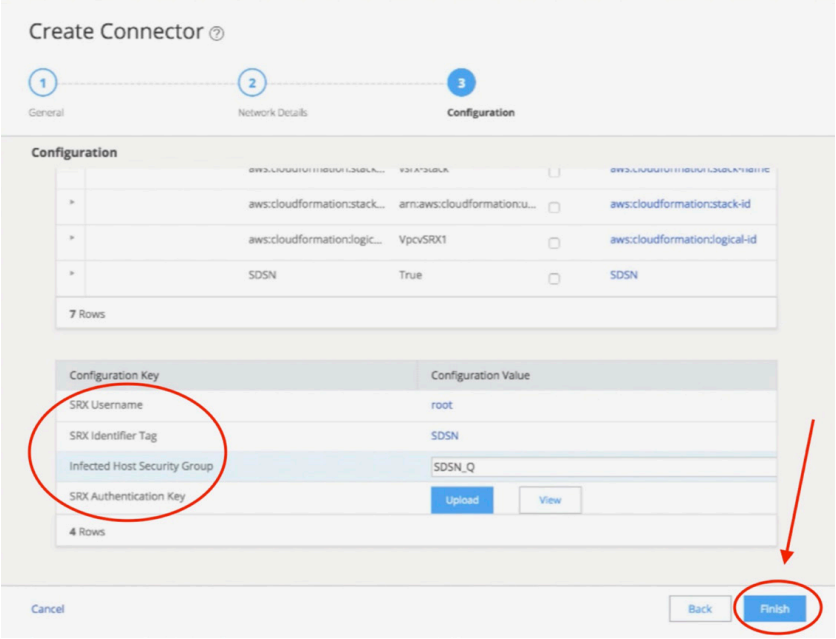

Figure 9.8 *Policy Enforcer AWS Connector Configuration Parameters*

Here is a quick explanation of what's needed:

- *SRX Username*: Username of the vSRX deployed in the target VPC.
- *SRX Identifier Tag*: AWS tag identifying the vSRX. This tag must be associated to your vSRX VM deployed in the target VPC.

- *Infected Host Security Group*: If threat remediation is required, you must ensure that a security group in the same AWS account/region is created. All infected hosts will be added to that security group. Specify the name of the security group here.

SRX Authentication Key: Upload SSH private key of your SSH key pair. The Policy Enforcer AWS Connector will use this information along with the provided user name to connect to the vSRX.

Click Finish to create your Policy Enforcer AWS connector. Figure 9.9 shows a list of your Policy Enforcer AWS Connectors including the AWS connector that was just created.

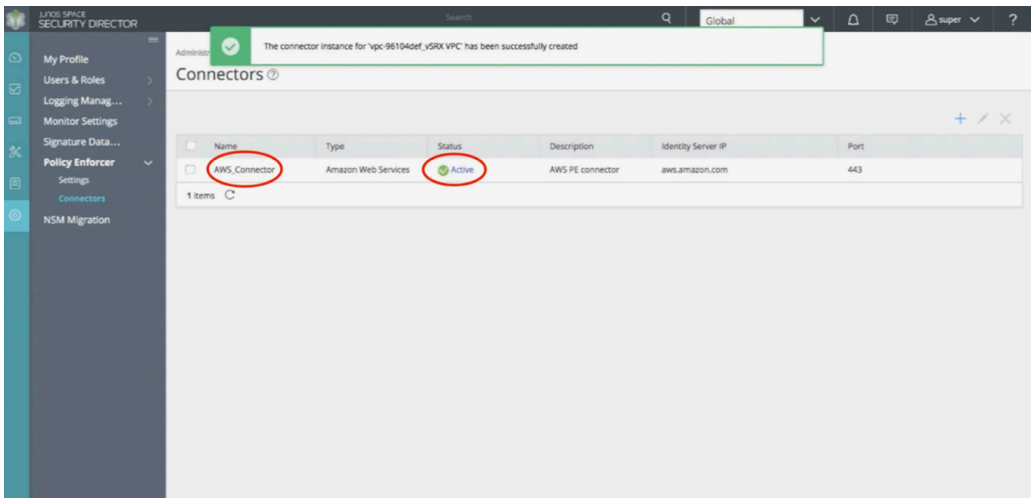

Figure 9. 9 *List of Policy Enforcer AWS Connectors*

Now let's go back to the Security Devices home page and from the left sidebar menu navigate to Devices > Security Devices. And verify that your newly added vSRX device has been successfully added to the list of security devices as shown in Figure 9.10.

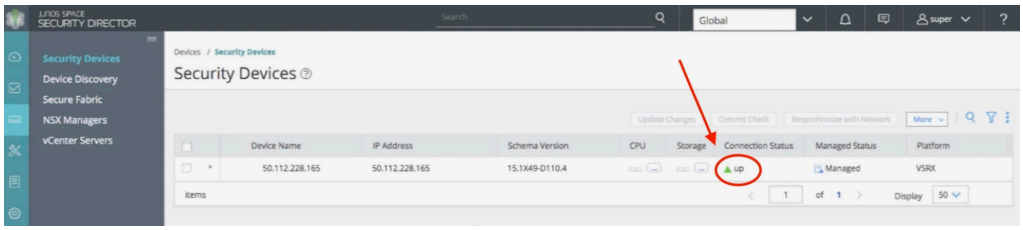

Figure 9.10 *List of Junos Space Security Director-Managed Security Devices*

In order to import policies already programmed in the vSRX device, right-click at the device row level and choose "Import Device" from a popup menu . Now you can choose any policies that you would like to be imported.

Figure 9.11 shows an example summary of the imported policies.

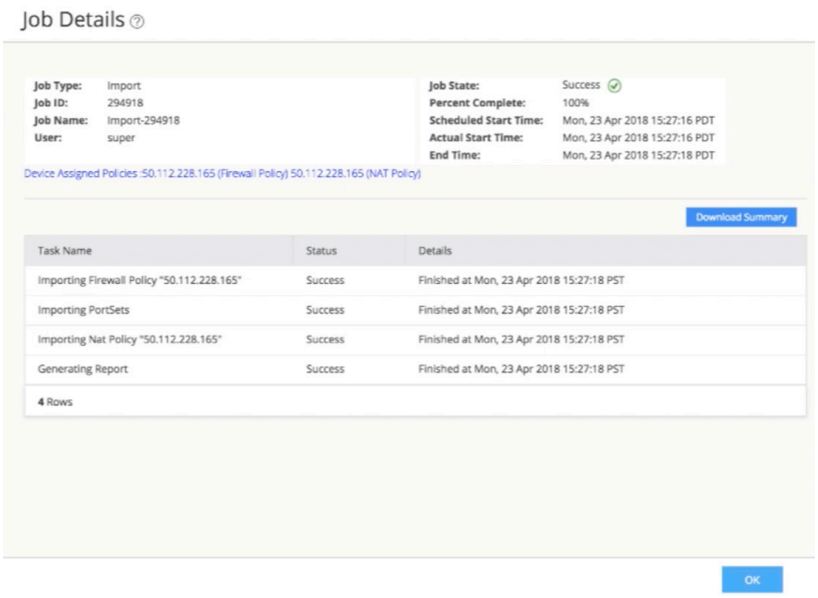

Figure 9.11 *Importing Policies to the Junos Space Security Director*

Once you have imported the selected policies, it's time to publish them to the Junos Space Security Director. From the left sidebar at the home page, click "Configuration" and then "Firewall Policies" to publish the firewall policies as shown in Figure 9.12. Note that to publish other policies, such as NAT policies, you follow the same steps.

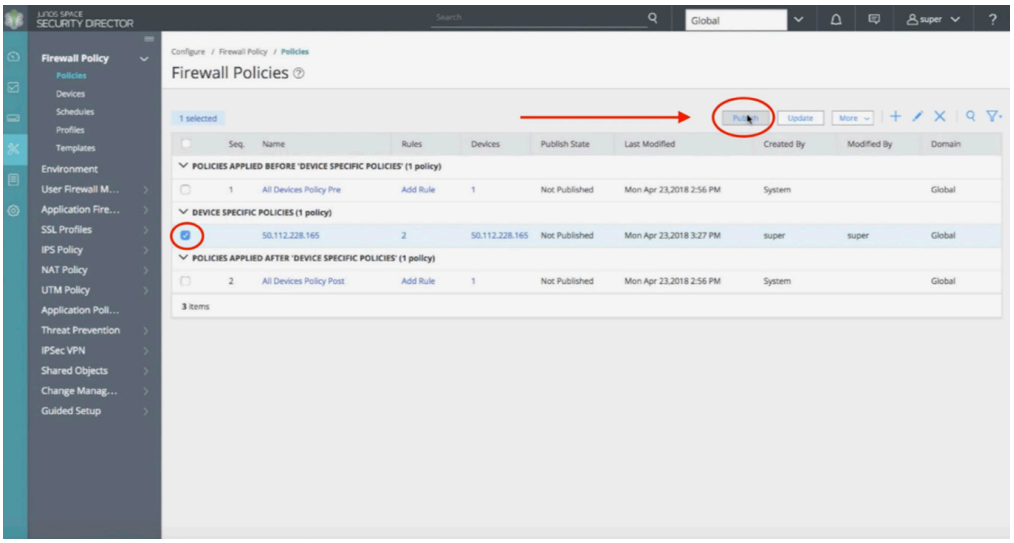

Figure 9.12 *Publishing the Imported Firewall Policies to the Junos Space Security Director*

Metadata-Based Firewall Policy

Earlier in this recipe, we introduced you to the concept of metadata-based policies and how they can be imported as part of creating the Policy Enforcer AWS connectors. Now let's illustrate how those tags (metadata information) can be used to define and change firewall rules.

To start, click on the firewall policy that was published in the previous step. You will be directed to the Rules page as shown in Figure 9.13.

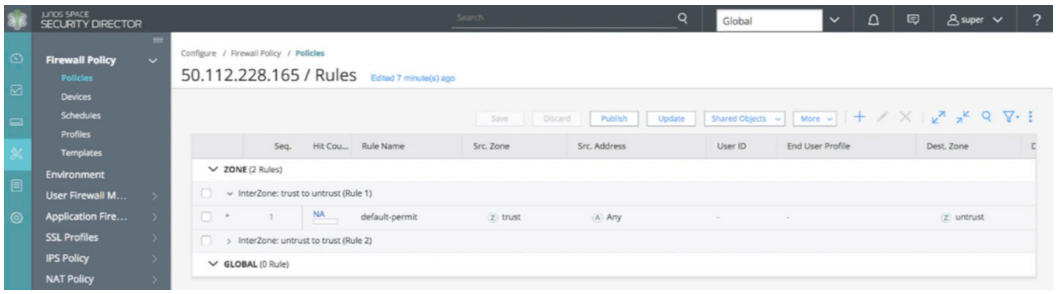

Figure 9.13 *Firewall Rules Page*

In the example shown in Figure 9.13, a firewall rule has been defined to allow traffic from all source addresses when flowing from trust zone to untrust zone. With a metadata-based policy, one can restrict the allowed source address to only a dynamic address group that fulfils certain meta-data requirements. Figure 9.14 depicts a scenario where only VMs that are associated with "Department=Finance" and "PCI=False" tags are the allowed source addresses.

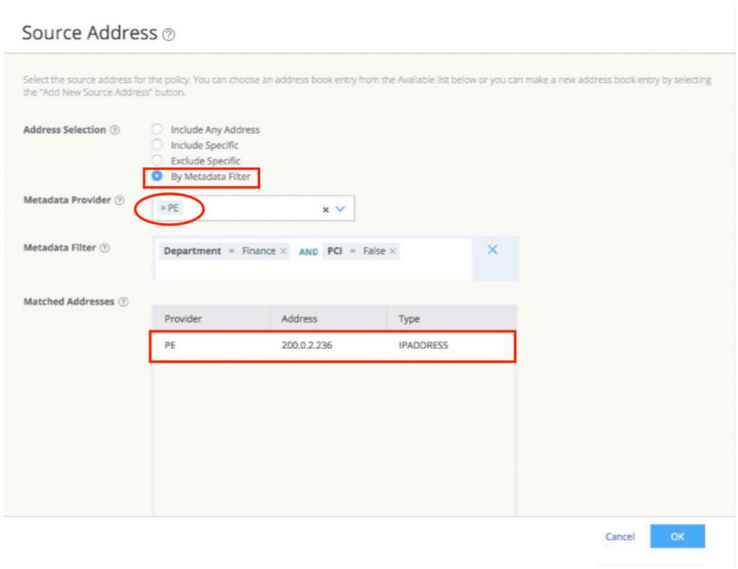

Figure 9.14 *Metadata-Based Dynamic Address Group*

In Figure 9.14, only one VM (with the IP address of 200.0.2.236) satisfies the metadata filter rules. Click OK and then save your changes. Finally, click Publish to sync-up the newly defined metadata policies with the actual firewall device deployed in your AWS VPC.

Threat Remediation

So far this recipe has discussed how to *enforce metadata-based policies*. However, most users would also like to be able to take some remediation actions when a workload is infected. Luckily, Policy Enforcer AWS Connector enables remediation by quarantining the infected hosts. Figure 9.15 is a high-level depiction of threat remediation on AWS using the Policy Enforcer AWS Connector.

