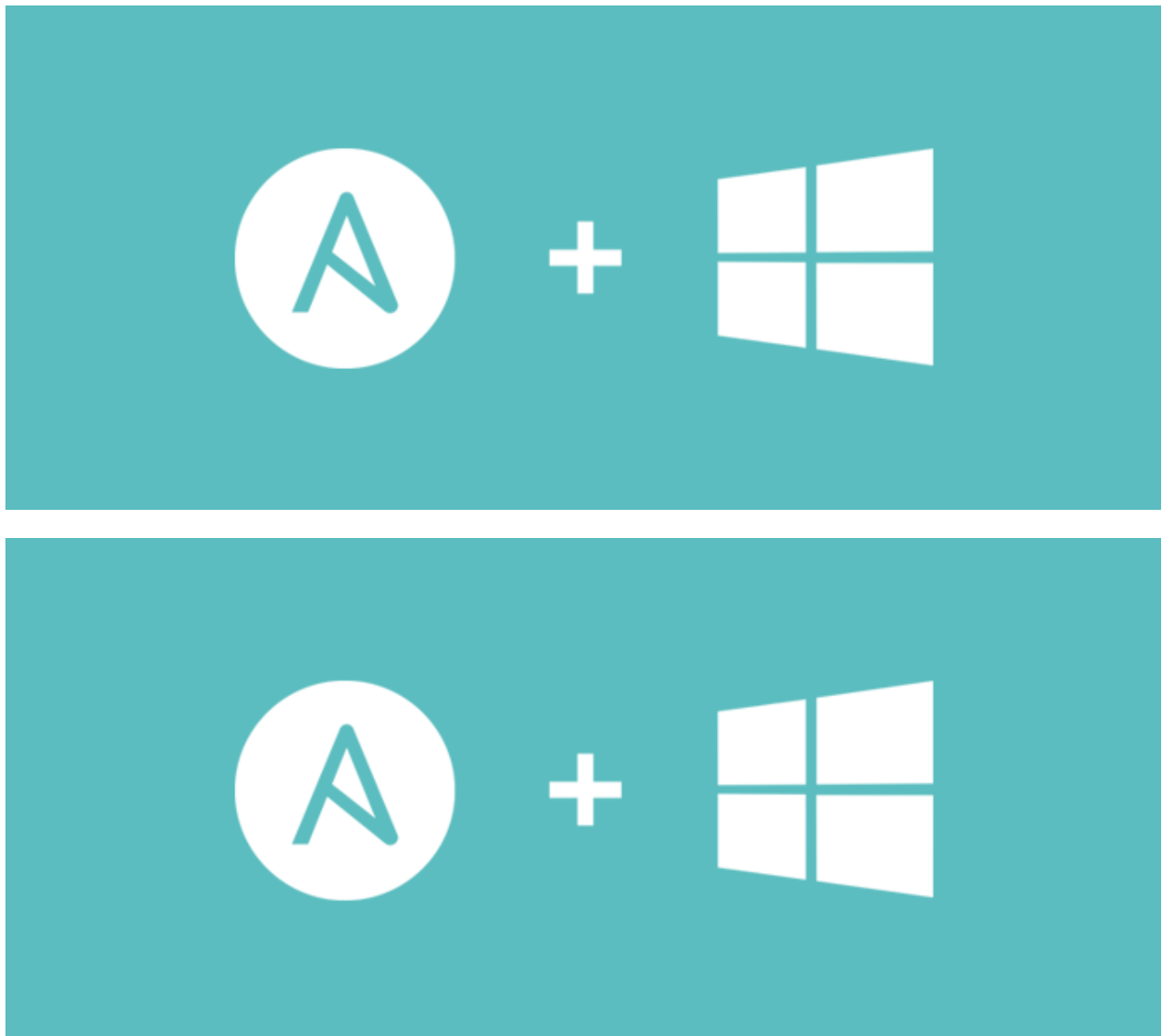


Managing Windows Servers with Ansible

 bloggingforlogging.com/2017/10/12/managing-windows-servers-with-ansible

October 12, 2017



Since version 1.7, Ansible has been able to manage Windows hosts like it can with normal unix OS'. Granted, the meaning of "support" at that time was fairly basic with a lot of the killer features like check mode, become privilege escalation, and others were not available for Windows hosts but it was a start. Seven releases later and at version 2.4, a lot of these features are now available and the gap between Unix/Linux OS' is closing. Now for too many people, this post won't really be anything that may interest them, but to myself and others who have to deal with some of the Microsoft-ism's that are prevalent in our domains this can really make our lives easier. In this post I will be talking about how to use Ansible and get it to manage Windows Servers. I'll give some basic info for people new to Ansible about how to set it up and get yourself up and running.

