Network and Security Automation Workshop

This repository contains an entire class on creating network automation. It utilizes vSRX and the NetDevOps VM (an Ubuntu development environment). In the course of the lab you will build a multi-node vSRX topology connecting to a single headend.

Course Steps

The course is divided into multiple steps.

Overview

1. Getting a copy of the lab
2. Topology Overview
3. vSRX Topology
4. vSRX Headend Topology
5. Connecting to your VMs
6. Software Overview
7. Installing Software
8. Using Python and PyEZ Library
9. Using Ansible
10. Basic API Review
11. NETCONF Magic

Hands On Lab

1. Configuring NAT
    1. Configuring NAT with NETCONF (Example)
    2. Configuring NAT with Ansible
2. Basic Firewall Policies
3. VPN connection to headend
4. Enabling Dynamic Routing
5. Creating VPN Firewall Policies

6. Automating Licenses (skip if you dont have licenses)
7. Creating Application Policies (skip if you dont have licenses)
8. Creating IPS Policies (skip if you dont have licenses)
9. Disaster Strikes! (add disaster creating steps)
10. Recovering the Lab
    - Recovering the lab with licenses
    - Recovering the lab without licenses
11. Reviewing the lab

VM Access Information

- VM Host Passwords

Proctor Instructions

- Classroom Handout

TODO

1. Support VMware Workstation, Fusion, and Virtual Box
    - Add in VMware instructions

Installing the Lab Repository

The lab is a set of files and directories that are hosted on github. This allows the lab to be completely open for everyone to participate in creation of the lab. While this may seem like a new concept, github is the new was that developers share ideas and code to make the world a better place.

Installing Git on your system

Installing Git on your host will depend on the version of the operating system that you are running.

The simplest thing to do would be to install the GitHub tool for either Windows or Mac.

- Mac GitHub Tool
- Windows GitHub Tool

However if you wish to install the git tool suite

- Install Full Git

Windows Install

On Windows there is a challenge when it comes to working with Vagrant. Vagrant ideally needs access to an ssh client. Windows does not come with an SSH client. However if you install the Git package with the default settings you will be able to use ssh via its built in BASH shell. Installing this will be the easiest method to get all of the value out of Vagrant.

- Installing Git + SSH

Cloning the Repository

Ideally you want to use git) to download the the repository. This provides a few key benefits over downloading it as a zip file.

- The ability to review the history of changes to the lab
- Make changes and save them to your own branch
- Easily pull new updates to the lab as they are created
- Learn git

To clone the repository you can use one of the following command sets:

- (SSH) git clone git@github.com:JNPRAutomate/JNPRAutomateDemo-Class.git
- (HTTPS) git clone https://github.com/JNPRAutomate/JNPRAutomateDemo-Class.git

Downloading the Repository

If you don't want to deal with getting git just yet you can also download the repository as a zip file. This way you can get started without git.

- Zip File of Current Repository

Once the Zip file is downloaded please open it.

Learning More about Git

Git is an amazing piece of software. There is so much to learn how git works and all of the available tips and tricks. Here are some great learning resources if you wish to know more about Git.

- Git Intro
- Git Book
- Git Docs
- Git Videos
- Git Resources

- Git GUI Clients
- libgit2 bindings

Entering the repository

Once you have cloned or downloaded the repository you will have the minimum requirements to get started. All of the required information to run the lab is contained within the repository. Any commands referenced within the lab such as using vagrant are done within the repository.

Lab Hardware requirements

- Computer running Windows, Mac or Linux
- 8GB Ram and 12GB Disk Space
- Dual or quad core CPU
- Vagrant
- VirtualBox
- Student Vagrant VMs
- vSRX Host
- NetDevOps Ubuntu 14.04

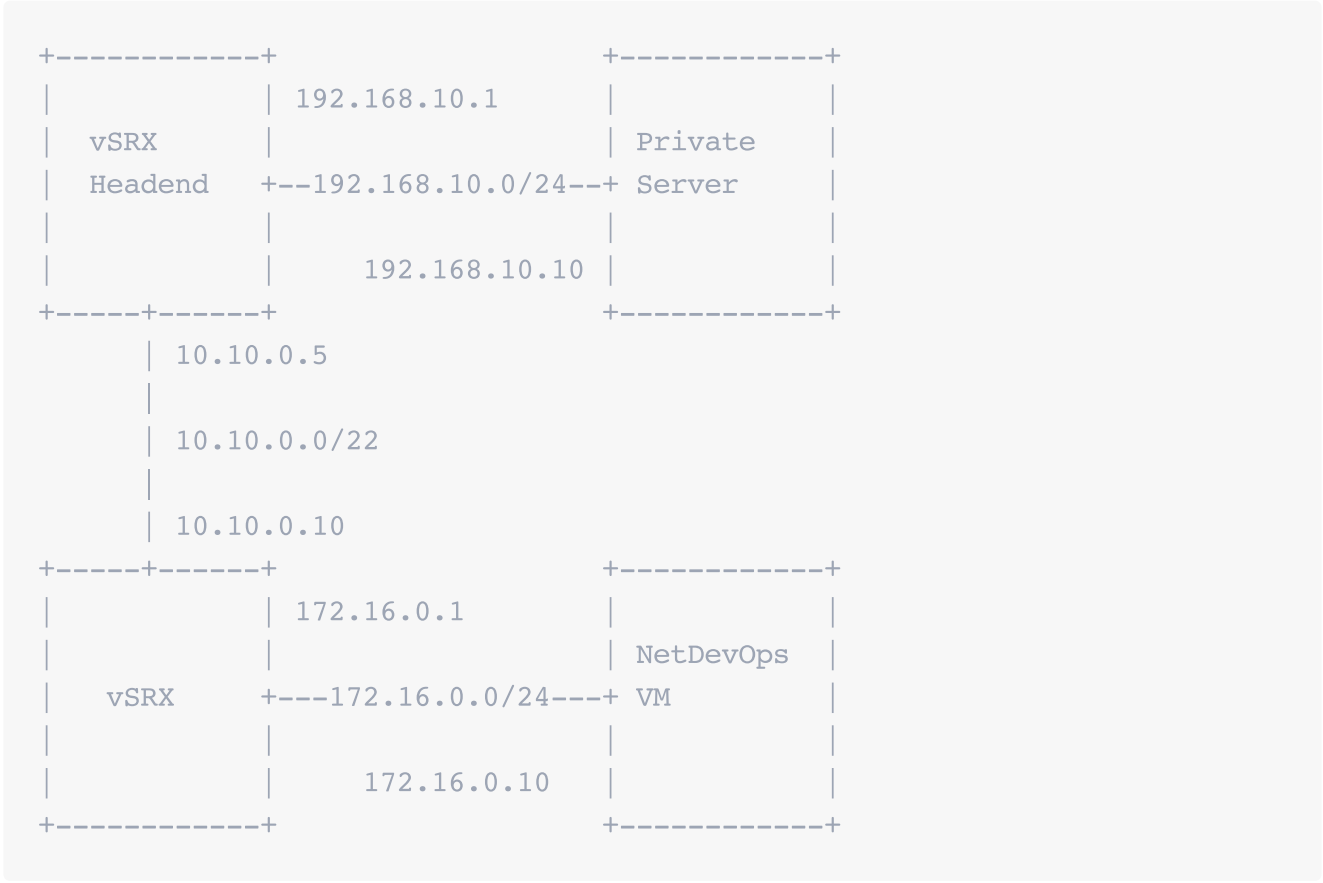
For RAM and CPU more is always better!

Topology

The lab IP Address ranges are broken out for each section of the lab to be visually different than any other part. The benefit here is that it is very clear to you what part of the lab that you are using. After staring at IP addresses for a while they tend to blend together and this can cause confusion.

LAB IP Subnets

- Shared LAN
  - 10.10.0.0/22
  - This is the network used between the vSRX and the vSRX headend
- Student Lab
  - 172.16.0.0/24
  - NAT will be used when communicating outside of student vSRX
- Private Server
  - 192.168.10.0/24
  - This portion of the lab will only be accessible via a VPN or NAT
  - This ensures that you know you have correctly configured your device if you can access this part of the network



Student Side

Each student is expected to have their own laptop to run the lab on. The tools used are supported on Windows, Mac OS X, and Linux.

Using Vagrant the topology below will be created on your laptop.

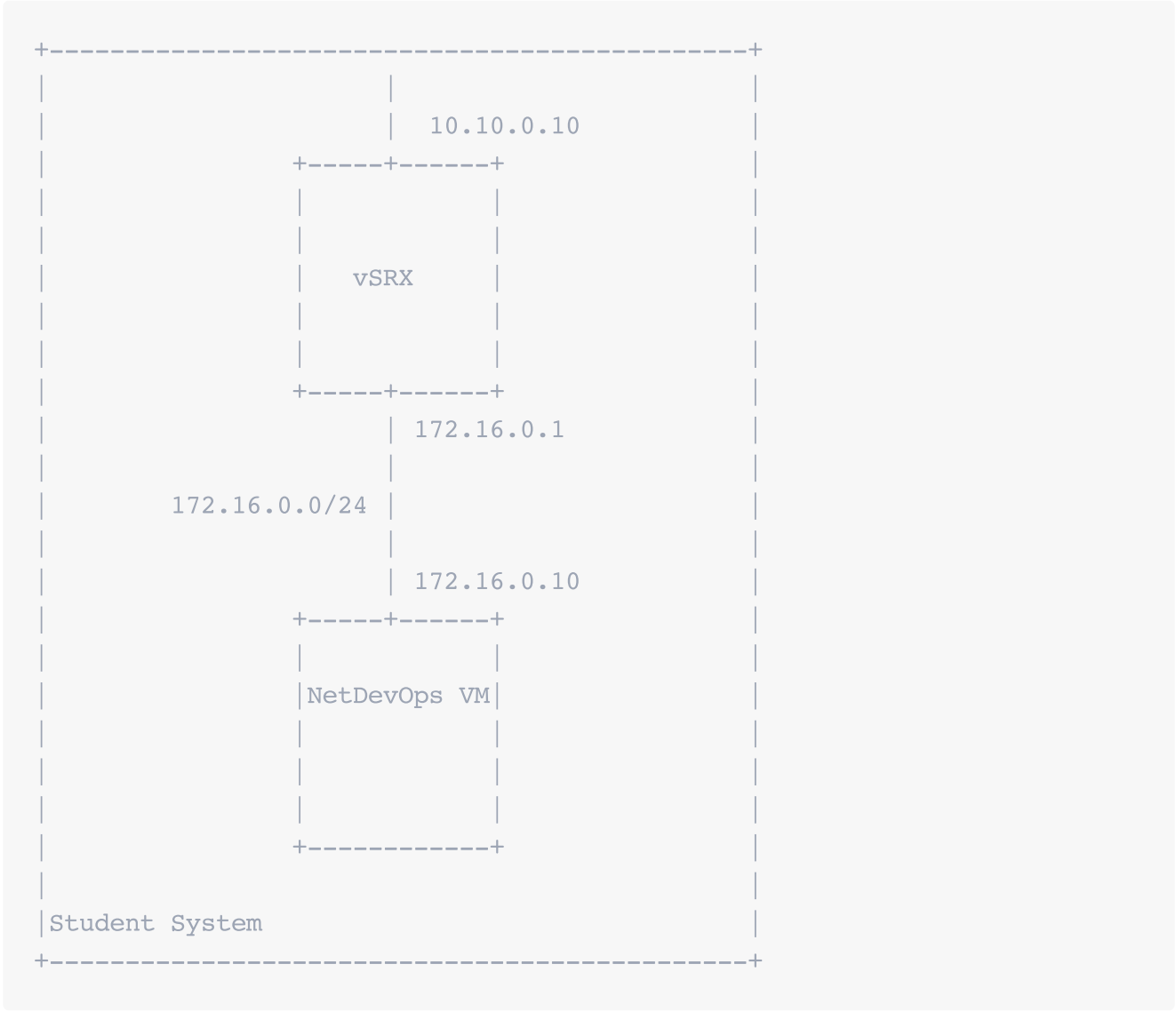
- NetDevOps VM

The NetDevOps VM is a preconfigured Ubuntu (or optionally CentOS) VM that comes preloaded with all of the required libraries and tools needed to operate in the lab. The goal is to get the student started without needing to deal with the installation of all of the prerequisites. The VM can also be used after the class to continue as a development environment for automation.

The VM sits behind a vSRX instance that it uses as a default gateway. This way you can have your own personal Junos device to configure and manage.

- vSRX VM

The vSRX VM is based upon Junos version 12.1X47-D20. This version gives you a wide variety of features to utilize and automate against.



Headend

The headend runs on the same host as the student side of the VM. The topology is simplified so this can be run on one single host.

- vSRX Headend
  - This device will act as the headend for all IPsec terminations
  - It will also secure access to the private server
- NetDevOps Private Server
  - This server provides services that are only accessible over NAT or a VPN connection behind the vSRX headend
  - This server should only be accessible if the student has correctly configured steps of their labs
  - Services
    - Web Server
    - DNS Server

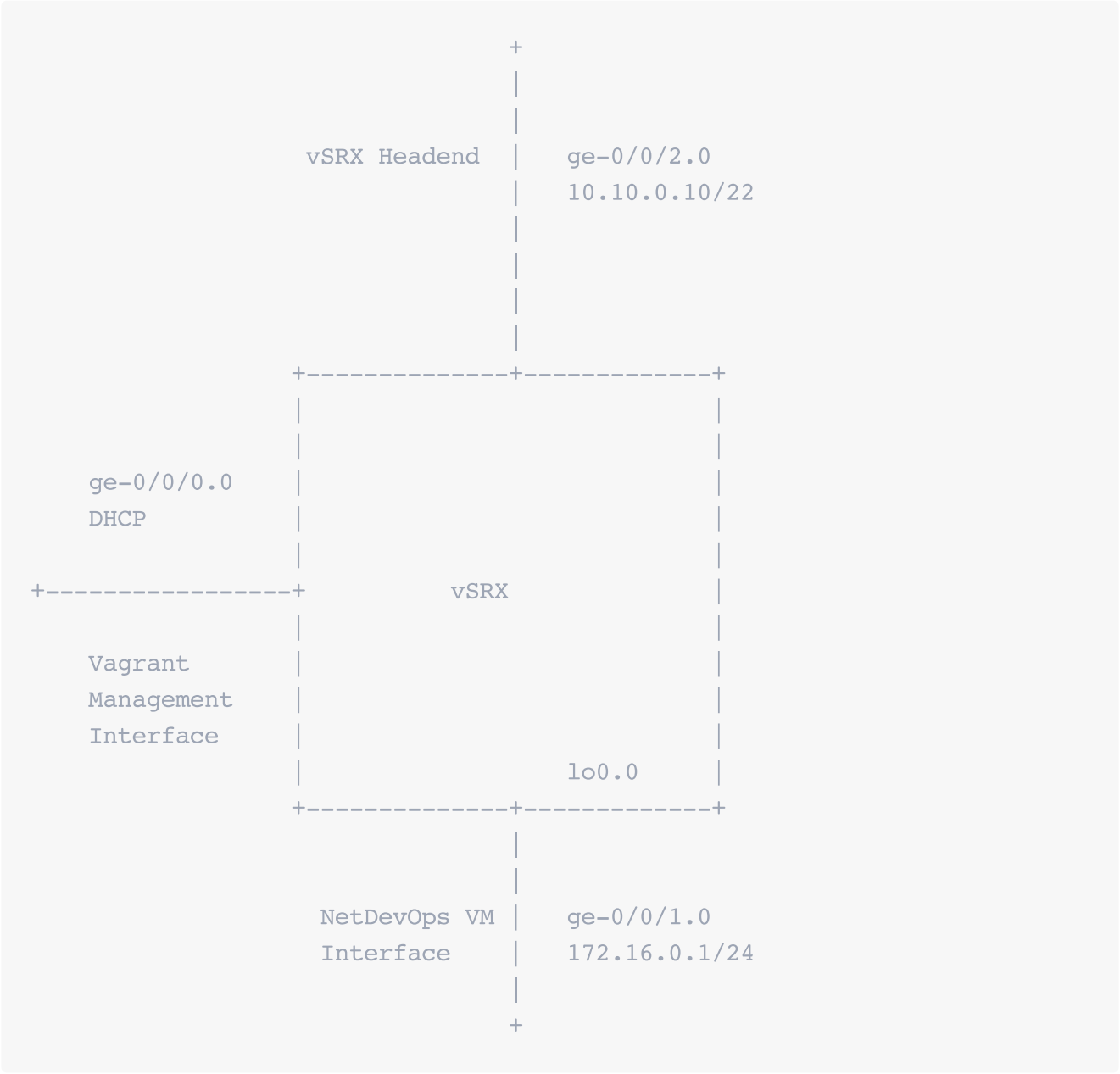

Headend
NetDevOps
Private
Server
192.168.10.10
192.168.10.0/24
192.168.10.1
vSRX
Headend
10.10.0.5

Student vSRX Interface Topology

The vSRX used in the lab has three physical interfaces in the topology.

- ge-0/0/0.0
  - This interface is used by Vagrant to talk to the device
  - When using commands such as "vagrant ssh" this is the Interface it connects to
  - The interface is configured in a management zone and is not for transit
  - Uses DHCP for IP address allocation
- ge-0/0/1.0
  - Connected to the NetDevOps VM
  - Has a static IP address of 172.16.0.1/24
  - When needed to connect to the vSRX from the NetDevOps VM this is the interface and IP to use
- ge-0/0/2.0
  - This interface connects to the headend vSRX
  - Has a static IP address of 10.10.0.10/22

Diagram

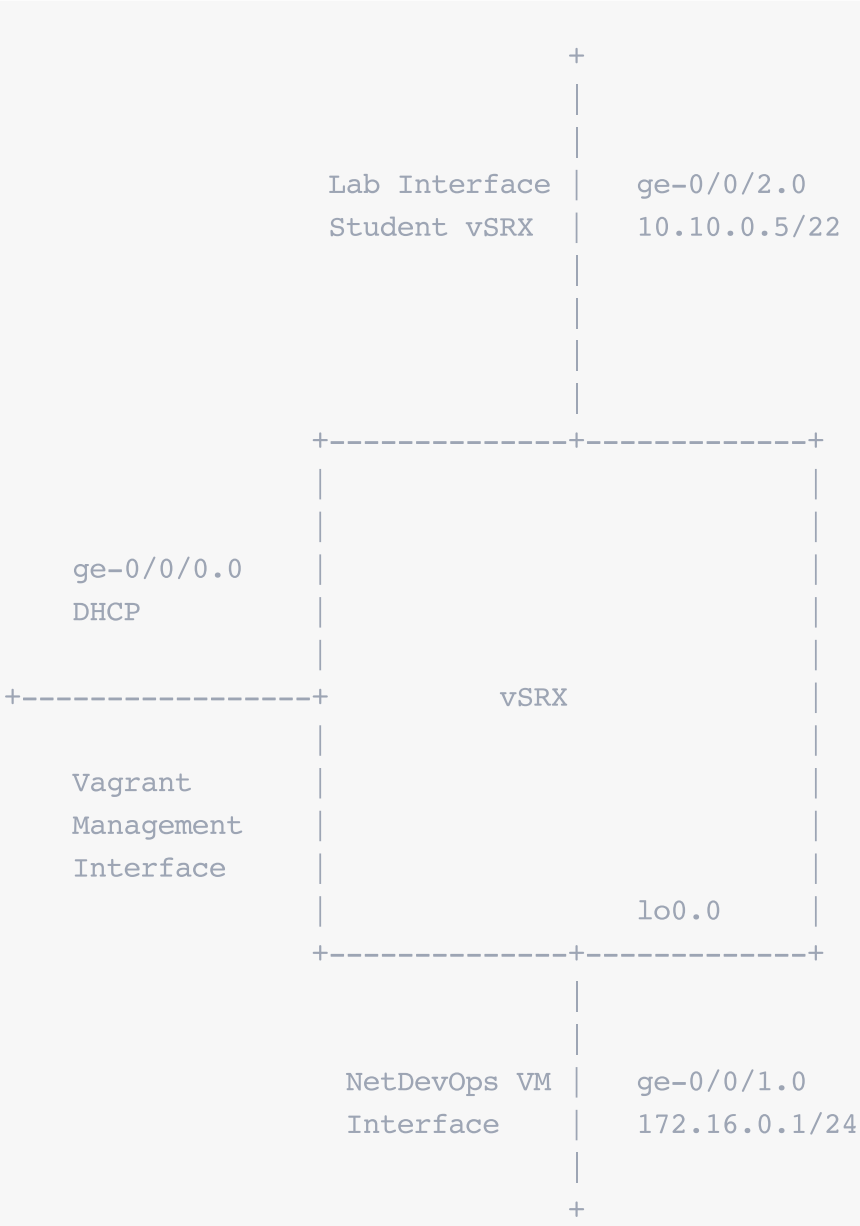

vSRX Headend
ge-0/0/2.0
10.10.0.10/22
ge-0/0/0.0
DHCP
vSRX
Vagrant
Management
Interface
lo0.0
NetDevOps VM
Interface
ge-0/0/1.0
172.16.0.1/24

Headend vSRX Interface Topology

The vSRX used in the lab has three physical interfaces in the topology.