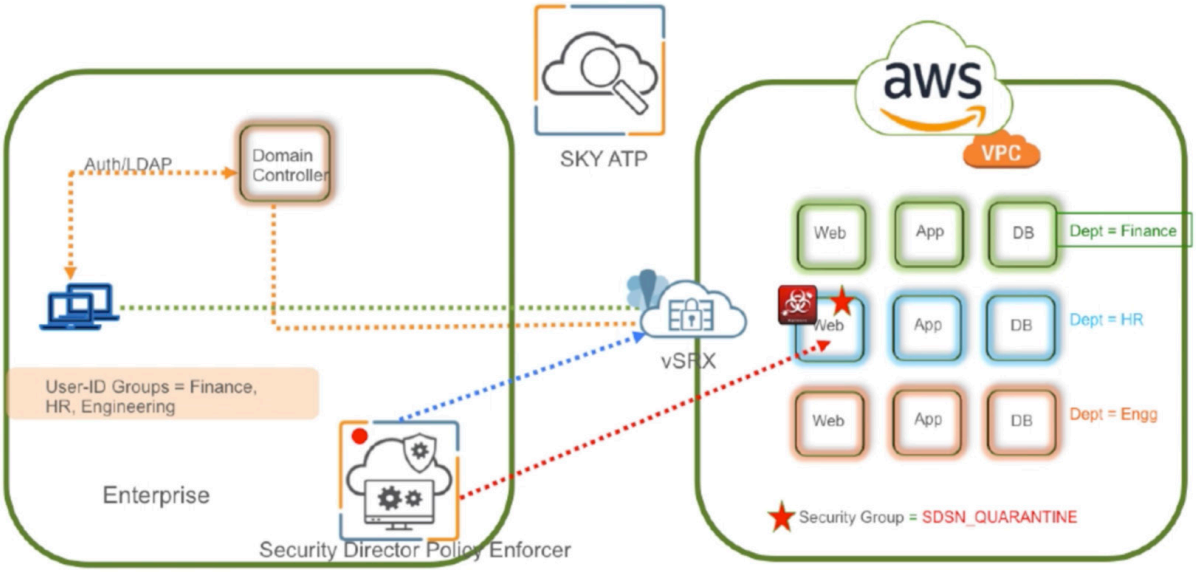

Figure 9.15 *Threat Remediation on AWS*

You can see in Figure 9.15 that the vSRX firewall receives feeds from the SKY ATP malware detection engine via the Security Director Policy Enforcer.

In its latest release, Policy Enforcer AWS Connector quarantines the infected hosts by placing them in a pre-created security group, which must then be specified when creating the Policy Enforcer AWS Connector (that was shown in Figure 9.8). Upon detecting an infected host, the Policy Enforcer will override the security groups associated with that host into the pre-created security group which will only contain the infected hosts. This security group stops lateral propagation of threats in the VPC.

Okay, with that brief introduction into the basics of threat remediation on AWS, it's time to set up and enable threat remediation on AWS through the Junos Space Security Director with the following step-by-step instructions:

From the Junos Space Security Director home page navigate the path Configuration > Guided Setup > Threat Prevention and you will arrive at the Threat Prevention Policy Setup as shown in Figure 9.16.

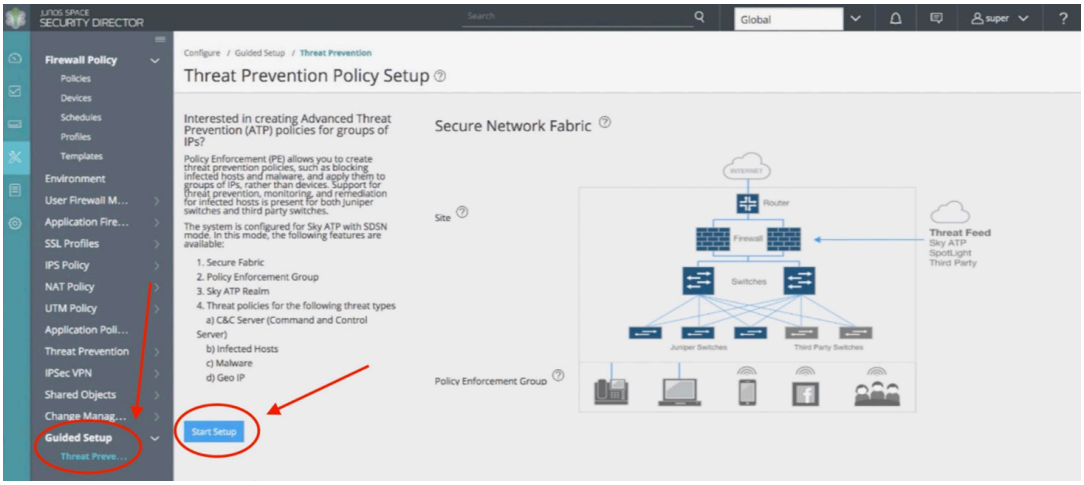

Figure 9.16 *Threat Prevention Policy Setup*

Click "Start Setup" to initiate the threat prevention policy set up process that has five numbered steps. First, select the site (AWS VPC) to which you are going to apply the threat prevention policy as shown in Figure 9.17.

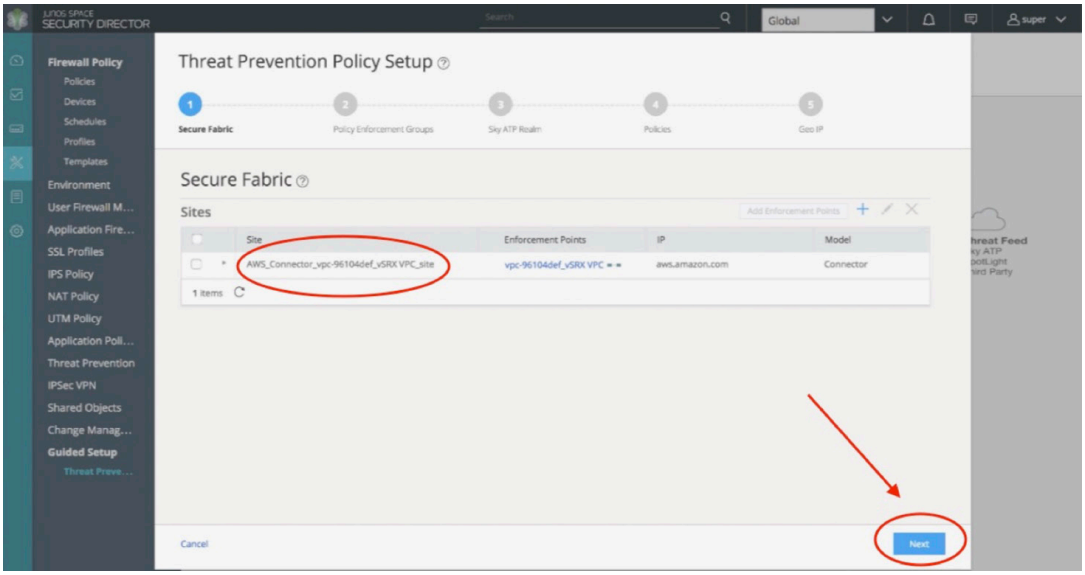

Figure 9.17 *Threat Prevention Policy Setup – Secure Fabric*

Click "Next" to be directed to the second step in the process. In this step a policy enforcement group needs to be selected. If you need to create a new group, you can do so by clicking at the "+" sign located at the upper right-hand corner of the page

and then follow the steps to complete creation of a new policy enforcement group as shown in Figure 9.18. Note that in the case of AWS Connector, a policy enforcement group is comprised of only the subnets within the VPC that are selected in this step.

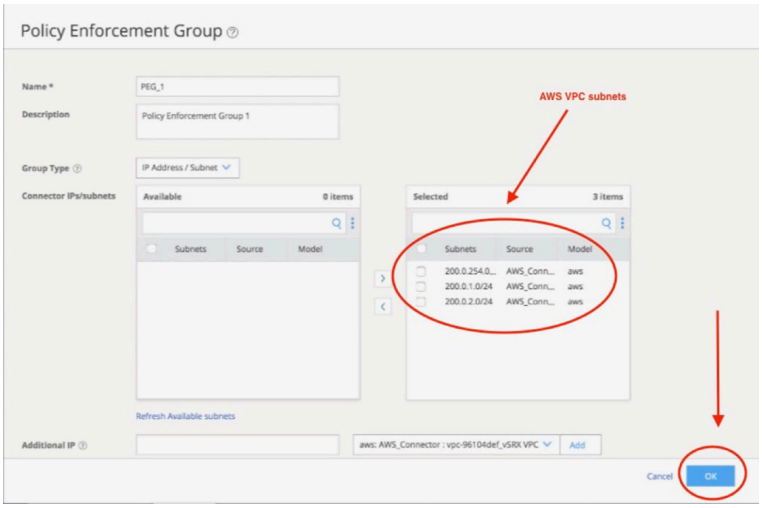

Figure 9.18 *Policy Enforcement Group Creation*

Click OK to finish this step. Verify that a new policy enforcement group has been created as shown in Figure 9.19.

NOTE To continue with the rest of this wizard, you must ensure that you have a Sky ATP realm created.

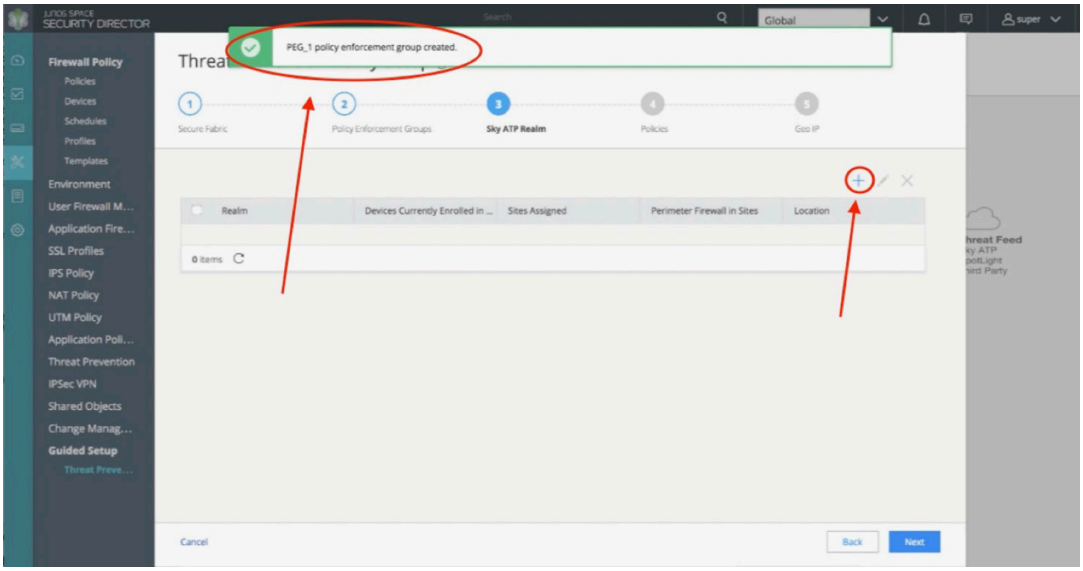

Figure 9.19 *Threat Prevention Policy Setup – Sky ATP Realm*

Click the "+" button located at the upper right-hand corner of the page to add details of your Sky ATP Realm as shown in Figure 9.19. Once added, click at assigned sites in the "Sites Assigned" column as shown in Figure 9.20. Select the appropriate site(s) (AWS VPC) and click OK as shown in Figure 9.21.

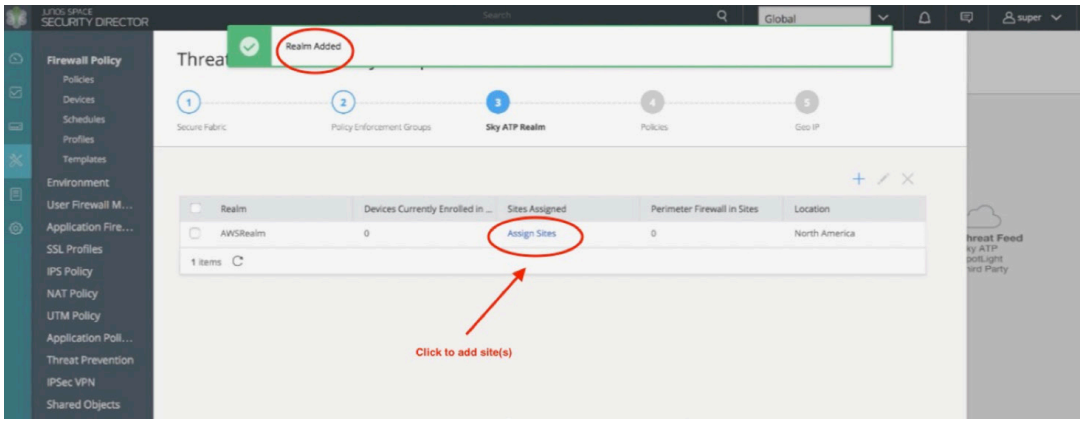

Figure 9.20 *Sky ATP Realm – Assign Sites*

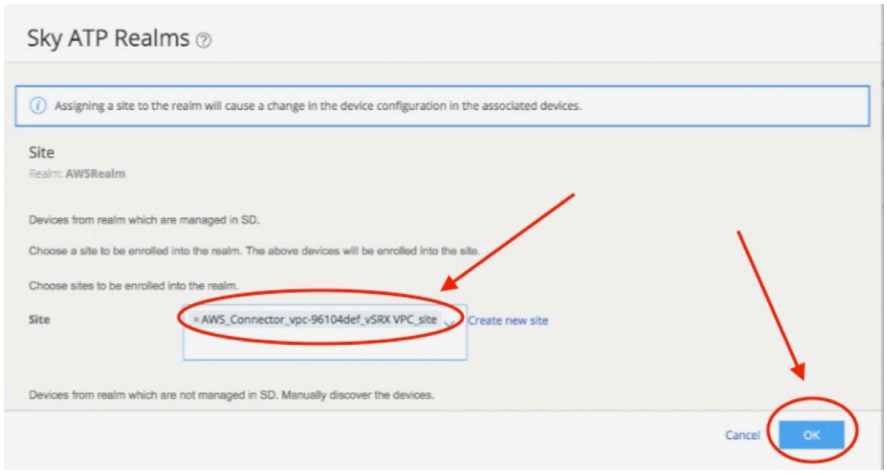

Figure 9.21 *Sky ATP Realm – Assign Sites*

After you have completed this step you will proceed to Step 4. Click "Next" to be directed to the "Policies" page.

