Ansible to Manage Windows Servers – Step by Step

argonsys.com/microsoft-cloud/articles/configuring-ansible-manage-windows-servers-step-step/

By Robert Keith

Introduction

Ansible is quickly becoming the dominant DevOps platform for automating software provisioning, configuration management and application deployment in a heterogeneous datacenter and hybrid cloud environment. Ansible has facilities to integrate and manage various technologies including Microsoft Windows, systems with REST API support and of course Linux.

This article will step through the steps of deploying the Ansible controlling node on CentOS 7, and the configuration of Windows Server 2016 for management and create Ansible playbook examples with custom Powershell Ansible modules.

Windows and Ansible integration is documented in the official Ansible documentation.

By following the instructions in this article, you will be able to manage Windows systems using Ansible as easily as managing any other environment, including Linux.

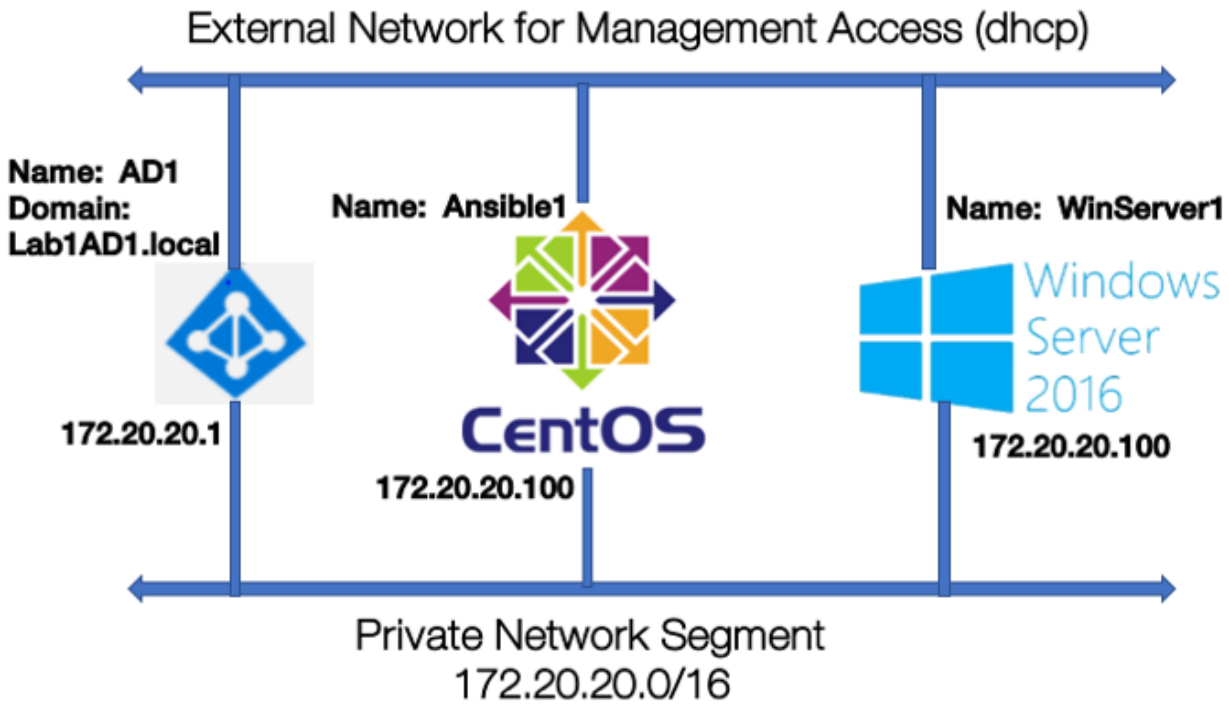
Example Lab

The examples in this article will reference this minimal configuration:

- One Ansible Controlling Node running Linux CentOS 7
- One Windows Server 2016 server to be managed
- Once Windows Server 2016 Active Directory server providing DNS services

The Windows systems are not required to be domain joined. In this example, the Windows system is a standalone WORKGROUP machine.
This lab is built on three VMs running on Hyper-V on a Windows 10 desktop. Here is an article describing a similar a similar scenario, How to Build Windows Storage Spaces Direct on a Virtual Lab.

Below is a diagram of the example lab environment.



All communications between systems runs on a private network segment for simplicity.

Install Lab Servers

Build the following servers in your environment.