What is Ansible?

What is Ansible some of you may ask? Well according to ansible.com

Ansible delivers simple IT automation that ends repetitive tasks and frees up DevOps teams for more strategic work.

Now this is a very vague statement that can mean multiple things based on whoever is looking at it. To myself, Ansible is an easy way to manage multiple servers and have it conform to a state that you define without having to write any shell scripts.

Compared to other configuration management tools like Chef, Puppet, SaltStack, Ansible can achieve the same thing but in my mind the key areas where it shines are;

- No agent's are required on the client hosts, Ansible uses in built technologies like SSH, and for Windows WinRM, to communicate
- Ansible uses yaml to organise actions to be run, unlike other tools there is no pre-requisites to know any programming languages like Ruby or Python to create an Ansible playbook
- It works with a wide variety of platforms, Ansible can run manage pretty much anything with an SSH server which can cover network devices, Unix OS' like AIX and Solaris and the standard Linux distros like RHEL, Ubuntu. I've even seen it been used to manage an IBM mainframe running Z/OS
- It is open-source project with a large and passionate community behind it, this allows you to get down and dirty into the internals to see how it works as well as having multiple people worldwide contributing to Ansible and adding more and more modules daily

Now there are many blog posts, talks, videos about Ansible that will give it more justice than what I can do so I'll stop while I'm ahead

Installing Ansible

There are multiple ways in which Ansible can be installed, the easiest ones from my perspective are from source and through pip. There are other methods that can be used to install Ansible but I won't cover them here.

Creating a Virtualenv

While not necessary, it is good practice to create a virtualenv to install Ansible in. A virtualenv is;

A virtual environment that is an isolated copy of Python which allows you to work on a specific project without affecting other products.

It makes it simple to isolate each project and install their required dependencies without stepping on the toes of another project which may have different dependencies. To create a new venv with the default system Python run the following;

```
virtualenv ansible-venv
source ansible-venv/bin/activate
```

The first command will create a new venv in the current directory called **ansible-venv** while the second command will activate the new venv in the current bash session. An activated venv means any Python command like **python** or **pip** are run under that venvs' copy. To exit a venv, just run **deactivate** and the bash prompt will return to the default system's Python.

Installing from Pip

Hopefully you created a new virtualenv from the above steps but if not, these steps are still relevant. To install the latest release of Ansible, run the following;

```
(ansible-venv) jborean:~$ pip install ansible
```

You can verify what version is installed by running **pip list** or **ansible --version**. Here is the output on a brand new venv

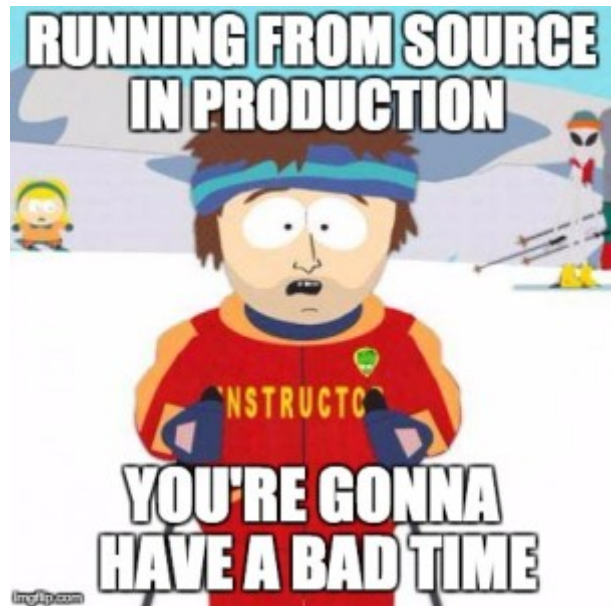
```
(ansible-venv) jborean:~$ pip list
ansible (2.4.0.0)
asn1crypto (0.23.0)
bcrypt (3.1.3)
cffi (1.11.2)
cryptography (2.0.3)
enum34 (1.1.6)
idna (2.6)
ipaddress (1.0.18)
Jinja2 (2.9.6)
MarkupSafe (1.0)
paramiko (2.3.1)
pip (9.0.1)
pyasn1 (0.3.7)
pyparser (2.18)
PyNaCl (1.1.2)
PyYAML (3.12)
setuptools (36.5.0)
six (1.11.0)
wheel (0.30.0)
```

```
(ansible-venv) jborean:~$ ansible --version
ansible 2.4.0.0
config file = None
configured module search path = [u'/Users/jborean/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
ansible python module location = /Users/jborean/ansible-venv/lib/python2.7/site-
packages/ansible
executable location = /Users/jborean/ansible-venv/bin/ansible
python version = 2.7.10 (default, Jul 15 2017, 17:16:57) [GCC 4.2.1 Compatible
Apple LLVM 9.0.0 (clang-900.0.31)]
```