To create policies, click at the "+" button. A new dialog box will open where you can configure your desired policies. Fill in the details and click OK as shown in Figure 9.22.

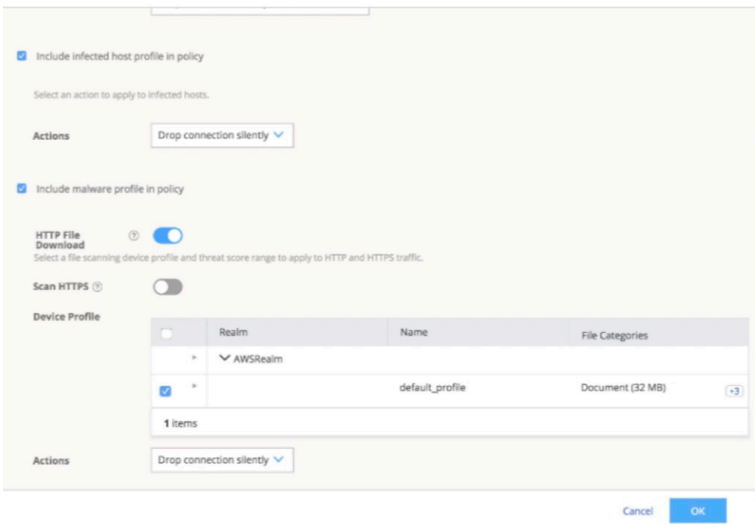

Figure 9.22 *Threat Prevention Policy Setup – Create Policy*

Now that a policy has been created, it's time to assign it to a policy enforcement group created in an earlier step. This is shown in Figure 9.23 and Figure 9.24.

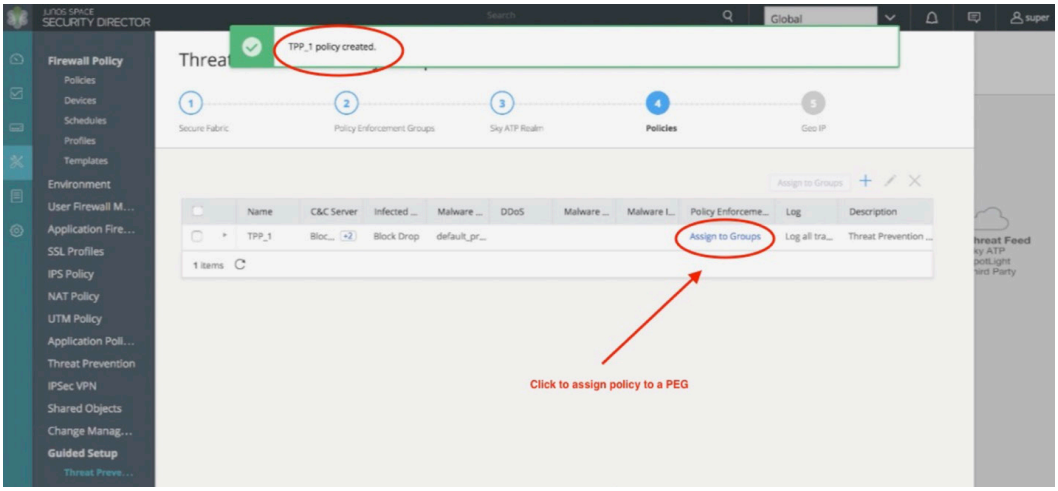

Figure 9.23 *Threat Prevention Policy Setup – Assign Policy to PEG*

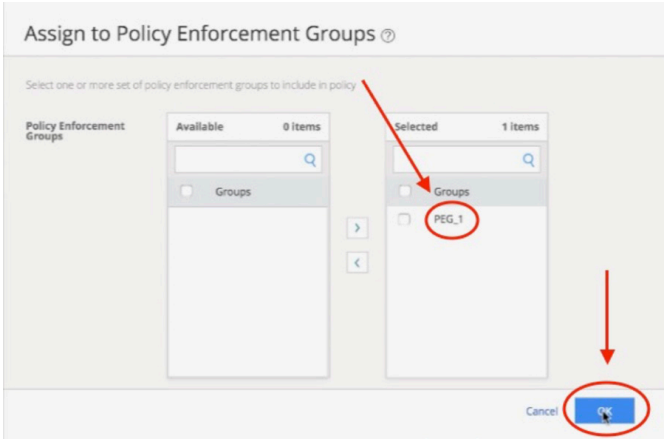

Figure 9.24 *Threat Prevention Policy Setup – Assign Policy to PEG*

Click OK to assign the created policy to the selected Policy Enforcement Group (PEG). Then follow the steps on your Threat Prevention Policy Setup to update and push the generated rules to your vSRX firewall as shown in Figure 9.25.

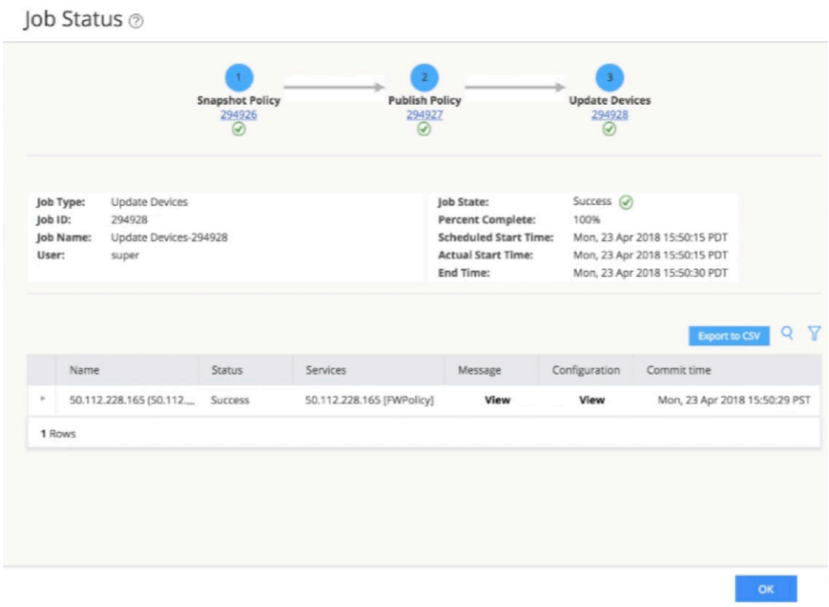

Figure 9.25 *Publish Policy to the Managed Security Device*

Click OK one more time to land at the final step of the Threat Prevention Policy Setup wizard. This is an optional step and we choose not to configure any Geo IP. Click "Finish" to complete the Threat Prevention Policy Setup as shown in Figures 9.26 and 9.27.

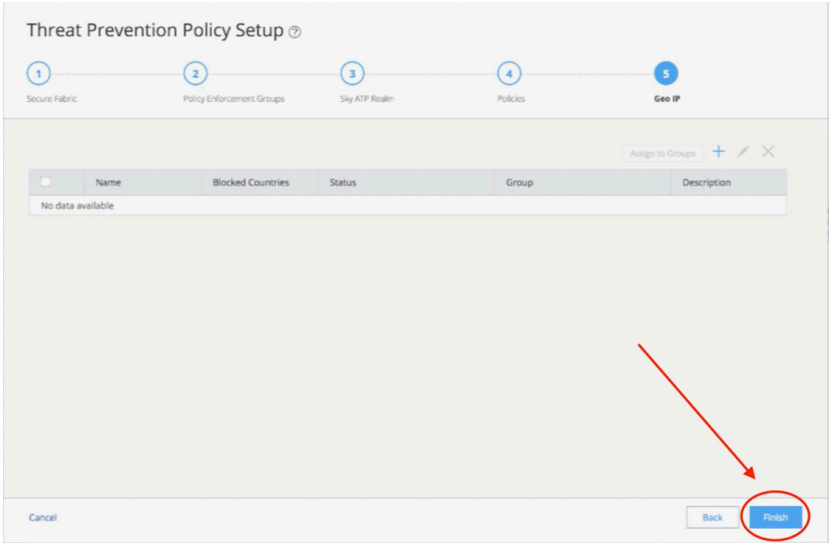

Figure 9.26 *Threat Prevention Policy Setup – Geo IP*

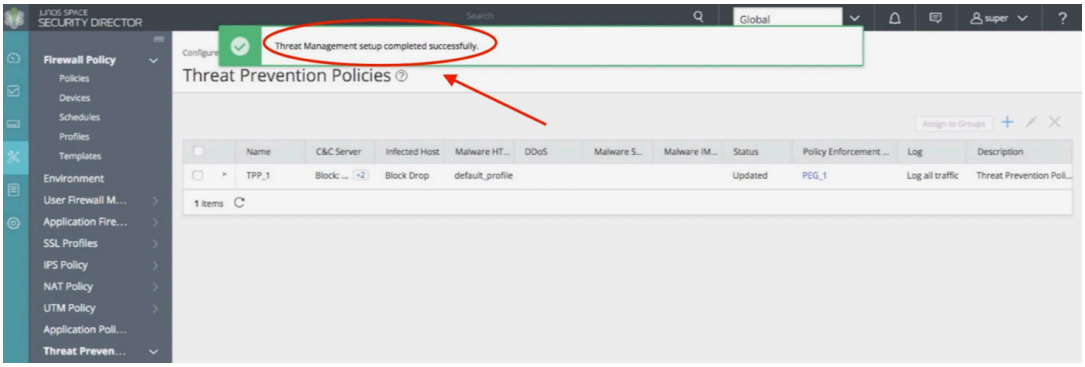

Figure 9.27 *Threat Prevention Policy – Setup Completed*

Now all the infected hosts in the subnets included in the policy enforcement group (that was created earlier in this section) will be quarantined as per Policy Enforcer AWS Connector threat remediation steps.

To verify, log in to one of your VMs in a protected VPC subnet and then try to contact a known command and control (C&C) address, for example, ping the address. You will notice that no response will be received because the vSRX NGFW blocks any traffic going to those IP addresses. Note that this can be a valid test only if you included "C&C" profile in your threat prevention policy.

Another way to verify that this host is added to the list of infected hosts is to navigate to the infected hosts page (Monitor > Threat Prevention > Hosts) from your Junos Space Security Director as shown in Figure 9.28.

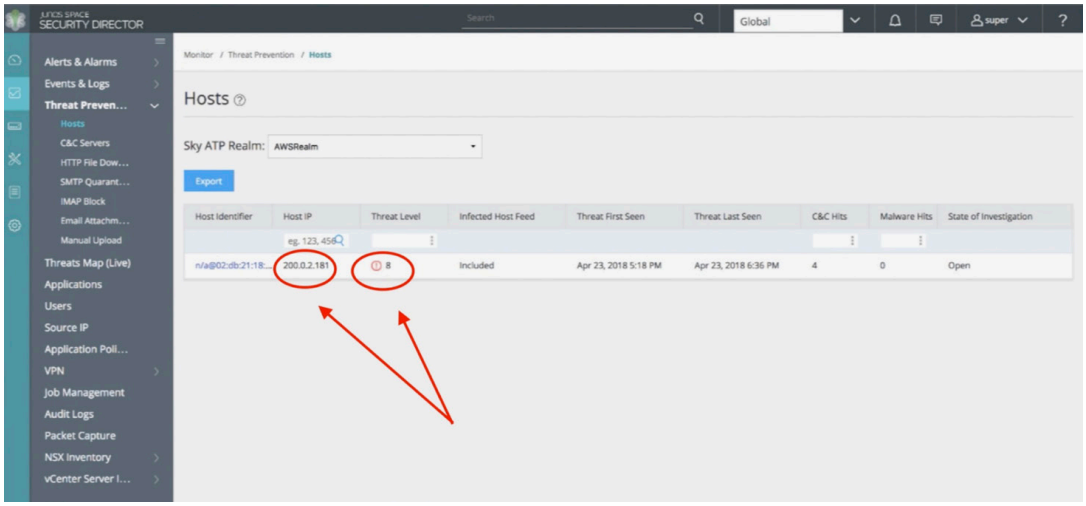

Figure 9.28 *List of Infected Hosts*

As explained earlier in this recipe, any infected host in AWS deployments that are managed by Policy Enforcer AWS connector and selected to be part of threat remediation, will be subject to a quarantine action and will be placed in a separate security group. This is done to prevent the infected host from laterally propagating the threat (to other hosts in the same VPC). To verify, navigate to your Amazon EC2 home page and then click at "EC2 Dashboard), a sample is shown in Figure 9.29.