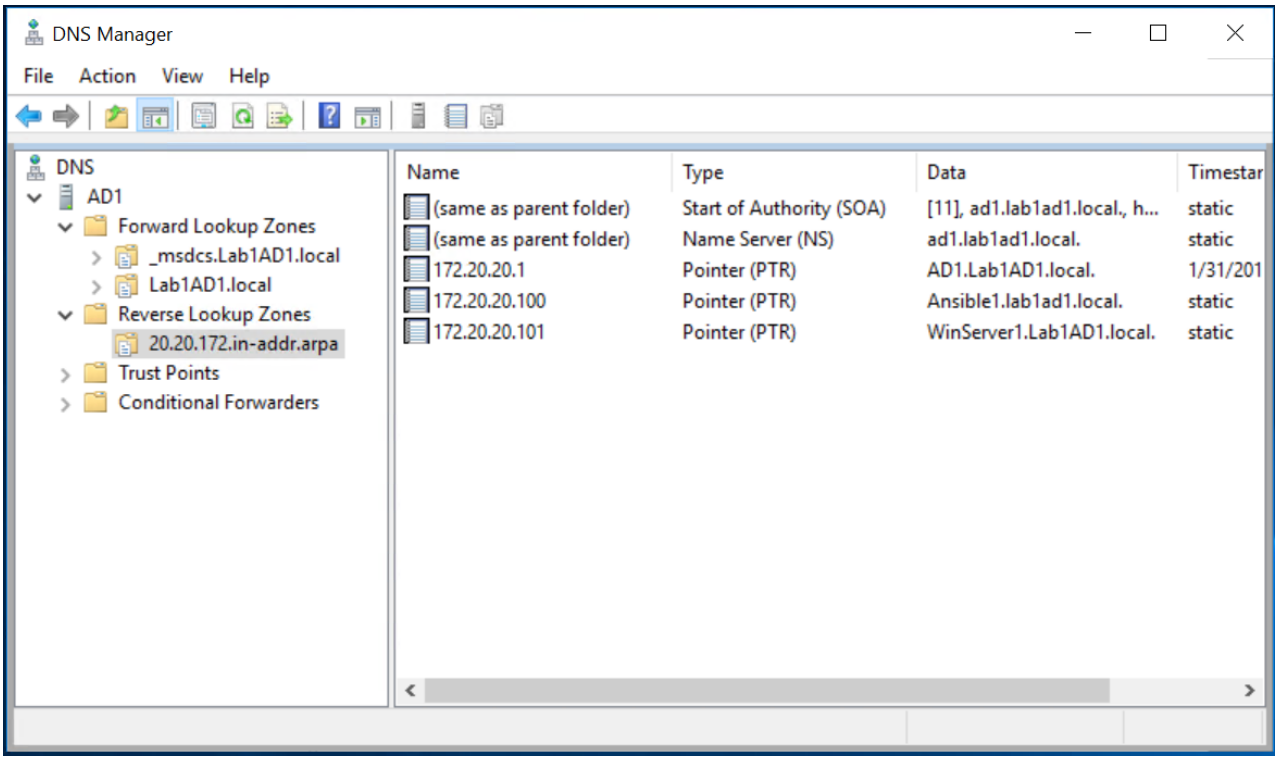
Name	Notes
Ansible1	This will be a CentOS version 7 system with the minimal software selection. Make sure you can access this system via SSH from a client such as Putty.
WinServer1	This is a Windows Server 2016. • After installing Windows Server 2016, apply all the latest Microsoft Updates • Rename the server to WinServer1 (or whatever you like)
AD1	This is a Windows Server 2016 with the Active Directory Domain Services role configured. This system is not required for domain services for the examples below. This node only provides DNS services for this environment. • Apply all the latest Microsoft Updates • Rename the server to AD1 (or whatever) • Install the Active Directory Domain Services role • Configure the Active Directory • Domain: Lab1AD1.local