- ge-0/0/0.0
  - This interface is used by Vagrant to talk to the device
  - When using commands such as "vagrant ssh" this is the Interface it connects to
  - The interface is configured in a management zone and is not for transit
  - Uses DHCP for IP address allocation
- ge-0/0/1.0
  - Connected to the Private server
  - Has a static IP address of 192.168.10.1/24
- ge-0/0/2.0
  - This interface connects to the student vSRX
  - Has a static IP address of 10.10.0.5/22

Diagram


Lab Interface
Student vSRX
ge-0/0/2.0
10.10.0.5/22
ge-0/0/0.0
DHCP
vSRX
Vagrant
Management
Interface
lo0.0
NetDevOps VM
Interface
ge-0/0/1.0
172.16.0.1/24

Connecting to your VMs

We have focused around discussing the values of in managing a topology. But Vagrant also simplifies connecting to the virtual machines as well. When a virtual machine is booted using Vagrant there is a special management interface on each VM. This provides access from Vagrant to the VM, Internet access for the VM, and it can also allow for port forwarding for other services on the VM such as a web server.

When each VM is booted Vagrant attempts to connect to it via SSH. Once booted Vagrant can run all of the various provisioning steps that are associated with the VM. When the VM is done booting the user can access each host with the command "vagrant ssh HOSTNAME". For our labs all of the commands that we will run will be via the command line which is accessed via ssh. On Mac OS X and Linux SSH is a default installed binary, however this is not the case for Windows. Below explains the easiest method for using SSH on Windows. You can choose to use another method but for all of the documentation we will assume you will be using the Git bash shel.

SSH on Windows

On Windows there is a challenge when it comes to working with Vagrant. Vagrant ideally needs access to an ssh client. Windows does not come with an SSH client. However if you install the Git package with the default settings you will be able to use ssh via its built in BASH shell. Installing this will be the easiest method to get all of the value out of Vagrant.

- Installing Git + SSH

VirtualBox

VirtualBox is an open source and free virtualization software platform for Windows, Mac, Linux, and other operating systems. It's a great fit for Vagrant and development environments because it's free and consistent in usage. For networking, it's also a great fit, as you can name the virtual network segments. VMware is faster for desktop virtualization, but costs money, has two separate versions (Workstation or Fusion) that operate a little differently, and requires a paid plugin to work with Vagrant.

Vagrant

Vagrant is a tool that takes the pain out of creating and using development environments. In the past, users used their own PC, procured a server, or used a virtual machine on their desktop or a server. Over the last decade, the virtualization revolution has dictated that these are most-commonly virtual machines (VMs). Using VMs leverages the modern hardware on your PC that could duplicate the power of a rack of computers from years ago, but let's look at the few of things that Vagrant helps with:

- **Consistent**: VMs are snapshot installs of an installed OS, but Vagrant can take this to the next level across complex environments of multiple VMs. Your environment and mine will always be the same.
- **Resilient**: your development environment can now be infinitely reinstalled. Of course, you can reinstall a VM, but with Vagrant, you type `vagrant destroy`, and `vagrant up`. If your data resides somewhere else, Vagrant can use shared folders on your host machine to keep working data unaffected.
- **Portable**: your Vagrant Box (a VM that's been "Vagrantized") can be shared with anyone via a URL or Atlas. You can also give someone a Vagrantfile, which is a simple text file, and they can type `vagrant up` to launch your environment!
- **Shareable**: You can share your running instance with another user with Vagrant Share. This is great for classes and troubleshooting.

PyEZ Library

PyEZ (TechWiki)(GitHub)(PyPI) is Juniper's premiere NETCONF-based automation

library. Originally authored by Junos Automation Hall of Famer Jeremy Schulman, PyEZ (aka junos-eznc for "EZ" NETCONF) is now curated by another automation rockstar, Rick Sherman. With thousands of downloads a month, the Python library is definitely the hottest open source network automation library out there!

Before PyEZ (and our other EZ libraries), network automation usually meant programming, but PyEZ helps make this consumable for non-programmers.

Python as a Power Shell techwiki page:

> This means that non-programmers, for example, the Network Engineer, can use the native Python shell on their management server (laptop, tablet, phone, and so on) as their point-of-control for remotely managing Junos OS devices. The Python shell is an interactive environment that provides the necessary means to perform common automation tasks, such as conditional testing, for-loops, macros, and templates. These building blocks are similar enough to other "shell" environments, like Bash, to enable the non-programmer to use the Python shell as a power tool, instead of a programming language. From the Python shell, a user can manage Junos OS devices using native hash tables, arrays, and so on, instead of using device-specific Junos OS XML or resorting to "screen scraping" the actual Junos OS CLI.

Most network vendors don't have truly open libraries that are free as in freedom, vs. free as in beer. You might have to sign up on a webpage, fill out a form, or actually buy something. Juniper's commitment to real open source projects and tooling is best presented with our GitHub pages (Juniper) (JNPRAutomate).

Ansible

You hear a lot about Puppet, Chef, Ansible, and Salt, but Ansible definitely wins the award for easiest configuration management framework to learn. That doesn't mean it's less powerful, as it's currently powering some of the largest clouds in the world. Ansible's strengths for us:

- rapid configuration of multiple devices (up to thousands), simultaneously over SSH
- easy configuration through use of YAML for describing your environment
- easy templating with Jinja2 to dynamically

- easy operation with CLI command suite Swiss army knife tools, `ansible` and `ansible-playbook`
- easy module installation with `ansible-galaxy`

Did we mention it's easy? Ansible will make your day-to-day job easier, with immediately beneficial network automation prowess. After checking out Ansible, you'll be that much more prepared to learn Puppet, Chef, and Salt and see if they're a better fit for you.

Software Prerequisites

To run the lab you need a few applications loaded on your laptop.

These tools are Vagrant and VirtualBox. This tool set is very commonly used in a development environment. It makes common tasks such as bringing up a VM, provisioning it, and building multi-box topologies a breeze.

Vagrant

Vagrant is a virtual machine management tool. It manages the downloading, provisioning, and state management of the virtual machine. It is a common tool used in software development.

Download Locations

- Mac OS X
- Windows
- Linux 64-bit DEB
- Linux 32-bit DEB
- Linux 64-bit RPM
- Linux 32-bit RPM

Installation Instructions

Please download and install the correct versions

Install this executable on your operating system of choice just as you would any software for that platform. On Windows, you will need to reboot, but you can wait until after VirtualBox is installed.

Vagrant Plugins

To provide additional features into Vagrant we must first install a few additional plugins for Vagrant. The first plugin "vagrant-junos" is a driver to help configure and control Junos. The second plugin, "vagrant-host-shell", is used to assist in the configuration of Junos.

The vSRX boxes require two plugins, namely:

- vagrant-junos
- vagrant-host-shell

These can be installed with the following commands (launch a command prompt on Windows (`cmd`), or a Terminal session on Mac (located in your Applications/Utilities folder):

Installation Instructions

This will fetch the latest version of the vagrant plugins from the Internet. Both plugins are less than 1MB in size so the installation should be simple.

```
vagrant plugin install vagrant-junos
vagrant plugin install vagrant-host-shell
```

To check for updates in the future, a simple `vagrant plugin update` will ensure everything is up-to-date.

VirtualBox

For the lab we will be using VirtualBox as our virtualization manager. This tool is free to use and most importantly it is free to use with Vagrant. It also offers some support that VMware Workstation or Fusion do not offer. This includes support for the VirtIO driver and the ability to more simply manage virtual networks.

When installing on Windows it will typically install a network driver. It will prompt you to click "Continue" to take this action. Please hit continue to install the driver.

- Mac OS X
- Windows
- Linux 64-bit DEB
- Linux 32-bit DEB
- Other Linux

Installation Instructions

Please download and install Virtual Box before continuing.

> Windows users: you will need to reboot your laptop before Vagrant and VirtualBox will both work.

VMware

TODO

Installing Boxes

> Normally, Vagrant boxes are added with `vagrant box add vendor/name` from the web, but for the size of this class, we need to install them locally. You will still be able to type `vagrant box outdated` to get new versions of these boxes in the future.
>
> NOTE: Vagrant stores boxes in `$HOME/vagrant.d` on Linux & Mac, or `%userprofile%\.vagrant.d\boxes` on Windows

Vagrant boxes can be installed with the following commands (launch a command prompt on Windows (`cmd`), or a Terminal session on Mac (located in your Applications/Utilities folder):

Installation Instructions

When you attempt to bring up the virtual machines for the first time Vagrant will automatically download the correct boxes for you. However you may want to run the class on an airplane or a location with unreliable Internet access. If that is the case then you can also pre download the machines using the vagrant command.

```
vagrant box add juniper/netdevops-ubuntu1404-headless
vagrant box add juniper/ffp-12.1X47-D20.7
```

Validating Install

- `vagrant version` should tell you that you're running 1.7.3, aka the latest and greatest
- `vagrant plugin list` should include:
  - `vagrant-host-shell` 0.0.4 or newer
  - `vagrant-junos` 0.2.0 or newer

Vagrant Global Status

With the workflow of vagrant it is possible to have many virtual machines running without you being aware of this happening. The only indication of this being a problem is your laptop will begin to run very slowly. To see if you have vagrant hosts running you can use the command "vagrant global-status"

```
[rcameron:~/code/JNPRAutomateDemo-Class] master(+31/-0) ± vagrant global-st
id       name       provider      state       directory
---------------------------------------------------------------------------
86a431a  default    vmware_fusion not running /Users/rcameron/vagrant/ffp-1
4407dc1  trusty1    virtualbox    saved       /Users/rcameron/vagrant/3host
55b10f6  default    virtualbox    saved       /Users/rcameron/vagrant/happy
7879af8  default    virtualbox    saved       /Users/rcameron/vagrant/preci
b67e217  default    virtualbox    saved       /Users/rcameron/vagrant/ffppm
2c59020  ndo        virtualbox    running     /Users/rcameron/code/JNPRAuto
ec8e606  srx        virtualbox    running     /Users/rcameron/code/JNPRAuto

The above shows information about all known Vagrant environments
on this machine. This data is cached and may not be completely
up-to-date. To interact with any of the machines, you can go to
that directory and run Vagrant, or you can use the ID directly
with Vagrant commands from any directory. For example:
"vagrant destroy 1a2b3c4d"
[rcameron:~/code/JNPRAutomateDemo-Class] master(+31/-0) ±
```

Run command to stop running machines

You must get the ID of the running VM. This is the hex code in the first column. Simply copy and paste or type that to run the suspend command against that host.

```
vagrant suspend ec8e606
==> srx: Saving VM state and suspending execution...
[rcameron:~/code/JNPRAutomateDemo-Class] master(+52/-0) 7s ±
```

Optional Text Editor

During the class we are going to be editing or reviewing a set of files during the lab. You are welcome to use any text editor that your are familiar with. However if you do not have an editor in mind then please use the Atom editor from Github. It is a high-end and open source editor that can be completely customized for your needs.

Atom Downloads

- Windows
- Mac
- Ubuntu Linux
- Redhat/Centos Linux

Getting Started with Python

Python is a general-purpose, high-level programming language. It was created by Guido van Rossum in the late 1980's with its first implementation started in December 1989. Named after the Monty Python's Flying Circus as Guido was a huge fan of the show. Python introduced the idea where indentation was used rather than brackets to specify closures in the code. As Python became popular this indentation became a huge point of contention for new developers migrating from other languages such as PERL. However its benefit is that the code is similarly formatted across multiple developers applications. This was a huge departure from PERL where often there were little coding standards and it made PERL typically hard to maintain.

Python has become the "gold standard" for programming languages. It has a great deal of available libraries that provide support for just about every possible thing that you can imagine. Advanced scientific computing NumPy, gaming pygame, to automation with Ansible. The ability to write code a single time and not having to worry about porting to multiple platforms is a huge advantage. With the ubiquitous nature of Python and its adherence to formatting standards as defined in PEP8 it gives programmers an advantage to solve various domain problems in a simmilar fashon.

It is a dynamically typed language in which you do not have to specify the type of a variable. When Python is run it takes the script file and interprets it into byte code. This can be seen when you run a **.py** file as a **.pyc** file is generated in the same folder. This byte code is then run in a virtual machine, similar to Java. This gives the language the ability to be written once and run across multiple platforms. While this has some performance implications generally Python performs quite well. The language is used heavily in many different fields from math, science simulations, various application services, and of course automation.

Some of the downsides to Python stem from its need to move forward. Python created a new version in the 3.X branch in December of 2008. The goal was to remove some of the languages rough edges and increase overall performance. Adoption for the new version has been slow due to the fact that the code, while similar, is not 100% backwards compatible. Due to the wide variety of packages available in 2.X most developers tend to stick with the older release. In the end the competition between the two versions is positive as it helps drive new features into the 3.X line such as the inclusion of strong-typing (predefined data structures).

Python Language Syntax

Programming comes down to two basic concepts: data structures and algorithms. No matter which language you choose to use these are the fundamentals that will be used. A data structure is the format in which you store data. Each language generally has a similar set of data structures that are used such as dictionaries, a key/value type of structure. This is because no matter what language you use you are looking to solve similar problems. Some languages, such as R (http://www.r-project.org/), have structures that are more friendly to solving its specific problems. In the case of R it has advanced matrices that are used for multidimensional data. The goal of a language is to give the programmer flexibility and yet help them solve problems in an easy manner.

Once you have your data or data structures defined then we need to go about using them. Data is read, manipulated, removed, and created throughout a program. An algorithm is defined as a formula or procedure for solving a problem. Or in simpler terms how we interact with the data. When you have a program it usually takes some sort of data into it to solve a specific problem. In the case of automation we take in data structures such as the Junos config and then use that data in various ways. We may manipulate the data or config, provide a report or output the data, or even remove elements and commit the change back to the Junos device.

Variables

A variable is a named data structure that contains information used by the program. There are several type of variables that can be used in Python. All Python data structures are known as objects. An object is a data structure that not only contains data but it also contains methods. A method gives the object certain actions that it can take. We will take a more detailed look at this in the next section.

Numbers

A number in Python is an integer that can contain either a positive or negative value.

Python has four different types of numbers that you can use. This varies depending on how large the number is or how precise the number needs to be.

- int
    - A signed integer that can be a positive or negative number

    - 1 or -1 are both ints
- long
    - A large number specified with a capital L at the end of the number
    - 51924361L or -51924361L
- float
    - A number with floating points specified with a decimal point and a following set of integers
    - 1.01 or -20.2
- complex
    - A number that is very large or complex and is followed with a lower case j
    - 3.14j

Strings

A string is a series of characters. This can include letters or numbers. All Python strings are encoded in UTF-8 as opposed to ASCII. This lets users utilize non-English characters and ultimately simplifies dealing with text.

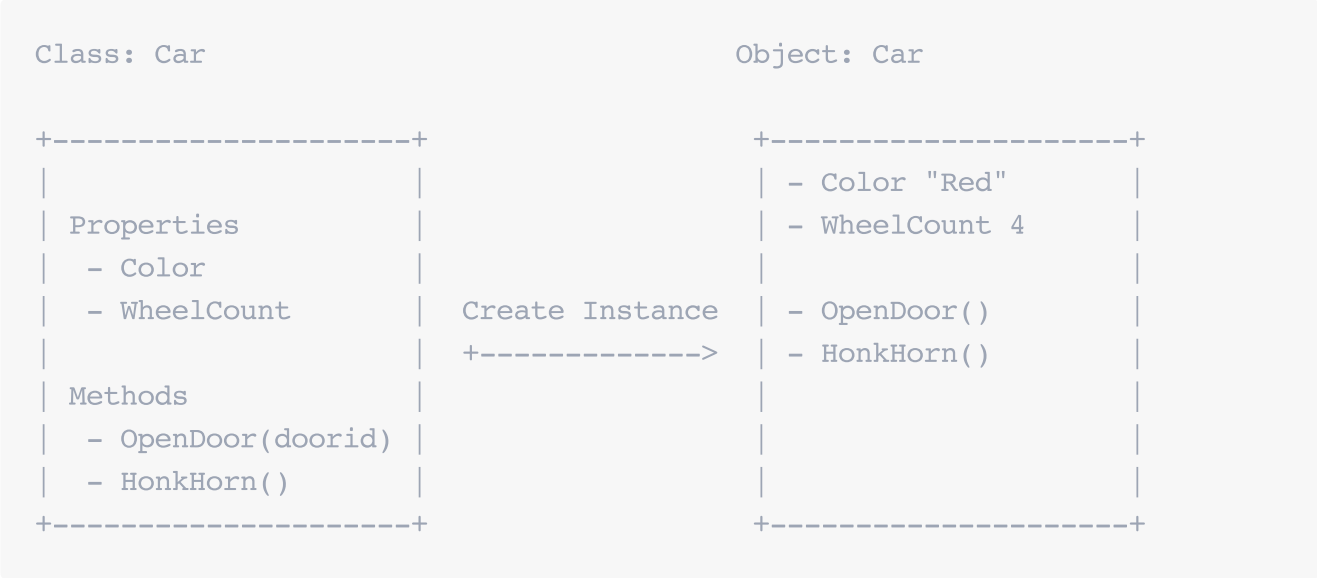
Lists and Tuples

A list or tuple is a series of data. They can contain any type of data and the types within a list can be mixed. So you could have an object, a number, and a string contained in either. The list and tuple is the same, except for that a tuple can not be manipulated and is immutable. A list can be reordered or have items deleted from it as needed.

Classes and Objects

A class is a complex data structure. It contains variables or properties as well as methods. A method is a function that can be used as actions on the object. The classic example for class is to think of it as a blueprint to create an object. A car has certain properties such as color or number of wheels. If you wanted to open a car door that would be a method or action that the object could take. Object oriented programming was invented by Alan Kay at Xerox PARC in the 1970s as a way of coupling data with methods of operation. Classes and object-oriented programming is quite a tall subject we won't go into all the details here.