Pip does allow you to specify a specific version of a package or packages with the **==** option, e.g. **pip install ansible==2.3.2.0** will install Ansible 2.3.2.0.

Installing from Source

Installing from source is good if you want the latest and greatest but comes with the danger of having an untested change. There are numerous commits that happen every day so running this at separate times will produce a different install of Ansible each time. The dangers of this is that something that works at one point may have been broken with an untested commit, these commits happen from time to time and are ultimately resolved in a few days. This method is very useful if you are developing playbooks, modules, or just trying out new and upcoming features.



To install from source, the Python pre-requisites must be installed, this can easily be done by running `pip install ansible`. Once they are installed, run the following to clone a copy of the Ansible source code and replace the pip installed version with the latest devel version.

```
(ansible-venv) jborean:~$ pip uninstall ansible -y
Uninstalling ansible-2.4.0.0:
Successfully uninstalled ansible-2.4.0.0
```

```
(ansible-venv) jborean:~$ git clone https://github.com/ansible/ansible.git
Cloning into 'ansible'...
remote: Counting objects: 263625, done.
remote: Compressing objects: 100% (76/76), done.
remote: Total 263625 (delta 46), reused 44 (delta 17), pack-reused 263523
Receiving objects: 100% (263625/263625), 85.24 MiB | 1.66 MiB/s, done.
Resolving deltas: 100% (170389/170389), done.
```

```
(ansible-venv) jborean:~$ cd ansible
(ansible-venv) jborean:~/ansible$ source hacking/env-setup -q
```

When running `ansible --version` now, you can see it is running on the devel branch version which at the time of writing this is 2.5.

```
(ansible-venv) jborean:~/ansible$ ansible --version
ansible 2.5.0 (devel 4e22677e7d) last updated 2017/10/10 10:38:29 (GMT +1000)
config file = None
configured module search path = [u'/Users/jborean/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
ansible python module location = /Users/jborean/ansible/lib/ansible
executable location = /Users/jborean/ansible/bin/ansible
python version = 2.7.10 (default, Jul 15 2017, 17:16:57) [GCC 4.2.1 Compatible
Apple LLVM 9.0.0 (clang-900.0.31)]
```

Installing Windows Requirements

Now that Ansible is installed, there are a few more packages that must be installed which are not packaged with Ansible. Ansible usually communicates over SSH to managed hosts but because Windows does not natively support SSH right now, the WinRM protocol is used instead. The Python library `pywinrm` is used by Ansible to talk over WinRM and to install it, run the following:

```
(ansible-venv) jborean:~/ansible$ pip install pywinrm
Collecting pywinrm
Using cached pywinrm-0.2.2-py2.py3-none-any.whl
Requirement already satisfied: six in /Users/jborean/ansible-venv/lib/python2.7/site-packages (from pywinrm)
Collecting requests-ntlm==0.3.0 (from pywinrm)
Using cached requests_ntlm-1.0.0-py2.py3-none-any.whl
Collecting requests==2.9.1 (from pywinrm)
Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting xmltodict (from pywinrm)
Using cached xmltodict-0.11.0-py2.py3-none-any.whl
Collecting ntlm-auth==1.0.2 (from requests-ntlm==0.3.0->pywinrm)
Using cached ntlm_auth-1.0.5-py2.py3-none-any.whl
Collecting certifi==2017.4.17 (from requests==2.9.1->pywinrm)
Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests==2.9.1->pywinrm)
Using cached chardet-3.0.4-py2.py3-none-any.whl
Requirement already satisfied: idna<2.7,>=2.5 in /Users/jborean/ansible-venv/lib/python2.7/site-packages (from requests==2.9.1->pywinrm)
Collecting urllib3<1.23,>=1.21.1 (from requests==2.9.1->pywinrm)
Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting ordereddict (from ntlm-auth==1.0.2->requests-ntlm==0.3.0->pywinrm)
Installing collected packages: certifi, chardet, urllib3, requests, ordereddict, ntlm-auth, requests-ntlm, xmltodict, pywinrm
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 ntlm-auth-1.0.5 ordereddict-1.1 pywinrm-0.2.2 requests-2.18.4 requests-ntlm-1.0.0 urllib3-1.22 xmltodict-0.11.0
```

This will install `pywinrm` with support for authentication over `basic`, `certificate`, and `ntlm` but it can also run over `kerberos` and `credssp` with a few extra steps. Because `kerberos` and `credssp` require extra dependencies on host they are not included in the base package.