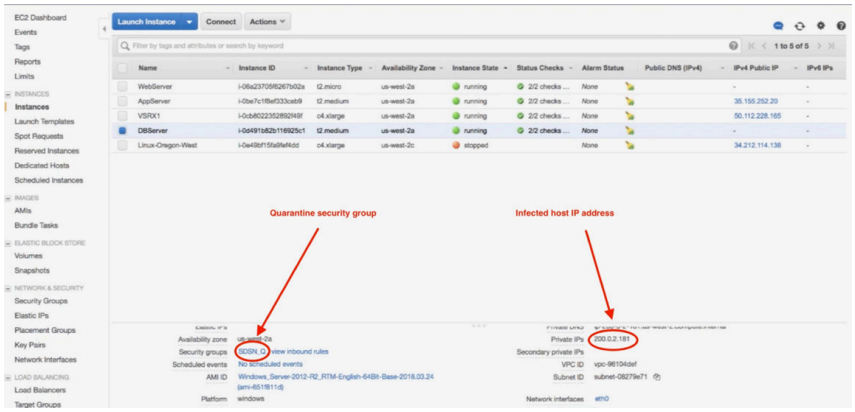

Figure 9.29 *Security Group Associated with Infected Host(s)*

Select the infected host and note the details section located at the bottom part of the screen and look for security groups associated with this VM. The security group name should match the name specified during the creation of the Policy Enforcer AWS connector done earlier in this recipe.

Conclusion

This recipe covered an overview of Juniper Software Defined Secure Networks (SDSN) and discussed multi-cloud metadata-based policy that allows for pre-created security policies to be dynamically enforced for all existing and new applications as they are launched in the cloud, and that dynamic address group ensures that changes in the cloud are enforced on the perimeter vSRX NGFW instantly without the need for a commit. A brief step-by-step to create Policy Enforcer AWS connector, a how to create firewall rules with metadata-based policy, and steps for threat prevention policy were outlined. Finally, the recipe provided steps to verify that infected AWS resources are detected and their security group membership is altered to quarantine infected resources.

Recipe 10: Securely Extending the Data Center

by Chirag Patel

This recipe focuses on securely extending an existing data center into the AWS cloud. Traditional data centers are physically situated at some geographical location for any number of reasons. While those reasons are outside the context of this cookbook, it's important to note that it's not easy to eliminate an existing physical data center, yet customers are seeking to incorporate some of the benefits of using cloud computing by extending their existing physical data center into the public cloud (as with AWS). These benefits may include reducing costs such as hosting certain applications within the AWS Cloud. Also, by extending data centers into the cloud, customers can create a hybrid environment enabling the best of both worlds to be integrated into the environment.

Extending the data center into the AWS Cloud also brings a number of security challenges such as ensuring that any connections from the data center to the AWS Cloud are securely protected and wrapped around your organization's security policies. In addition, the components within the AWS environment must still be bound to the same security principles that are applied to the physical data center. This recipe explores this arena in greater detail.

Some of the uses cases for extending a data center into the AWS Public Cloud include:

- *Private Networking*: Supports private networks that span over multiple AWS Regions.
- *Shared Connectivity*: Multiple VPCs can share connectivity to data centers, partner networks, and other clouds.
- Cross-Account AWS Usage: The VPCs and AWS resources can reside in multiple AWS accounts.

Problem

How do I extend an existing data center into the AWS Public Cloud while meeting all necessary security requirements?

Solution

Extending an existing physical data center into the AWS Cloud is not as complicated as it may sound. The ultimate goal is to simply make the AWS Cloud environment appear to be a simple extension of an existing data center that incorporates all the required security that you would expect in a typical, physical data center.

Let's first start at exploring how a traditional data center will connect to the AWS Cloud, the components that will be involved, and their purposes. Figure 10.1 depicts what will essentially be configured on the following pages what the end goal looks like and what this recipe is trying to accomplish.

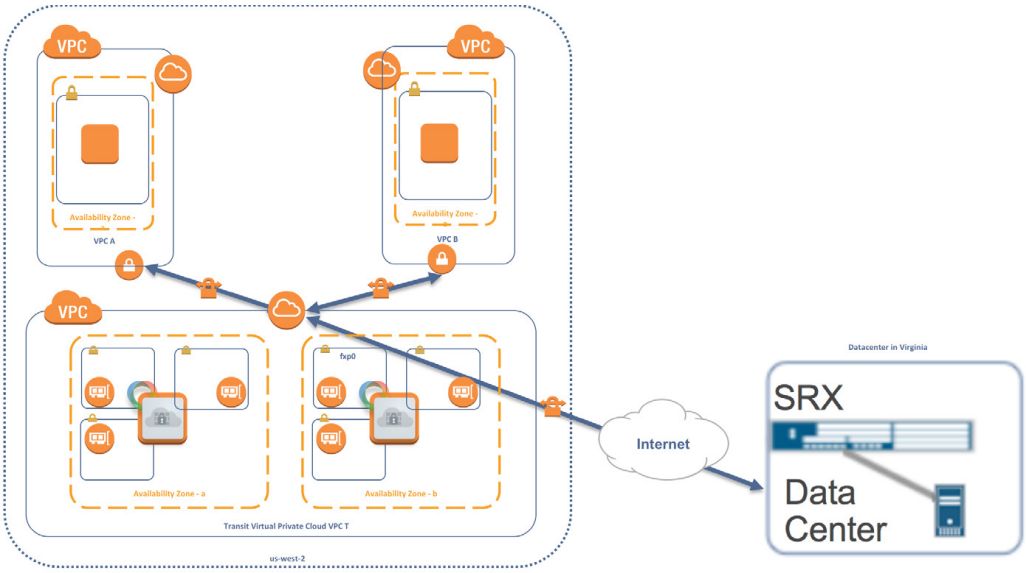

Figure 10.1 *Overview*

You can see in Figure 10.1, at a very high level mind you, a physical data center (on the right) connecting over the Internet to AWS (on the left), and in specific, to a AWS Transit VPC. Since the connection between AWS and the data center is over the Internet, you need to address this from a security perspective, which is explored when we look at the low-level configurations later on in this recipe. However, note that we will be configuring a VPN over the Internet to connect the data center to AWS securely.

Figure 10.2 takes a little closer look at the setup within the AWS environment. Note that while the focus here is to look at the configurations of the vSRX in AWS that is

needed to make this topology work, technically, Ansible scripts can be created in which the AWS VPC topology is built. Those scripts are outside the scope of this recipe and working with Ansible is discussed in other recipes of this *Day One* cookbook.

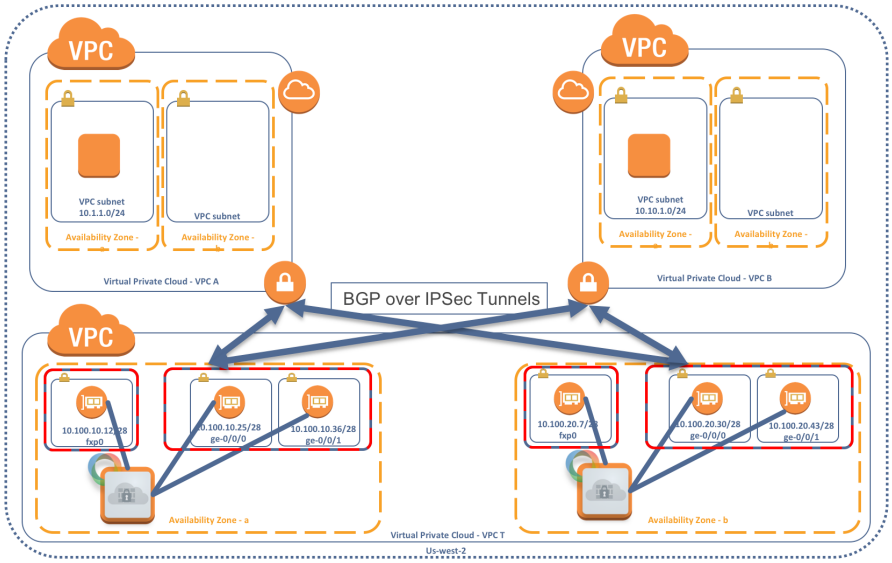

Figure 10.1 *Detailed Look at AWS Setup*

The concept of a Transit VPC is to connect multiple, geographically dispersed VPCs (same or different regions) and remote networks, and in our case, it would network within our traditional data center. Therefore, a Transit VPC can be considered as a specialized way of adding capability and flexibility to VPCs. The Transit VPC interconnects other VPCs, acting as the hub for data flow between spoke VPCs and other on-premise customer resources that exist within the data center. The vSRX within the Transit VPC creates a hub and spoke IPsec topology with the AWS VPNs on other VPCs, and also with the data center router and firewall. The Transit VPC allows data flow between AWS VPCs to stay in the cloud, minimizing limitations such as bandwidth, latency, and availability, that typically occur when introducing non-cloud-based resources to the data flow.

The vSRX inside the Transit VPC is the data flow hub between the other VPCs, which are spoke VPCs. These spoke VPCs provide the typical VPC functions: private address space, elastic public address space, network access control (NAC) using security groups, and virtual private gateways (VGWs) to build IPsec VPN connections. The primary characteristic of this Transit VPC use case is that spoke VPC, VGW, and VPN connections are established with the vSRX virtual firewall instances in the Transit VPC.

In Figure 10.2 you will see there are two vSRXs within the Transit VPC. Each vSRX is within its own Availability Zone across a single region. The purpose of this design is to provide High Availability through AWS VPCs and routing.

NOTE This is *different* from using native Junos High Availability that is typically found on physical SRX devices.

Any discussion regarding security will not be complete without talking about IPsec tunnels. And in this scenario IPsec tunnels are critical component of the recipe. IPsec tunnels are set up between the AWS VPN client and the vSRX in the Transit VPC. An IPsec tunnel is also set up between the SRX firewall located within the data center and does not need to be reconfigured when new VPCs are added. All of the routing is done in the Transit VPC.

The final point to note in Figure 10.2 is the use of BGP routing protocol over the IPsec tunnels. There are IPsec tunnels with BGP running between the following:

- The two IPsec tunnels between VPC A-VPN to each vSRX in VPC-T. From each vSRX, ge-0/0/0 will be used to route to VPC-A-VPN.
- The two IPsec tunnels between VPC B-VPN to each vSRX in VPC-T. From each vSRX, ge-0/0/0 will be used to route to VPC-B-VPN.

So, the objectives of this recipe are:

- Create three VPCs. VPC A and VPC B will be the Spokes. VPC T will be the Hub (Transit VPC).
- Two vSRXs will be instantiated in VPC T.
- VGWs are installed on VPC A and VPC B.
- IPsec VPNs are created between the VGWs (in VPC A and VPC B) to the vSRXs in VPC T. Because of the High Availability nature of AWS, two IPsec tunnels will be created between VGWs and vSRXs.
- Two Linux hosts will be instantiated in VPC A and VPC B. The data flow from these hosts will be directed out through the IPsec tunnels.

Now, let's switch our attention to the actual data center. Look closely at the setup illustrated in Figure 10.3. We are still employing the same configuration principles applied within the AWS vSRX, and the configurations on the physical SRX should not look any different, or introduce anything new, except that the configurations are essentially flipped.

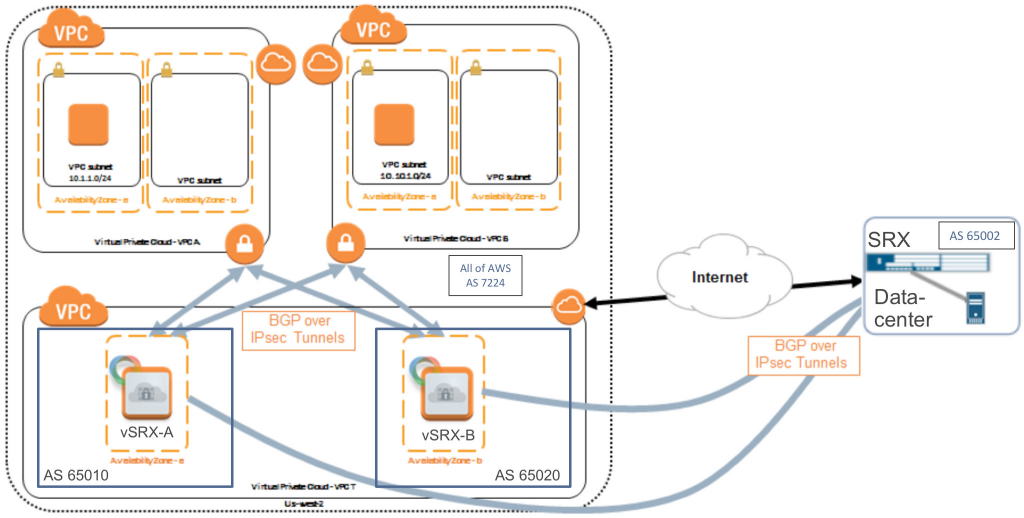

Figure 10.3 BGP Over IPsec

The SRX that is used in the data center as the *on-premise* SRX has IPsec tunnels to the two vSRXs created in the AWS Transit VPC-T. The SRX has a public IP address of 10.10.10.1/27, which is used as the IKE Gateway IP on ge-0/0/0.

That's all the high-level design and topology, so let's look at the configurations of the three main devices: the two vSRXs in AWS and the physical on-premise SRX in the data center. To simplify the configuration, only the necessary configurations that apply security and connectivity will be examined.

