Windows Remote Management

Unlike Linux/Unix hosts, which use SSH by default, Windows hosts are configured with WinRM. This topic covers how to configure and use WinRM with Ansible.

Topics

What is WinRM?

WinRM is a management protocol used by Windows to remotely communicate with another server. It is a SOAP-based protocol that communicates over HTTP/HTTPS, and is included in all recent Windows operating systems. Since Windows Server 2012, WinRM has been enabled by default, but in most cases extra configuration is required to use WinRM with Ansible.

Ansible uses the pywinrm package to communicate with Windows servers over WinRM. It is not installed by default with the Ansible package, but can be installed by running the following:

```
pip install "pywinrm>=0.3.0"
```

> **Note**
>
> on distributions with multiple python versions, use pip2 or pip2.x, where x matches the python minor version Ansible is running under.

> **Warning**
>
> Using the `winrm` or `psrp` connection plugins in Ansible on MacOS in the latest releases typically fail. This is a known problem that occurs deep within the Python stack and cannot be changed by Ansible. The only workaround today is to set the environment variable `no_proxy=*` and avoid using Kerberos auth.

Authentication Options

When connecting to a Windows host, there are several different options that can be used when authenticating with an account. The authentication type may be set on inventory hosts or groups with the `ansible_winrm_transport` variable.

The following matrix is a high level overview of the options:

Option	Local Accounts	Active Directory Accounts	Credential Delegation	HTTP Encryption
Basic	Yes	No	No	No
Certificate	Yes	No	No	No
Kerberos	No	Yes	Yes	Yes
NTLM	Yes	Yes	No	Yes
CredSSP	Yes	Yes	Yes	Yes

Basic

Basic authentication is one of the simplest authentication options to use, but is also the most insecure. This is because the username and password are simply base64 encoded, and if a secure channel is not in use (eg, HTTPS) then it can be decoded by anyone. Basic authentication can only be used for local accounts (not domain accounts).

The following example shows host vars configured for basic authentication:

```
ansible_user: LocalUsername
ansible_password: Password
ansible_connection: winrm
ansible_winrm_transport: basic
```

Basic authentication is not enabled by default on a Windows host but can be enabled by running the following in PowerShell:

```
Set-Item -Path WSMan:\localhost\Service\Auth\Basic -Value $true
```

Certificate

Certificate authentication uses certificates as keys similar to SSH key pairs, but the file format and key generation process is different.

The following example shows host vars configured for certificate authentication:

```
ansible connection: winrm
ansible winrm cert pem: /path/to/certificate/public/key.pem
ansible winrm cert key pem: /path/to/certificate/private/key.pem
ansible_winrm_transport: certificate
```

Certificate authentication is not enabled by default on a Windows host but can be enabled by running the following in PowerShell:

```
Set-Item -Path WSMan:\localhost\Service\Auth\Certificate -Value $true
```

> **Note**
>
> Encrypted private keys cannot be used as the urllib3 library that is used by Ansible for WinRM does not support this functionality.

Generate a Certificate

A certificate must be generated before it can be mapped to a local user. This can be done using one of the following methods:

- OpenSSL

- PowerShell, using the `New-SelfSignedCertificate` cmdlet
- Active Directory Certificate Services

Active Directory Certificate Services is beyond of scope in this documentation but may be the best option to use when running in a domain environment. For more information, see the Active Directory Certificate Services documentation.

> **Note**
>
> Using the PowerShell cmdlet `New-SelfSignedCertificate` to generate a certificate for authentication only works when being generated from a Windows 10 or Windows Server 2012 R2 host or later. OpenSSL is still required to extract the private key from the PFX certificate to a PEM file for Ansible to use.

To generate a certificate with `OpenSSL`:

```
# Set the name of the local user that will have the key mapped to
USERNAME="username"

cat > openssl.conf << EOL
distinguished_name = req_distinguished_name
[req_distinguished_name]
[v3_req_client]
extendedKeyUsage = clientAuth
subjectAltName = otherName:1.3.6.1.4.1.311.20.2.3;UTF8:$USERNAME@localhost
EOL

export OPENSSL_CONF=openssl.conf
openssl req -x509 -nodes -days 3650 -newkey rsa:2048 -out cert.pem -outform PEM -keyout cert_key.pem -subj "/CN=$USERNAME" -extensions v
rm openssl.conf
```