Classes and Objects Examples



Object Oriented Programming

- Object Oriented Programming Defined
- List of OOP Languages
- OOP Design Patterns
  - This book is written by the "Gang of Four" and considered the ultimate guide to OOP design patterns
- Multi-paradigm Languages

Examples of variables

```
#number
simple_int = 1

#string
simple_str = "foo"

#a list of ordered objects
simple_list = ["foo","bar",1]

#an tuple is an immutable list
simple_tuples = ("foo", "bar", 1)

#a dict or dictionary is a key/value structure
simple_dict = {'foo':{'bar':True},'shoe':1}

# a class is a structure that defines data storage and operations on the da
class Calculator:
    foo = 0
    bar = 0
    def __init__(self,foo,bar):
        foo = foo
        bar = bar

    def add(self):
        return foo + bar

simple_object = SimpleClass(1,2)
simple_return = simple_object.add()

print simple_return
# Prints 3
```

Functions and Methods

As discussed a function or method is a bit of code that can be repeatedly reused. This is typical for a common set of code that we would use several times. By making this into a function or method it leaves the code in a singular place in which it can be reused but also maintained. If you were to have a function to add numbers, as can be seen below in our example "add_numbers", we can call this function any time we need to return the sum of two numbers. If we want to change this behavior in the code we can update this single function and then have that change impact the code within the rest of the program.

A method is simply a function that is bound to an object or class. This is usually used to describe the actions or methods of operation over a class. They are written nearly identically. However when defining a method on a class the special object "self" must be the first parameter used. This allows the object to reference itself and change internal elements to its own data structure. This means you are changing the instantiated object, rather than changing the class all together.

```
# Function
def add_numbers(i1, i2):
    return i1 + i2

# Method is a function bound to an object
class Calculator:
    def add(self, i1, i2):
        return i1 + i2
```

Conditionals

A conditional is a basic set of logic in programming. The most common here is the "if" statement.

```
foo = True
bar = False

# a simple if statement
if foo:
    print "success"

# a simple if else statement
if foo:
    print "success"
else:
    print "fail"

# a simple if else if and else statement
if foo != True:
    print "success foo"
elif bar != True:
    print "success bar"
else:
    print "fail"
```

Loops

A loop is the ability to loop over a data structure. This lets you take a structure and act upon each element. This is also known as iteration.

A "for" loop takes a variable and iterates over it. It provides one value from the list and allows you to utilize it. In the example below we want to take one of each of the items and then print it out. In our example we take a list of four alcoholic drinks and then have it request an order for one.

Also you can use a "while" loop. A "while" loop continues to loop until the while condition changes. In this example we are saying "while true" to continue the loop. This loop will run indefinitely unless we choose to break from the loop. In our infinite loop we increment the value of x by one. After 11 iterations, since we start at 0, we detect that x is equal to 10. Once this is detected we print out the current value of x and declare we are complete. Lastly we call the command break to exit the execution of the loop.

```
# a simple for loop
items = ["gimlet","greyhound","old fashioned","beer"]

for item in items:
    print "Order me a {0} please!".format(item)

# Prints
# Order me a gimlet please!
# Order me a greyhound please!
# Order me a old fashioned please!
# Order me a beer please!

# a simple while loop
x = 0

while True:
    if x == 10:
        print "Value of X is {0}".format(x)
        print "Complete"
        break
    x = x + 1
```

Python Summary

Learning programming or a specific language is much like playing the game of golf. It is something you can do for years to get better and better at it. Experience in using Python will be your greatest advantage to getting better at it. Find use cases for it in your daily life. You can simply drop into an interactive Python shell, just type "python" at your prompt and go for it, and use it. Use it as a calculator, use it to read in a spreadsheet and iterate over the data, use it to automate your network. What at first seems impossible, becomes completely possible, and eventually it becomes natural. Only by using it and solving new problems will you see growth in your skill sets.

- What is Python
- Python Site
- Python for Network Engineers
- Learn Python the Hard Way

Getting Started with PyEZ

The PyEZ library is a simple way to utilize Python to interact with Junos. It uses some special programming techniques that keep it free from needing to be updated to support new RPC calls in Junos. It takes a complex process of connecting to the device, running RPCs, and then parsing the output and makes it a snap. However just because the name includes EZ for easy this does not inheritly make it easy to use. There are some important basics that you would need to know before you get started.

Metaprogramming

Metaprogramming is a concept that is applicable in several programming languages. It effectively allows you to call code that does not exist and it utilizes the program as the data. This means that the program can dynamically interpet the code that is called and turn it into other code for you. The feelings around this technique vary, but in the case of PyEZ it is a perfect use case. The downside to consider is that this code is dynamically generated with no specific documentation around the various available methods on the device. Developers tend to spend a lot of time reading documentation and because the code doesn't actually exist this can be a problem. However the target of the PyEZ library is that it is to be used by network engineers who cross the lines between Junos knowledge and Python. This ability to meet in the middle is very powerful in the right hands.

In Junos there are potentially thousands of different RPC calls that the user can call. These RPC calls are well documented on the [Juniper website]TODO undate link (http://foo). This documentation tells you what the various RPC calls do and the expected responses.

Mapping RPCs to Methods - The method to the madness

As someone familiar with Junos you have a distinct advantage over the average programmer. As you know, or at least know how, to find the RPCs that you are looking to call. There is a mapping between the RPC calls. If you know the command or can figure it out it is simple to learn the correct RPC call. You can use the format of "show interfaces terse ge-0/0/0 | display xml rpc". This will show you what you need to use to call the command in PyEZ.

RPC Mapping Examples

```
# Command "show interfaces terse ge-0/0/0"
# RPC - get-interface-information
# PyEZ - get_interface_information

# Command on the CLI
root@NetDevOps-SRX01> show interfaces terse ge-0/0/0
Interface               Admin Link Proto    Local                 Remote
ge-0/0/0                up    up
ge-0/0/0.0              up    up   inet     10.0.2.15/24

root@NetDevOps-SRX01>

# Show XML output from the CLI
root@NetDevOps-SRX01> show interfaces terse ge-0/0/0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <interface-information xmlns="http://xml.juniper.net/junos/12.1X47/juno
        <physical-interface>
            <name>ge-0/0/0</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
            <logical-interface>
                <name>ge-0/0/0.0</name>
                <admin-status>up</admin-status>
                <oper-status>up</oper-status>
                <filter-information>
                </filter-information>
                <address-family>
                    <address-family-name>inet</address-family-name>
                    <interface-address>
                        <ifa-local junos:emit="emit">10.0.2.15/24</ifa-loca
                    </interface-address>
                </address-family>
            </logical-interface>
        </physical-interface>
    </interface-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

#show XML RPC from the command line
root@NetDevOps-SRX01> show interfaces terse ge-0/0/0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <rpc>
        <get-interface-information>
                <terse/>
                <interface-name>ge-0/0/0</interface-name>
        </get-interface-information>
```

```
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

root@NetDevOps-SRX01>
```

PyEZ Simple Example

In this example:

- Initialize device object
- Open a connection to the device
- Print the facts
- Close the connection

```
from jnpr.junos import Device

dev = Device( user='root', host='172.16.0.1', password='Juniper' )

dev.open()

print dev.facts

dev.close()
```

PyEZ Example with RPC

- Initialize device object
- Open a connection to the device
- Request the chassis inventory
- Print the text representation of the XML
- Close the connection

```
from jnpr.junos import Device
import xml.etree.ElementTree as ET

junos_dev = Device(user='root', host='172.16.0.1', password='Juniper' )

junos_dev.open()

chassis_output = junos_dev.rpc.get_chassis_inventory()

print ET.tostring(chassis_output,encoding="utf8", method="text")

junos_dev.close()
```

Tables and Views

As we have seen previously, Junos maintains all output data in an XML structure that allows for easy processing. While we could use standard tools like Regular Expressions, or more likely the very powerful XPath tools, for translating these XML elements into Python variables, we have simplified this process.

For the many use cases for collecitng data from a Junos device we follow the same process:

1. Send RPC to device
2. Receive XML
3. Parse XML and loop through data

To parse this data from XML nested information, into useful Python data objects, the Juniper team created two primary components:

- **Tables:** A table essentially represents all the data collected for a certain RPC request, sorted and keyed on a particular set of values and presented to the user as a collection of native Python data structures. The end user only needs to describe the data sets in YAML format without knowing the Python objects below.
    - Think about Tables as defining the data set and the rows of a table
- **Views:** A view is applied to a table to create a custom combination of the data in the table. As such, a combination of table and view definitions can be created with just a handful of lines of code yet scale infinitely.
    - Think about Views as defining the columns of a table

These two features are very powerful, but are somewhat abstract concepts. To better

demonstrate the way tables and views work together it's best to show you a real world example.

In this example, we're going to build on the XML examples shown previously in the "RPC Mapping Examples" section, and show how we can use PyEZ to programmatically collect details about port states. This might feel like a simplistic use case, but this is actually the script that the author first wrote when starting out with the PyEZ library to solve a real world "on the job" customer problem.

The first thing we need to do is map the <rpc> for the "show interface" command to a table structure. A table takes all of the XML output and collects into an ordered fashion. Table and view definitions are written in the YAML format, which is an easy to read format that is also programming language independent. The PyEZ library stores these definitions in the "lib/jnpr/junos/op/" directory in the folder where the PyEZ library is stored.

First, lets have a look at the following table definition that ships with the PyEZ library.

PhyPort Table Definition:

```
PhyPortTable:
  rpc: get-interface-information
  args:
    interface_name: '[fgx]e*'
  args_key: interface_name
  item: physical-interface
  view: PhyPortView
```

If you recall from our investigations in the "DMI and the NETCONF Protocol" the RPC command associated with "show interface" command is <get-interface-information>. The second line of the output above requests that this table definition should map to the XML output from that <rpc-reply>. We then filter down our interface list to just Fast Ethernet, Gigabit Ethernet and 10 Gigabit Ethernet interfaces ("[fge]e*"). The "item" section creates the equivalent of the table row in this data structure. By defining "physical-interface" as the "item" each interface from the XML output will be represented separately in the Table Structure.

The last line in the output above is used to feed this table into a "view". The view definition below takes each of the XML tags from the <rpc-reply> and maps them to dictionary keys on the left. These keys are the equivalent of columns in this data structure.

PhyPort View Definition:

```
PhyPortView:
  fields:
    oper : oper-status
    admin : admin-status
    description: description
    mtu: { mtu : int }
    link_mode: link-mode
    speed: speed
    macaddr: current-physical-address
    flapped: interface-flapped
```

With these Tables and Views defined, we can now write some very simple Python code to connect to our switch and collect interface statistics using these definitions. The example file below is a very simple script designed to loop through all the Ethernet interfaces on the switch and print out a “CSV formatted” output of the port statistics on the switch.

port-report.py:

```
from jnpr.junos.op.phyport import *
from jnpr.junos import Device

dev = Device( user='netconf-test', host='lab-switch', password='lab123' )
dev.open()

ports = PhyPortTable(dev).get()
print "Port,Status,Flapped" #Print Header for CSV

for port in ports:
        print("%s,%s,%s" % (port.key, port.oper, port.flapped))
```

The Python code show here is quite simple, and we have only introduced three new components from our helloEZ.py example.

- In the first line we load the library module for the PhyPort table and view. This is required to ensure that our script knows where to find our Table and View definitions.
- After “dev.open()” we call our module to with the “PhyPortTable(dev).get()”. We store the result of this function in the variable “ports”. Once assigned, this variable now contains the result of the Table+View combination.
    - This is represented in Python as a List structure, where each item is a Python

Dictionary (Collection of Key/Value pairs).

- Now that we have the data from our switch represented in simple iterable Python objects we can process them using standard Python tools. To present the output to the use, we then loop through our table printing out the key (which happens to be our interface name), the Operational State of the port, and the last time the port changed state.

When we run this script we get the following output:

```
vagrant@NetDevOps-Student:/vagrant/examples$ python ./port-report.py
Port,Status,Flapped
ge-0/0/0,up,2015-03-30 21:30:40 UTC (02:05:54 ago)
ge-0/0/1,up,2015-03-30 21:30:40 UTC (02:05:54 ago)
ge-0/0/2,up,2015-03-30 21:30:40 UTC (02:05:54 ago)
vagrant@NetDevOps-Student:/vagrant/examples$
```

This example output shows that we now have a simple Comma Delimited list containing the port name, current operational status and how long since the port changed status for each port on our switch. If you wanted to earn bonus points with your boss, it is trivial to take the CSV output and put pretty formatting around it in Excel to present management style reports. Extra bonus points if you create the reports using all Python!

This was just a simple example of automating the operational tasks on a Junos platform, but by combining the power of the Junos DMI system with the NETCONF protocol and the Tables and Views structures from the PyEZ library, we can easily automate many of our troubleshooting and data collection tasks.

Templating with Jinja2

In Python the most popular templating language is called Jinja. The original use case for Jinja was to provide developers a simpler method for developing web applications. Today most of the HTML pages that you see are actually created via templating. In the use case of of Junos configurations we most likely do not need all of the features of Jinja.

The complete documentation can be found here Jinja Dev Documentation. However we will highlight the specific language use cases here.

Utilized Jinja features

Jinja has a huge set of features that can be utilized to accomplish your goals. But the hardest part in getting started is learning where to start. For this demo we will use only a small subset of the Jinja template engine.

Variable Substitution

The most basic example of Jinja is variable substitution. Think of this as a simple search and replace within a text editor or using the UNIX command *sed*. There are several benefits to utilizing a Python script to accomplish this task.

- Repeatability
  - The ability to repeatedly call a script with fixed input and specified output
- Chaining
  - The ability to chain scripts and or actions together

To create a Jinja template it is quite simple. Below is an example of a Jinja substitution variable.

Hello World Template

```
"Hello, {{ world }}!"
```

In our example you see "Hello, " which is just treated as plain text. When run through the Jinja engine this text will be output as is. What Jinja is interested in is the contents of the double curly braces "{{ world }}". Jinja sees this as a variable named "world". When applying a variable you are simply applying a key and value to the variable. In this case "world" is the variable. In the Python code we would simply apply world="Automation". The message outputted would be "Hello, Automation!" if you applied this simple key and value.

Here is the full example of what we just discussed.

Full Hello World Template

```
# Template
"Hello, {{ world }}!"

# Key/Value
world="Automation"

# Final Output
"Hello, Automation!"
```

The documentation for this can be found here.

Using loops

The second step in substitution is utilizing loops. Loops provide true automation advantage to not require the user to repetitively type in commands. We can apply a more complex data structure in Python to a template and iterate over the items. To do this we use the age old "for loop". A "for" loop is used in many different languages to loop or iterate over items.

A Jinja for loop example:

```
{% for item in items %}
    Hello, {{ item }}!
{% endfor %}
```

In this example we use an array or list called *items*. This contains a list of strings. When run Jinja will take each item out of the list *items*. A list can contain mixed types or different types of information. The same list could contain numbers/integers, strings, or even more complex items such as dictionaries. The loop takes the next item in the list, starting at item 0, and substitutes it in the loop.

Full for loop example

```
# Template
{% for item in items %}
    Hello, {{ item }}!
{% endfor %}

# Key/Value
items = ["Automation","Shoe","Socks","Pants"]

# Final Output
Hello, Automation!
Hello, Shoe!
Hello, Socks!
Hello, Pants!
```

Best practices in writing tools

While there is a mountain of wisdom that you can use in programming, here are a few best practices in writing tools that are critical to think about.

1. Keep it simple
    - Don't try and cram too many goals into a single tool
2. Keep tools to a singular workflow
    - The best automation is a set of loosely coupled tools that are strung together
    - This simplifies the debugging and maintenance of the tools
    - Chaining multiple workflows together is called orchestration
    - Use an orchestration tool to chain the tools together
3. Think about the consumer of the tool
    - Can they use it easily
    - Are there a lot of dependencies to run it
    - Does the tool solve the problem of the user
    - Will they need your help after you give them the tool
    - How will they get access to the tool
        - github, package distribution, email, website
4. Maintainability
    - Can the tool be easily maintained
    - Are there other use cases for the tool
    - Does it have clear documentation on how to use it

Getting Started with Ansible

Ansible is a command-line automation tool that simplifies the large scale management of devices. It is one of the simplest tools that you can use to automate a large scale topology. There are only a few basics that you need to learn to use Ansible.

Helpful Links

- Ansible Best Practices
- Ansible Inventories
- Ansible Variables
- Built-in Modules

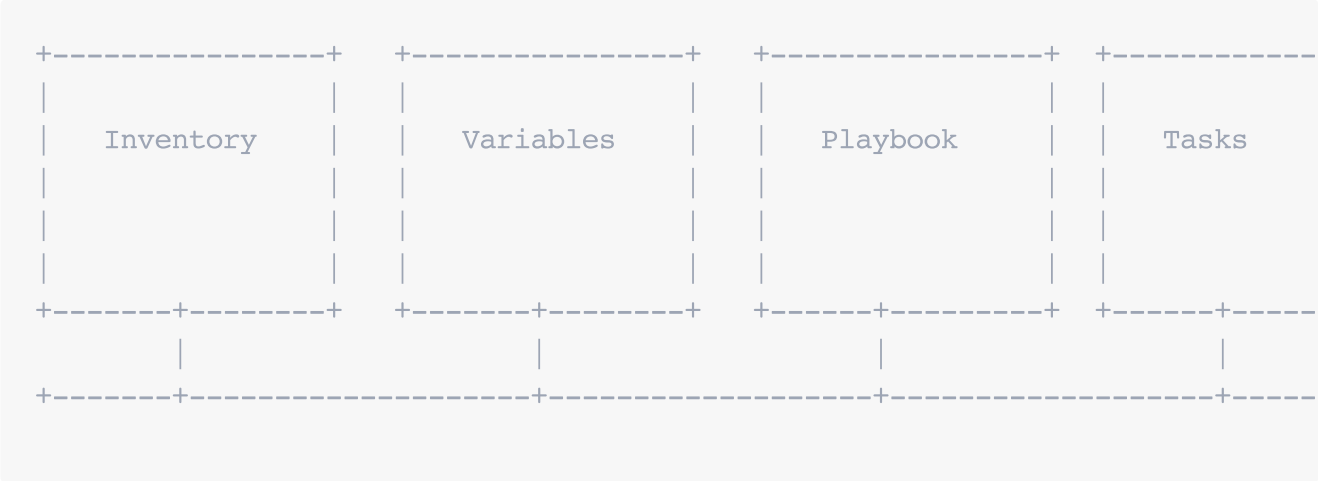
Benefits

- Tasks are run step-by-step easily identifying any issues during a deployment
- Can manage not only Junos devices but configure servers as well
- Extremely flexible ordering of tasks
- Simple to create playbooks with only YAML templates
- Easy to learn
- Easy to extend with custom modules
    - Python is first class language for this
    - But any language can be used to run scrips (Bash, Ruby, Perl)

Drawbacks

- Unable to manage Windows hosts
- Managing a large scale of devices requires a strong structure
    - SSH Keys at scale
    - Large scale variables
- Extremely flexible ordering of tasks
- Difficult to master

Execution Diagram



Ansible Technologies

Ansible at its core uses "Yet Another Markup Language" YAML as the syntax for building playbooks. YAML is a simplified language structure that has become quite popular for use due to its simplicity. In fact it is in use today in the PyEZ libraries for doing tables and views.