Configure Lab Network

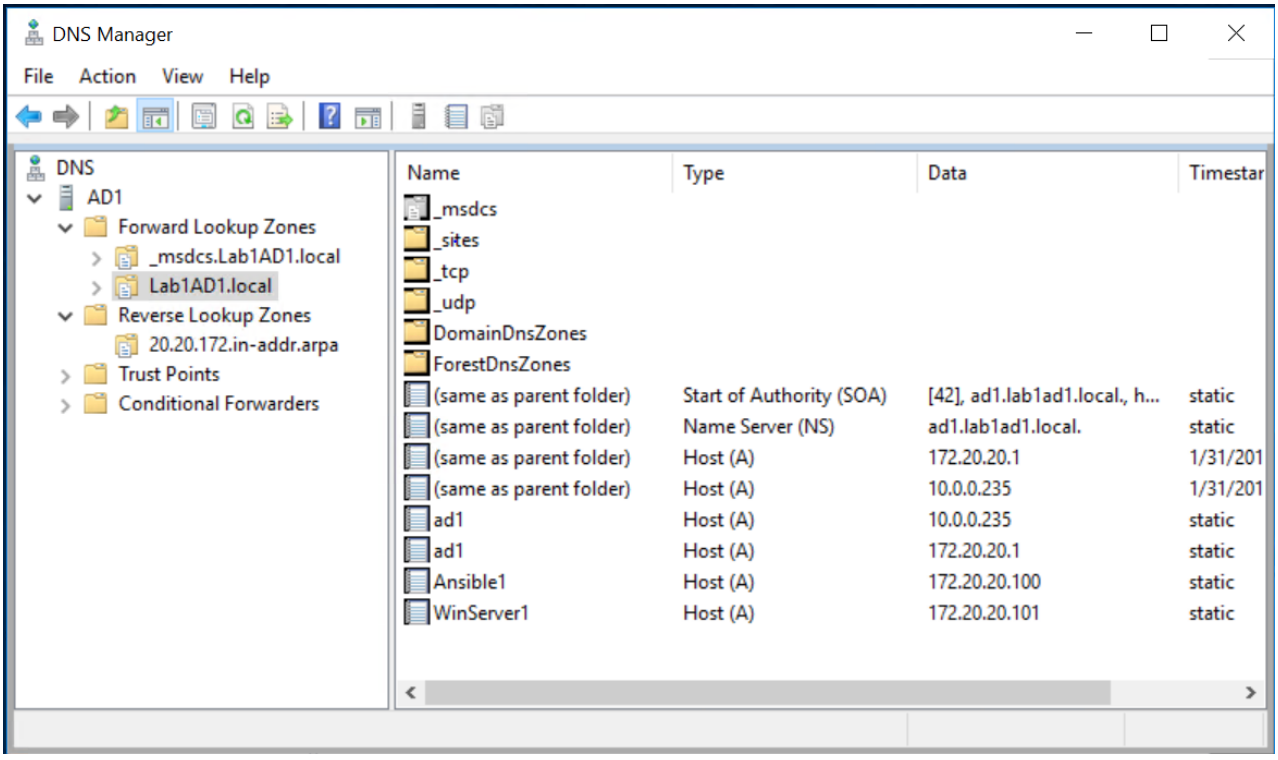
Configure AD1 DNS services

Configure DNS Forward and Reverse Lookups, Kerberos requires both forward and reverse DNS lookup to resolve correctly.

Configure the DNS Reverse Lookup Zone. This will look like the screenshot below.



The DNS Forward Lookup Zone will be like the screenshot below



Add **Ansible1** and **WinServer1** A records:

- Ansible1: 172.20.20.100 – Select to create the PTR record automatically
- WinServer1: 172.20.20.101 – Select to create the PTR record automatically

Review the Reverse Look Zone 20.20.172.in-addr.arpa. You may need to select **Right-Click** and **REFRESH** to see the new records.

Configure Ansible1 CentOS System Network

Update the following files. Be sure to modify as appropriate for your own environment.

The eth0 interface in this example uses DHCP addresses to access the Internet and provide SSH access for administration.

1. Edit: /etc/sysconfig/network-scripts/ifcfg-eth0

PowerShell

```
TYPE="Ethernet"
PROXY_METHOD="none"
BROWSER_ONLY="no"
BOOTPROTO="dhcp"
DEFROUTE="yes"
PEERDNS="no"
IPV4_FAILURE_FATAL="no"
NAME="eth0"
DEVICE="eth0"
ONBOOT="yes"
```

PEERDNS is set to "no" to prevent DHCP from inserting external DNS records. The only DNS records should come from AD1.
The eth1 interface is the private network and is shared only with the other servers in this lab.

2. Edit /etc/sysconfig/network-scripts/ifcfg-eth1

PowerShell

```
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=yes
IPADDR=172.20.20.100
NETMAST=255.255.0.0
DNS1=172.20.20.1
IPV4_FAILURE_FATAL=no
NAME=eth1
DEVICE=eth1
ONBOOT=yes
```

The resolv.conf configures the Linux DNS client.