Let's start off with configuring the on-premise physical SRX within our data center. First let's add some a basic housekeeping configuration that is standard to any SRX deployment – it is not specific to our extending the data center configuration, but in order to put down a minimum set of recommended configurations:

NOTE This will differ in your own environment.

```
set groups cookbook system host-name Data-Center-SRX
set groups cookbook system time-zone America/New_York
set groups cookbook system authentication-order radius
set groups cookbook system authentication-order password
set groups cookbook system root-authentication encrypted-password "$1$mzUQjfuc$XjaJIv4Un5Dl6iwvWAi
1u/"
set groups cookbook system name-server 8.8.8.8
set groups cookbook system login message "\n \n This is a recipe for extending data center to AWS \n \n
*** Please log out if you are not an authorized user ***\n \n"
set groups cookbook system login user lab uid 2000
set groups cookbook system login user lab class superuser
set groups cookbook system login user lab authentication encrypted-password "$1$mzUQjfuc$XjaJIv4Un5Dl6
iwvWAi1u/"
set groups cookbook system services ssh protocol-version v2
```

```
set groups cookbook system services netconf ssh
set groups cookbook system syslog host 172.25.45.6 authorization any
set groups cookbook system syslog file messages any info
set groups cookbook system ntp server 66.129.255.62 prefer
set groups cookbook snmp community public
```

And apply the above configuration:

```
set apply-groups cookbook
```

Next, create sample login user accounts:

```
set system login user aws uid 2002
set system login user aws class super-user
set system login user aws authentication encrypted-password "$5$cESAUjC1$ "
set system login user user uid 2003
set system login user user class super-user
set system login user user authentication encrypted-password "$5$OvtO1eW/$ MDv0h/WEO9"
```

Now let's create the configurations that are more directly related to the extension of the data center into AWS. Here, define the interfaces along with IP addresses:

```
set interfaces ge-0/0/0 description "to public internet"
set interfaces ge-0/0/0 unit 0 family inet address 10.10.10.1/27
set interfaces ge-0/0/1 description "to trusted network"
set interfaces ge-0/0/1 unit 0 family inet address 192.168.100.1/24
```

You need to define the ST tunnels that will later bind to the VPN configuration. Notice the two ST tunnels that are to each vSRX in the AWS Transit VPC that exists in two different availability zones:

```
set interfaces st0 unit 1 description "to AWS vSRX-A tunnel endpoint st0.1"
set interfaces st0 unit 1 family inet address 10.0.251.20/24
set interfaces st0 unit 2 description "to AWS vSRX-B tunnel endpoint st0.1"
set interfaces st0 unit 2 family inet address 10.0.200.2/30
```

Configure static routing:

```
set routing-options static route 0.0.0.0/0 next-hop 66.129.238.129
set routing-options static route 66.129.241.13/32 next-hop 192.168.2.1
set routing-options static route 66.129.241.14/32 next-hop 192.168.2.1
set routing-options static route 66.129.241.12/32 next-hop 192.168.2.1
set routing-options static route 66.129.241.11/32 next-hop 192.168.2.1
set routing-options static route 66.129.241.10/32 next-hop 192.168.2.1
```

Configure BGP to route traffic over the VPN tunnels. Note we are using a different AS number and will therefore be using eBGP across the tunnels. In addition, there are two eBGP links to two separate ASs. We will discuss later on in this recipe why there are two different ASs in AWS for each vSRX in the section on configuring vSRX in AWS.

```
set routing-options autonomous-system 65002
set protocols bgp export trusted-routes
set protocols bgp group aws type external
set protocols bgp group aws neighbor 10.0.251.10 peer-as 65010
set protocols bgp group aws-vSRX-B type external
set protocols bgp group aws-vSRX-B neighbor 10.0.200.1 peer-as 65020
```

Configure routing policies:

```
set policy-options policy-statement trusted-routes term interface-route from protocol direct
set policy-options policy-statement trusted-routes term interface-route from interface ge-0/0/1.0
set policy-options policy-statement trusted-routes term interface-route from interface ge-0/0/2.0
set policy-options policy-statement trusted-routes term interface-route then accept
```

Next configure the actual VPN Tunnels. Again, note there are two VPN tunnels to different vSRXs within the transit VPC in AWS that exist in two different availability zones. Also, note the external interface we are using to create the VPN tunnels is ge-0/0/0. Finally, we are configuring route-based VPNs with the two tunnel interfaces, st0.1 and st0.2:

```
set security ike proposal AWS_IKE_proposal authentication-method pre-shared-keys
set security ike proposal AWS_IKE_proposal dh-group group2
set security ike proposal AWS_IKE_proposal authentication-algorithm sha-256
set security ike proposal AWS_IKE_proposal encryption-algorithm aes-256-cbc
set security ike proposal AWS_IKE_proposal lifetime-seconds 1800
set security ike policy AWS mode aggressive
set security ike policy AWS proposals AWS_IKE_proposal
set security ike policy AWS pre-shared-key ascii-text "$9$CFZ5pORXxdVs4LxUjkqf5Rhc"
set security ike gateway AWS ike-policy AWS
set security ike gateway AWS address 18.219.31.75
set security ike gateway AWS local-identity user-at-hostname "herndon1@juniper.net"
set security ike gateway AWS remote-identity user-at-hostname "aws-herndon1@juniper.net"
set security ike gateway AWS external-interface ge-0/0/0
set security ike gateway AWS-vSRX-B ike-policy AWS
set security ike gateway AWS-vSRX-B address 13.59.116.210
set security ike gateway AWS-vSRX-B local-identity user-at-hostname "herndon2@juniper.net"
set security ike gateway AWS-vSRX-B remote-identity user-at-hostname "aws-herndon2@juniper.net"
set security ike gateway AWS-vSRX-B external-interface ge-0/0/0
set security ipsec traceoptions flag all
set security ipsec proposal AWS_IPSEC protocol esp
set security ipsec proposal AWS_IPSEC authentication-algorithm hmac-sha1-96
set security ipsec proposal AWS_IPSEC encryption-algorithm aes-256-cbc
set security ipsec policy AWS_IPSEC_policy proposals AWS_IPSEC
set security ipsec vpn AWS bind-interface st0.1
set security ipsec vpn AWS ike gateway AWS
set security ipsec vpn AWS ike ipsec-policy AWS_IPSEC_policy
set security ipsec vpn AWS establish-tunnels immediately
set security ipsec vpn AWS-vSRX-B bind-interface st0.2
set security ipsec vpn AWS-vSRX-B ike gateway AWS-vSRX-B
set security ipsec vpn AWS-vSRX-B ike ipsec-policy AWS_IPSEC_policy
```

Now configure address books to define which subnets will be allowed to pass through the SRX:

```
set security address-book global address aws-subnet1 10.0.20.0/24
set security address-book global address aws-subnet2 10.0.0.0/16
```

Define the security zones, assign relevant interfaces to the security zones, and permit which traffic is allowed inbound on those specific zones. Configure all these for the untrust zone:

```
set security zones security-zone untrust host-inbound-traffic system-services ping
set security zones security-zone untrust host-inbound-traffic system-services ike
set security zones security-zone untrust interfaces ge-0/0/0.0
```

Let's now configure the same as the previous step, but for the trust zone:

```
set security zones security-zone trust host-inbound-traffic system-services all
set security zones security-zone trust host-inbound-traffic protocols all
set security zones security-zone trust interfaces ge-0/0/1.0
set security zones security-zone trust interfaces st0.1
set security zones security-zone trust interfaces st0.2
```

Configure the security policies that will permit traffic to pass the SRX firewall from zone trust to trust:

```
set security policies from-zone trust to-zone trust policy trust-to-trust match source-address any
set security policies from-zone trust to-zone trust policy trust-to-trust match destination-address
any
set security policies from-zone trust to-zone trust policy trust-to-trust match application any
set security policies from-zone trust to-zone trust policy trust-to-trust then permit
```

And configure the security policies that will permit traffic to pass the SRX firewall from zone untrust to trust:

```
set security policies from-zone untrust to-zone trust policy aws-resources match source-address
aws-subnet1
set security policies from-zone untrust to-zone trust policy aws-resources match source-address
paul-subnet1
set security policies from-zone untrust to-zone trust policy aws-resources match source-address
aws-subnet2
set security policies from-zone untrust to-zone trust policy aws-resources match destination-address
herndon-subnet-1
set security policies from-zone untrust to-zone trust policy aws-resources match application any
set security policies from-zone untrust to-zone trust policy aws-resources then permit
set security policies from-zone untrust to-zone trust policy aws-resources then log session-close
```

Configure the security policies that will permit traffic to pass the SRX Firewall from zone trust to untrust:

```
set security policies from-zone trust to-zone untrust policy trust-to-untrust match source-address any
set security policies from-zone trust to-zone untrust policy trust-to-untrust match destination-
address any
set security policies from-zone trust to-zone untrust policy trust-to-untrust match application any
set security policies from-zone trust to-zone untrust policy trust-to-untrust then permit
set security policies default-policy permit-all
```

And you are done! As you can see, configuring an on-premise SRX to connect to AWS is no different than configuring a physical SRX in a data center. You applied some basic housekeeping configurations to get started, and then created VPN configurations to ensure you had security connectivity to the data center. We added some BGP to help move traffic up the AWS environment, and finally configured almost every SRX firewall, security polices, address book entries, and zones that govern what is permitted and what is not permitted through the firewall.

Now let's shift our focus to configuring the other end of connectivity in the AWS environment, by applying these same configuration principles and components. Start by configuring the vSRX in availability zone A within the AWS Environment

with some housekeeping items such as access, root-authentication, hostnames, syslog, NTP, etc., the type of configuration that is considered a baseline configuration in the vSRX:

```
set groups aws-default system root-authentication ssh-rsa
"ssh-rsaAAAAB3NzaC1yc2EAAAADAQABAAABAQCdVIsagm8QOhbkXvArqavsNldnKgAdhlJ6dD4Glp++D93ALVUvacETOFmOqj
8E8Yq2eCoxhJGhZ4wZQMGAOlskpqtQxsR0k/DTb9b+kzb3EKtyidvetlx63NMjr4xEzrwJ4xZqrDUZ7TCYKsfXNihzWDOOdm6cVA
Tru3KVo5sqdBYEHbVBZqi22MKlc1i01wY1dRGhEL0UZA4RAniX jnpr_aws_us-east-2_key"
set groups aws-default system services ssh no-passwords
set groups aws-default system services netconf ssh
set groups aws-default system services web-management https system-generated-certificate
set groups aws-default interfaces fxp0 unit 0 family inet address 10.100.10.12/28
set groups aws-default routing-options static route 0.0.0.0/0 next-hop 10.100.10.1
set apply-groups aws-default
set system host-name vSRX_A
set system root-authentication encrypted-password "$5$zTVMlqhk$9qMrvzRJ/
ARmU1CF8ZgvZkuWFCDnY6R7FnLDAAtssGA"
set system name-server 169.254.169.253
set system syslog user * any emergency
set system syslog file messages any notice
set system syslog file messages authorization info
set system syslog file interactive-commands interactive-commands any
set system license autoupdate url https://ae1.juniper.net/junos/key_retrieval
set system ntp server 69.164.213.136
set system ntp server 50.22.155.163
```

Once the basic configuration is out of the way, let's start configuring the VPN. This is a rather large section so we'll break it down into a number of subsections. Start with the IKE proposals:

```
set security ike proposal ike-prop-vpn-aad0049d-1 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-aad0049d-1 dh-group group2
set security ike proposal ike-prop-vpn-aad0049d-1 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-aad0049d-1 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-aad0049d-1 lifetime-seconds 28800
set security ike proposal ike-prop-vpn-aad0049d-2 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-aad0049d-2 dh-group group2
set security ike proposal ike-prop-vpn-aad0049d-2 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-aad0049d-2 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-aad0049d-2 lifetime-seconds 28800
set security ike proposal ike-prop-vpn-a9d0049e-1 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-a9d0049e-1 dh-group group2
set security ike proposal ike-prop-vpn-a9d0049e-1 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-a9d0049e-1 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-a9d0049e-1 lifetime-seconds 28800
set security ike proposal ike-prop-vpn-a9d0049e-2 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-a9d0049e-2 dh-group group2
set security ike proposal ike-prop-vpn-a9d0049e-2 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-a9d0049e-2 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-a9d0049e-2 lifetime-seconds 28800
set security ike proposal AWS_IKE-H_proposal authentication-method pre-shared-keys
set security ike proposal AWS_IKE-H_proposal dh-group group2
set security ike proposal AWS_IKE-H_proposal authentication-algorithm sha-256
set security ike proposal AWS_IKE-H_proposal encryption-algorithm aes-256-cbc
set security ike proposal AWS_IKE-H_proposal lifetime-seconds 1800
```