A playbook consists of a few required elements.

- Name
  - The name of the running playbook
- Hosts
  - Hosts to apply the tasks to
- Tasks
  - Tasks to apply to the hosts
- (Optionally) Variables
  - Variables allow for the customization of a running task

Playbook Example

```
---
- name: Configure basic firewall policies   #defines playbook
  hosts: mysrx                              #defines hosts to apply
  connection: local                         #defines execution environment,
  gather_facts: no                          #gathers facts for the devices
  vars:                                     #variables to be used in the pl
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"
    address_entries: [ {'name':'LocalNet','prefix':'172.16.0.0/24'},{'name'
    fw_policy_info: [ {'policy_name':'Allow_Policy','src_zone':'trust','dst

  tasks:                                    #set of tasks to run
    - name: Build address book entries      #Name of task
      template: src=templates/fw_address_book_global.set.j2 dest={{build_di
      with_items: address_entries           #Add in additional variables to

    - name: Apply address book entries
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use


    - name: Build firewall policies config template
      template: src=templates/fw_policy.set.j2 dest={{build_dir}}/fw_policy
      with_items: fw_policy_info

    - name: Apply firewall policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

Inventory

The inventory defines which hosts you can run Ansible against. This can consist of a simple text file or also utilize an API to gather this information. The format of file is in the traditional INI style format. The listing consists of a single host per line. You can also have groups of hosts that may have a common role. An example is if you had multiple web servers or database servers and you want to apply the same tasks to that group. You can also include ranges of alphanumeric characters as well.

Ansible Inventories

```
mail.example.com         #A single host
host[a:z].example.com    #26 different hosts defined by a range
172.16.0.1               #A host defined by an IP
172.16.0.[1:254]         #Hosts defined by an IP range

[webservers]         #A group of hosts
foo.example.com
bar.example.com

[dbservers]              #A second group of hosts
one.example.com
two.example.com
three.example.com
```

It is also possible to query the inventory from a script or API. There are existing tools that allow you to plug into things like AWS. With a simple API call to AWS it pulls in your entire inventory from the list of existing VMs.

Variables

Variables are the special sauce that makes Ansible so tasty to use. This allows you to take a playbook and customize it for your specific set of hosts you want to run against. So imagine you have two data centers. Each data center has a set of DNS, NTP, and syslog servers that are specific to the data center. In this case you can use the same playbook for both data centers, but specify different DNS, NTP, and syslog servers for each datacenter.

Variable Example

```
---
- name: Configure basic firewall policies
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:                                  #variables to be used in the pl
    junos_user: "root"                   #username for our Junos devices
    junos_password: "Juniper"            #password for our Junos devices
    build_dir: "/tmp/"                   #directory for us build templat
                                         #a complex variable, this is ju
    address_entries: [ {'name':'LocalNet','prefix':'172.16.0.0/24'},{'name'
```

Variable scope

A variable can be applied to several locations within your Ansible environment. The most specific application of a variable becomes the value that is used when applied. Using the ordering capabilities of variables allows you to further customize how your tasks are run.

Variable Order

1. Host
2. Group
3. Role
4. Variable File
5. Playbook

Ansible Galaxy

Ansible not only includes a host of included modules, but we also have a repository that users can contribute to for Ansible. This is called Ansible galaxy and it allows you to easily install 3rd party modules for use in your Ansible environment.

Example of installing Junos Ansible modules

```
[root@ansible-cm]# ansible-galaxy install Juniper.junos
downloading role 'junos', owned by Juniper
no version specified, installing 1.0.0
- downloading role from
https://github.com/Juniper/ansible-junos-stdlib/archive/1.0.0.tar.gz
- extracting Juniper.junos to /etc/ansible/roles/Juniper.junos
Juniper.junos was installed successfully
```

Ansible Tower

While all of this may seem great to use, how do you scale these scripts to a larger environment. For this Ansile has the tool Ansible Tower. It gives you a GUI that is wrapped around the management of Ansible tasks. This has a free trial version but it is not free to use. This can assist you in the management of a large scale environment.

Ansible Tower

Getting to Know NETCONF

NETCONF is an XML-RPC based API. The current RFC 6241 specifies the RFC standard. However you can also see the original RFC 4741 as well to see where it all started.

NETCONF Introduction via the RFC

Quoted from RFC 6241 Introduction

The NETCONF protocol defines a simple mechanism through which a network device can be managed, configuration data information can be retrieved, and new configuration data can be uploaded and manipulated. The protocol allows the device to expose a full, formal application programming interface (API). Applications can use this straightforward API to send and receive full and partial configuration data sets.

The NETCONF protocol uses a remote procedure call (RPC) paradigm. A client encodes an RPC in extensible markup language (XML) (TODO is this suppose to link to http://www.w3.org/TR/2000/REC-xml-20001006) [W3C.REC-xml-20001006] and sends it to a server using a secure, connection-oriented session. The server responds with a reply encoded in XML. The contents of both the request and the response are fully described in XML document type definitions (DTDs) or XML schemas, or both, allowing both parties to recognize the syntax constraints imposed on the exchange.

A key aspect of NETCONF is that it allows the functionality of the management protocol to closely mirror the native functionality of the device. This reduces implementation costs and allows timely access to new features. In addition, applications can access both the syntactic and semantic content of the device's native user interface.

NETCONF allows a client to discover the set of protocol extensions supported by a server. These "capabilities" permit the client to adjust its behavior to take advantage of the features exposed by the device. The capability definitions can be easily extended in a noncentralized manner. Standard and non-standard capabilities can be defined with semantic (meaning of expressions, statements, and program units) and syntactic (form of expessions, statements, and program units) rigor. Capabilities are discussed in Section 8.

The NETCONF protocol is a building block in a system of automated configuration.

XML is the lingua franca of interchange, providing a flexible but fully specified encoding mechanism for hierarchical content. NETCONF can be used in concert with XML-based transformation technologies, such as XSLT TODO : should a link be added? [W3C.REC-xslt-19991116], to provide a system for automated generation of full and partial configurations. The system can query one or more databases for data about networking topologies, links, policies, customers, and services. This data can be transformed using one or more XSLT scripts from a task-oriented, vendor-independent data schema into a form that is specific to the vendor, product, operating system, and software release. The resulting data can be passed to the device using the NETCONF protocol.

What does this all mean?

NETCONF was designed to achieve a few specific goals

- To closely mirror the native functionallity of the device

This keeps the way we interact with the device close to how the device is configured. This simplifies the API usage because the way you interact with it is the same as doing the native configuration

- To use an extensible interchange language that can be easily manipulated

Computers do a horrible job of interpreting data that does not have context. XML was chosen as the interchange format for NETCONF. It allows for data exchange in a way that makes sense to a computer and it can easily be parsed. XML is highly extensible and does a great job of representing the data so it can be managed.

- Secure access to the API

NETCONF requires secure access to the API as defined in RFC 6241 Section 2. The goal is the communication must be authenticated, have data integrity, confidentiality, and replay protection. Junos uses SSH as this transport mechanism. NETCONF while it can support other protocols such as TLS or BEEP it MUST support SSH as per RFC 6241 section 2.3.

Interacting with NETCONF manually

Canonical Data Terminator

The character set "]]>]]>" is used to specify the end of a NETCONF message. This is used to tell the host that the message is completed. While some protocols such as HTTP use content length for this, by defining a specific terminator that is all that an application needs to look for at the end of a message. This is typically done by looking for this set of bytes as "5D 5D 3E 5D 5D 3E". Because XML can contain new lines, tabs, and carriage returns this byte set terminator allows the usage of these typical end elements to be included freely in the XML sent to the user. You can read more here at the Juniper documentation NETCONF end of document.

Connecting to NETCONF

Connecting to NETCONF host can easily be done from your NetDevOps VM.

- Connect into your host with "vagrant ssh ndo"
- Run the command "ssh root@172.16.0.1 -s netconf" to request an SSH session with the device. The "-s netconf" flag is used to specify the NETCONF subsystem.
    - We can also connect to the NETCONF port directly using "ssh root@172.16.0.1 -p 830 -s netconf" by specifying the port of 830
- Use the root password of "Juniper" to connect to the device

```
ssh root@172.16.0.1 -s netconf
```

This should connect you to the NETCONF service on the vSRX. Once connected you will see it spit out a bunch of XML back to you. We now are bound to a socket connection that we can freely send and receive data over.

Each NETCONF session is given a session ID. This is used to tell you which process id or PID is being used on the server. If you have multiple NETCONF sessions active it will help you determine which one you are connected to. The session ID is included as part of the hello that is part of the initial connection.

Response

```
<!-- No zombies were killed during the creation of this user interface -->
<!-- user root, class super-user -->
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</ca
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</cap
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>1985</session-id>
</hello>
]]>]]>
```

The hello response also includes a set of capabilities on the device. These included URLs are used as part of the namespace for the document. These aren't active websites that are used.

Hello RPC

Once connected we need to send back a hello message to the the NETCONF server. On Junos this is not required to be sent. However the best practice is to send this information. Also in the event that we DO enforce it in the future, it is better to be compliant to the specification of the protocol.

Request

```
<hello>
    <capabilities>
        <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
        <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
        <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-com
        <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0<
        <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?proto
        <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
        <capability>http://xml.juniper.net/dmi/system/1.0</capability>
    </capabilities>
</hello>
]]>]]>
```

Response

```
Junos will not send a response
```

Message ID

When sending RPC messages it is possible to include a message-id in the request. When the response is returned it will allow you to match the response to the request. This is hugely helpful for when you need to troubleshoot a connection in which you are sending multiple messages at the same time. It helps you map a response to a request. This is optional and it is up to the implementor to use.

Gathering some information

Now that we have an active NETCONF connection to the device we can start sending commands to the host.

Request

```
<rpc message-id="1">
    <get-software-information/>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <software-information>
        <host-name>NetDevOps-SRX01</host-name>
        <product-model>firefly-perimeter</product-model>
        <product-name>firefly-perimeter</product-name>
        <package-information>
            <name>junos</name>
            <comment>JUNOS Software Release [12.1X47-D10.4]</comment>
        </package-information>
    </software-information>
</rpc-reply>
]]>]]>
```

Adding to the configuration via a manual RPC

Now we are going to make a simple configuration change. We do this just as we would from the command line: Open configuration, load change, and commit. For our example we will simply change the hostname.

- Open Configuration
- Load Configuration
- Commit Configuration

Opening the configuration

To change the configuration we first must open up a candidate configuration. This is just as if you were to type "edit" or "configure" from the CLI.

Request

```
<rpc message-id="2">
  <open-configuration>
  </open-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
</rpc-reply>
]]>]]>
```

Loading a configuration

Now that the configuration is open we will load our configuration in the standard XML format that is expected. We can also load other formats such as "set commands" or "Junos config". But to best demonstrate the API we will use the standard XML formatting.

Request

```
<rpc message-id="3">
    <load-configuration action="merge">
    <configuration>
        <system>
            <host-name>NETCONFED</host-name>
        </system>
    </configuration>
    </load-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <load-configuration-results>
        <ok/>
    </load-configuration-results>
</rpc-reply>
]]>]]>
```

Committing the configuration

We must now commit the configuration just as we would by do by hand. But again in this case we will use the XML formatting.

Request

```
<rpc message-id="4">
    <commit-configuration>
        <log>Committed via NETCONF by hand!</log>
    </commit-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <ok/>
</rpc-reply>
]]>]]>
```

Validating the change

Now we want to validate that our change successfully occured. There are numerous ways to do this, but lets reuse an RPC we did earlier in the lab. We will get the software information and this response also includes the hostname. The best practice would be to look at the diff of configuration or by comparing the entire configuration.

Request

```
<rpc message-id="5">
    <get-software-information/>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <software-information>
        <host-name>NETCONFED</host-name>
        <product-model>firefly-perimeter</product-model>
        <product-name>firefly-perimeter</product-name>
        <package-information>
            <name>junos</name>
            <comment>JUNOS Software Release [12.1X47-D10.4]</comment>
        </package-information>
    </software-information>
</rpc-reply>
]]>]]>
```

Rolling back the change

To keep the lab intact we want to rollback to the previous configuration. This will keep us all on the same page as we progress, plus it gives us an excuse to run a few more RPCs. In this request we are going to rollback to the previous configuration and then commit the change.

Request

```
<rpc message-id="6">
    <get-rollback-information>
        <rollback>1</rollback>
    </get-rollback-information>
</rpc>
```

Response

The response is long and contains the entire configuration. For this example the response has been truncated.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <rollback-information>
        <ok/>
        <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:cha
            <version>12.1X47-D10.4</version>
            <system>
                <host-name>NetDevOps-SRX01</host-name>
            </system>
            <!-- RESPONSE TRUNCATED -->
        </configuration>
    </rollback-information>
</rpc-reply>
]]>]]>
```

Lastly commit the configuration to apply the rollback.

Request

```
<rpc message-id="7">
    <commit-configuration>
        <log>Committed via NETCONF by hand!</log>
    </commit-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <ok/>
</rpc-reply>
]]>]]>
```

Review

In this section we reviewed how to make a manual connection to the NETCONF service. This is not ideal to do by hand. Humans aren't very good at parsing XML data by hand. This is why the best way to interact with a Junos device manually is through the CLI. It gives you the exact same capabilities of NETCONF but in a human friendly format. Although you are welcome to use NETCONF in this manner it probably won't be a very efficent way to interact with the device.

What is the NETCONF subsystem on Junos?

When creating a NETCONF connection we mentioned that it is part of an SSH subsystem. An SSH subsytem allows you to bind various services or accomplish remote tasks on the host. More details can be found in RFC 4254. By default when using SSH you are getting a shell on the device. It is also possible to execute a single command on the device. For NETCONF when calling the subsystem in invokes the below script.

NETCONF subsystem script

```
#!/bin/sh

#Source file: /usr/libexec/ui/netconf

CLI=/usr/sbin/cli

if [ -f $CLI ]; then
    exec $CLI xml-mode netconf need-trailer "$@"
fi
cat << EOFEOFEOF
<rpc-error>
<error-severity>error</error-severity>
<error-message>
NETCONF initialization error
</error-message>
</rpc-error>
EOFEOFEOF
exit 1
```

Lets take a deeper look at what this script does

That is correct. Effectively launching a NETCONF session gives you a Junos CLI instance that is running in XML mode. It is even possible to invoke this from the command line via a hidden command.

```
root@NetDevOps-SRX01> junoscript netconf interactive
```

All options for hidden command

```
root@NetDevOps-SRX01> junoscript ?
Possible completions:
  <[Enter]>            Execute this command
  attributes           List of attributes to pass to management daemon
  interactive          Start interactive session (not for scripts)
  netconf              Run in NETCONF mode
  version              JUNOScript version number
  |                    Pipe through a command
root@NetDevOps-SRX01>
```

Configuring NAT via NETCONF

!!!While this example works it is not recommended to run manually as it can cause insanity!!!

This is an example of how to configure NAT via the NETCONF interface directly. It CAN be run to configure NAT and it will work correctly. However doing so is not for the feint of heart. Be careful if you are copy and pasting XML as there can be issues in incorrectly inserting characters into the NETCONF commands.

Network Address Translation or NAT is a key feature on the SRX. This technology is used to swap IP addresses out in packets to hide internal IP addressing or to use a single IP address that can hide thousands behind it.

In this section of the lab we will be using the manual NETCONF protocol to configure NAT. The goal here is to let you learn how to use NETCONF manually to configure NAT for your lab.

Tools

- NetDevOps VM
- ssh command line
- XML

Network Interface SNAT

To access the rest of the lab you must first configure source NAT or SNAT on your vSRX instance. This will allow the NetDevOps VM to access hosts outside of your own laptop. Each student has the same IP address block between their NetDevOps appliance and the vSRX instance. This allows for a consistent experience for each student so each of the exercises can be completed with the same documentation. The downside of this topology is that your NetDevOps VM can not access the rest of the lab. However we have a vSRX sitting directly in front of the NetDevOps VM. We will use this with SNAT on our external interface in the lab to hide our internal IP address.

Opening a NETCONF session

First lets open up a NETCONF session from the command line of the NetDevOps VM

```
vagrant@NetDevOps-Student:~$ ssh root@172.16.0.1 -s netconf
```

Once connected you will get the typical hello response

Response

```
<!-- No zombies were killed during the creation of this user interface -->
<!-- user root, class super-user -->
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</ca
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</cap
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>1985</session-id>
</hello>
]]>]]>
```

To be a good citizen you should also send a hello back to the server. **This step is not required, but it is good to be polite.**

Request

```
<hello>
    <capabilities>
        <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
        <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
        <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-com
        <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0<
        <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?proto
        <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
        <capability>http://xml.juniper.net/dmi/system/1.0</capability>
    </capabilities>
</hello>
]]>]]>
```

Response

```
Junos will not send a response. It already said hello to you.
```

Open configuration for changes

We must now open the configuration so we can load in the NAT configuration. This is the same as if we typed "configure" from the CLI. This gives us a candidate configuration to work with.

Request

```
<rpc message-id="1">
  <open-configuration>
  </open-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt

</rpc-reply>
]]>]]>
```

Configuring NAT

Now that we have a candidate configuration we can load in XML which allows for NAT to occur on the untrust interface. We are specifying the source address to be that of the NetDevOps VM. For simplicity we specify the entire subnet even though there is only a single host.

Request

```
<rpc message-id="2">
    <load-configuration action="merge">
    <configuration>
            <security>
                <nat>
                    <source>
                        <rule-set>
                            <name>LabNAT</name>
                            <from>
                                <zone>trust</zone>
                            </from>
                            <to>
                                <zone>untrust</zone>
                            </to>
                            <rule>
                                <name>1</name>
                                <src-nat-rule-match>
                                <source-address>172.16.0.0/24</source-addre
                                </src-nat-rule-match>
                                <then>
                                    <source-nat>
                                        <interface>
                                        </interface>
                                    </source-nat>
                                </then>
                            </rule>
                        </rule-set>
                    </source>
                </nat>
            </security>
    </configuration>
</load-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <load-configuration-results>
        <ok/>
    </load-configuration-results>
</rpc-reply>
]]>]]>
```

Commit NAT configuration

Now it is time to enact the configuration. This is done with a simple commit. If you get disconnected from your netconf connection your candidate configuration is still open and you can still commit it. This is a nice feature of the Junos configuration model. It leaves the state of the configuration on the device and you optionally do not have to worry about that in your application.

Request

```
<rpc message-id="3">
    <commit-configuration>
        <log>Add SNAT configuration</log>
    </commit-configuration>
</rpc>
]]>]]>
```

Response

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="htt
    <ok/>
</rpc-reply>
]]>]]>
```

Verifying NAT

Now the NAT configuration should be correctly applied to your vSRX instance.

Validate connectivity

To validate the connectivity you can issue a ping command from your NetDevOps VM. If everything is working correctly you should see a response from the host.