To add Kerberos auth to `pywinrm`, run the following:

```
# for Debian/Ubuntu/etc:
sudo apt-get install gcc python-dev libkrb5-dev

# for RHEL/CentOS/etc:
sudo yum install gcc python-devel krb5-devel krb5-workstation

pip install pywinrm[kerberos]
```

To add CredSSP auth to `pywinrm`, run the following:

```
# for Debian/Ubuntu/etc:
sudo apt-get install gcc python-dev libssl-dev

# for RHEL/CentOS/etc:
sudo yum install gcc python-devel openssl-devel

pip install pywinrm[credssp]
```

From a testing perspective, `ntlm` is the best to use as it works with both local and domain accounts but it is an older protocol and does not support credential delegation. When moving to use your playbooks in production, I would definitely recommend using either `kerberos` or `credssp` if in a domain environment.

Setting up a Test Windows Host

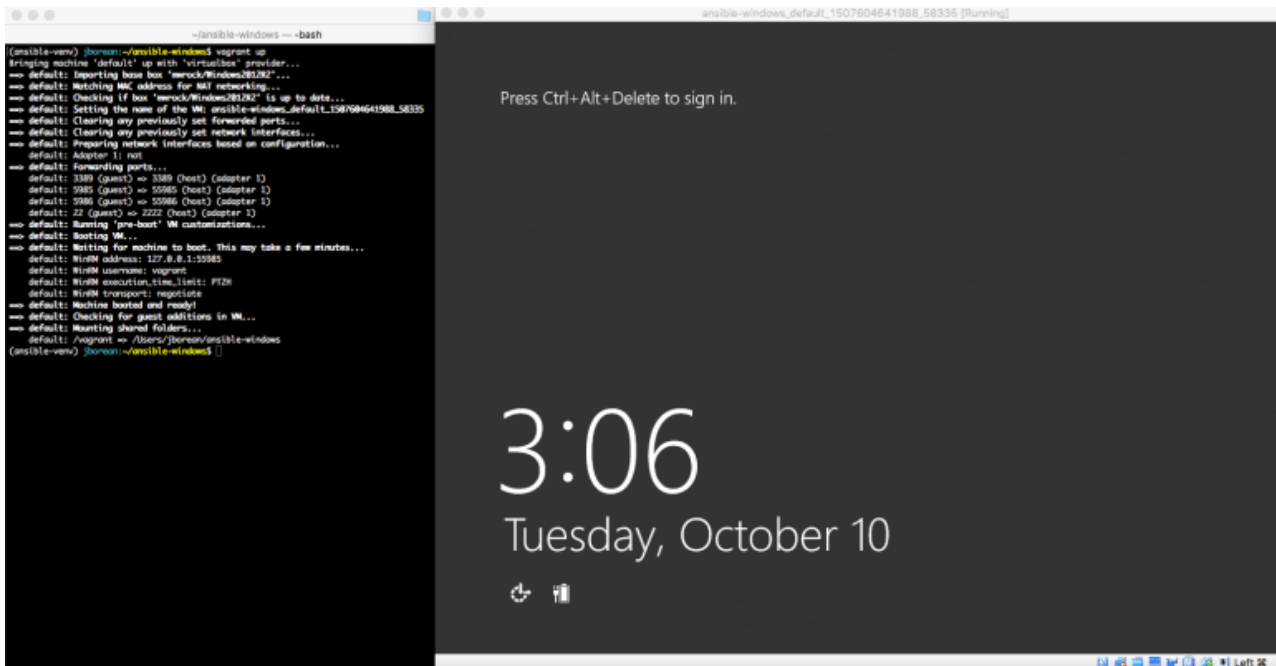
You probably already have a host ready to go for testing, but what's good about a blog post that cannot show you how to do this. In this post I am going to use vagrant to download a pre-baked image online and spin that up. To start off, create a file called `Vagrantfile` with the following contents:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure(2) do |config|
  config.vm.box = "mwrock/Windows2012R2"
  config.vm.guest = :windows
  config.vm.communicator = "winrm"
  config.vm.boot_timeout = 300
  config.vm.network :forwarded_port, guest: 3389, host: 3389, id: 'rdp',
auto_correct: true