3. Edit /etc/resolv/conf

PowerShell

```
search Lab1AD1.local
nameserver 172.20.20.1
```

If Network Manager is running (the default configuration), it will overwrite this configuration. You can disable the Network Manager with the command `systemctl disable NetworkManager`

Set the hostname

4. Edit: /etc/hostname

PowerShell

```
Ansible1
```

5. Reboot the server

Type: `reboot`

6. Test your network. Make sure you can resolve DNS correctly.

Type: `nslookup WinServer1`

This command will query the AD1 DNS server for **WinServer1** without a fully qualified name.

```
[root@Ansible1 ~]# nslookup WinServer1
Server:         172.20.20.1
Address:        172.20.20.1#53

Name:   WinServer1.lab1ad1.local
Address: 172.20.20.101
```

Type: `nslookup 172.20.20.101`

This command will do a reverse lookup on the IP address. This should return WinServer1 as the name.

```
[root@Ansible1 ~]# nslookup 172.20.20.101
Server:         172.20.20.1
Address:        172.20.20.1#53

101.20.20.172.in-addr.arpa      name = WinServer1.Lab1AD1.local.
```

Configure Ansible Environment

Other versions of Linux will work equally well. The configuration commands will have to be adjusted for each version of Linux.

Install Prerequisite Packages

Use Yum to install the following packages.

1. Install GCC required for Kerberos

PowerShell

```
yum -y group install “Development Tools”
```

2. Install EPEL

PowerShell

```
yum -y install epel-release
```

3. Install Ansible

PowerShell

```
yum -y install ansible
```

4. Install Kerberos

PowerShell

```
yum -y install python-devel krb5-devel krb5-libs krb5-workstation
```

5. Install Python PIP

PowerShell

```
yum -y install python-pip
```

6. Install BIND utilities for nslookup

PowerShell

```
yum -y install bind-utils
```

7. Bring all packages up to the latest version

PowerShell

```
yum -y update
```

Check Ansible and Python is Installed

1. Run the commands:

PowerShell

```
ansible - - version | head -l 1
python - - version
```

```
[root@Ansible1 ~]# ansible --version | head -n 1
ansible 2.4.2.0
[root@Ansible1 ~]# python --version
Python 2.7.5
```

The versions of Ansible and Python here are 2.4.2 and 2.7.5. Ansible is developing extremely rapidly so these instructions will likely change in the near future.

Configure Kerberos

There are other options than Kerberos, but Kerberos is generally the best option, though not the simplest.

1. Install the Kerberos wrapper:

PowerShell

```
pip install pywinrm[Kerberos]
```

Kerberos packages were installed previously which will have created /etc/krb5.conf

2. Edit /etc/krb5.conf

Add:

PowerShell

```
[realms]
LAB1AD1.LOCAL = {
  kdc = AD1.LAB1AD1.LOCAL
}
```

Add:

PowerShell

```
[domain_realm]
.lab1ad1.local = LAB1AD1.LOCAL
lab1ad1.local = LAB1AD1.LOCAL
```

The /etc/krb5.conf file when complete will be similar to:

```
# Configuration snippets may be placed in this directory as well
includedir /etc/krb5.conf.d/

[logging]
 default = FILE:/var/log/krb5libs.log
 kdc = FILE:/var/log/krb5kdc.log
 admin_server = FILE:/var/log/kadmind.log

[libdefaults]
 dns_lookup_realm = false
 ticket_lifetime = 24h
 renew_lifetime = 7d
 forwardable = true
 rdns = false
 default_ccache_name = KEYRING:persistent:%{uid}

[realms]
LAB1AD1.LOCAL = {
  kdc = AD1.LAB1AD1.LOCAL

[domain_realm]
.lab1ad1.local = LAB1AD1.LOCAL
lab1ad1.local = LAB1AD1.LOCAL
```

Only the [realms] and [domain_realm] were updated manually.

Test Kerberos

1. Run the following commands to test Kerberos:

PowerShell

```
1  kinit administrator@LAB1AD1.LOCAL
```

You will be prompted for the administrator password `klist`
You should see a Kerberos KEYRING record.

```
[root@Ansible1 ~]# kinit administrator@LAB1AD1.LOCAL
Password for administrator@LAB1AD1.LOCAL:
[root@Ansible1 ~]#
[root@Ansible1 ~]# klist
Ticket cache: KEYRING:persistent:0:0
Default principal: administrator@LAB1AD1.LOCAL

Valid starting       Expires              Service principal
01/31/2018 23:59:09  02/01/2018 09:59:09  krbtgt/LAB1AD1.LOCAL@LAB1AD1.LOCAL
        renew until 02/07/2018 23:59:04
```

Configure Ansible

Ansible is complex and is sensitive to the environment. Troubleshooting an environment which has never initially worked is complex and confusing. We are going to configure Ansible with the least complex possible configuration. Once you have a working environment, you can make extensions and enhancements in small steps.

The core configuration of Ansible resides at /etc/ansible

```
[root@Ansible1 ansible]# ls /etc/ansible
ansible.cfg  group_vars  hosts  roles
```

We are only going to update two files for this exercise.