```
vagrant@NetDevOps-Student:~$ ping 10.10.0.5
PING 10.10.0.10 (10.10.0.5) 56(84) bytes of data.
64 bytes from 10.10.0.5: icmp_seq=1 ttl=64 time=0.482 ms
64 bytes from 10.10.0.5: icmp_seq=2 ttl=64 time=0.480 ms
64 bytes from 10.10.0.5: icmp_seq=3 ttl=64 time=0.520 ms
64 bytes from 10.10.0.5: icmp_seq=4 ttl=64 time=0.548 ms

--- 10.10.0.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.480/0.507/0.548/0.036 ms
vagrant@NetDevOps-Student:~$
```

If the ping request is not working? Then why?

Adding NAT Policies

The first step we need to do is add in NAT policies so your NetDevOps VM can access the outside world.

Using Ansible to enable NAT

For this step we will use Ansible to create both our firewall policies and the required address book objects. This is one of the most requested automation elements when dealing with a firewall. Typically once a firewall is deployed and integrated into the network there are few changes that are required on the networking side of things. However the majority of heavy lifting is centered around managing security policies and the associated address book objects. Address book objects are typically quite painful due to the need to create, manage, and have the correct association to zones. Luckily using Ansible makes these tasks a snap.

Reviewing the playbook

First let's take a look at the playbook that is used to accomplish this task. We briefly looked at this playbook during the Ansible overview section, but now we will dive deeper into the applied steps.

Playbook Review

1. Define the name of the playbook - Configure basic firewall policies
    - This will be displayed and logged as you start to run the playbook
2. Define the hosts the playbook should be applied to
    - In this case we use the group "**mysrx**" to apply to
        - The host list is picked up from either the default Ansible host list in "/etc/ansible/hosts"
        - Alternatively when the playbook is run you can specify your own custom inventory
3. Connection is defined to as local - Typically when Ansible runs it transports an execution environment over to the host and runs it
    - Because this will not work on Junos hosts we use connection defined to local to run the execution environment

4. Gather facts
    - Ansible will gather local facts about the host such as interfaces and hostnames
    - Because this isn't possible on Junos we disable this feature
5. Vars - These are the variables that we will use to apply to our tasks
    - They can be applied at many different locations for our run
    - But to keep everything together we have included the variables into the playbook
    - address_entires will be used to generate the address book entries
    - nat_policy_info will be used to generate the NAT policies
6. Tasks - These are the tasks that we will use
    - The build phase for the playbook generates the Junos config from the templates
    - The apply phase will apply the configuration to the device
    - This will be run as two separate commits, but in doing so we can simplify the tasks and see which step fails

Playbook

```
---
- name: Configure basic NAT policies
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"
    address_entries: [ {'name':'LocalNet','prefix':'172.16.0.0/24'},{'name'
    nat_policy_info: [{'rule_set':'fw-nat', 'src_zone':'trust', 'dst_zone':

  tasks:
    - name: Build address book entries
      template: src=templates/fw_address_book_global.set.j2 dest={{build_di
      with_items: address_entries

    - name: Apply address book entries
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use

    - name: Build NAT policies
      template: src=templates/nat_src_policy.set.j2 dest={{build_dir}}/nat_
      with_items: nat_policy_info

    - name: Apply NAT policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

Address book template

- The address book template loops through the address entries in the variable
- It generates one "set" configuration line per loop
- Here we are generating one

```
{% for i in address_entries %}
set security address-book global address {{ i.name }} {{ i.prefix }}
{% endfor %}
```

Output after generation

- This is the generated output from the template being applied with variables
- These commands are then committed to Junos
- If one or more of the entries are already created it will recognize this as "OK"

```
set security address-book global address NetDevOpsVM 172.16.0.10/32
```

NAT Template

This template is a bit more complex. We need to loop through source IPs, Destination IPs, and applications. Each loop through these variables will generate a single line of "set" commands. Creating the template this way allows us to reuse it in the future when we have a larger list of addresses and applications to apply.

```
{% for item in nat_policy_info %}
set security nat source rule-set {{ item.rule_set }} from zone {{ item.src_
set security nat source rule-set {{ item.rule_set }} to zone {{ item.dst_zo
    {% for rule in item.rules %}
        {% for src_ip in rule.src_ips %}
        set security nat source rule-set {{ item.rule_set }} rule {{ rule.n
        {% endfor %}
        {% for dst_ip in rule.dst_ips %}
        set security nat source rule-set {{ item.rule_set }} rule {{ rule.n
        {% endfor %}
        {% if rule.interface is defined %}
        set security nat source rule-set {{ item.rule_set }} rule {{ rule.n
        {% endif %}
        {% if rule.pool_name is defined %}
        set security nat source rule-set {{ item.rule_set }} rule {{ rule.n
        {% endif %}
    {% endfor %}
{% endfor %}
```

Output after generation

Once run here are the set commands that will be loaded onto the device. Again if additional elements are added they will be generated into individual set commands. While we could create the applications with a single command using a list of applications this keeps it simple to implement using individual commands.

```
set security nat source rule-set fw-nat from zone trust
set security nat source rule-set fw-nat to zone untrust
set security nat source rule-set fw-nat rule rule1 match source-address 172
set security nat source rule-set fw-nat rule rule1 match destination-addres
set security nat source rule-set fw-nat rule rule1 then source-nat interfac
```

Running the playbook

To run the playbook you must use the "ansible-playbook" command.

Playbook Command

Ensure before running the command you are in the "**ansible**" directory.

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y
```

Playbook Run Example

Once run the output should look like the following

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y
sudo password:

PLAY [Configure basic NAT policies] ************************************

TASK: [Build address book entries] *************************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNe
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] *************************************
changed: [172.16.0.1]

TASK: [Build NAT policies] *********************************************
changed: [172.16.0.1] => (item={'rules': [{'interface': True, 'dst_ips': ['

TASK: [Apply NAT policies] *********************************************
changed: [172.16.0.1]

PLAY RECAP *************************************************************
172.16.0.1                 : ok=4    changed=4    unreachable=0    failed=0
```

Validate connectivity

To validate the connectivity you can issue a ping command from your NetDevOps VM. If everything is working correctly you should see a response from the host.

```
vagrant@NetDevOps-Student:~$ ping 10.10.0.5
PING 10.10.0.10 (10.10.0.5) 56(84) bytes of data.
64 bytes from 10.10.0.5: icmp_seq=1 ttl=64 time=0.482 ms
64 bytes from 10.10.0.5: icmp_seq=2 ttl=64 time=0.480 ms
64 bytes from 10.10.0.5: icmp_seq=3 ttl=64 time=0.520 ms
64 bytes from 10.10.0.5: icmp_seq=4 ttl=64 time=0.548 ms

--- 10.10.0.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.480/0.507/0.548/0.036 ms
vagrant@NetDevOps-Student:~$
```

If the ping request is not working? Then why?

Adding Firewall Policies

At this point in the lab your vSRX will have access to the rest of the lab. However your NetDevOps appliance will not. To allow this we must first enable firewall policies on your vSRX device.

Using Ansible to enable firewall policies

For this step we will use Ansible to create both our firewall policies and the required address book objects. This is one of the most requested automation elements when dealing with a firewall. Typically once a firewall is deployed and integrated into the network there are few changes that are required on the networking side of things. However the majority of heavy lifting is centered around managing security policies and the associated address book objects. Address book objects are typically quite painful due to the need to create, manage, and have the correct association to zones. Luckily using Ansible makes these tasks a snap.

Reviewing the playbook

First let's take a look at the playbook that is used to accomplish this task. We briefly looked at this playbook during the Ansible overview section, but now we will dive deeper into the applied steps.

Playbook Review

1. Define the name of the playbook - Configure basic firewall policies
    - This will be displayed and logged as you start to run the playbook
2. Define the hosts the playbook should be applied to
    - In this case we use the group "**mysrx**" to apply to
        - The host list is picked up from either the default Ansible host list in "/etc/ansible/hosts"
        - Alternatively when the playbook is run you can specify your own custom inventory
3. Connection is defined to as local - Typically when Ansible runs it transports an execution environment over to the host and runs it

    - Because this will not work on Junos hosts we use connection defined to local to run the execution environment
4. Gather facts
    - Ansible will gather local facts about the host such as interfaces and hostnames
    - Because this isn't possible on Junos we disable this feature
5. Vars - These are the variables that we will use to apply to our tasks
    - They can be applied at many different locations for our run
    - But to keep everything together we have included the variables into the playbook
    - address_entires will be used to generate the address book entries
    - fw_policy_info will be used to define our policies
6. Tasks - These are the tasks that we will use
    - The build phase for the playbook generates the Junos config from the templates
    - The apply phase will apply the configuration to the device
    - This will be run as two separate commits, but in doing so we can simplify the tasks and see which step fails

Playbook

```
---
- name: Configure basic firewall policies
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"
    address_entries: [ {'name':'LocalNet','prefix':'172.16.0.0/24'},{'name'
    fw_policy_info: [ {'policy_name':'Allow_Policy','src_zone':'trust','dst

  tasks:
    - name: Build address book entries
      template: src=templates/fw_address_book_global.set.j2 dest={{build_di
      with_items: address_entries

    - name: Apply address book entries
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use


    - name: Build firewall policies config template
      template: src=templates/fw_policy.set.j2 dest={{build_dir}}/fw_policy
      with_items: fw_policy_info

    - name: Apply firewall policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

Address book template

- The address book template loops through the address entries in the variable
- It generates one "set" configuration line per loop
- Here we are generating three lines

```
{% for i in address_entries %}
set security address-book global address {{ i.name }} {{ i.prefix }}
{% endfor %}
```

Output after generation

- This is the generated output from the template being applied with variables
- These commands are then committed to Junos
- If one or more of the entries are already created it will recognize this as "OK"

```
set security address-book global address LocalNet 172.16.0.0/24
set security address-book global address PrivateNet 192.168.10.0/24
set security address-book global address PublicNet 10.10.0.0/22
```

Policy Template

This template is a bit more complex. We need to loop through source IPs, Destination IPs, and applications. Each loop through these variables will generate a single line of "set" commands. Creating the template this way allows us to reuse it in the future when we have a larger list of addresses and applications to apply.

```
{% for item in fw_policy_info %}
    {% for i in item.src_ips %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
    {% endfor %}
    {% for i in item.dst_ips %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
    {% endfor %}
    {% for i in item.apps %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
    {% endfor %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
{% endfor %}
```

Output after generation

Once run here are the set commands that will be loaded onto the device. Again if additional elements are added they will be generated into individual set commands.

```
set security policies from-zone trust to-zone untrust policy Allow_Policy m
set security policies from-zone trust to-zone untrust policy Allow_Policy m
set security policies from-zone trust to-zone untrust policy Allow_Policy m
set security policies from-zone trust to-zone untrust policy Allow_Policy t
```

Running the playbook

To run the playbook you must use the "ansible-playbook" command. We must specify the inventory file and the playbook to apply. The templates will be automatically loaded from the playbook. Since cowsay is installed it will also add the comical cow

for our enjoyment. If you dislike our bovine friend then you can simply remove cowsay from your running host.

Playbook Command

Ensure before running the command you are in the "**ansible**" directory.

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
```

Playbook Run Example

Once run the output should look like the following

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
 ________________________________________
< PLAY [Configure basic firewall policies] >
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


 ___________________________________
< TASK: Build address book entries >
 -----------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNe
changed: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'Priva
changed: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNe

 ___________________________________
< TASK: Apply address book entries >
 -----------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

```
changed: [172.16.0.1]

 ________________________________________________
< TASK: Build firewall policies config template >
 ------------------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'untrust',

 ______________________________
< TASK: Apply firewall policies >
 ------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1]

 ____________
< PLAY RECAP >
 ------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


172.16.0.1                 : ok=4    changed=4    unreachable=0    failed=0
```

Validating the playbook run

Now connect to your vSRX instance from your NetDevOpsVM and validate the change

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ssh root@172.16
Password:
--- JUNOS 12.1X47-D20.7 built 2015-03-03 21:53:50 UTC
croot@NetDevOps-SRX01% cli
root@NetDevOps-SRX01> show configuration security address-book
```

```
global {
    address LocalNet 172.16.0.0/24;
    address PrivateNet 192.168.10.0/24;
    address PublicNet 10.10.0.0/22;
}

root@NetDevOps-SRX01> show configuration security policies
from-zone trust to-zone trust {
    policy default-permit {
        match {
            source-address any;
            destination-address any;
            application any;
        }
        then {
            permit;
        }
    }
}
from-zone trust to-zone untrust {
    policy default-permit {
        match {
            source-address any;
            destination-address any;
            application any;
        }
        then {
            permit;
        }
    }
    policy Allow_Policy {
        match {
            source-address LocalNet;
            destination-address any;
            application any;
        }
        then {
            permit;
        }
    }
}
from-zone untrust to-zone trust {
    policy default-deny {
        match {
            source-address any;
            destination-address any;
            application any;
        }
        then {
```

```
            deny;
        }
    }
}

root@NetDevOps-SRX01> exit

root@NetDevOps-SRX01% exit
logout
Connection to 172.16.0.1 closed.
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$
```

Connecting to the VPN Headend

For this step we will use a simple Jinja2 Template and a pre-built Python script to push a prepared template to the vSRX to configure a VPN tunnel and NAT egress traffic. One of the most common jobs for a network engineer is to make the same set of changes on a regular basis with just a few minor changes such as IP address or VLAN assignments. By using a prepared template, you can drastically reduce the time spent making common network changes.

Reviewing the Template

First let's take a look at the template of changes we are going to push to this device. The configuration below is almost entirely standard Junos set commands with a view variables sprinkled in.

Template Review

1. The first two lines demonstrate using *loops* inside a Jijna template
    - Here we say "For every VPN defined for each student in the YAML file, perform the follow actions"
2. **Line 4** sets the TCP Maximum Segment Size for IPSec traffic to 1350 to take into account the additional overhead of the VPN headers
3. **Line 6 and 7** create an st0 interface that will be used by the Route Based VPN and assigns both an IP address and puts the interface into a new "vpn" zone.
4. **Line 9** opens up the firewall to accept "ike" traffic to come in via the external interface on the vSRX.
5. **Line 11-13** configures the IPSec Phase 1 VPN requirements between the local vSRX and the Headend VPN run by the Proctor
    - Note that in Line 11 we are referencing a variable from "Headend" and not our previous interfaces. This allows us to seperate Local vs remote device configuration variables as needed.
6. **Line 19-22** configures the IPSec Phase 2 VPN requirements between the local vSRX and the Headend VPN run by the Proctor
7. **Line 23-24** close out the Jinja loops from Step 1.
8. **Line 26-30** ensure that all traffic leaving over the st0 tunnel inteface between the Student network and the "Private Network behind the HeadEnd will be NAT'd using the interface egress address

Jinja2 Template

The template used for this example is found located in **"examples/vpn/vpn.j2"**

```
 {% for student in students -%}
  {% for vpn in student.vpns -%}
set security flow tcp-mss ipsec-vpn mss 1350

set interfaces {{ vpn.tunnel_int }} family inet address {{ vpn.vpn_tunnel_i
set security zones security-zone vpn interfaces {{ vpn.tunnel_int }}

set security zones security-zone untrust host-inbound-traffic system-servic

set security ike gateway ike-gate address {{ HeadEnd.vpn_ext_ip }}
set security ike gateway ike-gate external-interface {{ vpn.vpn_ext_interfa
set security ike gateway ike-gate ike-policy ike-policy1

set security ike policy ike-policy1 mode main
set security ike policy ike-policy1 proposal-set standard
set security ike policy ike-policy1 pre-shared-key ascii-text "{{ vpn.share

set security ipsec policy vpn-policy1 proposal-set standard
set security ipsec vpn ike-vpn ike gateway ike-gate
set security ipsec vpn ike-vpn ike ipsec-policy vpn-policy1
set security ipsec vpn ike-vpn bind-interface {{ vpn.tunnel_int }}
  {% endfor -%}
 {% endfor -%}

set security nat source rule-set vpn_nat_out from zone trust
set security nat source rule-set vpn_nat_out to zone vpn
set security nat source rule-set vpn_nat_out rule interface-nat match sourc
set security nat source rule-set vpn_nat_out rule interface-nat match desti
set security nat source rule-set vpn_nat_out rule interface-nat then source
```

YAML Variables file

Here is a look at the YAMAL structure used by the above example. One of the keys to good templating is to build sensible and scalable variable structures. Note the following elements in this example:

- We seperate "HeadEnd" and "students" into seperate structures to allow us to refer to each seperately.
- Inside each "student" we include a list of "vpns", which could allow us to configure multiple VPNs with this template.

The template used for this example is found located in **"examples/vpn/vpn.yml"**

```
---
---
HeadEnd:
  subnets:
    - "10.10.0.0/18"
    - "10.11.0.0/18"
  vpn_ext_interface: ge-0/0/1.0
  vpn_ext_ip: 10.10.0.5

students:
  - pod-1:
    shortname: "Pod 1"
    desc: "This is Student Pod 1"
    active: true
    vpns:
      - vpn:
        active: true
        tunnel_int: st0.1
        int_descr: "Customer A Primary Link"
        active: true
        vpn_ext_interface: ge-0/0/2.0
        vpn_local_ip: 10.10.0.100
        vpn_tunnel_ip: 10.255.1.2
        vpn_zone: "vpn"
        shared_secret: "AwesomePassword123"
```

Loading the Templates

To allow us start immediately applying templates to devices without writing any Python code, we have prepared a small Python script to handle this for you.

The script is called "template-load.py" and is found in the **examples** direcory. The output below shows the arguments accepted by this script.

```
vagrant@NetDevOps-Student:/vagrant/examples$ python ./template-load.py
usage: template-load.py [-h] -b BUNDLE [-u USER] [-d DEVICE] [-p PASSWORD]
                        [-f {text,set,xml}]
template-load.py: error: argument -b/--bundle is required
vagrant@NetDevOps-Student:/vagrant/examples$
```

Once run here are the set commands that will be loaded onto the device. Again if

additional elements are added they will be generated into individual set commands.