  config.vm.provider "virtualbox" do |vb|
    vb.gui = true
    vb.memory = 2048
  end
end
```

In the same directory, run the command `vagrant up` and it will automatically download the image and start up a new VM in Virtualbox. Windows images are quite large so be careful running these commands on a connection with limited data. These images have been created by Matt Wrock and are designed to be as lightweight as possible but unfortunately lightweight and Windows does not fit in the same sentence. If you want to know more about this process and how these images were generated, Matt's article on this is fantastic and can be found [here](#). This image is stored in a local cache so this step will only need to run the download process once, subsequent runs should be a lot quicker.



In a few minutes you should be presented with this glorious site

The username and password required to log onto the new box is just **vagrant**. To finish the prep required by Ansible, log onto the box and open a new PowerShell window (as an Administrator) and run:

```
New-Item -Path C:\Temp -ItemType Directory
$config_script = "C:\Temp\ConfigureRemotingForAnsible.ps1"
(New-Object -TypeName
System.Net.WebClient).DownloadFile("https://raw.githubusercontent.com/ansible/ansible/devel/examples/scripts/ConfigureRemotingForAnsible.ps1", $config_script)
powershell.exe -ExecutionPolicy ByPass -File $config_script -EnableCredSSP
```

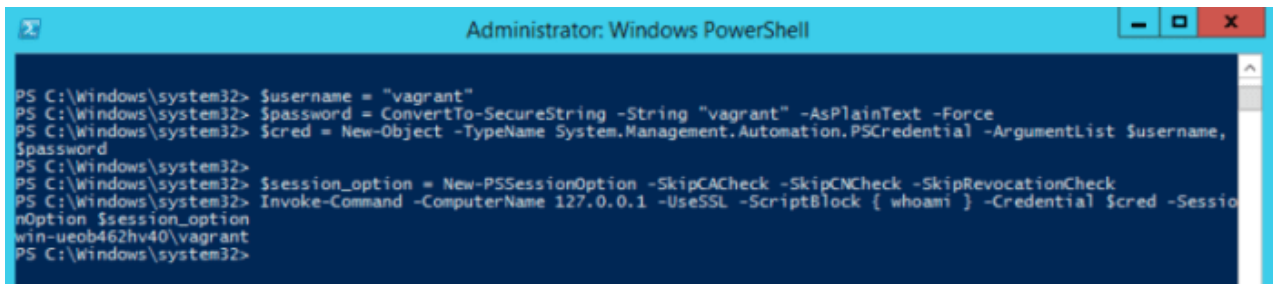
These commands will download and run the script

ConfigureRemotingForAnsible.ps1 which sets up a HTTPS WinRM listener and enable various authentication protocols. The script itself is quite useful for quickly setting up a dev box but I would try and avoid using it in a production environment. This is because it enables certain protocols useful for development and testing but it enables all of them which just become more attack vectors to exploit. I am hoping to expand on this topic in the future but as WinRM is a complex service, it is a bit too much to delve into for this blog post at this time.

To test that everything is worked fine, run the following command to verify WinRM is up and running over a HTTPS listener:

```
$username = "vagrant"
$password = ConvertTo-SecureString -String "vagrant" -AsPlainText -Force
$cred = New-Object -TypeName System.Management.Automation.PSCredential -
ArgumentList $username, $password

$session_option = New-PSSessionOption -SkipCACheck -SkipCNCheck -
SkipRevocationCheck
Invoke-Command -ComputerName 127.0.0.1 -UseSSL -ScriptBlock { whoami } -Credential
$cred -SessionOption $session_option
```

```
Administrator: Windows PowerShell
PS C:\Windows\system32> $username = "vagrant"
PS C:\Windows\system32> $password = ConvertTo-SecureString -String "vagrant" -AsPlainText -Force
PS C:\Windows\system32> $cred = New-Object -TypeName System.Management.Automation.PSCredential -ArgumentList $username,
$password
PS C:\Windows\system32> $session_option = New-PSSessionOption -SkipCACheck -SkipCNCheck -SkipRevocationCheck
PS C:\Windows\system32> Invoke-Command -ComputerName 127.0.0.1 -UseSSL -ScriptBlock { whoami } -Credential $cred -SessionOption $session_option
win-ueob462hv40\vagrant
PS C:\Windows\system32>
```

A successful test of WinRM over HTTPS

Configuring the Ansible Inventory

Now that we have Ansible installed and a fully functioning Windows host up and running, we need to create an inventory file so Ansible knows how to connect to the Windows host. Create a new file called `inventory.yml` with the following contents:

```
windows:
  hosts:
    vagrant-windows2012R2:
      ansible_host: 127.0.0.1
  vars:
    ansible_user: vagrant
    ansible_password: vagrant
    ansible_connection: winrm
    ansible_port: 55986 # this port is usually 5986 but Vagrant forwards this as
55986
    ansible_winrm_transport: ntlm
    ansible_winrm_server_cert_validation: ignore
```

What this inventory tell's Ansible is that there is a group called `windows` and it contains 1 host called `vagrant-windows2012R2`. It then set's the actual hostname it connects with to `127.0.0.1` and the various args to let Ansible know how to connect over WinRM. In a normal scenario, the variable `ansible_host` does not need to be defined and Ansible will connect to a host using the host name specified but in our case we want to still connect over the localhost loopback but display a more human friendly name in the output.

Now that Ansible knows what host to connect to, it needs to know how to connect to it. By default it will use SSH but this won't work for Windows so we need to specify `ansible_connection: winrm`. There are a lot more options available to configure WinRM but I find the ones set above are what most people need or will ultimately use. Here is a brief explanation of what each variable does:

- `ansible_user`: simply the username that will be used in the connection, this can be a local or domain user and it should be a member of the local Administrators group
- `ansible_password`: do I need to explain this one
- `ansible_connection`: tells Ansible to use the WinRM connection plugin, this must be set for Ansible to connect otherwise it will try and fail to use SSH

- `ansible_port`: will default to 5985 or 5986 depending on what `ansible_winrm_scheme` is set to and if the scheme isn't set will default to 5986. In our case we need to set a custom port as Vagrant forwards 5986 to 55986 so that is what we will connect to
- `ansible_winrm_transport`: a weirdly named argument that specifies what auth protocol to use, I like setting this explicitly so I know for sure what it is using.
- `ansible_winrm_server_cert_validation`: when using self signed certificates, this must be set to `ignore` otherwise it will fail.

The variable `ansible_winrm_transport` is pretty important as it determines how your credentials are sent and validated on the server. It also affects on what type of account that can be used in the connection. Here is a basic overview on the options that are available

- `basic`: the credentials are base64 encoded and sent to the server, this is quite insecure, especially when not running over HTTPS, and should be avoided where possible
- `certificate`: similar to SSH keys but not quite the same, certificate auth uses X509 certificates that are mapped to a local user but due to the complexity of the setup required it is not very popular
- `ntlm`: the all-rounder of auth options, an older Microsoft protocol that works in the majority of situations but is less secure than others
- `kerberos`: the best option to use when dealing with domain accounts, supports credential delegation and is one of the most secure options to choose from
- `credssp`: useful when credential delegation is required but Kerberos is not available

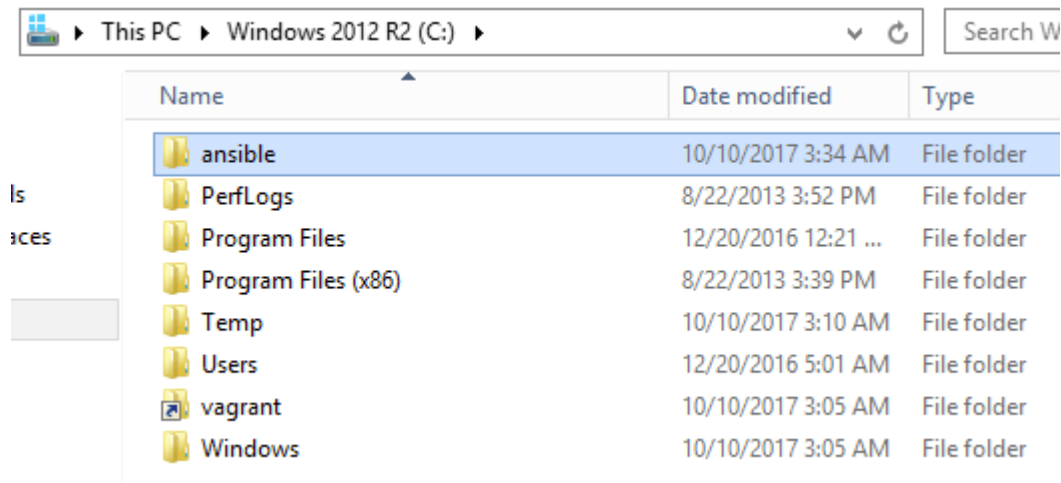
Once the inventory has been set up, you can test your setup by running