Next configure the IKE policy:

```
set security ike policy ike-pol-vpn-aad0049d-1 mode main
set security ike policy ike-pol-vpn-aad0049d-1 proposals ike-prop-vpn-aad0049d-1
set security ike policy ike-pol-vpn-aad0049d-1 pre-shared-key ascii-text "$9$QNOTz3/
p01IESKMNVsgUDF36AIhKMLX-waZznCtOB-VwgaGHqm3n9lK4aUD.m5Tz3CpeK8L7-oJn/ApREsYgJDHmfzAuBcy"
set security ike policy ike-pol-vpn-aad0049d-2 mode main
set security ike policy ike-pol-vpn-aad0049d-2 proposals ike-prop-vpn-aad0049d-2
set security ike policy ike-pol-vpn-aad0049d-2 pre-shared-key ascii-text "$9$/l18A0IEcyvWxVwH.5zCArev
MXNVw24JGfTESev7NfTzntuEcyvMLcS2aJDq.Ap01Rc-VsgoGz3v8Lxbw4oJZUin6CB1hpu"
set security ike policy ike-pol-vpn-a9d0049e-1 mode main
set security ike policy ike-pol-vpn-a9d0049e-1 proposals ike-prop-vpn-a9d0049e-1
set security ike policy ike-pol-vpn-a9d0049e-1 pre-shared-key ascii-text
"$9$GdjPTFn9tuBoJ9pOBcSLX7Nw2ZUjPTFUDOIEcvMDikmfQFnC01EwYH.PfzFKM87Y4ZUiqP5GUHmP5n6lKvMxNs2ojHmSr"
set security ike policy ike-pol-vpn-a9d0049e-2 mode main
set security ike policy ike-pol-vpn-a9d0049e-2 proposals ike-prop-vpn-a9d0049e-2
set security ike policy ike-pol-vpn-a9d0049e-2 pre-shared-key ascii-text
"$9$dNVgof5Fn9tuOlMLXVbF369O18L7wYo1R2aJU.m69Au1hevW-b2qm1EhyW8DiH.5FCA0OIEjHBRcr8LN-VsYoGDkqPT9A"
set security ike policy AWS-policy-H mode aggressive
set security ike policy AWS-policy-H proposals AWS_IKE-H_proposal
```

Configure the IKE gateway. Note there is one IKE gateway located in the customer data center and two IKE gateways in each Spoke VPC, and there are two IPsec Tunnels to each spoke VPN for High Availability:

```
set security ike policy AWS-policy-H pre-shared-key ascii-text "$9$gCaGi6/tpORn/reMWx7ikq"
set security ike gateway gw-vpn-aad0049d-1 ike-policy ike-pol-vpn-aad0049d-1
set security ike gateway gw-vpn-aad0049d-1 address 52.14.84.89
set security ike gateway gw-vpn-aad0049d-1 dead-peer-detection interval 10
set security ike gateway gw-vpn-aad0049d-1 dead-peer-detection threshold 3
set security ike gateway gw-vpn-aad0049d-1 no-nat-traversal
set security ike gateway gw-vpn-aad0049d-1 external-interface ge-0/0/1.0
set security ike gateway gw-vpn-aad0049d-2 ike-policy ike-pol-vpn-aad0049d-2
set security ike gateway gw-vpn-aad0049d-2 address 52.15.221.249
set security ike gateway gw-vpn-aad0049d-2 dead-peer-detection interval 10
set security ike gateway gw-vpn-aad0049d-2 dead-peer-detection threshold 3
set security ike gateway gw-vpn-aad0049d-2 no-nat-traversal
set security ike gateway gw-vpn-aad0049d-2 external-interface ge-0/0/1.0
set security ike gateway gw-vpn-a9d0049e-1 ike-policy ike-pol-vpn-a9d0049e-1
set security ike gateway gw-vpn-a9d0049e-1 address 18.217.46.218
set security ike gateway gw-vpn-a9d0049e-1 dead-peer-detection interval 10
set security ike gateway gw-vpn-a9d0049e-1 dead-peer-detection threshold 3
set security ike gateway gw-vpn-a9d0049e-1 no-nat-traversal
set security ike gateway gw-vpn-a9d0049e-1 external-interface ge-0/0/1.0
set security ike gateway gw-vpn-a9d0049e-2 ike-policy ike-pol-vpn-a9d0049e-2
set security ike gateway gw-vpn-a9d0049e-2 address 52.15.95.96
set security ike gateway gw-vpn-a9d0049e-2 dead-peer-detection interval 10
set security ike gateway gw-vpn-a9d0049e-2 dead-peer-detection threshold 3
set security ike gateway gw-vpn-a9d0049e-2 no-nat-traversal
set security ike gateway gw-vpn-a9d0049e-2 external-interface ge-0/0/1.0
set security ike gateway AWS-gw-H ike-policy AWS-policy-H
set security ike gateway AWS-gw-H address 10.10.10.1/27
set security ike gateway AWS-gw-H local-identity user-at-hostname "aws-herndon1@juniper.net"
set security ike gateway AWS-gw-H remote-identity user-at-hostname "herndon1@juniper.net"
set security ike gateway AWS-gw-H external-interface ge-0/0/0
```

Configure the IPsec proposal:

```
set security ipsec proposal ipsec-prop-vpn-aad0049d-1 protocol esp
set security ipsec proposal ipsec-prop-vpn-aad0049d-1 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-aad0049d-1 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-aad0049d-1 lifetime-seconds 3600
set security ipsec proposal ipsec-prop-vpn-aad0049d-2 protocol esp
set security ipsec proposal ipsec-prop-vpn-aad0049d-2 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-aad0049d-2 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-aad0049d-2 lifetime-seconds 3600
set security ipsec proposal ipsec-prop-vpn-a9d0049e-1 protocol esp
set security ipsec proposal ipsec-prop-vpn-a9d0049e-1 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-a9d0049e-1 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-a9d0049e-1 lifetime-seconds 3600
set security ipsec proposal ipsec-prop-vpn-a9d0049e-2 protocol esp
set security ipsec proposal ipsec-prop-vpn-a9d0049e-2 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-a9d0049e-2 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-a9d0049e-2 lifetime-seconds 3600
set security ipsec proposal AWS_IPSEC-H protocol esp
set security ipsec proposal AWS_IPSEC-H authentication-algorithm hmac-sha1-96
set security ipsec proposal AWS_IPSEC-H encryption-algorithm aes-256-cbc
```

Configure the IPsec policy:

```
set security ipsec policy ipsec-pol-vpn-aad0049d-1 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-aad0049d-1 proposals ipsec-prop-vpn-aad0049d-1
set security ipsec policy ipsec-pol-vpn-aad0049d-2 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-aad0049d-2 proposals ipsec-prop-vpn-aad0049d-2
set security ipsec policy ipsec-pol-vpn-a9d0049e-1 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-a9d0049e-1 proposals ipsec-prop-vpn-a9d0049e-1
set security ipsec policy ipsec-pol-vpn-a9d0049e-2 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-a9d0049e-2 proposals ipsec-prop-vpn-a9d0049e-2
set security ipsec policy AWS_IPSEC-H_policy proposals AWS_IPSEC-H
```

Finally let's wrap up the VPN configuration and bring everything together by configuring the IPsec VPN and binding to the ST interfaces. Note that we have not actually defined the ST interface yet or assigned them to a zone, but will do so soon:

```
set security ipsec vpn vpn-aad0049d-1 bind-interface st0.10
set security ipsec vpn vpn-aad0049d-1 df-bit clear
set security ipsec vpn vpn-aad0049d-1 ike gateway gw-vpn-aad0049d-1
set security ipsec vpn vpn-aad0049d-1 ike ipsec-policy ipsec-pol-vpn-aad0049d-1
set security ipsec vpn vpn-aad0049d-2 bind-interface st0.11
set security ipsec vpn vpn-aad0049d-2 df-bit clear
set security ipsec vpn vpn-aad0049d-2 ike gateway gw-vpn-aad0049d-2
set security ipsec vpn vpn-aad0049d-2 ike ipsec-policy ipsec-pol-vpn-aad0049d-2
set security ipsec vpn vpn-a9d0049e-1 bind-interface st0.12
set security ipsec vpn vpn-a9d0049e-1 df-bit clear
set security ipsec vpn vpn-a9d0049e-1 ike gateway gw-vpn-a9d0049e-1
set security ipsec vpn vpn-a9d0049e-1 ike ipsec-policy ipsec-pol-vpn-a9d0049e-1
set security ipsec vpn vpn-a9d0049e-2 bind-interface st0.13
set security ipsec vpn vpn-a9d0049e-2 df-bit clear
set security ipsec vpn vpn-a9d0049e-2 ike gateway gw-vpn-a9d0049e-2
set security ipsec vpn vpn-a9d0049e-2 ike ipsec-policy ipsec-pol-vpn-a9d0049e-2
set security ipsec vpn AWS-H bind-interface st0.1
set security ipsec vpn AWS-H ike gateway AWS-gw-H
set security ipsec vpn AWS-H ike ipsec-policy AWS_IPSEC-H_policy
```

Almost done with the VPN configuration, one last command, to configure the tunnels to establish immediately upon committing the configuration (this is a Juniper Networks best practice recommendation):

```
set security ipsec vpn AWS-H establish-tunnels immediately
```

And another Juniper Networks best practice, set the maximum segment size:

```
set security flow tcp-mss ipsec-vpn mss 1379
```

Configure some basic screens on the untrust zone:

```
set security screen ids-option untrust-screen icmp ping-death
set security screen ids-option untrust-screen ip source-route-option
set security screen ids-option untrust-screen ip tear-drop
set security screen ids-option untrust-screen tcp syn-flood alarm-threshold 1024
set security screen ids-option untrust-screen tcp syn-flood attack-threshold 200
set security screen ids-option untrust-screen tcp syn-flood source-threshold 1024
set security screen ids-option untrust-screen tcp syn-flood destination-threshold 2048
set security screen ids-option untrust-screen tcp syn-flood timeout 20
set security screen ids-option untrust-screen tcp land
```

Apply the screen created into the previous step to the untrust zone:

```
set security zones security-zone untrust screen untrust-screen
```

Configure security policies:

```
set security policies from-zone trust to-zone trust policy default-permit match source-address any
set security policies from-zone trust to-zone trust policy default-permit match destination-address
any
set security policies from-zone trust to-zone trust policy default-permit match application any
set security policies from-zone trust to-zone trust policy default-permit then permit
set security policies from-zone trust to-zone trust policy intra-zone match source-address any
set security policies from-zone trust to-zone trust policy intra-zone match destination-address any
set security policies from-zone trust to-zone trust policy intra-zone match application any
set security policies from-zone trust to-zone trust policy intra-zone then permit
set security policies from-zone trust to-zone untrust policy default-permit match source-address any
set security policies from-zone trust to-zone untrust policy default-permit match destination-address
any
set security policies from-zone trust to-zone untrust policy default-permit match application any
set security policies from-zone trust to-zone untrust policy default-permit then permit
```

Configure the security zones, assign interfaces to the zones, and allow protocols inbound on the zone:

```
set security zones security-zone trust host-inbound-traffic system-services ping
set security zones security-zone trust host-inbound-traffic protocols bgp
set security zones security-zone trust interfaces st0.10
set security zones security-zone trust interfaces st0.11
set security zones security-zone trust interfaces st0.12
set security zones security-zone trust interfaces st0.13
set security zones security-zone trust interfaces st0.1
set security zones security-zone untrust host-inbound-traffic system-services ping
set security zones security-zone untrust host-inbound-traffic system-services ike
set security zones security-zone untrust interfaces ge-0/0/0.0
set security zones security-zone untrust interfaces ge-0/0/1.0
```

Configure the interfaces, including the VPN interfaces:

```
set interfaces ge-0/0/0 unit 0 description eni-6477bc35
set interfaces ge-0/0/0 unit 0 family inet address 10.100.10.25/28
set interfaces ge-0/0/1 unit 0 description eni-778b4126
set interfaces ge-0/0/1 unit 0 family inet address 10.100.10.36/28
set interfaces st0 unit 1 family inet address 10.0.251.10/24
set interfaces st0 unit 10 family inet mtu 1436
set interfaces st0 unit 10 family inet address 169.254.59.222/30
set interfaces st0 unit 11 family inet mtu 1436
set interfaces st0 unit 11 family inet address 169.254.59.62/30
set interfaces st0 unit 12 family inet mtu 1436
set interfaces st0 unit 12 family inet address 169.254.59.14/30
set interfaces st0 unit 13 family inet mtu 1436
set interfaces st0 unit 13 family inet address 169.254.57.162/30
```

Configure the routing policies:

```
set policy-options policy-statement AWS-policy term accept-aws from as-path AWS-orig
set policy-options policy-statement AWS-policy term accept-aws then accept
set policy-options policy-statement AWS-policy term all_else then reject
set policy-options policy-statement EXPORT-DEFAULT term default from route-filter 0.0.0.0/0 exact
set policy-options policy-statement EXPORT-DEFAULT term default then accept
set policy-options policy-statement EXPORT-DEFAULT term reject then reject
set policy-options as-path AWS-orig ".* 7224"
```

Create a routing instance:

```
set routing-instances AWS-data instance-type virtual-router
```

Assign all interfaces to the routing instance:

```
set routing-instances AWS-data interface ge-0/0/0.0
set routing-instances AWS-data interface ge-0/0/1.0
set routing-instances AWS-data interface st0.1
set routing-instances AWS-data interface st0.10
set routing-instances AWS-data interface st0.11
set routing-instances AWS-data interface st0.12
set routing-instances AWS-data interface st0.13
```

Configure static routes:

```
set routing-instances AWS-data routing-options static route 52.14.84.89/32 next-hop 10.100.10.33
set routing-instances AWS-data routing-options static route 52.15.221.249/32 next-hop 10.100.10.33
set routing-instances AWS-data routing-options static route 18.217.46.218/32 next-hop 10.100.10.49
set routing-instances AWS-data routing-options static route 52.15.95.96/32 next-hop 10.100.10.49
set routing-instances AWS-data routing-options static route 10.10.10.1/27 next-hop 10.100.10.17
```

Time to configure BGP. Figure 10.4 is shows at a high level how BGP will be configured. The goal is to run BGP over IPsec. We will have BGP connectivity to each spoke VPC from the Transit VPC (Hub) and a BGP connection between AWS and the data center. Note that there are two BGP sessions between the Transit VPC and the customer data center. Also, note in this recipe that we use a different AS per vSRX in AWS so that we can have ECMP.