```
vagrant@NetDevOps-Student:/vagrant/examples$ python ./template-load.py -u n
##############################################################################
# 2015-03-30 08:39:28 Connecting to Device (172.16.0.1):
##############################################################################

##############################################################################
# 2015-03-30 08:39:29 Loading Template to Candidate Config:
##############################################################################

##############################################################################
# 2015-03-30 08:39:30 Performing Config Diff:
##############################################################################


[edit interfaces]
+    st0 {
+        unit 1 {
+            family inet {
+                address 10.255.1.2/30;
+            }
+        }
+    }
[edit security]
+    ike {
+        policy ike-policy1 {
+            mode main;
+            proposal-set standard;
+            pre-shared-key ascii-text "$9$7y-dwJGiPfzDi/tu1yrWLx7bYg4ZjkPaZ
+        }
+        gateway ike-gate {
+            ike-policy ike-policy1;
+            address 10.10.0.10;
+            external-interface ge-0/0/2.0;
+        }
+    }
+    ipsec {
+        policy vpn-policy1 {
+            proposal-set standard;
+        }
+        vpn ike-vpn {
+            bind-interface st0.1;
+            ike {
+                gateway ike-gate;
+                ipsec-policy vpn-policy1;
+            }
+        }
```

```
+    }
+    flow {
+        tcp-mss {
+            ipsec-vpn {
+                mss 1350;
+            }
+        }
+    }
+    nat {
+        source {
+            rule-set vpn_nat_out {
+                from zone trust;
+                to zone vpn;
+                rule interface-nat {
+                    match {
+                        source-address-name LocalNet;
+                        destination-address-name PrivateNet;
+                    }
+                    then {
+                        source-nat {
+                            interface;
+                        }
+                    }
+                }
+            }
+        }
+    }
[edit security zones security-zone untrust]
+     host-inbound-traffic {
+         system-services {
+             ike;
+         }
+     }
[edit security zones]
     security-zone untrust { ... }
+    security-zone vpn {
+        interfaces {
+            st0.1;
+        }
+    }

################################################################################
# 2015-03-30 08:39:30 Performing Commit Check:
################################################################################

True
Proceed with Commit? [y | N]: y
################################################################################
# 2015-03-30 08:39:33 Performing Config Commit:
################################################################################
```

```
##############################################################################

##############################################################################
# 2015-03-30 08:39:33 Disconnecting from Device (172.16.0.1):
##############################################################################

vagrant@NetDevOps-Student:/vagrant/examples$
```

In the above example you see from the output logs that the script went through the following steps:

- Establish a NETCONF connection to the vSRX
- Load the template
- Perform a "show | compare" and print the output to screen
- Ensure that the configuration is a valid candidate config
- Ask you if you wish to commit the change
    - If you say "Y", the configuration is applied
    - If you say "N", the configuration is left as a candidate config
- Lastly, close out the NETCONF session to the vSRX and cleanup.

Validating the template

Now connect to your vSRX instance from your NetDevOpsVM and validate the change.

First, lets verify our Phase 1 configuration:

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ssh root@172.16
Password:
--- JUNOS 12.1X47-D20.7 built 2015-03-03 21:53:50 UTC
root@NetDevOps-SRX01% cli
root@NetDevOps-SRX01> show configuration security ike
policy ike-policy1 {
    mode main;
    proposal-set standard;
    pre-shared-key ascii-text "$9$7y-dwJGiPfzDi/tu1yrWLx7bYg4ZjkPaZA0IcvMaZ
}
gateway ike-gate {
    ike-policy ike-policy1;
    address 10.10.0.10;
    external-interface ge-0/0/2.0;
}
```

Second, lets verify our Phase 2 configuration:

```
root@NetDevOps-SRX01> show configuration security ipsec
policy vpn-policy1 {
    proposal-set standard;
}
vpn ike-vpn {
    bind-interface st0.1;
    ike {
        gateway ike-gate;
        ipsec-policy vpn-policy1;
    }
}
```

Lastly, lets check that we have the correct NAT configuration:

```
root@NetDevOps-SRX01> show configuration security nat
source {
    rule-set vpn_nat_out {
        from zone trust;
        to zone vpn;
        rule interface-nat {
            match {
                source-address-name LocalNet;
                destination-address-name PrivateNet;
            }
            then {
                source-nat {
                    interface;
                }
            }
        }
    }
}

root@NetDevOps-SRX01>
```

Enable Dynamic Routing

In the previous example, we configured our IPSec VPN between the Student Pod and the HeadEnd VPN server. At this stage we have a secure path between both networks, but we are not aware of any subnets on the other side of the VPN. We have two options to address this:

- Create Static routes for all the remote networks
- Configure a dynamic routing protocol such as OSPF

While creating static routes is usually an easier answer, we decided that OSPF would be a far more scalable answer to this problem. The advantage of using a dynamic routing protocol is that we can also learn about all the other student pods as they come on line.

Look at the code

In this example, we are going to use a simple Python script to connect to our lab device and commit a single configuration.

To keep things as simple as possible, we have embedded all of our Junos configuration directly into the script itself.

```
from jnpr.junos.utils.config import Config
from jnpr.junos import Device


dev = Device( user='netconf', host='172.16.0.1', password='test123' )
dev.open()

pod_number = "1"

set_cfg = """
set interfaces lo0.0 family inet address 10.255.255.{0}/32
set security zones security-zone trust interfaces lo0.0

set security zones security-zone vpn host-inbound-traffic system-services a
set security zones security-zone vpn host-inbound-traffic protocols ospf

set protocols ospf area 0 interface st0.{0}
set protocols ospf area 0 interface lo0.0 passive
""".format(pod_number)

'''
If you run the script with the wrong variables, the below delete commands s

delete interfaces lo0.0
delete security zones security-zone trust interfaces lo0.0
delete security zones security-zone vpn host-inbound-traffic system-service
delete security zones security-zone vpn host-inbound-traffic protocols ospf
delete protocols ospf
'''

dev.bind(cu=Config)
dev.cu
dev.cu.lock()

dev.cu.load(set_cfg, format='set')

commit_diff = dev.cu.diff()
print "Diff: %s" % (commit_diff)
dev.cu.commit()
dev.cu.unlock()

dev.close()
```

Lets step through the process:

- In the first two lines we tell Python to load both the Configuration elements of the PyEZ library as well as the standard device libraries
- Next we establish a NETCONF session to our lab SRX using fixed credentials
- To make this script a little more scalable we define a variable called "pod_number" and we assign it the value we received from the regisrtation process.
    - **Please modify this value in your editor to match your correct pod number.**
- We create a simple multiline string called "set_cfg" and insert normal set commands we would use on the Junos CLI
    - In this example you can see that we have replaced some of the set commands with "{0}" which is used by Python to indicate that we should insert some variables at this point in the string
    - The ".format(pod_number)" is used to indicate which variable should be inserted into the formatted string.
        - Note: Even though we reference the "{0}" varable twice in the string, we only need to reference the variable once in the format().
- Mistakes are bound to happen, so we have included the commands you may need to paste to the SRX to role back these changes if you made a mistake with your pod number
- The next three lines open a new candidate configuration on the SRX and ensure that we lock the configuration so that nobody else can make changes until we have completed our changes.
- To load the our formatted string into the SRX we call the "dev.cu.load(set_cfg, format='set')" command to send our string and tell Python which format we are using (In this case our string is using Junos set format)
- Following good configuration practices we perform a Junos "show | compare" using the "dev.cu.diff()" function and print that value to screen.
- The remaining lines commit the configuration, unlock the configuration and close the NETCONF session.

NOTE: Best Practice programming should include Exception Handling code to catch any errors, but in the interest of keeping this example simple we have not included those features here.

Running the script

To execute the script for this exercise, ensure that you are in the examples directory and call the script as shown below.

```
vagrant@NetDevOps-Student:~$ cd /vagrant/examples/
vagrant@NetDevOps-Student:/vagrant/examples$
vagrant@NetDevOps-Student:/vagrant/examples$ python ./conf-ospf.py
Diff:
[edit interfaces]
+   lo0 {
+       unit 0 {
+           family inet {
+               address 10.255.255.1/32;
+           }
+       }
+   }
[edit]
+  protocols {
+      ospf {
+          area 0.0.0.0 {
+              interface st0.1;
+              interface lo0.0 {
+                  passive;
+              }
+          }
+      }
+  }
[edit security zones security-zone trust interfaces]
      ge-0/0/1.0 { ... }
+     lo0.0;
[edit security zones security-zone vpn]
+     host-inbound-traffic {
+         system-services {
+             all;
+         }
+         protocols {
+             ospf;
+         }
+     }

vagrant@NetDevOps-Student:/vagrant/examples$
```

If everything works to plan, your output should show the output of the "show | compare" and then return you to the prompt.

Verify Our Changes

From our SRX session we can verify that our changes are applied correctly

```
root@NetDevOps-SRX01> show configuration interfaces lo0
unit 0 {
    family inet {
        address 10.255.255.1/32;
    }
}

root@NetDevOps-SRX01>
root@NetDevOps-SRX01> show configuration protocols
ospf {
    area 0.0.0.0 {
        interface st0.1;
        interface lo0.0 {
            passive;
        }
    }
}

root@NetDevOps-SRX01>
```

Adding VPN Firewall Policies

At this point in the lab your vSRX will have access to the rest of the lab. This run will generate the necessary policies to allow us to access the host over the VPN.

Using Ansible to enable firewall policies

For this step we will use Ansible to create both our firewall policies and the required address book objects. This is one of the most requested automation elements when dealing with a firewall. Typically once a firewall is deployed and integrated into the network there are few changes that are required on the networking side of things. However the majority of heavy lifting is centered around managing security policies and the associated address book objects. Address book objects are typically quite painful due to the need to create, manage, and have the correct association to zones. Luckily using Ansible makes these tasks a snap.

Reviewing the playbook

First let's take a look at the playbook that is used to accomplish this task. We briefly looked at this playbook during the Ansible overview section, but now we will dive deeper into the applied steps.

Playbook Review

1. Define the name of the playbook - Configure basic firewall policies
    - This will be displayed and logged as you start to run the playbook
2. Define the hosts the playbook should be applied to
    - In this case we use the group "**mysrx**" to apply to
        - The host list is picked up from either the default Ansible host list in "/etc/ansible/hosts"
        - Alternatively when the playbook is run you can specify your own custom inventory
3. Connection is defined to as local - Typically when Ansible runs it transports an execution environment over to the host and runs it
    - Because this will not work on Junos hosts we use connection defined to

local to run the execution environment

4. Gather facts
    - Ansible will gather local facts about the host such as interfaces and hostnames
    - Because this isn't possible on Junos we disable this feature
5. Vars - These are the variables that we will use to apply to our tasks
    - They can be applied at many different locations for our run
    - But to keep everything together we have included the variables into the playbook
    - address_entires will be used to generate the address book entries
    - fw_policy_info will be used to define our policies
        - In this run we will be generating a policy for the VPN zone
6. Tasks - These are the tasks that we will use
    - The build phase for the playbook generates the Junos config from the templates
    - The apply phase will apply the configuration to the device
    - This will be run as two separate commits, but in doing so we can simplify the tasks and see which step fails

Playbook

```
---
- name: Configure VPN firewall policies
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"
    address_entries: [ {'name':'NetDevOpsVM','prefix':'172.16.0.10/32'} ]
    fw_policy_info: [ {'policy_name':'Allow_Policy','src_zone':'trust','dst

  tasks:
    - name: Build address book entries
      template: src=templates/fw_address_book.set.j2 dest={{build_dir}}/fw_
      with_items: address_entries

    - name: Apply address book entries
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use


    - name: Build firewall policies config template
      template: src=templates/fw_policy.set.j2 dest={{build_dir}}/fw_policy
      with_items: fw_policy_info

    - name: Apply firewall policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

Address book template

- The address book template loops through the address entries in the variable
- It generates one "set" configuration line per loop
- Here we are generating one

```
{% for i in address_entries %}
set security address-book global address {{ i.name }} {{ i.prefix }}
{% endfor %}
```

Output after generation

- This is the generated output from the template being applied with variables
- These commands are then committed to Junos
- If one or more of the entries are already created it will recognize this as "OK"

```
set security address-book global address NetDevOpsVM 172.16.0.10/32
```

Policy Template

This template is a bit more complex. We need to loop through source IPs, Destination IPs, and applications. Each loop through these variables will generate a single line of "set" commands. Creating the template this way allows us to reuse it in the future when we have a larger list of addresses and applications to apply. In this example we generate multiple applications for the policy.

```
{% for item in fw_policy_info %}
    {% for i in item.src_ips %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
    {% endfor %}
    {% for i in item.dst_ips %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
    {% endfor %}
    {% for i in item.apps %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
    {% endfor %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
{% endfor %}
```

Output after generation

Once run here are the set commands that will be loaded onto the device. Again if additional elements are added they will be generated into individual set commands. While we could create the applications with a single command using a list of applications this keeps it simple to implement using individual commands.

```
set security policies from-zone trust to-zone vpn policy Allow_Policy match
set security policies from-zone trust to-zone vpn policy Allow_Policy match
set security policies from-zone trust to-zone vpn policy Allow_Policy match
set security policies from-zone trust to-zone vpn policy Allow_Policy match
set security policies from-zone trust to-zone vpn policy Allow_Policy match
set security policies from-zone trust to-zone vpn policy Allow_Policy match
set security policies from-zone trust to-zone vpn policy Allow_Policy then
```

Running the playbook

To run the playbook you must use the "ansible-playbook" command. We must specify the inventory file and the playbook to apply. The templates will be automatically loaded from the playbook. Since cowsay is installed it will also add the comical cow for our enjoyment. If you dislike our bovine friend then you can simply remove cowsay from your running host.

Playbook Command

Ensure before running the command you are in the "**ansible**" directory.

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
```

Playbook Run Example

Once run the output should look like the following

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
 ____________________________________
< PLAY [Configure firewall policies] >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


 _____________________________________
< TASK: Build address book entries >
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1] => (item={'prefix': '172.16.0.10/32', 'name': 'NetDev
 _____________________________________
< TASK: Apply address book entries >
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (  )\       )\/\
```

```
                ||----w |
                ||     ||


changed: [172.16.0.1]

 ________________________________________________
< TASK: Build firewall policies config template >
 ------------------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'vpn', 'sr

 ______________________________
< TASK: Apply firewall policies >
 ------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1]

 _____________
< PLAY RECAP >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


172.16.0.1                 : ok=4    changed=4    unreachable=0    failed=0
```

Validating the playbook run

Now connect to your vSRX instance from your NetDevOpsVM and validate the change

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ssh root@172.16
Password:
--- JUNOS 12.1X47-D20.7 built 2015-03-03 21:53:50 UTC
clclroot@NetDevOps-SRX01% cli
root@NetDevOps-SRX01> show configuration security address-book
global {
    address LocalNet 172.16.0.0/24;
    address PrivateNet 192.168.10.0/24;
    address PublicNet 10.10.0.0/22;
    address NetDevOpsVM 172.16.0.10/32;
}

root@NetDevOps-SRX01> show configuration security policies from-zone trust
policy Allow_Policy {
    match {
        source-address NetDevOpsVM;
        destination-address PrivateNet;
        application [ junos-http junos-ping junos-ssh junos-https ];
    }
    then {
        permit;
    }
}

root@NetDevOps-SRX01> exit

root@NetDevOps-SRX01% exit
logout
Connection to 172.16.0.1 closed.
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$
```

Automating License and IDP Package Installation

Important The licenses for this step are not included within the lab due to legal reasons. You will need to place the license files "utm.txt" and "appsecure.txt" into the licenses directory within the repository.

An example license file will look something like this. It may also contain additional comments but the actual license component will look like this.

```
JUNOS604043 xxxqic a6ajc4 a2ob24 bhhqdj yhlqeq 6anha5
            xxx5yb u4dvyc dpagtq oxbmhd smjxgm yecmbq
            xxxtqo brg4ga 4tcbij lwk3tz ouqfu2 dbnztq
            xxxdtn 6mhr4d bldwa7 26bcdg inj3wb p6tjq4
            xxxq4x ms5lt4 5coxjv 52756a n3bark dw64
```

Important

When deploying IPS there are a few steps you must take on a device before it is ready to utilize an IPS policy.

First a license is required to installed on the vSRX. This gives the device the ability to run IPS, AppID, and UTM. Once installed these features are enabled and ready to be used. The IPS and UTM licenses are required to be installed separately.

Once installed you can then download and install the IPS signature packs. Doing this on a single device isn't the most challenging thing to do, but doing it for dozens or hundreds of devices quickly becomes a pain.

In this section we will look at two tools that can be used to simplify the process. Both tools are written in Python and demonstrate two different methods of running commands on the vSRX.

Tools

- tools/licensetool.py
- tools/idpsecpack.py
- Python interpreter

Automating License Installation

Installing a license on an SRX is a bit of a challenge. It requires you to enter a command on the CLI. Unfortunately adding a license is not an available RPC nor can use use the "" RPC to call it. Fortunately with automation we can work around this issue. The tool "licensetool.py" can be found in the "tools" directory in the base of the student repository.

The "licensetool.py" uses a Python library called Paramiko that allows you to automation SSH connections. In fact this tool is used in the Junos PyEZ libraries as well. This tool provides a few different command line flags.

```
usage: licensetool.py [-h] [--host HOST] [--username USERNAME]
                      [--password PASSWORD] [--url URL]

Process user input

optional arguments:
  -h, --help           show this help message and exit
  --host HOST          Specify host to connect to
  --username USERNAME  Specify the username
  --password PASSWORD  Specify the password
  --url URL            Specify the license URL
```

The code for the tool is fairly simple. You can open the file or look at the code below. To see an example of how it is written.

Steps in Script

- 1 Gathers command line options from user
- 2 Connects to the host using the specified username and password
- 3 Executes the command `/usr/sbin/cli -c request system license add URL`
- 4 Prints the output of the result of the command

```
#!/usr/bin/env python

import paramiko
import argparse
import logging

logger = paramiko.util.logging.getLogger()
logger.setLevel(logging.WARN)


parser = argparse.ArgumentParser(description='Process user input')
parser.add_argument("--host", dest="host", default="",metavar="HOST",help="
parser.add_argument("--username", dest="username", metavar="USERNAME",help=
parser.add_argument("--password", dest="password", metavar="PASSWORD",help=
parser.add_argument("--url", dest="url",metavar="URL",help="Specify the lic
args = parser.parse_args()

host = args.host
username = args.username
password = args.password
licenseURL = args.url

client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(host,username=username,password=password,look_for_keys=False
stdin, stdout, stderr = client.exec_command('/usr/sbin/cli -c "request syst

data = stdout.read()
print data
```

You can see that we first take a set of command line arguments using the argparse library. This library was added in Python 2.7 and it is extremely useful for dealing with command line arguments. Once the arguments are received then the script makes an SSH connection to the host on your behalf using the provided credentials to authenticate to the vSRX. Once connected we request SSH to execute a command on the remote device. The command we call is the cli binary. This is the same CLI that you use when you connect into a device. We simply use this with the "-c" flag to run a command. The command chosen is "request system license add URL". This fetches the license from a specific URL and then loads it onto the device. This may take a few moments depending on how busy the vSRX is when you run the command. Once completed it will print out the response from the command. The tool does not have any built in error checking so it may throw errors in the event it runs into any problems.

Using the tool to fetch your licenses

To run the tool change directories to "tools/". From here run the commands specified below. This will fetch your licenses from the proctors server.

```
python licensetool.py  --user root --password Juniper --host 172.16.0.1 --u
```

```
python licensetool.py  --user root --password Juniper --host 172.16.0.1 --u
```

Automating both the IDP Package and Template Installation

Now that we have a license to the device we can now download and install the signature pack. To do this we have created a separate tool. The commands that we need to run can be called using NETCONF RPCs. This means we can utilize the Junos PyEZ library and do not have to use the lower level Paramiko library.

Much like we did in the license tool we have a set of command line flags. In fact it is the same set of command line flags that we used before, minus the "--url" option as that is not required.

```
usage: idpsecpack.py [-h] [--host HOST] [--username USERNAME]
                     [--password PASSWORD]

Process user input

optional arguments:
  -h, --help           show this help message and exit
  --host HOST          Specify host to connect to
  --username USERNAME  Specify the username
  --password PASSWORD  Specify the password
```

The code for the script introduces the usage of loops and regular expressions. The loops are used to periodically request the status of the download and installation of the security package. We use regular expressions to match the output of the command to verify that the download has been completed. While we would hope that the XML response would be descriptive enough for us to look for a completed tag,

sometimes it will not be. In this case the the XML response tag is the same no matter the status of the download or installation. Because of this we must search for the response to validate the output. The particular string that we are looking for is "DONE;". This means that the transaction has completed and we no longer need to poll for the status of the change. For each RPC that it runs it outputs the response to the standard out.