```
(ansible-venv) jborean:~/ansible-windows$ ansible -i inventory.yml windows -m win_ping
vagrant-windows2012R2 | SUCCESS => {
  "changed": false,
  "failed": false,
  "ping": "pong"
}
```

If you don't get a success message, then something has gone wrong so double check your setup and verify you followed each step. The `win_ping` module shows that Ansible can communicate with the new host over WinRM but say we wanted to create a new directory we can use the `win_file` module instead. The `-m` argument specifies the module to use while `-a` allows you to specify the arguments to the module like `path`, `state` and so on. To create a new folder called `ansible` in the C drive, run the following command:

```
(ansible-venv) jborean:~/ansible-windows$ ansible -i inventory.yml windows -m win_file -a 'path=C:\\ansible state=directory'
vagrant-windows2012R2 | SUCCESS => {
  "changed": true,
  "failed": false
}
```

On the Virtualbox session, you should now see the newly created folder.



The beauty of most Ansible modules is that they are idempotent, if you were to run the command again no changed will occur.

```
(ansible-venv) jboean:~/ansible-windows$ ansible -i inventory.yml windows -m
win_file -a 'path=C:\\ansible state=directory'
vagrant-windows2012R2 | SUCCESS => {
  "changed": false,
  "failed": false
}
```

While you can continue to run Ansible modules in this ad-hoc fashion, it is best to start using playbooks to group tasks together. The added benefit of using a playbook is that there is not escaping rules when dealing with backslashes and you can take advantage of host vars.

Creating a Playbook

Playbooks can contain multiple plays which are then comprised of multiple tasks or roles. Depending on who you ask and what the situation is, people will give you multiple different answers as how to create and maintain playbooks. This is a topic that I can spend endless hours writing about but right now let's create a very basic playbook with some tasks. Create a new file called `playbook.yml` with the following text:

```

- name: example play to run on a Windows server
  hosts: windows
  tasks:
    - name: create a folder to store IIS site
      win_file:
        path: C:\ansible
        state: directory

    - name: create a basic html page
      win_copy:
        content: |
          <html>
          <head><title>IIS Test</title></head>
          <body>Body</body>
          </html>
        dest: C:\ansible\index.html

    - name: install the IIS features
      win_feature:
        name: Web-Server
        state: present
        include_sub_features: yes
        include_management_tools: no

    - name: remove the default IIS website
      win_iis_website:
        name: Default Web Site
        state: absent

    - name: create new IIS website
      win_iis_website:
        name: Ansible Demo
        state: started
        port: 8080
        physical_path: C:\ansible

```

What this means is that Ansible will run 5 tasks which are:

1. Make sure the path `C:\ansible` exists as a folder
2. Create a file at `C:\ansible\index.html` and set the content of that file to the HTML text set
3. Make sure the `Web-Server` Windows feature is installed
4. Remove the website `Default Web Site` if it exists
5. Create and start a new website called `Ansible Demo` on port 8080

To run the playbook, run `ansible-playbook -i inventory.yml playbook.yml -vv`, it will go through each task and ensure the Windows host resource matches that state specified.

```

(ansible-venv) jlborean:~/ansible-windows$ ansible-playbook -i inventory.yml playbook.yml

PLAY [example play to run on a Windows server] *****

TASK [Gathering Facts] *****
ok: [vagrant-windows2012R2]

TASK [create a folder to store IIS site] *****
changed: [vagrant-windows2012R2]

TASK [create a basic html page] *****
changed: [vagrant-windows2012R2]

TASK [install the IIS features] *****
changed: [vagrant-windows2012R2]

TASK [remove the default IIS website] *****
changed: [vagrant-windows2012R2]

TASK [create new IIS website] *****
changed: [vagrant-windows2012R2]

PLAY RECAP *****
vagrant-windows2012R2      : ok=6    changed=5    unreachable=0    failed=0

```

First run of playbook

After running the playbook you can log onto the server and open up IE and see the IIS site up and running.



New IIS Site up and Running

When running the playbook command one more time, it will run a lot quicker than before and no changes should be reported:

```

[(ansible-venv) jborean:~/ansible-windows$ ansible-playbook -i inventory.yml playbook.yml

PLAY [example play to run on a Windows server] *****

TASK [Gathering Facts] *****
ok: [vagrant-windows2012R2]

TASK [create a folder to store IIS site] *****
ok: [vagrant-windows2012R2]