To generate a certificate with `New-SelfSignedCertificate`:

```
# Set the name of the local user that will have the key mapped
$username = "username"
$output_path = "C:\temp"

# Instead of generating a file, the cert will be added to the personal
# LocalComputer folder in the certificate store
$cert = New-SelfSignedCertificate -Type Custom `
    -Subject "CN=$username" `
    -TextExtension @("2.5.29.37={text}1.3.6.1.5.5.7.3.2","2.5.29.17={text}upn=$username@localhost") `
    -KeyUsage DigitalSignature,KeyEncipherment `
    -KeyAlgorithm RSA `
    -KeyLength 2048

# Export the public key
$pem_output = @()
$pem_output += "-----BEGIN CERTIFICATE-----"
$pem_output += [System.Convert]::ToBase64String($cert.RawData) -replace ".{64}", "$&`n"
$pem_output += "-----END CERTIFICATE-----"
[System.IO.File]::WriteAllLines("$output_path\cert.pem", $pem_output)

# Export the private key in a PFX file
[System.IO.File]::WriteAllBytes("$output_path\cert.pfx", $cert.Export("Pfx"))
```

> **Note**
>
> To convert the PFX file to a private key that pywinrm can use, run the following command with OpenSSL `openssl pkcs12 -in cert.pfx -nocerts -nodes -out cert_key.pem -passin pass: -passout pass:`

Import a Certificate to the Certificate Store

Once a certificate has been generated, the issuing certificate needs to be imported into the `Trusted Root Certificate Authorities` of the `LocalMachine` store, and the client certificate public key must be present in the `Trusted People` folder of the `LocalMachine` store. For this example, both the issuing certificate and public key are the same.

Following example shows how to import the issuing certificate:

```
$cert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 "cert.pem"

$store_name = [System.Security.Cryptography.X509Certificates.StoreName]::Root
$store_location = [System.Security.Cryptography.X509Certificates.StoreLocation]::LocalMachine
$store = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Store -ArgumentList $store_name, $store_location
$store.Open("MaxAllowed")
$store.Add($cert)
$store.Close()
```

> **Note**
>
> If using ADCS to generate the certificate, then the issuing certificate will already be imported and this step can be skipped.

The code to import the client certificate public key is:

```
$cert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 "cert.pem"

$store_name = [System.Security.Cryptography.X509Certificates.StoreName]::TrustedPeople
$store_location = [System.Security.Cryptography.X509Certificates.StoreLocation]::LocalMachine
$store = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Store -ArgumentList $store_name, $store_location
$store.Open("MaxAllowed")
$store.Add($cert)
$store.Close()
```

Mapping a Certificate to an Account

Once the certificate has been imported, map it to the local user account:

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

# This is the issuer thumbprint which in the case of a self generated cert
# is the public key thumbprint, additional logic may be required for other
# scenarios
$thumbprint = (Get-ChildItem -Path cert:\LocalMachine\root | Where-Object { $_.Subject -eq "CN=$username" }).Thumbprint

New-Item -Path WSMan:\localhost\ClientCertificate `
    -Subject "$username@localhost" `
    -URI * `
    -Issuer $thumbprint `
    -Credential $credential `
    -Force
```

Once this is complete, the hostvar `ansible_winrm_cert_pem` should be set to the path of the public key and the `ansible_winrm_cert_key_pem` variable should be set to the path of the private key.

NTLM

NTLM is an older authentication mechanism used by Microsoft that can support both local and domain accounts. NTLM is enabled by default on the WinRM service, so no setup is required before using it.

NTLM is the easiest authentication protocol to use and is more secure than `Basic` authentication. If running in a domain environment, `Kerberos` should be used instead of NTLM.

Kerberos has several advantages over using NTLM:

- NTLM is an older protocol and does not support newer encryption protocols.
- NTLM is slower to authenticate because it requires more round trips to the host in the authentication stage.
- Unlike Kerberos, NTLM does not allow credential delegation.

This example shows host variables configured to use NTLM authentication:

```
ansible_user: LocalUsername
ansible_password: Password
ansible_connection: winrm
ansible_winrm_transport: ntlm
```

Kerberos

Kerberos is the recommended authentication option to use when running in a domain environment. Kerberos supports features like credential delegation and message encryption over HTTP and is one of the more secure options that is available through WinRM.

Kerberos requires some additional setup work on the Ansible host before it can be used properly.

The following example shows host vars configured for Kerberos authentication:

```
ansible_user: username@MY.DOMAIN.COM
ansible_password: Password
ansible_connection: winrm
ansible_port: 5985
ansible_winrm_transport: kerberos
```

As of Ansible version 2.3, the Kerberos ticket will be created based on `ansible_user` and `ansible_password`. If running on an older version of Ansible or when `ansible_winrm_kinit_mode` is `manual`, a Kerberos ticket must already be obtained. See below for more details.

There are some extra host variables that can be set:

```
ansible_winrm_kinit_mode: managed/manual (manual means Ansible will not obtain a ticket)
ansible_winrm_kinit_cmd: the kinit binary to use to obtain a Kerberos ticket (default to kinit)
ansible_winrm_service: overrides the SPN prefix that is used, the default is ``HTTP`` and should rarely ever need changing
ansible_winrm_kerberos_delegation: allows the credentials to traverse multiple hops
ansible_winrm_kerberos_hostname_override: the hostname to be used for the kerberos exchange
```

Installing the Kerberos Library

Some system dependencies that must be installed prior to using Kerberos. The script below lists the dependencies based on the distro:

```
# Via Yum (RHEL/Centos/Fedora)
yum -y install gcc python-devel krb5-devel krb5-libs krb5-workstation

# Via Apt (Ubuntu)
sudo apt-get install python-dev libkrb5-dev krb5-user

# Via Portage (Gentoo)
emerge -av app-crypt/mit-krb5
emerge -av dev-python/setuptools

# Via Pkg (FreeBSD)
sudo pkg install security/krb5

# Via OpenCSW (Solaris)
pkgadd -d http://get.opencsw.org/now
/opt/csw/bin/pkgutil -U
/opt/csw/bin/pkgutil -y -i libkrb5_3

# Via Pacman (Arch Linux)
pacman -S krb5
```

Once the dependencies have been installed, the `python-kerberos` wrapper can be install using `pip`:

```
pip install pywinrm[kerberos]
```

> **Note**
>
> While Ansible has supported Kerberos auth through `pywinrm` for some time, optional features or more secure options may only be available in newer versions of the `pywinrm` and/or `pykerberos` libraries. It is recommended you upgrade each version to the latest available to resolve any warnings or errors. This can be done through tools like `pip` or a system package manager like `dnf`, `yum`, `apt` but the package names and versions available may differ between tools.

Configuring Host Kerberos

Once the dependencies have been installed, Kerberos needs to be configured so that it can communicate with a domain. This configuration is done through the `/etc/krb5.conf` file, which is installed with the packages in the script above.

To configure Kerberos, in the section that starts with:

```
[realms]
```

Add the full domain name and the fully qualified domain names of the primary and secondary Active Directory domain controllers. It should look something like this:

```
[realms]
    MY.DOMAIN.COM = {
        kdc = domain-controller1.my.domain.com
        kdc = domain-controller2.my.domain.com
    }
```

In the section that starts with:

```
[domain_realm]
```

Add a line like the following for each domain that Ansible needs access for:

```
[domain_realm]
    .my.domain.com = MY.DOMAIN.COM
```

You can configure other settings in this file such as the default domain. See krb5.conf for more details.

Automatic Kerberos Ticket Management

Ansible version 2.3 and later defaults to automatically managing Kerberos tickets when both `ansible_user` and `ansible_password` are specified for a host. In this process, a new ticket is created in a temporary credential cache for each host. This is done before each task executes to minimize the chance of ticket expiration. The temporary credential caches are deleted after each task completes and will not interfere with the default credential cache.

To disable automatic ticket management, set `ansible_winrm_kinit_mode=manual` via the inventory.

Automatic ticket management requires a standard `kinit` binary on the control host system path. To specify a different location or binary name, set the `ansible_winrm_kinit_cmd` hostvar to the fully qualified path to a MIT krbv5 `kinit`-compatible binary.

Manual Kerberos Ticket Management

To manually manage Kerberos tickets, the `kinit` binary is used. To obtain a new ticket the following command is used:

```
kinit username@MY.DOMAIN.COM
```

> **Note**
>
> The domain must match the configured Kerberos realm exactly, and must be in upper case.

To see what tickets (if any) have been acquired, use the following command:

```
klist
```

To destroy all the tickets that have been acquired, use the following command:

```
kdestroy
```

Troubleshooting Kerberos

Kerberos is reliant on a properly-configured environment to work. To troubleshoot Kerberos issues, ensure that:

- The hostname set for the Windows host is the FQDN and not an IP address.
- The forward and reverse DNS lookups are working properly in the domain. To test this, ping the windows host by name and then use the ip address returned with `nslookup`. The same name should be returned when using `nslookup` on the IP address.
- The Ansible host's clock is synchronized with the domain controller. Kerberos is time sensitive, and a little clock drift can cause the ticket generation process to fail.
- Ensure that the fully qualified domain name for the domain is configured in the `krb5.conf` file. To check this, run:

  ```
  kinit -C username@MY.DOMAIN.COM
  klist
  ```

  If the domain name returned by `klist` is different from the one requested, an alias is being used. The `krb5.conf` file needs to be updated so that the fully qualified domain name is used and not an alias.
- If the default kerberos tooling has been replaced or modified (some IdM solutions may do this), this may cause issues when installing or upgrading the Python Kerberos library. As of the time of this writing, this library is called `pykerberos` and is known to work with both MIT and Heimdal Kerberos libraries. To resolve `pykerberos` installation issues, ensure the system dependencies for Kerberos have been met (see: Installing the Kerberos Library), remove any custom Kerberos tooling paths from the PATH environment variable, and retry the installation of Python Kerberos library package.

CredSSP

CredSSP authentication is a newer authentication protocol that allows credential delegation. This is achieved by encrypting the username and password after authentication has succeeded and sending that to the server using the CredSSP protocol.

Because the username and password are sent to the server to be used for double hop authentication, ensure that the hosts that the Windows host communicates with are not compromised and are trusted.

CredSSP can be used for both local and domain accounts and also supports message encryption over HTTP.

To use CredSSP authentication, the host vars are configured like so:

```
ansible_user: Username
ansible_password: Password
ansible_connection: winrm
ansible_winrm_transport: credssp
```

There are some extra host variables that can be set as shown below:

```
ansible_winrm_credssp_disable_tlsv1_2: when true, will not use TLS 1.2 in the CredSSP auth process
```

CredSSP authentication is not enabled by default on a Windows host, but can be enabled by running the following in PowerShell:

```
Enable-WSManCredSSP -Role Server -Force
```

Installing CredSSP Library

The `requests-credssp` wrapper can be installed using `pip`:

```
pip install pywinrm[credssp]
```

CredSSP and TLS 1.2

By default the `requests-credssp` library is configured to authenticate over the TLS 1.2 protocol. TLS 1.2 is installed and enabled by default for Windows Server 2012 and Windows 8 and more recent releases.

There are two ways that older hosts can be used with CredSSP:

- Install and enable a hotfix to enable TLS 1.2 support (recommended for Server 2008 R2 and Windows 7).
- Set `ansible_winrm_credssp_disable_tlsv1_2=True` in the inventory to run over TLS 1.0. This is the only option when connecting to Windows Server 2008, which has no way of supporting TLS 1.2

See :ref:`winrm_tls12` for more information on how to enable TLS 1.2 on the Windows host.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]windows_winrm.rst`, line 523); *backlink***
>
> Unknown interpreted text role "ref".

Set CredSSP Certificate

CredSSP works by encrypting the credentials through the TLS protocol and uses a self-signed certificate by default. The `CertificateThumbprint` option under the WinRM service configuration can be used to specify the thumbprint of another certificate.

> **Note**
>
> This certificate configuration is independent of the WinRM listener certificate. With CredSSP, message transport still occurs over the WinRM listener, but the TLS-encrypted messages inside the channel use the service-level certificate.

To explicitly set the certificate to use for CredSSP:

```
# Note the value $certificate thumbprint will be different in each
# situation, this needs to be set based on the cert that is used.
$certificate_thumbprint = "7C8DCBD5427AFEE6560F4AF524E325915F51172C"

# Set the thumbprint value
Set-Item -Path WSMan:\localhost\Service\CertificateThumbprint -Value $certificate_thumbprint
```

Non-Administrator Accounts

WinRM is configured by default to only allow connections from accounts in the local `Administrators` group. This can be changed by running:

```
winrm configSDDL default
```

This will display an ACL editor, where new users or groups may be added. To run commands over WinRM, users and groups must have at least the `Read` and `Execute` permissions enabled.

While non-administrative accounts can be used with WinRM, most typical server administration tasks require some level of administrative access, so the utility is usually limited.

WinRM Encryption

By default WinRM will fail to work when running over an unencrypted channel. The WinRM protocol considers the channel to be encrypted if using TLS over HTTP (HTTPS) or using message level encryption. Using WinRM with TLS is the recommended option as it works with all authentication options, but requires a certificate to be created and used on the WinRM listener.

The `ConfigureRemotingForAnsible.ps1` creates a self-signed certificate and creates the listener with that certificate. If in a domain environment, ADCS can also create a certificate for the host that is issued by the domain itself.

If using HTTPS is not an option, then HTTP can be used when the authentication option is `NTLM`, `Kerberos` or `CredSSP`. These protocols will encrypt the WinRM payload with their own encryption method before sending it to the server. The message-level encryption is not used when running over HTTPS because the encryption uses the more secure TLS protocol instead. If both transport and message encryption is required, set `ansible_winrm_message_encryption=always` in the host vars.

> **Note**
>
> Message encryption over HTTP requires pywinrm>=0.3.0.

A last resort is to disable the encryption requirement on the Windows host. This should only be used for development and debugging purposes, as anything sent from Ansible can be viewed, manipulated and also the remote session can completely be taken over by anyone on the same network. To disable the encryption requirement:

```
Set-Item -Path WSMan:\localhost\Service\AllowUnencrypted -Value $true
```

> **Note**
>
> Do not disable the encryption check unless it is absolutely required. Doing so could allow sensitive information like credentials and files to be intercepted by others on the network.

Inventory Options

Ansible's Windows support relies on a few standard variables to indicate the username, password, and connection type of the remote hosts. These variables are most easily set up in the inventory, but can be set on the `host_vars/` `group_vars` level.

When setting up the inventory, the following variables are required:

```
# It is suggested that these be encrypted with ansible-vault:
# ansible-vault edit group_vars/windows.yml
ansible_connection: winrm

# May also be passed on the command-line via --user
ansible_user: Administrator

# May also be supplied at runtime with --ask-pass
ansible_password: SecretPasswordGoesHere
```

Using the variables above, Ansible will connect to the Windows host with Basic authentication through HTTPS. If `ansible_user` has a UPN value like `username@MY.DOMAIN.COM` then the authentication option will automatically attempt to use Kerberos unless `ansible_winrm_transport` has been set to something other than `kerberos`.

The following custom inventory variables are also supported for additional configuration of WinRM connections:

- `ansible_port`: The port WinRM will run over, HTTPS is `5986` which is the default while HTTP is `5985`
- `ansible_winrm_scheme`: Specify the connection scheme (`http` or `https`) to use for the WinRM connection. Ansible uses `https` by default unless `ansible_port` is `5985`
- `ansible_winrm_path`: Specify an alternate path to the WinRM endpoint, Ansible uses `/wsman` by default
- `ansible_winrm_realm`: Specify the realm to use for Kerberos authentication. If `ansible_user` contains `@`, Ansible will use the part of the username after `@` by default
- `ansible_winrm_transport`: Specify one or more authentication transport options as a comma-separated list. By default, Ansible will use `kerberos, basic` if the `kerberos` module is installed and a realm is defined, otherwise it will be `plaintext`
- `ansible_winrm_server_cert_validation`: Specify the server certificate validation mode (`ignore` or `validate`). Ansible defaults to `validate` on Python 2.7.9 and higher, which will result in certificate validation errors against the Windows self-signed certificates. Unless verifiable certificates have been configured on the WinRM listeners, this should be set to `ignore`
- `ansible_winrm_operation_timeout_sec`: Increase the default timeout for WinRM operations, Ansible uses `20` by default
- `ansible_winrm_read_timeout_sec`: Increase the WinRM read timeout, Ansible uses `30` by default. Useful if there are intermittent network issues and read timeout errors keep occurring
- `ansible_winrm_message_encryption`: Specify the message encryption operation (`auto`, `always`, `never`) to use, Ansible uses `auto` by default. `auto` means message encryption is only used when `ansible_winrm_scheme` is `http` and `ansible_winrm_transport` supports message encryption. `always` means message encryption will always be used and `never` means message encryption will never be used
- `ansible_winrm_ca_trust_path`: Used to specify a different cacert container than the one used in the `certifi` module. See the HTTPS Certificate Validation section for more details.
- `ansible_winrm_send_cbt`: When using `ntlm` or `kerberos` over HTTPS, the authentication library will try to send channel binding tokens to mitigate against man in the middle attacks. This flag controls whether these bindings will be sent or not (default: `yes`).
- `ansible_winrm_*`: Any additional keyword arguments supported by `winrm.Protocol` may be provided in place of `*`

In addition, there are also specific variables that need to be set for each authentication option. See the section on authentication above for more information.

> **Note**
>
> Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_pass`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_password`, `ansible_host`, and `ansible_port`. If using a version of Ansible prior to 2.0, the older style (`ansible_ssh_*`) should be used instead. The shorter variables are ignored, without warning, in older versions of Ansible.

> **Note**
>
> `ansible_winrm_message_encryption` is different from transport encryption done over TLS. The WinRM payload is still encrypted with TLS when run over HTTPS, even if `ansible_winrm_message_encryption=never`.

IPv6 Addresses

IPv6 addresses can be used instead of IPv4 addresses or hostnames. This option is normally set in an inventory. Ansible will attempt to parse the address using the ipaddress package and pass to pywinrm correctly.

When defining a host using an IPv6 address, just add the IPv6 address as you would an IPv4 address or hostname:

```
[windows-server]
2001:db8::1

[windows-server:vars]
ansible_user=username
ansible_password=password
ansible_connection=winrm
```

> **Note**
>
> The ipaddress library is only included by default in Python 3.x. To use IPv6 addresses in Python 2.7, make sure to

run `pip install ipaddress` which installs a backported package.

HTTPS Certificate Validation

As part of the TLS protocol, the certificate is validated to ensure the host matches the subject and the client trusts the issuer of the server certificate. When using a self-signed certificate or setting `ansible_winrm_server_cert_validation: ignore` these security mechanisms are bypassed. While self signed certificates will always need the `ignore` flag, certificates that have been issued from a certificate authority can still be validated.

One of the more common ways of setting up a HTTPS listener in a domain environment is to use Active Directory Certificate Service (AD CS). AD CS is used to generate signed certificates from a Certificate Signing Request (CSR). If the WinRM HTTPS listener is using a certificate that has been signed by another authority, like AD CS, then Ansible can be set up to trust that issuer as part of the TLS handshake.

To get Ansible to trust a Certificate Authority (CA) like AD CS, the issuer certificate of the CA can be exported as a PEM encoded certificate. This certificate can then be copied locally to the Ansible controller and used as a source of certificate validation, otherwise known as a CA chain.

The CA chain can contain a single or multiple issuer certificates and each entry is contained on a new line. To then use the custom CA chain as part of the validation process, set `ansible_winrm_ca_trust_path` to the path of the file. If this variable is not set, the default CA chain is used instead which is located in the install path of the Python package certifi.

Note

Each HTTP call is done by the Python requests library which does not use the systems built-in certificate store as a trust authority. Certificate validation will fail if the server's certificate issuer is only added to the system's truststore.

TLS 1.2 Support

As WinRM runs over the HTTP protocol, using HTTPS means that the TLS protocol is used to encrypt the WinRM messages. TLS will automatically attempt to negotiate the best protocol and cipher suite that is available to both the client and the server. If a match cannot be found then Ansible will error out with a message similar to:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]windows_winrm.rst, line 764)

Cannot analyze code. No Pygments lexer found for "ansible-output".

```
.. code-block:: ansible-output

    HTTPSConnectionPool(host='server', port=5986): Max retries exceeded with url: /wsman (Caused by SSLError(SSLError(1, '[SSL: UN
```

Commonly this is when the Windows host has not been configured to support TLS v1.2 but it could also mean the Ansible controller has an older OpenSSL version installed.

Windows 8 and Windows Server 2012 come with TLS v1.2 installed and enabled by default but older hosts, like Server 2008 R2 and Windows 7, have to be enabled manually.

Note

There is a bug with the TLS 1.2 patch for Server 2008 which will stop Ansible from connecting to the Windows host. This means that Server 2008 cannot be configured to use TLS 1.2. Server 2008 R2 and Windows 7 are not affected by this issue and can use TLS 1.2.

To verify what protocol the Windows host supports, you can run the following command on the Ansible controller:

```
openssl s_client -connect <hostname>:5986
```

The output will contain information about the TLS session and the `Protocol` line will display the version that was negotiated:

```
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1
    Cipher    : ECDHE-RSA-AES256-SHA
    Session-ID: 962A00001C95D2A601BE1CCFA7831B85A7EEE897AECDBF3D9ECD4A3BE4F6AC9B
    Session-ID-ctx:
    Master-Key: ....
    Start Time: 1552976474
    Timeout   : 7200 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---

New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: AE16000050DA9FD44D03BB8839B64449805D9E43DBD670346D3D9E05D1AEEA84
    Session-ID-ctx:
    Master-Key: ....
    Start Time: 1552976538
    Timeout   : 7200 (sec)
    Verify return code: 21 (unable to verify the first certificate)
```

If the host is returning `TLSv1` then it should be configured so that TLS v1.2 is enable. You can do this by running the following PowerShell script:

```
Function Enable-TLS12 {
    param(
        [ValidateSet("Server", "Client")]
        [String]$Component = "Server"
    )

    $protocols path = 'HKLM:\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols'
    New-Item -Path "$protocols path\TLS 1.2\$Component" -Force
    New-ItemProperty -Path "$protocols path\TLS 1.2\$Component" -Name Enabled -Value 1 -Type DWORD -Force
    New-ItemProperty -Path "$protocols_path\TLS 1.2\$Component" -Name DisabledByDefault -Value 0 -Type DWORD -Force
}

Enable-TLS12 -Component Server

# Not required but highly recommended to enable the Client side TLS 1.2 components
Enable-TLS12 -Component Client

Restart-Computer
```

The below Ansible tasks can also be used to enable TLS v1.2:

```
- name: enable TLSv1.2 support
```

```
  win_regedit:
    path: HKLM:\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols\TLS 1.2\{{ item.type }}
    name: '{{ item.property }}'
    data: '{{ item.value }}'
    type: dword
    state: present
  register: enable_tls12
  loop:
  - type: Server
    property: Enabled
    value: 1
  - type: Server
    property: DisabledByDefault
    value: 0
  - type: Client
    property: Enabled
    value: 1
  - type: Client
    property: DisabledByDefault
    value: 0

- name: reboot if TLS config was applied
  win_reboot:
  when: enable_tls12 is changed
```

There are other ways to configure the TLS protocols as well as the cipher suites that are offered by the Windows host. One tool that can give you a GUI to manage these settings is IIS Crypto from Nartac Software.

Limitations

Due to the design of the WinRM protocol , there are a few limitations when using WinRM that can cause issues when creating playbooks for Ansible. These include:

- Credentials are not delegated for most authentication types, which causes authentication errors when accessing network resources or installing certain programs.
- Many calls to the Windows Update API are blocked when running over WinRM.
- Some programs fail to install with WinRM due to no credential delegation or because they access forbidden Windows API like WUA over WinRM.
- Commands under WinRM are done under a non-interactive session, which can prevent certain commands or executables from running.
- You cannot run a process that interacts with `DPAPI`, which is used by some installers (like Microsoft SQL Server).

Some of these limitations can be mitigated by doing one of the following:

- Set `ansible_winrm_transport` to `credssp` or `kerberos` (with `ansible_winrm_kerberos_delegation=true`) to bypass the double hop issue and access network resources
- Use `become` to bypass all WinRM restrictions and run a command as it would locally. Unlike using an authentication transport like `credssp`, this will also remove the non-interactive restriction and API restrictions like WUA and DPAPI
- Use a scheduled task to run a command which can be created with the `win_scheduled_task` module. Like `become`, this bypasses all WinRM restrictions but can only run a command and not modules.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]windows_winrm.rst`, line 923)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`playbooks_intro`
       An introduction to playbooks
   :ref:`playbooks_best_practices`
       Tips and tricks for playbooks
   :ref:`List of Windows Modules <windows_modules>`
       Windows specific module list, all implemented in PowerShell
   `User Mailing List <https://groups.google.com/group/ansible-project>`_
       Have a question?  Stop by the google group!
   :ref:`communication_irc`
       How to join Ansible chat channels
```