Steps in Script

1. Gathers command line options from user
2. Connects to the host using the specified username and password
3. Requests to download the security package
4. Checks the status of the download
    - Sleeps two seconds at the end of the loop to give the package time to download
    - Once the "DONE;" message is seen then we break out of the loop
5. Requests to install the security package
6. Checks the status of the installation
    - Sleeps two seconds at the end of the loop to give the package time to install
    - Once the "DONE;" message is seen then we break out of the loop
7. Requests to download the policy templates
8. Checks the status of the download
    - Sleeps two seconds at the end of the loop to give the templates time to download
    - Once the "DONE;" message is seen then we break out of the loop
9. Requests to install the policy templates
10. Checks the status of the installation
    - Sleeps two seconds at the end of the loop to give the templates time to install
    - Once the "DONE;" message is seen then we break out of the loop

```
#!/usr/bin/env python

import argparse
import time
import re
import xml.etree.ElementTree as ET

from jnpr.junos import Device

parser = argparse.ArgumentParser(description='Process user input')
parser.add_argument("--host", dest="host", default="",metavar="HOST",help="
```

```
parser.add_argument("--username", dest="username", metavar="USERNAME",help=
parser.add_argument("--password", dest="password", metavar="PASSWORD",help=
args = parser.parse_args()

host = args.host
username = args.username
password = args.password

#first we instantiate an instance of the device
junos_dev = Device(host=host, user=username, password=password)
#now we can connect to the device
junos_dev.open()
#run the RPC to download the security package
download_result = junos_dev.rpc.request_idp_security_package_download()
#output to a sting
print ET.tostring(download_result,encoding="utf8", method="text")

#REGEX to match the done condition
matchDone = re.compile('Done;.*',re.MULTILINE)

#loop through status until OK is seen
while True:
    download_status = junos_dev.rpc.request_idp_security_package_download(s
    print ET.tostring(download_status,encoding="utf8", method="text")
    info = download_status.findtext('secpack-download-status-detail')
    if matchDone.match(info):
        print "Download completed"
        break
    time.sleep(2)

#run the RPC to install the security package
install_result = junos_dev.rpc.request_idp_security_package_install()
#output to a sting
print ET.tostring(install_result,encoding="utf8", method="text")

#loop through status until OK is seen
while True:
    install_status = junos_dev.rpc.request_idp_security_package_install(sta
    print ET.tostring(install_status,encoding="utf8", method="text")
    info = install_status.findtext('secpack-status-detail')
    if matchDone.match(info):
        print "Installation completed"
        break
    time.sleep(2)

#run the RPC to download the download policy templates
download_tmpl_result = junos_dev.rpc.request_idp_security_package_download(
#output to a sting
print ET.tostring(download_tmpl_result,encoding="utf8", method="text")
```

```
#loop through status until OK is seen
matchDone = re.compile('Done;.*',re.MULTILINE)

while True:
    download_status = junos_dev.rpc.request_idp_security_package_download(s
    print ET.tostring(download_status,encoding="utf8", method="text")
    info = download_status.findtext('secpack-download-status-detail')
    if matchDone.match(info):
        print "Download completed"
        break
    time.sleep(2)

#run the RPC to install the download policy templates
install_tmpl_result = junos_dev.rpc.request_idp_security_package_install(po
#output to a sting
print ET.tostring(install_tmpl_result,encoding="utf8", method="text")

#loop through status until OK is seen
while True:
    install_status = junos_dev.rpc.request_idp_security_package_install(sta
    print ET.tostring(install_status,encoding="utf8", method="text")
    info = install_status.findtext('secpack-status-detail')
    if matchDone.match(info):
        print "Installation completed"
        break
    time.sleep(2)
```

Using the tool to download and install the package

To run the tool change directories to "tools/". From here run the commands specified below. This will download and install the packages from the Internet.

```
python idpsecpack.py --user root --password Juniper --host 172.16.0.1
```

Creating Application Policies

Now it is time to start using some of the more advanced firewall capabilities. The first set of policies we will create are application firewall or AppFW policies. This feature allows us to look into the data being sent over the connection.

Creating Application Policies with Ansible

Before we created basic firewall policies using Ansible. Now we will create application firewall policies. While the concept is the same there is an additional challenge. In a basic firewall policy you can add what amounts to just ports now you have to manage the applications that go over those ports. Typically you will want to apply many more specific applications that you want to block. You can also add other elements such as application groups. Because of this managing AppFW policies can be quite tedious. But as we will see there are a variety of methods and tools we can employ to simplify the process.

Reviewing the playbook

First let's take a look at the playbook that is used to accomplish this task.

Playbook Review

1. Define the name of the playbook - Configure AppFirewall policies
    - This will be displayed and logged as you start to run the playbook
2. Define the hosts the playbook should be applied to
    - In this case we use the group "**mysrx**" to apply to
        - The host list is picked up from either the default Ansible host list in "/etc/ansible/hosts"
        - Alternatively when the playbook is run you can specify your own custom inventory
3. Connection is defined to as local - Typically when Ansible runs it transports an execution environment over to the host and runs it
    - Because this will not work on Junos hosts we use connection defined to local to run the execution environment

4. Gather facts
    - Ansible will gather local facts about the host such as interfaces and hostnames
    - Because this isn't possible on Junos we disable this feature
5. Vars - These are the variables that we will use to apply to our tasks
    - They can be applied at many different locations for our run
    - But to keep everything together we have included the variables into the playbook
    - appfw_to_policy_info will be used to apply the AppFW policy to our stateful policy
    - appfw_policy_info will be used to define our policies
6. Tasks - These are the tasks that we will use
    - The build phase for the playbook generates the Junos config from the templates
    - The apply phase will apply the configuration to the device
    - This will be run as two separate commits, but in doing so we can simplify the tasks and see which step fails

Playbook

```
---
- name: Configure AppFirewall policies
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"
    appfw_to_policy_info: [{"src_zone":"trust","dst_zone":"untrust","policy
    appfw_policy_info: [{"rule_set":"ruleset1","rule_set_default_action":"p

  tasks:
    - name: Build app firewall policies
      template: src=templates/appfw_policy.set.j2 dest={{build_dir}}/appfw_
      with_items: appfw_policy_info

    - name: Apply app firewall policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use

    - name: Apply app firewall rules to policy
      template: src=templates/appfw_to_policy.set.j2 dest={{build_dir}}/app
      with_items: appfw_to_policy_info

    - name: Apply firewall policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

AppFW Policy Template

- It generates one "set" configuration line per loop
- Here we are generating three lines

```
{% for item in appfw_policy_info %}
    {% for i in item.rules %}
        {% for app in i.dynapps %}
set security application-firewall rule-sets {{ item.rule_set }} rule {{ i.n
        {% endfor %}
set security application-firewall rule-sets {{ item.rule_set }} rule {{ i.n
    {% endfor %}
set security application-firewall rule-sets {{ item.rule_set }} default-rul
{% endfor %}
```

Output after generation

- This is the generated output from the template being applied with variables
- These commands are then committed to Junos
- If one or more of the entries are already created it will recognize this as "OK"

```
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 match dynam
set security application-firewall rule-sets ruleset1 rule rule1 then deny
set security application-firewall rule-sets ruleset1 default-rule permit
```

Template to apply AppFW policy to a firewall policy

In this template we simply apply the AppFW policy to the stateful rule. We still use it as a loop in the event that we want to apply multiple policies at the same time.

```
{% for item in appfw_to_policy_info %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
{% endfor %}
```

Output after generation

Once run here are the set commands that will be loaded onto the device. Again if additional elements are added they will be generated into individual set commands.

```
set security policies from-zone trust to-zone untrust policy Allow_Policy t
```

Running the playbook

To run the playbook you must use the "ansible-playbook" command. We must specify the inventory file and the playbook to apply. The templates will be automatically loaded from the playbook. Since cowsay is installed it will also add the comical cow for our enjoyment. If you dislike our bovine friend then you can simply remove cowsay from your running host.

Playbook Command

Ensure before running the command you are in the "**ansible**" directory.

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
```

Playbook Run Example

Once run the output should look like the following

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
 ______________________________________
< PLAY [Configure AppFirewall policies] >
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

```
                ||      ||


 ____________________________________
< TASK: Build app firewall policies >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1] => (item={'rules': [{'action': 'deny', 'dynapps': ['j

 ____________________________________
< TASK: Apply app firewall policies >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


ok: [172.16.0.1]

 _____________________________________________
< TASK: Apply app firewall rules to policy >
 ---------------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


ok: [172.16.0.1] => (item={'appfw_rule_set': 'ruleset1', 'src_zone': 'trust

 _______________________________
< TASK: Apply firewall policies >
 -------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


ok: [172.16.0.1]

 ____________
< PLAY RECAP >
 ------------
```

```
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


172.16.0.1                 : ok=4    changed=1    unreachable=0    failed=0
```

Validating the playbook run

Now connect to your vSRX instance from your NetDevOpsVM and validate the change

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ssh root@172.16
Password:
--- JUNOS 12.1X47-D20.7 built 2015-03-03 21:53:50 UTC
root@NetDevOps-SRX01% cli

root@NetDevOps-SRX01> show security application-firewall rule-set all
Rule-set: ruleset1
    Rule: rule1
        Dynamic Applications: junos:GOOGLE, junos:GOOGLE-ACCOUNTS, junos:GO
        junos:GOOGLE-ANALYTICS-TRACKING, junos:GOOGLE-APPENGINE, junos:GOOG
        junos:GOOGLE-DOCS-DRAWING, junos:GOOGLE-DOCS-FORM, junos:GOOGLE-DOC
        junos:GOOGLE-DOCS-WORD-DOCUMENT, junos:GOOGLE-DRIVE, junos:GOOGLE-E
        junos:GOOGLE-MOBILE-MAPS-APP, junos:GOOGLE-PICASA, junos:GOOGLE-PLU
        junos:GOOGLE-SAFEBROWSE-UPDATE, junos:GOOGLE-SKYMAP, junos:GOOGLE-S
        junos:GOOGLE-TRANSLATE, junos:GOOGLE-UPDATE, junos:GOOGLE-VIDEOS, j
        SSL-Encryption: any
        Action:deny
        Number of sessions matched: 0
        Number of sessions redirected: 0
Default rule:permit
        Number of sessions matched: 0
        Number of sessions redirected: 0
Number of sessions with appid pending: 0

root@NetDevOps-SRX01> show configuration security application-firewall
rule-sets ruleset1 {
    rule rule1 {
        match {
            dynamic-application [ junos:GOOGLE junos:GOOGLE-ACCOUNTS junos:
        }
        then {
            deny;
```

```
        }
    }
    default-rule {
        permit;
    }
}

root@NetDevOps-SRX01> show security policies from-zone trust to-zone untrus
From zone: trust, To zone: untrust
  Policy: default-permit, State: enabled, Index: 5, Scope Policy: 0, Sequen
    Source addresses: any
    Destination addresses: any
    Applications: any
    Action: permit
  Policy: Allow_Policy, State: enabled, Index: 7, Scope Policy: 0, Sequence
    Source addresses: LocalNet
    Destination addresses: PrivateNet
    Applications: any
    Action: permit, application services
  Application firewall:ruleset1

root@NetDevOps-SRX01> show configuration security policies from-zone trust
policy default-permit {
    match {
        source-address any;
        destination-address any;
        application any;
    }
    then {
        permit;
    }
}
policy Allow_Policy {
    match {
        source-address LocalNet;
        destination-address PrivateNet;
        application any;
    }
    then {
        permit {
            application-services {
                application-firewall {
                    rule-set ruleset1;
                }
            }
        }
    }
}

root@NetDevOps-SRX01>
```

```
root@NetDevOps-SRX01>

root@NetDevOps-SRX01> exit

root@NetDevOps-SRX01% exit
logout
Connection to 172.16.0.1 closed.
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$
```

Testing

```
curl http://10.10.0.10:8080 -X GET -H "Host: google.com"
```

Creating IPS Policies

Much like AppFW policies IDP policies can also be tasking to manager. In this section we will use Ansible to create our IDP policies as well.

Creating IPS Policies with Ansible

In our third example of policy management policies we will now apply the same ideas to IPS. Much like AppFW policies IDP policies can use thousands of different objects to create the the policies. While the majority of customers use attack groups there are many that still manage individual attacks due to their complex security requirements.

Reviewing the playbook

First let's take a look at the playbook that is used to accomplish this task.

Playbook Review

1. Define the name of the playbook - Configure IPS policies
    - This will be displayed and logged as you start to run the playbook
2. Define the hosts the playbook should be applied to
    - In this case we use the group "**mysrx**" to apply to
        - The host list is picked up from either the default Ansible host list in "/etc/ansible/hosts"
        - Alternatively when the playbook is run you can specify your own custom inventory
3. Connection is defined to as local - Typically when Ansible runs it transports an execution environment over to the host and runs it
    - Because this will not work on Junos hosts we use connection defined to local to run the execution environment
4. Gather facts
    - Ansible will gather local facts about the host such as interfaces and hostnames
    - Because this isn't possible on Junos we disable this feature
5. Vars - These are the variables that we will use to apply to our tasks
    - They can be applied at many different locations for our run

- But to keep everything together we have included the variables into the playbook
- idp_policy_name will be used to apply the IDP policy to our stateful policy
- idp_policy_info will be used to define our policies

6. Tasks - These are the tasks that we will use
   - The build phase for the playbook generates the Junos config from the templates
   - The apply phase will apply the configuration to the device
   - This will be run as two separate commits, but in doing so we can simplify the tasks and see which step fails

Playbook

```
---
- name: Configure IPS policies
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"
    idp_policy_name: "ips-policy1"
    idp_to_policy_info: [{"src_zone":"trust","dst_zone":"untrust","policy_n
    idp_policy_info: [{"policy_name":"ips-policy1","rules":[{"name":"rule1"

  tasks:
    - name: Build ips policies config template
      template: src=templates/idp_policy.set.j2 dest={{build_dir}}/idp_poli
      with_items: ips_policy_info

    - name: Apply ips policies
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use

    - name: Build ips policy apply template
      template: src=templates/idp_policy_activate.set.j2 dest={{build_dir}}

    - name: Activate ips policy
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

IDP Policy Template

- It generates one "set" configuration line per loop
- Here we are generating several line

- Due to the potential complexities of an IPS policy this is our most complex template yet
- Each section will only generate the appropriate configuration if the elements exist
    - For example if rule.predef_attacks is defined then it will be looped through
    - If not then that part of the template will be ignored
    - If it does exist then it will be executed and the configuration will be generated

```
{% for item in idp_policy_info %}
set security idp idp-policy {{ item.policy_name }}
    {% for rule in item.rules %}
        {% for i in rule.src_ips %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
        {% endfor %}
        {% for i in rule.dst_ips %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
        {% endfor %}
        {% for i in rule.apps %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
        {% endfor %}
        {% if rule.predef_attacks is defined %}
            {% for x in rule.predef_attacks %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
            {% endfor %}
        {% endif %}
        {% if rule.predef_attack_groups is defined %}
            {% for x in rule.predef_attack_groups %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
            {% endfor %}
        {% endif %}
        {% if rule.custom_attacks is defined %}
            {% for x in item.custom_attacks %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
            {% endfor %}
        {% endif %}
        {% if rule.custom_attack_groups is defined %}
            {% for x in item.custom_attack_groups %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
            {% endfor %}
        {% endif %}
        {% if rule.dyn_attack_groups is defined %}
            {% for x in item.custom_attacks %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
            {% endfor %}
        {% endif %}
        {% if rule.notification.log is defined and rule.notification.alert
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
        {% endif %}
```

```
        {% if rule.notification.log is defined %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
        {% endif %}
        {% if rule.severity is defined %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
        {% endif %}
set security idp idp-policy {{ item.policy_name }} rulebase-ips rule {{ rul
    {% endfor %}
{% endfor %}
```

Output after generation

- This is the generated output from the template being applied with variables
- These commands are then committed to Junos
- If one or more of the entries are already created it will recognize this as "OK"

```
set security idp idp-policy ips-policy1
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 match from-
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 match from-
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 match from-
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 match attac
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 match attac
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 then notifi
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 then notifi
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 then severi
set security idp idp-policy ips-policy1 rulebase-ips rule rule1 then action
```

Template to apply IPS policy to a firewall policy

In this template we apply the IDP policy to the stateful rule and activate the ips policy. We still use it as a loop in the event that we want to apply multiple policies at the same time.

```
set security idp active-policy {{ idp_policy_name }}
{% for item in idp_to_policy_info %}
set security policies from-zone {{ item.src_zone }} to-zone {{ item.dst_zon
{% endfor %}
```

Output after generation

Once run here are the set commands that will be loaded onto the device. Again if

additional elements are added they will be generated into individual set commands.

```
set security idp active-policy ips-policy1
set security policies from-zone trust to-zone untrust policy Allow_Policy t
```

Running the playbook

To run the playbook you must use the "ansible-playbook" command. We must specify the inventory file and the playbook to apply. The templates will be automatically loaded from the playbook. Since cowsay is installed it will also add the comical cow for our enjoyment. If you dislike our bovine friend then you can simply remove cowsay from your running host.

Playbook Command

Ensure before running the command you are in the "**ansible**" directory.

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
```

Playbook Run Example

Once run the output should look like the following

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$ ansible-playboo
 _______________________________
< PLAY [Configure IPS policies] >
 -------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


 ________________________________________
< TASK: Build ips policies config template >
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||     ||
```

```
                ||----w |
                ||     ||


ok: [172.16.0.1] => (item=ips_policy_info)

 ___________________________
< TASK: Apply ips policies >
 ---------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1]

 _____________________________________
< TASK: Build ips policy apply template >
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'untrust',

 ___________________________
< TASK: Activate ips policy >
 ---------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


changed: [172.16.0.1]

 ____________
< PLAY RECAP >
 ------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


172.16.0.1                 : ok=4    changed=3    unreachable=0    failed=0
```

```
vagrant@NetDevOps-Student:~/JNPRAutomateDemo-Class/ansible$
```

Validating the playbook run

Now connect to your vSRX instance from your NetDevOpsVM and validate the change

```
--- JUNOS 12.1X47-D20.7 built 2015-03-03 21:53:50 UTC
root@NetDevOps-SRX01% cli
root@NetDevOps-SRX01> show security idp status
State of IDP: Default,  Up since: 2015-03-25 22:39:58 UTC (05:30:12 ago)

Packets/second: 0               Peak: 0 @ 2015-03-26 04:09:11 UTC
KBits/second  : 0               Peak: 0 @ 2015-03-26 04:09:11 UTC
Latency (microseconds): [min: 0] [max: 0] [avg: 0]

Packet Statistics:
 [ICMP: 0] [TCP: 0] [UDP: 0] [Other: 0]

Flow Statistics:
  ICMP: [Current: 0] [Max: 0 @ 2015-03-26 04:09:11 UTC]
  TCP: [Current: 0] [Max: 0 @ 2015-03-26 04:09:11 UTC]
  UDP: [Current: 0] [Max: 0 @ 2015-03-26 04:09:11 UTC]
  Other: [Current: 0] [Max: 0 @ 2015-03-26 04:09:11 UTC]

Session Statistics:
 [ICMP: 0] [TCP: 0] [UDP: 0] [Other: 0]
  Policy Name : ips-policy1
  Running Detector Version : 12.6.130140822

root@NetDevOps-SRX01> show security idp policies
ID    Name                    Sessions    Memory      Detector
 3     ips-policy1             0           2720166     12.6.130140822

root@NetDevOps-SRX01> show configuration security idp
idp-policy ips-policy1 {
    rulebase-ips {
        rule rule1 {
            match {
                from-zone trust;
                source-address any;
                to-zone untrust;
                destination-address any;
                application default;
                attacks {
                    predefined-attack-groups [ "APP - Critical" "APP - Majo
```

```
                }
            }
            then {
                action {
                    drop-connection;
                }
                notification {
                    log-attacks {
                        alert;
                    }
                }
                severity critical;
            }
        }
    }
}
active-policy ips-policy1;

root@NetDevOps-SRX01> show configuration security policies from-zone trust
match {
    source-address LocalNet;
    destination-address any;
    application any;
}
then {
    permit {
        application-services {
            idp;
            application-firewall {
                rule-set ruleset1;
            }
        }
    }
}

root@NetDevOps-SRX01>
```

The Worst Day in DevOps

In a DevOps role you will have the **"occasional"** bad day. These days are marred by some intense anger by your customers as their service is degraded. These days are not fun and can be horribly stressful on you and your team. While you can not use automation to fix stupidity, you can have many of the recovery options of your topology available via automation. Automation can be used to reset the configuration of your network to the desired state.