Update the Ansible Inventory file

1. Edit /etc/ansible/hosts and add:

PowerShell

```
1  [windows]
2  WinServer1.lab1ad1.local
```

```
[root@Ansible1 ansible]# cat /etc/ansible/hosts

[windows]
WinServer1.lab1ad1.local
```

"[windows]" is a created group of servers called "windows". In reality this should be named something more appropriate for a group which would have similar configurations, such as "Active Directory Servers", or "Production Floor Windows 10 PCs", etc.

Update the Ansible Group Variables for Windows

Ansible Group Variables are variable settings for a specific inventory group. In this case, we will create the group variables for the "windows" servers created in the /etc/ansible/hosts file.

1. Create /etc/ansible/group_vars/windows and add:

PowerShell

```
1  ---
2  ansible_user: Administrator
3  ansible_password: Abcd1234
4  ansible_port: 5986
5  ansible_connection: winrm
6  ansible_winrm_server_cert_validation: ignore
```

This is a YAML configuration file, so make sure the first line is three dashes "---" Naturally change the Administrator password to the password for **WinServer1**.

```
[root@Ansible1 group_vars]# cat /etc/ansible/group_vars/windows
---
ansible_user: Administrator
ansible_password: Abcd1234
ansible_port: 5986
ansible_connection: winrm
ansible_winrm_server_cert_validation: ignore
```

For best practices, Ansible can encrypt this file into the Ansible Vault. This would prevent the password from being stored here in clear text. For this lab, we are attempting to keep

the configuration as simple as possible. Naturally in production this would not be appropriate.

Configure Windows Servers to Manage

To configure the Windows Server for remote management by Ansible requires a bit of work. Luckily the Ansible team has created a PowerShell script for this. Download this script from [here] to each Windows Server to manage and run this script as Administrator.

Log into **WinServer1** as Administrator, download ConfigureRemotingForAnsible.ps1 and run this PowerShell script without any parameters.

Once this command has been run on the **WinServer1**, return to the Ansible1 Controller host.

Test Connectivity to the Windows Server

If all has gone well, we should be able to perform an Ansible PING test command. This command will simply connect to the remote **WinServer1** server and report success or failure.

1. Type: `ansible windows -m win_ping`

```
[root@Ansible1 group_vars]# ansible windows -m win_ping
WinServer1.lab1ad1.local | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

This command runs the Ansible module "win_ping" on every server in the "windows" inventory group.

2. Type: `ansible windows -m setup` to retrieve a complete configuration of Ansible environmental settings.
3. Type: `ansible windows -c ipconfig`

If this command is successful, the next steps will be to build Ansible playbooks to manage Windows Servers.

Managing Windows Servers with Playbooks

Let's create some playbooks and test Ansible for real on Windows systems.

1. Create a folder on **Ansible1** for the playbooks, YAML files, modules, scripts, etc. For these exercises we created a folder under /root called win_playbooks.

```
[root@Ansible1 win_playbooks]# ls /root/win_playbooks/
get_version_script.yml  get_version.yml  ipconfig.yml  library  scripts
```

Ansible has some expectations on the directory structure where playbooks reside. Create the library and scripts folders for use later in this exercise.

Commands:

PowerShell