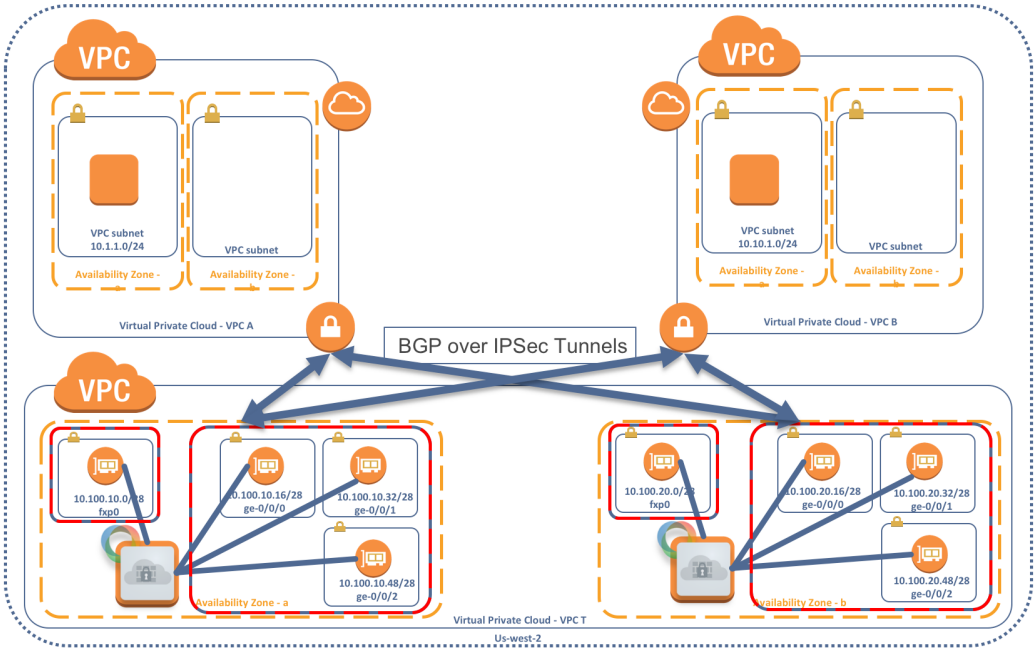

Figure 10.4 IPsec Over BGP Tunnels

```
set routing-instances AWS-data protocols bgp group ebgp type external
set routing-instances AWS-data protocols bgp group ebgp as-override
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.221 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.221 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.221 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.221 local-as 65010
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.61 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.61 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.61 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.61 local-as 65010
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.13 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.13 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.13 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.59.13 local-as 65010
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.161 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.161 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.161 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.161 local-as 65010
set routing-instances AWS-data protocols bgp group ebgp neighbor 10.0.251.20 peer-as 65002
set routing-instances AWS-data protocols bgp group ebgp neighbor 10.0.251.20 local-as 65010
```

You're done configuring the first vSRX in AWS for availability in zone A. Now let's configure the vSRX in availability zone B. Note the configuration is almost identical with the exception of the occasional IP address change and therefore only the key information is presented. You will notice the steps and parameters are almost all the same:

```
set groups aws-default system root-authentication ssh-rsa "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCdVIs/24eTTAJagm8QOhbkXvArqavsNldnKgAdhlJ6dD4Glp++D93ALVUvacETOFmOqjwU
```

```
8E8Yq2eCoxhJGhZ4wZQMGAOlskpqtQjsR0k/DTb9b+kzb3EKtyidvetlx63NMjr4xEzrwJ4xZqrDUZ7TCYKsfXNihzWDOOdm6cVA
Tru3KVo5sqdBYEHbVBZqi22MKlc1i01wY1dRGhEL0UZA4RAniX jnpr_aws_us-east-2_key"
set groups aws-default system services ssh no-passwords
set groups aws-default system services netconf ssh
set groups aws-default system services web-management https system-generated-certificate
set groups aws-default interfaces fxp0 unit 0 family inet address 10.100.20.7/28
set groups aws-default routing-options static route 0.0.0.0/0 next-hop 10.100.20.1
set apply-groups aws-default
set system host-name vSRX_B
set system root-authentication encrypted-password "$5$Bb9GDuiS$A/G4kyjz8vlc1dpzI7xBAehw.
aAysoMv7OruHw.mXX7"
set system name-server 169.254.169.253
set system syslog user * any emergency
set system syslog file messages any notice
set system syslog file messages authorization info
set system syslog file interactive-commands interactive-commands any
set system ntp server 162.210.111.4
set system ntp server 35.171.237.77
set security ike proposal ike-prop-vpn-abd0049c-1 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-abd0049c-1 dh-group group2
set security ike proposal ike-prop-vpn-abd0049c-1 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-abd0049c-1 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-abd0049c-1 lifetime-seconds 28800
set security ike proposal ike-prop-vpn-abd0049c-2 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-abd0049c-2 dh-group group2
set security ike proposal ike-prop-vpn-abd0049c-2 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-abd0049c-2 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-abd0049c-2 lifetime-seconds 28800
set security ike proposal ike-prop-vpn-a8d0049f-1 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-a8d0049f-1 dh-group group2
set security ike proposal ike-prop-vpn-a8d0049f-1 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-a8d0049f-1 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-a8d0049f-1 lifetime-seconds 28800
set security ike proposal ike-prop-vpn-a8d0049f-2 authentication-method pre-shared-keys
set security ike proposal ike-prop-vpn-a8d0049f-2 dh-group group2
set security ike proposal ike-prop-vpn-a8d0049f-2 authentication-algorithm sha1
set security ike proposal ike-prop-vpn-a8d0049f-2 encryption-algorithm aes-128-cbc
set security ike proposal ike-prop-vpn-a8d0049f-2 lifetime-seconds 28800
set security ike proposal AWS_IKE-H_proposal authentication-method pre-shared-keys
set security ike proposal AWS_IKE-H_proposal dh-group group2
set security ike proposal AWS_IKE-H_proposal authentication-algorithm sha-256
set security ike proposal AWS_IKE-H_proposal encryption-algorithm aes-256-cbc
set security ike proposal AWS_IKE-H_proposal lifetime-seconds 1800
set security ike policy ike-pol-vpn-abd0049c-1 mode main
set security ike policy ike-pol-vpn-abd0049c-1 proposals ike-prop-vpn-abd0049c-1
set security ike policy ike-pol-vpn-abd0049c-1 pre-shared-key ascii-text "$9$QNOTF6CBIEhSe-
VUHmPzFO1RhKMXx-wYoX7Ujq.TQhSrvX7NdboJGWLkPfQ9CbwY2Gj6/AIRSk.RSyl8LGDi.mTApBhclNd"
set security ike policy ike-pol-vpn-abd0049c-2 mode main
set security ike policy ike-pol-vpn-abd0049c-2 proposals ike-prop-vpn-abd0049c-2
set security ike policy ike-pol-vpn-abd0049c-2 pre-shared-key ascii-text "$9$/
l18tOIEcyKv836RclK8LZGDjm5/CtBRhM8s4JZjitp01IheKW87-eKoJUjq.TQFnCpEcyW8xEceWLxbwkqm536tpORclRh"
set security ike policy ike-pol-vpn-a8d0049f-1 mode main
set security ike policy ike-pol-vpn-a8d0049f-1 proposals ike-prop-vpn-a8d0049f-1
set security ike policy ike-pol-vpn-a8d0049f-1 pre-shared-key ascii-text
"$9$GdimfTz3p0IHqt0IEKvjHqmT3EcrvMLcywg4aUDqmfQzn/CtOBEApev8XbwzFn6CuRhyW8xrlNVbYJZ3n/9uBleMX7dtu"
set security ike policy ike-pol-vpn-a8d0049f-2 mode main
set security ike policy ike-pol-vpn-a8d0049f-2 proposals ike-prop-vpn-a8d0049f-2
set security ike policy ike-pol-vpn-a8d0049f-2 pre-shared-key ascii-text
```

```
"$9$dNsgJPfQ39pQzrKWLdVmfTQ69AtOcyKu0-bwsZG/9ApOIhSr8X7P5pOIRyrW8X-s2iHmfQFjiTF3/tpdVw2JUF36/A0Tz"
set security ike policy AWS-policy-H mode aggressive
set security ike policy AWS-policy-H proposals AWS_IKE-H_proposal
set security ike policy AWS-policy-H pre-shared-key ascii-text "$9$gCaGi6/tpORn/reMWx7ikq"
set security ike gateway gw-vpn-abd0049c-1 ike-policy ike-pol-vpn-abd0049c-1
set security ike gateway gw-vpn-abd0049c-1 address 18.221.49.182
set security ike gateway gw-vpn-abd0049c-1 dead-peer-detection interval 10
set security ike gateway gw-vpn-abd0049c-1 dead-peer-detection threshold 3
set security ike gateway gw-vpn-abd0049c-1 no-nat-traversal
set security ike gateway gw-vpn-abd0049c-1 external-interface ge-0/0/1.0
set security ike gateway gw-vpn-abd0049c-2 ike-policy ike-pol-vpn-abd0049c-2
set security ike gateway gw-vpn-abd0049c-2 address 52.15.112.224
set security ike gateway gw-vpn-abd0049c-2 dead-peer-detection interval 10
set security ike gateway gw-vpn-abd0049c-2 dead-peer-detection threshold 3
set security ike gateway gw-vpn-abd0049c-2 no-nat-traversal
set security ike gateway gw-vpn-abd0049c-2 external-interface ge-0/0/1.0
set security ike gateway gw-vpn-a8d0049f-1 ike-policy ike-pol-vpn-a8d0049f-1
set security ike gateway gw-vpn-a8d0049f-1 address 52.14.87.128
set security ike gateway gw-vpn-a8d0049f-1 dead-peer-detection interval 10
set security ike gateway gw-vpn-a8d0049f-1 dead-peer-detection threshold 3
set security ike gateway gw-vpn-a8d0049f-1 no-nat-traversal
set security ike gateway gw-vpn-a8d0049f-1 external-interface ge-0/0/1.0
set security ike gateway gw-vpn-a8d0049f-2 ike-policy ike-pol-vpn-a8d0049f-2
set security ike gateway gw-vpn-a8d0049f-2 address 52.15.136.211
set security ike gateway gw-vpn-a8d0049f-2 dead-peer-detection interval 10
set security ike gateway gw-vpn-a8d0049f-2 dead-peer-detection threshold 3
set security ike gateway gw-vpn-a8d0049f-2 no-nat-traversal
set security ike gateway gw-vpn-a8d0049f-2 external-interface ge-0/0/1.0
set security ike gateway AWS-gw-H ike-policy AWS-policy-H
set security ike gateway AWS-gw-H address 66.129.238.133
set security ike gateway AWS-gw-H local-identity user-at-hostname "aws-herndon2@juniper.net"
set security ike gateway AWS-gw-H remote-identity user-at-hostname "herndon2@juniper.net"
set security ike gateway AWS-gw-H external-interface ge-0/0/0
set security ipsec proposal ipsec-prop-vpn-abd0049c-1 protocol esp
set security ipsec proposal ipsec-prop-vpn-abd0049c-1 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-abd0049c-1 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-abd0049c-1 lifetime-seconds 3600
set security ipsec proposal ipsec-prop-vpn-abd0049c-2 protocol esp
set security ipsec proposal ipsec-prop-vpn-abd0049c-2 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-abd0049c-2 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-abd0049c-2 lifetime-seconds 3600
set security ipsec proposal ipsec-prop-vpn-a8d0049f-1 protocol esp
set security ipsec proposal ipsec-prop-vpn-a8d0049f-1 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-a8d0049f-1 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-a8d0049f-1 lifetime-seconds 3600
set security ipsec proposal ipsec-prop-vpn-a8d0049f-2 protocol esp
set security ipsec proposal ipsec-prop-vpn-a8d0049f-2 authentication-algorithm hmac-sha1-96
set security ipsec proposal ipsec-prop-vpn-a8d0049f-2 encryption-algorithm aes-128-cbc
set security ipsec proposal ipsec-prop-vpn-a8d0049f-2 lifetime-seconds 3600
set security ipsec proposal AWS_IPSEC-H protocol esp
set security ipsec proposal AWS_IPSEC-H authentication-algorithm hmac-sha1-96
set security ipsec proposal AWS_IPSEC-H encryption-algorithm aes-256-cbc
set security ipsec policy ipsec-pol-vpn-abd0049c-1 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-abd0049c-1 proposals ipsec-prop-vpn-abd0049c-1
set security ipsec policy ipsec-pol-vpn-abd0049c-2 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-abd0049c-2 proposals ipsec-prop-vpn-abd0049c-2
set security ipsec policy ipsec-pol-vpn-a8d0049f-1 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-a8d0049f-1 proposals ipsec-prop-vpn-a8d0049f-1
```

```
set security ipsec policy ipsec-pol-vpn-a8d0049f-2 perfect-forward-secrecy keys group2
set security ipsec policy ipsec-pol-vpn-a8d0049f-2 proposals ipsec-prop-vpn-a8d0049f-2
set security ipsec policy AWS_IPSEC-H_policy proposals AWS_IPSEC-H
set security ipsec vpn vpn-abd0049c-1 bind-interface st0.10
set security ipsec vpn vpn-abd0049c-1 df-bit clear
set security ipsec vpn vpn-abd0049c-1 ike gateway gw-vpn-abd0049c-1
set security ipsec vpn vpn-abd0049c-1 ike ipsec-policy ipsec-pol-vpn-abd0049c-1
set security ipsec vpn vpn-abd0049c-2 bind-interface st0.11
set security ipsec vpn vpn-abd0049c-2 df-bit clear
set security ipsec vpn vpn-abd0049c-2 ike gateway gw-vpn-abd0049c-2
set security ipsec vpn vpn-abd0049c-2 ike ipsec-policy ipsec-pol-vpn-abd0049c-2
set security ipsec vpn vpn-a8d0049f-1 bind-interface st0.12
set security ipsec vpn vpn-a8d0049f-1 df-bit clear
set security ipsec vpn vpn-a8d0049f-1 ike gateway gw-vpn-a8d0049f-1
set security ipsec vpn vpn-a8d0049f-1 ike ipsec-policy ipsec-pol-vpn-a8d0049f-1
set security ipsec vpn vpn-a8d0049f-2 bind-interface st0.13
set security ipsec vpn vpn-a8d0049f-2 df-bit clear
set security ipsec vpn vpn-a8d0049f-2 ike gateway gw-vpn-a8d0049f-2
set security ipsec vpn vpn-a8d0049f-2 ike ipsec-policy ipsec-pol-vpn-a8d0049f-2
set security ipsec vpn AWS-H bind-interface st0.1
set security ipsec vpn AWS-H ike gateway AWS-gw-H
set security ipsec vpn AWS-H ike ipsec-policy AWS_IPSEC-H_policy
set security ipsec vpn AWS-H establish-tunnels immediately
set security flow tcp-mss ipsec-vpn mss 1379
set security screen ids-option untrust-screen icmp ping-death
set security screen ids-option untrust-screen ip source-route-option
set security screen ids-option untrust-screen ip tear-drop
set security screen ids-option untrust-screen tcp syn-flood alarm-threshold 1024
set security screen ids-option untrust-screen tcp syn-flood attack-threshold 200
set security screen ids-option untrust-screen tcp syn-flood source-threshold 1024
set security screen ids-option untrust-screen tcp syn-flood destination-threshold 2048
set security screen ids-option untrust-screen tcp syn-flood timeout 20
set security screen ids-option untrust-screen tcp land
set security policies from-zone trust to-zone trust policy default-permit match source-address any
set security policies from-zone trust to-zone trust policy default-permit match destination-address
any
set security policies from-zone trust to-zone trust policy default-permit match application any
set security policies from-zone trust to-zone trust policy default-permit then permit
set security policies from-zone trust to-zone trust policy intra-zone match source-address any
set security policies from-zone trust to-zone trust policy intra-zone match destination-address any
set security policies from-zone trust to-zone trust policy intra-zone match application any
set security policies from-zone trust to-zone trust policy intra-zone then permit
set security policies from-zone trust to-zone untrust policy default-permit match source-address any
set security policies from-zone trust to-zone untrust policy default-permit match destination-address
any
set security policies from-zone trust to-zone untrust policy default-permit match application any
set security policies from-zone trust to-zone untrust policy default-permit then permit
set security zones security-zone trust tcp-rst
set security zones security-zone trust host-inbound-traffic system-services ping
set security zones security-zone trust host-inbound-traffic protocols bgp
set security zones security-zone trust interfaces st0.10
set security zones security-zone trust interfaces st0.11
set security zones security-zone trust interfaces st0.12
set security zones security-zone trust interfaces st0.13
set security zones security-zone trust interfaces st0.1
set security zones security-zone untrust screen untrust-screen
set security zones security-zone untrust host-inbound-traffic system-services ping
set security zones security-zone untrust host-inbound-traffic system-services ike
```

```
set security zones security-zone untrust interfaces ge-0/0/0.0
set security zones security-zone untrust interfaces ge-0/0/1.0
set interfaces ge-0/0/0 unit 0 description eni-67581533
set interfaces ge-0/0/0 unit 0 family inet address 10.100.20.30/28
set interfaces ge-0/0/1 unit 0 description eni-e95419bd
set interfaces ge-0/0/1 unit 0 family inet address 10.100.20.43/28
set interfaces st0 unit 1 family inet address 10.0.200.1/30
set interfaces st0 unit 10 family inet mtu 1436
set interfaces st0 unit 10 family inet address 169.254.57.246/30
set interfaces st0 unit 11 family inet mtu 1436
set interfaces st0 unit 11 family inet address 169.254.58.142/30
set interfaces st0 unit 12 family inet mtu 1436
set interfaces st0 unit 12 family inet address 169.254.56.174/30
set interfaces st0 unit 13 family inet mtu 1436
set interfaces st0 unit 13 family inet address 169.254.57.122/30
set policy-options policy-statement AWS-policy term accept-aws from as-path AWS-orig
set policy-options policy-statement AWS-policy term accept-aws then accept
set policy-options policy-statement AWS-policy term all_else then reject
set policy-options policy-statement EXPORT-DEFAULT term default from route-filter 0.0.0.0/0 exact
set policy-options policy-statement EXPORT-DEFAULT term default then accept
set policy-options policy-statement EXPORT-DEFAULT term reject then reject
set policy-options as-path AWS-orig ".* 7224"
set routing-instances AWS-data instance-type virtual-router
set routing-instances AWS-data interface ge-0/0/0.0
set routing-instances AWS-data interface ge-0/0/1.0
set routing-instances AWS-data interface st0.1
set routing-instances AWS-data interface st0.10
set routing-instances AWS-data interface st0.11
set routing-instances AWS-data interface st0.12
set routing-instances AWS-data interface st0.13
set routing-instances AWS-data routing-options static route 18.221.49.182/32 next-hop 10.100.20.33
set routing-instances AWS-data routing-options static route 52.15.112.224/32 next-hop 10.100.20.33
set routing-instances AWS-data routing-options static route 52.14.87.128/32 next-hop 10.100.20.49
set routing-instances AWS-data routing-options static route 52.15.136.211/32 next-hop 10.100.20.49
set routing-instances AWS-data routing-options static route 66.129.238.133/32 next-hop 10.100.20.17
set routing-instances AWS-data protocols bgp group ebgp type external
set routing-instances AWS-data protocols bgp group ebgp as-override
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.245 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.245 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.245 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.245 local-as 65020
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.58.141 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.58.141 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.58.141 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.58.141 local-as 65020
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.56.173 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.56.173 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.56.173 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.56.173 local-as 65020
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.121 hold-time 30
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.121 import AWS-policy
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.121 peer-as 7224
set routing-instances AWS-data protocols bgp group ebgp neighbor 169.254.57.121 local-as 65020
set routing-instances AWS-data protocols bgp group ebgp neighbor 10.0.200.2 peer-as 65002
set routing-instances AWS-data protocols bgp group ebgp neighbor 10.0.200.2 local-as 65020
```