Disaster Strikes

Oh no! Your firewalls are now no longer passing traffic. A team member accidentally made a mass change via a poorly written script. Several configuration elements have been changed across your topology and now your customers are furious they can't look at the new trailer for **Star Wars X: War of Automaters**. Calls are coming in from all over and managers are streaming into the room to watch your every move. How can you solve this problem?

Creating the Disaster

Because our lab is in a closed environment we need to simulate the creation of a disaster. We will do this using the WMD Ansible playbook

Playbook Review

This playbook is fairly simple. We are going to use it to generate commands to delete some hirearchies of the configuration. Even though we are not actually generating a configuration we still run through the template building step to stay in form with the remainder of the playbooks.

Playbook

```
---
- name: Weapons of Mass Destruction
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"

  tasks:
    - name: Build Weapons
      template: src=templates/mass_destruction.set.j2 dest={{build_dir}}mas

    - name: Launch destruction
      junos_install_config: host={{ inventory_hostname }} user={{ junos_use
```

Mass Destruction Template

This is the template that we are using to build the commands. As you can see there is no templating we are using a straight configuration to delete parts of the configuration.

```
delete security policies
delete security idp
delete security application-firewall
delete security nat
```

Playbook Command

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y
```

Playbook Run Example

Once run the output should look like the following.

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y

PLAY [Weapons of Mass Destruction] ************************************

TASK: [Build Weapons] *************************************************
ok: [172.16.0.1]

TASK: [Launch destruction] ********************************************
changed: [172.16.0.1]

PLAY RECAP ************************************************************
172.16.0.1                 : ok=2    changed=1    unreachable=0    failed=0
```

Validating the playbook run

Once the playbook is run the NetDevOpsVM that sits behind your SRX should no longer have access to the rest of the lab.

Recovering the Lab

With automation you are making an investment into the repeatability of your configuration steps. If you were to configure everything by hand then you will need to repeat the same steps by hand to recover a failure. With automation you will be able to repeat the configuration simply by rerunning your automation tools.

Initially the creation of tools requires an upfront investment of time greater than as if you were to configure something by hand. However you can quickly recoup the cost of automation by reusing the tooling several times. Their isn't an exact formula to determine the value of the time spent, however it can become quickly relevant once you have a nasty issue occur. Generally if your are choosing the correct actions to automate then it will end up being used frequently.

Another aspect to consider is your mental availability at the time of the event. A tool or script never forgets. It operates 100% of the time the exact same way. It can not have a hangover, be tired, or be upset that it is a weekend. You may not correctly remember the commands to enter, the details of the environment, or the results that are required. If your network consists of one firewall, then perhaps you can always remember the correct steps. Most likely this will not be the case.

Bringing Back your Firewall

As we went through each step of the lab we used many different automation methods. The one we generally settled on as the correct abstraction was to use Ansible. This provides you with a step based methodology to correctly apply all of the configuration elements to your vSRX. We have take these steps and rolled them into a single playbook. It references all of the other play books and then applies them in the same order that we went through the steps.

The ALL playbook

ONLY RUN THIS IF YOU HAVE LICENSES AVAILABLE

In this playbook we run all of the steps of the lab. We did not have to rewrite all of the existing play books, we simply included them into a single task list. This way all existing automation can be reused without a substantial rewrite of our existing code. Ansible will loop through each included play book running each of the tasks within the included playbook. This was also used by the authors to do rapid testing of the lab,

without needing to follow each of the steps.

Almost all of the tasks are done using the Ansible playbook methodology of generating a template configuration and then applying it. The AppSecure licenses and the AppSecure signature packs however use the same scripts.

```
---
- name: Run all tasks
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"

- include: basic_nat_policies.yml
- include: basic_firewall_policies.yml
- include: vpn_config.yml
- include: vpn_ospf_config.yml
- include: vpn_firewall_policies.yml
- include: vpn_nat_policies.yml
- include: idp_license.yml
- include: idp_secpak.yml
- include: appfw_policies.yml
- include: idp_policies.yml
```

Calling a script from a playbook

This play book will call the scripts the same way that we did from the command line. This allows us to reuse the tooling that was already built.

```
---
- name: Install IDP Licenses
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"

  tasks:
      - name: Install appsec Licenses
        script: ../../tools/licensetool.py  --user {{ junos_user }} --passw

      - name: Install utm Licenses
        script: ../../tools/licensetool.py  --user {{ junos_user }} --passw
```

This play book follows the same idea, however it is used to download the security pack.

```
---
- name: Install IDP Security Packages
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"

  tasks:
      - name: Install package
        script: ../../tools/idpsecpack.py --user {{ junos_user }} --passwor
```

Running the all Playbook

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y
```

Run Output

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y
```

```
PLAY [Run all tasks] ****************************************************

PLAY [Configure basic NAT policies] *************************************

TASK: [Build address book entries] **************************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNe
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] **************************************
changed: [172.16.0.1]

TASK: [Build NAT policies] **********************************************
changed: [172.16.0.1] => (item={'rules': [{'interface': True, 'dst_ips': ['

TASK: [Apply NAT policies] **********************************************
changed: [172.16.0.1]

PLAY [Configure basic firewall policies] ********************************

TASK: [Build address book entries] **************************************
ok: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNet'})
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] **************************************
ok: [172.16.0.1]

TASK: [Build firewall policies config template] *************************
changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'untrust',

TASK: [Apply firewall policies] *****************************************
changed: [172.16.0.1]

PLAY [Configure student vpn to headend] *********************************

TASK: [set flow tcp-mss] ************************************************
changed: [172.16.0.1] => (item={'mss': '1350', 'protocol': 'ipsec-vpn'})

TASK: [Apply flow tcp-mss] **********************************************
changed: [172.16.0.1]

TASK: [Build vpn tunnel interface] **************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'family': 'inet', 'zone': 'untrust', 'interface'

TASK: [Apply vpn tunnel interface] **************************************
changed: [172.16.0.1]
```

```
TASK: [Build vpn zone] ********************************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'family': 'inet', 'zone': 'untrust', 'interface'

TASK: [Apply vpn zone] ********************************************************
changed: [172.16.0.1]

TASK: [Build VPN Phase 1] *****************************************************
changed: [172.16.0.1] => (item={'ext_interface': 'ge-0/0/2.0', 'gateway_ip'

TASK: [Apply VPN Phase 1] *****************************************************
changed: [172.16.0.1]

TASK: [Build VPN Phase 2] *****************************************************
changed: [172.16.0.1] => (item={'ike_gateway': 'ike-vpn', 'tunnel_int': 'st

TASK: [Apply VPN Phase 2] *****************************************************
changed: [172.16.0.1]

PLAY [Configure student vpn ospf] *********************************************

TASK: [Build vpn tunnel interface] ********************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'addr': u'10.255.255.1/32', 'family': 'inet', 'i

TASK: [Apply vpn tunnel interface] ********************************************
changed: [172.16.0.1]

TASK: [Build vpn zone] ********************************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'addr': u'10.255.255.1/32', 'family': 'inet', 'i

TASK: [Apply vpn zone] ********************************************************
changed: [172.16.0.1]

TASK: [Build vpn OSPF] ********************************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'addr': u'10.255.255.1/32', 'family': 'inet', 'i

TASK: [Apply vpn OSPF] ********************************************************
changed: [172.16.0.1]

PLAY [Configure VPN firewall policies] ****************************************

TASK: [Build address book entries] ********************************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.10/32', 'name': 'NetDev

TASK: [Apply address book entries] ********************************************
changed: [172.16.0.1]
```

```
changed: [172.16.0.1]

TASK: [Build firewall policies config template] ***************************
changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'vpn', 'sr

TASK: [Apply firewall policies] *******************************************
changed: [172.16.0.1]

PLAY [Configure VPN NAT policies] *****************************************

TASK: [Build address book entries] ****************************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNe
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] ****************************************
ok: [172.16.0.1]

TASK: [Build NAT policies] ************************************************
changed: [172.16.0.1] => (item={'rules': [{'interface': True, 'dst_ips': ['

TASK: [Apply NAT policies] ************************************************
changed: [172.16.0.1]

PLAY [Install IDP Licenses] ***********************************************

TASK: [Install appsec Licenses] *******************************************
changed: [172.16.0.1]

TASK: [Install utm Licenses] **********************************************
changed: [172.16.0.1]

PLAY [Install IDP Security Packages] **************************************

TASK: [Install package] ***************************************************
changed: [172.16.0.1]

PLAY [Configure AppFirewall policies] *************************************

TASK: [Build app firewall policies] ***************************************
changed: [172.16.0.1] => (item={'rules': [{'action': 'deny', 'dynapps': ['j

TASK: [Apply app firewall policies] ***************************************
changed: [172.16.0.1]

TASK: [Apply app firewall rules to policy] ********************************
changed: [172.16.0.1] => (item={'appfw_rule_set': 'ruleset1', 'src_zone': '

TASK: [Apply firewall policies] *******************************************
changed: [172.16.0.1]
```

```
PLAY [Configure IPS policies] ********************************************

TASK: [Build ips policies config template] *******************************
changed: [172.16.0.1] => (item=ips_policy_info)

TASK: [Apply ips policies] ***********************************************
changed: [172.16.0.1]

TASK: [Build ips policy apply template] **********************************
changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'untrust',

TASK: [Activate ips policy] **********************************************
changed: [172.16.0.1]

PLAY RECAP ********************************************************************
172.16.0.1                 : ok=43   changed=40   unreachable=0    failed=0

vagrant@NetDevOps-Student:/vagrant/ansible$
```

Your device should now have the complete configuration on it.

Recovering the Lab

With automation you are making an investment into the repeatability of your configuration steps. If you were to configure everything by hand then you will need to repeat the same steps by hand to recover a failure. With automation you will be able to repeat the configuration simply by rerunning your automation tools.

Initially the creation of tools requires an upfront investment of time greater than as if you were to configure something by hand. However you can quickly recoup the cost of automation by reusing the tooling several times. Their isn't an exact formula to determine the value of the time spent, however it can become quickly relevant once you have a nasty issue occur. Generally if your are choosing the correct actions to automate then it will end up being used frequently.

Another aspect to consider is your mental availability at the time of the event. A tool or script never forgets. It operates 100% of the time the exact same way. It can not have a hangover, be tired, or be upset that it is a weekend. You may not correctly remember the commands to enter, the details of the environment, or the results that are required. If your network consists of one firewall, then perhaps you can always remember the correct steps. Most likely this will not be the case.

Bringing Back your Firewall

As we went through each step of the lab we used many different automation methods. The one we generally settled on as the correct abstraction was to use Ansible. This provides you with a step based methodology to correctly apply all of the configuration elements to your vSRX. We have take these steps and rolled them into a single playbook. It references all of the other play books and then applies them in the same order that we went through the steps.

The ALL playbook

ONLY RUN THIS IF YOU DO NOT HAVE LICENSES AVAILABLE

In this playbook we run all of the steps of the lab. We did not have to rewrite all of the existing play books, we simply included them into a single task list. This way all existing automation can be reused without a substantial rewrite of our existing code. Ansible will loop through each included play book running each of the tasks within the included playbook. This was also used by the authors to do rapid testing of the lab,

without needing to follow each of the steps.

Almost all of the tasks are done using the Ansible playbook methodology of generating a template configuration and then applying it. The AppSecure licenses and the AppSecure signature packs however use the same scripts.

```
---
- name: Run all tasks
  hosts: mysrx
  connection: local
  gather_facts: no
  vars:
    junos_user: "root"
    junos_password: "Juniper"
    build_dir: "/tmp/"

- include: basic_nat_policies.yml
- include: basic_firewall_policies.yml
- include: vpn_config.yml
- include: vpn_ospf_config.yml
- include: vpn_firewall_policies.ymla
- include: vpn_nat_policies.yml
- include: add_vpn_gateway.yml
#Since no licenses are avialable these tasjs are disabled
#- include: idp_license.yml
#- include: idp_secpak.yml
#- include: appfw_policies.yml
#- include: idp_policies.yml
```

Running the all Playbook

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y
```

Run Output

```
vagrant@NetDevOps-Student:/vagrant/ansible$ ansible-playbook -i inventory.y

PLAY [Run all tasks] ***********************************************

PLAY [Configure basic NAT policies] ********************************

TASK: [Build address book entries] *********************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNe
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
```

```
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] ************************************
changed: [172.16.0.1]

TASK: [Build NAT policies] ********************************************
changed: [172.16.0.1] => (item={'rules': [{'interface': True, 'dst_ips': ['

TASK: [Apply NAT policies] ********************************************
changed: [172.16.0.1]

PLAY [Configure basic firewall policies] ******************************

TASK: [Build address book entries] ************************************
ok: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNet'})
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] ************************************
ok: [172.16.0.1]

TASK: [Build firewall policies config template] ***********************
changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'untrust',

TASK: [Apply firewall policies] ***************************************
changed: [172.16.0.1]

PLAY [Configure student vpn to headend] *******************************

TASK: [set flow tcp-mss] **********************************************
changed: [172.16.0.1] => (item={'mss': '1350', 'protocol': 'ipsec-vpn'})

TASK: [Apply flow tcp-mss] ********************************************
changed: [172.16.0.1]

TASK: [Build vpn tunnel interface] ************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'family': 'inet', 'zone': 'untrust', 'interface'

TASK: [Apply vpn tunnel interface] ************************************
changed: [172.16.0.1]

TASK: [Build vpn zone] ************************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'family': 'inet', 'zone': 'untrust', 'interface'

TASK: [Apply vpn zone] ************************************************
changed: [172.16.0.1]
```

```
TASK: [Build VPN Phase 1] ***********************************************
changed: [172.16.0.1] => (item={'ext_interface': 'ge-0/0/2.0', 'gateway_ip'

TASK: [Apply VPN Phase 1] ***********************************************
changed: [172.16.0.1]

TASK: [Build VPN Phase 2] ***********************************************
changed: [172.16.0.1] => (item={'ike_gateway': 'ike-vpn', 'tunnel_int': 'st

TASK: [Apply VPN Phase 2] ***********************************************
changed: [172.16.0.1]

PLAY [Configure student vpn ospf] ***************************************

TASK: [Build vpn tunnel interface] **************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'addr': u'10.255.255.1/32', 'family': 'inet', 'i

TASK: [Apply vpn tunnel interface] **************************************
changed: [172.16.0.1]

TASK: [Build vpn zone] **************************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'addr': u'10.255.255.1/32', 'family': 'inet', 'i

TASK: [Apply vpn zone] **************************************************
changed: [172.16.0.1]

TASK: [Build vpn OSPF] **************************************************
changed: [172.16.0.1] => (item={'addr': u'10.255.1.2/30', 'family': 'inet',
ok: [172.16.0.1] => (item={'addr': u'10.255.255.1/32', 'family': 'inet', 'i

TASK: [Apply vpn OSPF] **************************************************
changed: [172.16.0.1]

PLAY [Configure VPN firewall policies] **********************************

TASK: [Build address book entries] **************************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.10/32', 'name': 'NetDev

TASK: [Apply address book entries] **************************************
changed: [172.16.0.1]

TASK: [Build firewall policies config template] *************************
changed: [172.16.0.1] => (item={'src_zone': 'trust', 'dst_zone': 'vpn', 'sr

TASK: [Apply firewall policies] *****************************************
changed: [172.16.0.1]
```

```
PLAY [Configure VPN NAT policies] ************************************

TASK: [Build address book entries] ***********************************
changed: [172.16.0.1] => (item={'prefix': '172.16.0.0/24', 'name': 'LocalNe
ok: [172.16.0.1] => (item={'prefix': '192.168.10.0/24', 'name': 'PrivateNet
ok: [172.16.0.1] => (item={'prefix': '10.10.0.0/22', 'name': 'PublicNet'})

TASK: [Apply address book entries] ***********************************
ok: [172.16.0.1]

TASK: [Build NAT policies] *******************************************
changed: [172.16.0.1] => (item={'rules': [{'interface': True, 'dst_ips': ['

TASK: [Apply NAT policies] *******************************************
changed: [172.16.0.1]

PLAY RECAP ***********************************************************
172.16.0.1                 : ok=32   changed=29   unreachable=0    failed=0

vagrant@NetDevOps-Student:/vagrant/ansible$
```

Your device should now have the complete configuration on it.

The Lab in Review

Today in the lab we covered many different steps and technologies. Let us take a brief review of what we learned and what we can take away from it.

Topics Covered

- Vagrant
  - Multi-node topologies
- VirtualBox
  - Using VirtualBox with Vagrant
  - Using a VirtualBox GUI
- Ubuntu 14.04.2
  - NetDevOps Development Environment
- vSRX
  - Firewall Policies
  - NAT Policies
  - AppFirewall Policies
  - IDP Policies
  - OSPF
  - IPSec VPNs
  - License Management
  - AppSecure Package Installation
- Python
  - PyEZ
    - Junos RPCs via Python
    - Tables and Views
  - Jinja Templating
    - Creating Junos Configs from Jinja Templates
  - Language Elements
    - Classes
    - Functions and Methods
    - Variables
    - Loops
  - Python Best Practices
- Ansible

    - Ansible Templating
    - Multi-include Playbooks
    - Using External Scripts from Ansible
    - Using Variables
    - Junos Ansible Modules
- SSH
    - SSH Subsystems
- NETCONF
    - XML
    - XML-RPC
    - XPATH
    - NETCONF Best Practices
- Markdown
    - All of the documentation is written in Github Flavored Markdown
- Git Basics

Take Aways

- A vSRX environment
    - vSRX that connects to the local network
    - A simple way to demo the capabilities of the vSRX with customers
- NetDevOps Development Environment
    - Develop and test automation tools
- A lab that you can share and use with customers
    - All materials shared here are publicly available
    - You can take a customer through many of the steps without the proctor headend
    - Easy to setup proctor headend
- A Git repository that you can take with you to the lab

Virtual Machine Passwords

Each of the virtual machine instances has been preconfigured with authentication for the VM. This is part of the standard vagrant build requirements.

Here is a better description of the Vagrant requirements

Since vagrant is used as a development and learning tool the need for a secure password is typically bypassed. However it is possible to change this during the creation of a virtual machine if this is required for your environment. However most open Vagrant VMs keep with this same model to keep things simple.

NetDevOps

To access the NetDevOps VM you can use the vagrant ssh command.

```
vagrant ssh ndo
```

In the event that you need a password to accomplish something on the VM you can use the default username and password.

- vagrant/vagrant

To do things as root you can use the "sudo" command. This command will elevate your privileges for running a specific command. No password is required to use "sudo" for the vagrant user.

Junos

Because Junos is a bit different than a typical Linux install we have chosen to use the root user to manage the device. This is done via a specific plugin that was created to map specific vagrant commands to Junos functions.

To access the vSRX host you can use the vagrant ssh command.

```
vagrant ssh srx
```

In the event that you need to run commands against the device here are the default passwords. This will be required in portions of the lab.

- root/Juniper
- vagrant/vagrant