TASK [create a basic html page] *****
ok: [vagrant-windows2012R2]

TASK [install the IIS features] *****
ok: [vagrant-windows2012R2]

TASK [remove the default IIS website] *****
ok: [vagrant-windows2012R2]

TASK [create new IIS website] *****
ok: [vagrant-windows2012R2]

PLAY RECAP *****
vagrant-windows2012R2      : ok=6    changed=0    unreachable=0    failed=0

```

Second run through showing idempotence

Now you know the basics and how to glue everything together, start experimenting with the modules and go for gold. I will come round to talking about some of the more advanced topics when it comes to Windows and Ansible such as credential delegation, limitations in WinRM and other gotchas in a later post.

Windows Modules

As of writing this post, there are over a 1000 modules available to use with Ansible, unfortunately the majority of them are written in Python and are not compatible with Windows. A decision was made when first adding support for Windows to have modules that run on Windows to be written in PowerShell. PowerShell allows Ansible to leverage all the functions available in the .NET Framework and in more advanced cases, access to the Win32 API. This makes it a lot easier to interact with the OS on a more fundamental layer that would be a lot harder with Python.

Each Windows module is prefixed with `win_` and the majority of the fundamental Python modules have a Windows equivalent like `win_file`, `win_stat`, `win_command` and so on. A full list of Windows modules can be found [here](#).

As well as the modules on that page, there are a few action plugins that also work with Windows host. This is not an exhaustive list but here are some plugins you can use as an Ansible task on a Windows host;

- debug
- meta
- script

- raw
- fetch
- slurp
- template/win_template
- assert
- fail
- pause
- set_fact
- wait_for_connection

win_command and win_shell

The `win_command` and `win_shell` modules allow you to run an executable or a shell command on the Windows host. This is quite useful in situations where there is no module available to complete a task or whether you need to query a particular setting on a Windows host. The `win_command` module is used to run an executable while `win_shell` is used to run shell commands. A common misconception is that `win_command` runs in the same environment as the command prompt but this is incorrect, any shell specific commands like `mkdir`, `copy` will not work. To get this to work, either use `win_shell` and change the executable to `cmd` or prefix the command run with `win_command` with `cmd.exe /c` as shown below.

```
# won't work
- win_command: mkdir C:\temp

# will work
- win_command: cmd.exe /c mkdir C:\temp

# will also work with win_shell
- win_shell: mkdir C:\temp
  args:
    executable: cmd
```

The `win_shell` module is different, where it runs the string as a command in a shell and the shell defaults to PowerShell. You can still use this module to run executables but you need to follow the same rules of how to call executables in the shell being used. One benefit of `win_shell` is that you can easily run multiple commands under the same session, here is an example below of how to write a multi-lined shell task.

```
- win_shell: Write-Host single line powershell string

- win_shell: |
    $string = "multi-line powershell string"
    Write-Host $string
```

What's Next

Ansible can be used to do pretty much anything and over time, more and more modules/features are being added to cover the missing ground. Some of the hurdles of using Ansible to control Windows hosts are;

- WinRM is a lot slower than SSH and things like task execution and copying files take longer to complete
- A base requirement for Ansible is PowerShell 3.0 or newer, the base image of Windows 7, Server 2008, and Server 2008 R2 do not meet this requirement. This means certain hosts require further bootstrapping steps before it can be used with Ansible
- WinRM runs in a special process with different security restrictions, things like accessing network drives or accessing certain internal APIs like Windows Updates do not normally work without workarounds

The 1st 2 hurdles are not something that can be easily fixed in Ansible as they are fundamental limitations of WinRM but the last hurdle can be overcome. Over time, more and more features like support for credential delegation over [kerberos](#) and [credssp](#) auth as well as become support have helped to alleviate these problems. Going forward here are some of the things being looked at;

- Using SSH as a transport mechanism with the [Win32-OpenSSH](#) project
- Further improve the become process to allow things like password-less become for local accounts
- Help alleviate the bootstrapping barrier and find ways to overcome some of the restrictions we have with base OS installed

There is definitely more in the pipeline than what I have mentioned above but they are some of the key areas seen as where Ansible for Windows should focus on. Even so, if you are someone that manages Windows hosts with Ansible and feel like something is lacking or are looking for a killer feature not currently available, feel free to share your ideas. The best places to share this stuff would be;

- raising a Github issue or PR at the [Ansible](#) github repo
- post a comment on the Ansible Google Group page for [Ansible Project](#)
- talk over [#ansible](#) or [#ansible-windows](#) on the Freenode IRC network