```
cd /root
mkdir win_playbooks
mkdir win_playbooks/library
mkdir win_playbooks/scripts
```

2. Create the first playbook example “netstate.yml”

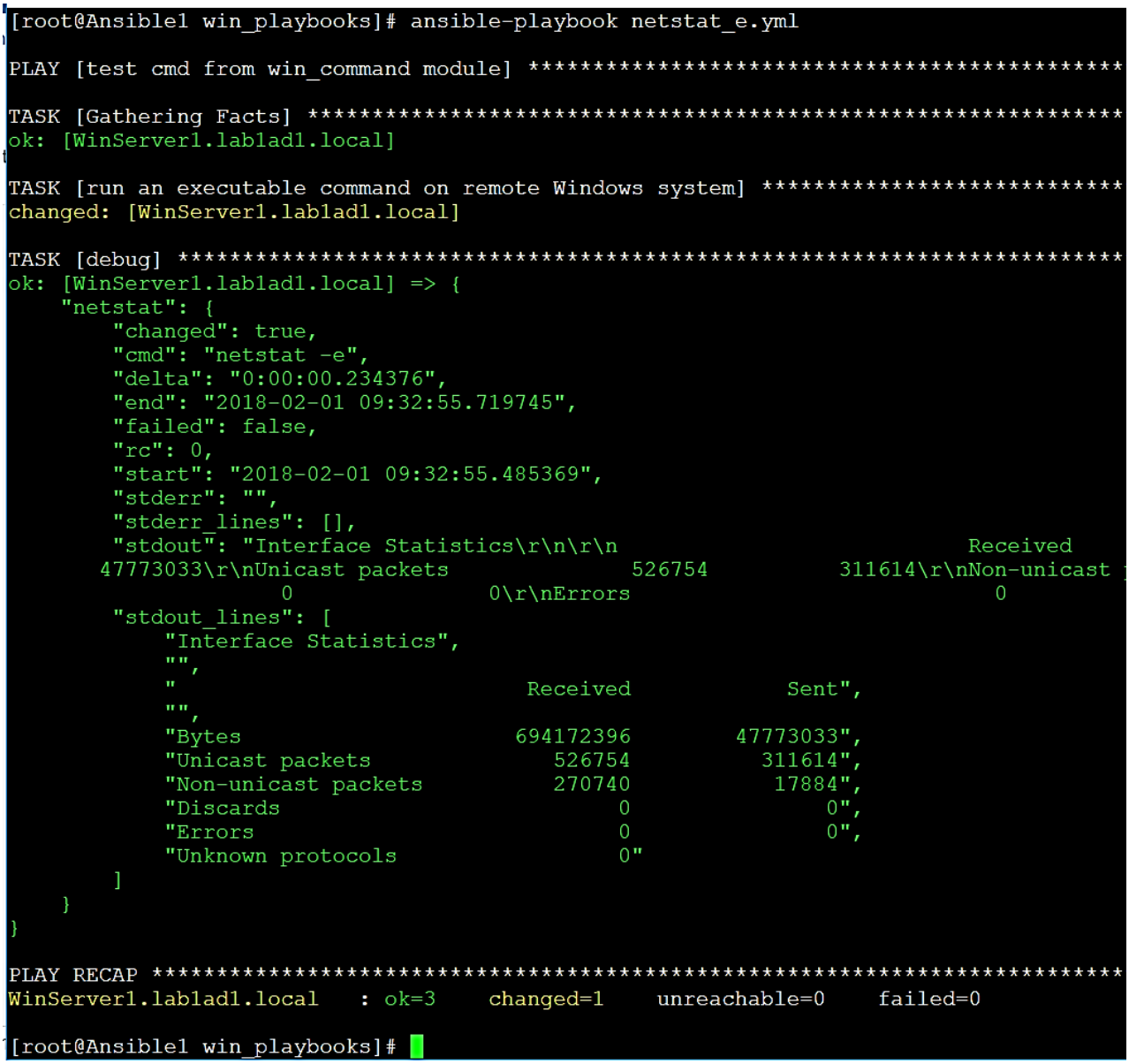
The contents are:

PowerShell

```
- name: test cmd from win_command module
  hosts: windows
  tasks:
    - name: run netstat and return Ethernet stats
      win_command: netstat -e
      register: netstat
    - debug: var=netstat
```

This playbook does only one task, to connect to the servers in the Ansible inventory group “windows” and run the command `netstat.exe -a` and return the results.

To run this playbook, run this command on **Ansible1**: `ansible-playbook netstat_e.yml`



OK, not exciting, but it did run, just not very friendly.

Also, notice that the "changed" flag is set. As of the time this article was created, all of the Windows commands return the "changed" flag as true. This is the same with running PowerShell scripts remotely. This makes much of the value of Ansible difficult as a configuration and deployment tool.

Interestingly enough, Ansible modules created with PowerShell do work correctly and return the "changed" flag correctly. PowerShell scripts and direct commands always return the "changed" flag as true. This is problematic for managing systems with Ansible. The rest of this article will focus on PowerShell modules which can perform complex management functions as well as integrate with other non-Windows systems.

Creating Ansible Modules with PowerShell

Ansible modules are plugin programs which are:

1.
    1. Loaded by Ansible when running a playbook test
    2. Ansible generates module input parameters in the JSON format
    3. Modifies the module into a generalized script and command
    4. Copies the modified script to the remote system(s)
    5. Executes the modified module on the remote system
    6. The module generates a response in JSON and this response is returned
    7. The returned JSON is parsed and values are saved or use by other tests
    8. The module returns a flag called "changed" which is important to maintain "Desired State Configurations". A return of "changed = True" will signal other tests to run to achieve "Desired State"

Modules are stored in several locations where Ansible will find them. One location is a folder named "library" located in the folder where the playbook is run. Modules stored in our library are first in the module search path and will override modules of the same name.

This example lab has a folder win_playbooks/library/