Discussion

This recipe presented a scenario of extending a data center into the AWS cloud using an IPsec tunnel across the Internet. There are various permutations of the design that can be implemented, while still fundamentally preserving the intent of the recipe provided in this section. For example, instead of using a connection across the Internet, the data center can be extended into the public cloud using AWS Direct Connect. Also, two different vSRXs were configured in AWS in two different availability zones to create a High Availability design in the Transit VPC. If High Availability is not required, then a single vSRX can be deployed within the Transit VPC. In addition, a limited amount of spoke configuration was shown in this recipe within AWS. You could easily scale out the number of spoke VPCs connected to the Transit VPC. The recipe presented can be very flexible to scale and meet your individual requirements.

The BGP configuration presented within this recipe could potentially be altered. Another possible scenario could be instead of having each vSRX in its own AS within the AWS Transit VPN, both vSRXs could have been configured to be in the same AS.

Finally, another permutation of this recipe is having separate individual interfaces on the vSRX within the Transit VPC to connect to spoke VPCs. Figure 10.5 shows you how this type of design might look and how you would use a separate individual interface such as ge-0/0/1, ge-0/0/2, ge-0/0/3, etc., to connect to the separate individual spoke VPCs. This is in contrast to the recipe you just read, which used a single interface to connect to all the separate spoke VPCs.

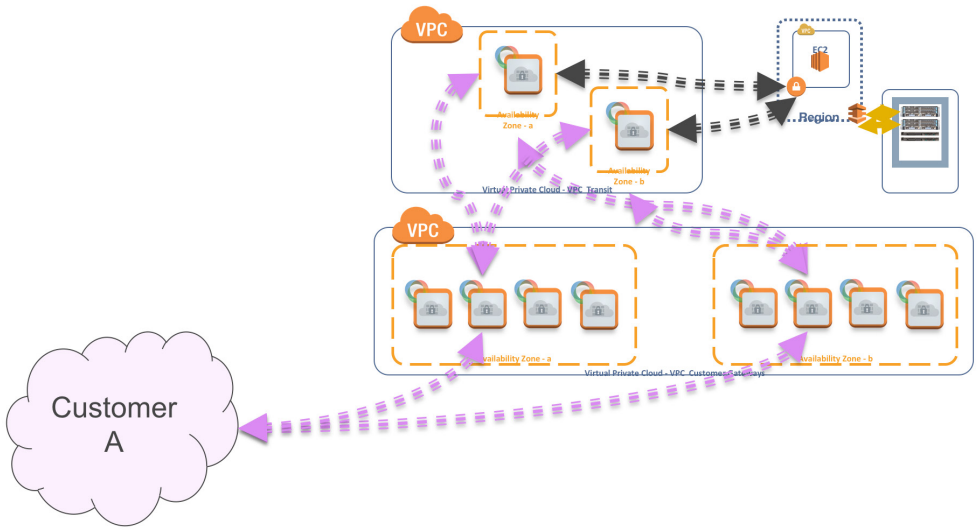

Figure 10.5 *Using Separate Individual Interfaces on the vSRX*

Figure 10.5's scenario has a limitation in the number of spoke VPCs you can have. It's tied directly to the number of interfaces available on the vSRX and therefore can pose a number of limitations in its implementation. Therefore, while this specific scenario is not recommended, it simply shows another design approach that can be taken if required. One use case of having individual separate interfaces connecting to each Spoke VPC is from a security point of view. However, on smaller instances it would limit how many places you can connect to.

And extending the discussion even further, Figure 10.6 illustrates yet another possible use case and scenario in which you can utilize Transit VPC design for extending a data center. It's an area not discussed so far and concerns how to handle the AWS account. Figure 10.6 shows one use case in which a customer could potentially create an account on a per-connection with a VPC per-region that will host the customer vSRX. It's a design in which Customer A is only connected to a pair of vSRXs.

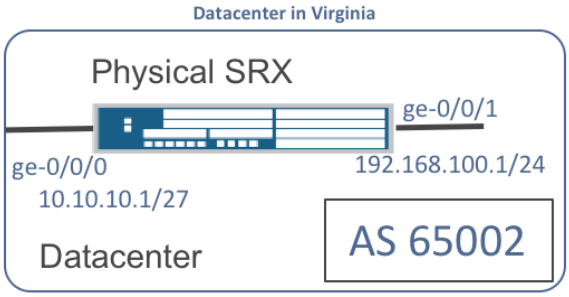

Figure 10.6 *Customer A Connected to a Pair of vSRXs*

The configuration within this recipe was manually created within the AWS environment. However, note that the configuration can also be generated automatically within AWS and downloaded. The manual approach was presented so that a more detailed explanation and an illustrative approach could be taken. AWS is constantly making changes that make it easier to generate configurations and deploy vSRX firewalls within AWS, and therefore subject to change much faster than this cookbook can update. So please, take this into account and use the best method applicable for your requirements.

Conclusion

This recipe securely extended a data center to the AWS public cloud by making use of Transit VPCs that add capability and flexibility to the VPCs. It is easy to configure Juniper vSRX and SRX firewalls to extend a data center into the AWS public cloud securely. The configurations that are applied to the vSRX in AWS and the physical SRX to AWS share similar configurations such as route-based VPNs and BGP configuration. While a transit VPC was configured with two spoke VPCs, this can easily be increased to a number of spoke VPCs by simply adding additional VPN and BGP configurations to the links, providing extensive flexibility in design implementations.

A Note About Professional Services

You might want to leverage Juniper Professional Services (PS) as a rapid and cost-effective way to implement any of this cookbook's recipes and solutions, or tailor them, or new ones, to your advantage. Professional Services is certified at meeting Juniper's stringent requirements for providing vMX and vSRX consulting services, and can enable rapid adoption allowing your development teams to focus on your core competencies.

If you are under pressure to rapidly deliver results, Professional Services can get a public, private, or multicloud environment up and running with minimal overhead and impact to your operations teams. All services are performed by highly-skilled engineers with a combination of remote and on-site delivery histories. Consider:

- PS is certified at meeting Juniper's stringent requirements for providing vMX and vSRX consulting services.
- PS gathers your deployment parameters for successful deployment and configuration of the AWS architecture.
- Based on your approved implementation plan, the deployment and configuration will be followed by a verification cycle that all is functioning as planned.
- PS knowledge transfer to you and your organization takes place throughout the engagement with your appropriate personnel (it requires continuous participation to gain maximum value).
- PS will identify corresponding use cases and services that are relevant to your environment and your needs, as a follow-up activity to the engagement.
- Juniper Networks provides services on a global scale.

For more details, visit: https://www.juniper.net/us/en/products-services/services/technical-services/professional-services/.

In addition to customized engagements, examples of predefined engagements and their descriptions can be found at https://www.juniper.net/us/en/solutions/pcm/.

Contact your Juniper Networks account manager to find out how Juniper Networks PS can help you advance your business and technology challenges. Don't have a Juniper account manager? Start here: https://www.juniper.net/us/en/how-to-buy/form/.