```
[root@Ansible1 ~]# ls win_playbooks/library/
get_version.ps1  get_version.py
```

PowerShell modules consist of two programs, a Python program and a PowerShell program. The Python program configures the local Ansible environment and the PowerShell program does the actual processing on the remote systems.

Create a file called "win_playbooks/library/get_version.py" and add the following contents:

get_version.py
Python

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#License: Public Domain
ANSIBLE_METADATA = {'metadata_version': '1.1',
                    'status': ['stableinterface'],
                    'supported_by': 'core'}
DOCUMENTATION = r'''
---
module: get_version
short_description: Windows get OS verison and return changed flag if below min
description:
  - Checks Windows OS version and returns the version data in JSON
  - Checks that the min version against the passed "major,minor,build" parameters
  - Returns "changed" if the remote system's OS version is below a minimum
  - The "changed" flag is t notify operations or to trigger another task
options:
  major:
    description:
      - Windows major version [MAJOR.minor.build]
    default: 5
  minor:
    description:
      - Windows minor version [major.MINOR.build]
    default: 1
  build:
    description:
      - Windows build version [major.minor.BUILD]
    default: None
author:
- Robert Keith (Argon Systems)
'''

EXAMPLES = r'''
# Example Ansible Playbook
- name: test remote Windows OS version
  hosts: windows
  tasks:
    - name: PowerShell module example to get Windows OS version
      get_version:
        major: "5"
        minor: "1"
        build: "13493"

'''

RETURN = '''
    description: Output of (Get-Host).Version powershell in JSON format
    returned: success
    type: string
'''
```

Next create a file “win_playbooks/library/get_version.ps1” and add the following contents:

get_version.ps1

PowerShell

```powershell
#!powershell
# WANT_JSON
# POWERSHELL_COMMON
$ErrorActionPreference = "Stop"
$params = Parse-Args $args -supports_check_mode $true
$MajorVer = Get-AnsibleParam -obj $params -name "major" -type "str"
$MinorVer = Get-AnsibleParam -obj $params -name "minor" -type "str"
$BuildVer = Get-AnsibleParam -obj $params -name "build" -type "str"
#Should validate input parameters here naturally
$Vers = (Get-Host).Version
$Message = $Message = "Windows OS is at desired version"
$BelowMin = $false
if ($Vers.Major -lt $MajorVer) { $BelowMin = $true }
if ($Vers.Minor -lt $MinorVer) { $BelowMin = $true }
if ($Vers.Build -lt $BuildVer) { $BelowMin = $true }
if ($BelowMin) { $Message = "Windows OS below desired version" }
$result = @{
  changed = $BelowMin
  version = $Vers
  message = $Message
  desired_major_ver = $MajorVer
  desired_minor_ver = $MinorVer
  desired_build_ver = $BuildVer
}
Exit-Json $result
```

Next, create the playbook YAML file to setup and run the new module.

From the win_playbooks folder, create a file with the following contents:

PowerShell

```powershell
- name: test simple powershell module - get OS version and test for minimum
desired version
  hosts: windows
  tasks:
    - name: simple powershell module example to get Windows OS version
      get_version:
        major: "5"
        minor: "1"
        build: "14393"
```

Execute this playbook with the command:

PowerShell

```powershell
ansible-playbook get_version.yml
```

```
[root@Ansible1 win_playbooks]# ansible-playbook get_version.yml

PLAY [test simple powershell module - get OS version and test for minimum desired version] ******

TASK [Gathering Facts] *********************************************************************
ok: [WinServer1.lab1ad1.local]

TASK [simple powershell module example to get Windows OS version] ******************************
ok: [WinServer1.lab1ad1.local]

PLAY RECAP *********************************************************************************
WinServer1.lab1ad1.local    : ok=2    changed=0    unreachable=0    failed=0

[root@Ansible1 win_playbooks]#
```

Notice the "changed" flag is set to False. In this case, this means the remote operating system matched the values in the get_version.yml playbook.

If we set the Verbose flag, we see the data returned from the PowerShell module in the JSON format.

PowerShell

```
1  ansible-playbook get_version.yml -v
```

```
[root@Ansible1 win_playbooks]# ansible-playbook get_version.yml -v
Using /etc/ansible/ansible.cfg as config file

PLAY [test simple powershell module - get OS version and test for minimum desired version] ****

TASK [Gathering Facts] *********************************************************************
ok: [WinServer1.lab1ad1.local]

TASK [simple powershell module example to get Windows OS version] ******************************
ok: [WinServer1.lab1ad1.local] => {"changed": false, "desired_build_ver": "14393", "desired_maj
or_ver": "5", "desired_minor_ver": "1", "message": "Windows OS is at desired version", "version
": {"Build": 14393, "Major": 5, "MajorRevision": 0, "Minor": 1, "MinorRevision": 206, "Revision
": 206}}

PLAY RECAP *********************************************************************************
WinServer1.lab1ad1.local    : ok=2    changed=0    unreachable=0    failed=0

[root@Ansible1 win_playbooks]#
```

Setting the Very Very Verbose flag give more information, and oddly enough formats the JSON into a more readable format.

PowerShell

```
1  ansible-playbook get_version.yml -vvv
```

```
TASK [simple powershell module example to get Windows OS version] ************
task path: /root/win_playbooks/get_version.yml:4
Using module file /root/win_playbooks/library/get_version.ps1
<WinServer1.lab1ad1.local> ESTABLISH WINRM CONNECTION FOR USER: Administrator o
inServer1.lab1ad1.local
EXEC (via pipeline wrapper)
ok: [WinServer1.lab1ad1.local] => {
    "changed": false,
    "desired_build_ver": "14393",
    "desired_major_ver": "5",
    "desired_minor_ver": "1",
    "message": "Windows OS is at desired version",
    "version": {
        "Build": 14393,
        "Major": 5,
        "MajorRevision": 0,
        "Minor": 1,
        "MinorRevision": 206,
        "Revision": 206
    }
}
```

Now if we modify the playbook expecting a different version. We set the major version from 5 to 9 (which does not exist).

PowerShell

```
1  - name: test simple powershell module - get OS version and test for minimum
2  desired version
3    hosts: windows
4    tasks:
5      - name: simple powershell module example to get Windows OS version
6        get_version:
7          major: "9"
8          minor: "1"
           build: "14393"
```

And run the playbook again

PowerShell

```
1  ansible-playbook get_version.yml
```

```
[root@Ansible1 win_playbooks]# ansible-playbook get_version.yml

PLAY [test simple powershell module - get OS version and test for minimum desired version] ***

TASK [Gathering Facts] *********************************************************************
ok: [WinServer1.lab1ad1.local]

TASK [simple powershell module example to get Windows OS version] **************************
changed: [WinServer1.lab1ad1.local]

PLAY RECAP *********************************************************************************
WinServer1.lab1ad1.local   : ok=2    changed=1    unreachable=0    failed=0

[root@Ansible1 win_playbooks]#
```

The “changed” flag is set.

We could extend this playbook with other PowerShell modules, for example:

PowerShell

```
- name: test simple powershell module - get OS version and test for minimum
desired version
  hosts: windows
  tasks:
    - name: simple powershell module example to get Windows OS version
      get_version:
        major: "9"
        minor: "1"
        build: "14393"
      register: get_version_output
    - debug:
        var: get_version_output.message
    - enable_win_updates: Enable unattended Windows Updates
      when get_version_output.changed
```

This playbook could check the Windows version, and if it is below a version threshold, the next task could then enable updates, manually install updates, or practically any possible action imaginable to maintain the Windows servers to a "Desired State".

Summary

Ansible is a powerful management and DevOps framework. It is complex to setup initially, but relatively simple to operate. Playbooks can be created which manage large and complex environments consisting of cloud systems, servers, storage systems, network systems, security systems, hardware and facilities management devices and any of a large set of disparate systems.

Configured correctly, managing and monitoring complex can be consolidated onto a single framework, and with the Ansible Windows support and some initial instructions detailed in this article, can include Windows Servers, Windows desktops and other Windows based systems.

Related Articles

- #Azure Storage Spaces direct #S2D Standard Storage vs Premium Storage
  I see this often in the Forums Should I use Standard Storage or should I use Premium storage. Well it Depends Premium cost more that Standard but even…
- Installation of #AzureDevOps Server 2019 RC1 for your Team Work #DevOps #Winserv
  What is Azure DevOps Server? Collaborative software development tools for the entire team Previously known as Team Foundation Server (TFS), Azure DevOps Server is a set of collaborative…
- Step-by-Step: Deploy Windows Server 2016 Storage Spaces Direct (S2D) Cluster in Microsoft Azure
  In this short video, we'll walk through deploying a Windows Server 2016 Storage Spaces Direct (S2D) Cluster to a Microsoft Azure subscription using Azure Resource Manager (ARM) Templates.…

- Time series analysis in Azure Data Explorer
  Azure Data Explorer (ADX) is a lightning fast service optimized for data exploration. It supplies users with instant visibility into very large raw datasets in near real-time to…

- Deploy #Azure WebApp with Visual Studio Code and Play with #Kudu and App Service Editor and #VSC
  When you have installed Microsoft Visual Studio Code which is Free and Open Source with Git integration, Debugging and lot of Extensions available, You activate the Microsoft Azure…