Demystifying WinRM


bloggingforlogging.com/2018/01/24/demystifying-winrm

January 24, 2018

One of the most common problems I come across today when it comes to remotely managing Windows is dealing with WinRM and its inconsistencies. I wanted to create a blog post that will help people understand what goes on with WinRM a bit more so that they can better use this resource on Windows. This blog post will cover the following part of WinRM

- Common restrictions that occur with WinRM
- Authentication with WinRM
- Authorization with WinRM

Before I continue further, I thought it best to note some of the major components and terms that are used or related to the WinRM protocol. Sometimes these terms are used interchangeably, but in reality, they represent distinct components. These components are;

- `WinRM`: Windows Remote Management, is Microsoft's implementation of the `WS-Management` protocol
- `WS-Management`: Web Services-Management, is an open standard that is based on SOAP messages to remotely exchange messaging data
- `WinRS`: Windows Remote Shell is a function of `WinRM` and is used to create a shell remotely on a Windows host and execute commands. This is usually what most open source libraries that claim `WinRM` support work with
- `PSRP`: PowerShell Remoting Protocol is a separate protocol that runs over `WinRM`, this is the protocol that is used when executing a command with `Invoke-Command` or `Enter-PSSession` and has some differences with `WinRS`

This post will mostly deal with `WinRM` and not `PSRP` but the same fundamental concepts apply to `PSRP` as well. The main difference between `WinRM` and `PSRP` is how it authorizes the user as they are protected by different ACL objects.

Restrictions

Once WinRM is up and running, it may seem simple to run commands and install programs but you are inevitably going to come across some of the many restrictions that are placed upon a WinRM session. Some of the common restrictions people encounter are;

- Cannot access network resources like a SMB share
- Windows Data Protection API DPAPI is not accessible unless using CredSSP
- Windows Update is mostly locked down and the user will get `Access is Denied` when trying to use it

- Anything relying on an interactive logon session will fail

Edit 2018-08-28: slightly modified the working to be more accurate and correct than before

Most of the time people think processes running under WinRM fail due to UAC and they need to “elevate” the process like they would locally but this is not entirely accurate. The complexity from this topic all comes down to the access token Windows creates when a user first signs on. This token contains the following components;

- The user SID
- The groups associated with the access token
- The logon session ID
- The privileges that are enabled/disabled on the access token
- The integrity level of the token
- The token elevation type and linked token info

The access token is not limited to the components above but they aren’t related to the topic at hand so we’ll ignore them for now. Each component is used in Windows to secure objects such as files, folders and API calls. When someone talks about “elevation” they are usually referring to the concept of running a process under a token that contains the full groups and rights usually restricted by default. The concept of split or linked tokens was introduced in Windows Vista to much vitriol under the name User Access Control (UAC). Personally I think UAC was a massive change in Windows and understand the frustrations some people had but it was definitely a step in the right direction from the wild wild west which was XP.

A linked or split token is when Windows produces two different tokens associated with a logon, a limited and full token. The full token contains all the groups and privileges of the user is associated with as well as the integrity level of high. A linked token is a copy of the full token but with the higher privileged groups like the `Administrators` group and privileges removed. This means that a process created under both a standard and admin user have similar rights by default. An admin can then explicitly run a process under the linked token with all their admin rights if they need to but this isn’t done by default.

Windows doesn’t always create a linked token but gives the user the full groups and rights they are allowed to have. A token without any filtering being done would happen in the following scenarios;

- The process is created from a standard user who has no admin privileges to create a linked token for
- The process is created from the builtin Administrator account and Use Admin Approval Mode for the built-in Administrator account is disabled (this mode is disabled by default)
- The process is created from a network logon and either a domain account is used or a local account with the LocalAccountTokenFilterPolicy set to 1
- UAC is disabled

The part that is of interest to WinRM is the `LocalAccountTokenFilterPolicy` setting which tells Windows whether to create a linked/filtered token for a network authenticated process like WinRM. By default this value is set to filter network logon tokens but the WinRM setup scripts from Microsoft disable this. This effectively means running `Enable-PSRemoting` or `winrm quickconfig`, the `LocalAccountTokenFilterPolicy` registry setting will be set to 1 (no filtering occurs). What this means is that any processes created from a network logon token, like WinRM or RPC, will have the full admin rights and integrity level associated with the user. Unfortunately this cannot be selectively controlled for specific processes or services so once on anything that is able to authenticate as a network logon will have the full rights of a user.

We can see this in action by running a few simple steps, here is the output when running a non-elevated process locally;

```
C:\Users\vagrant-domain>whoami /groups

GROUP INFORMATION
-----------------

Group Name                                 Type             SID                                            Attributes
========================================== ================ ============================================== ==================================================
Everyone                                   Well-known group S-1-1-0                                        Mandatory group, Enabled by default, Enabled group
BUILTIN\Users                              Alias            S-1-5-32-545                                   Mandatory group, Enabled by default, Enabled group
BUILTIN\Performance Log Users              Alias            S-1-5-32-559                                   Mandatory group, Enabled by default, Enabled group
BUILTIN\Administrators                     Alias            S-1-5-32-544                                   Group used for deny only
NT AUTHORITY\REMOTE INTERACTIVE LOGON      Well-known group S-1-5-14                                       Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\INTERACTIVE                   Well-known group S-1-5-4                                        Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users           Well-known group S-1-5-11                                       Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization             Well-known group S-1-5-15                                       Mandatory group, Enabled by default, Enabled group
LOCAL                                      Well-known group S-1-2-0                                        Mandatory group, Enabled by default, Enabled group
DOMAIN\Domain Admins                       Group            S-1-5-21-3242954042-3778974373-1659123385-512 Group used for deny only
Authentication authority asserted identity Well-known group S-1-18-1                                       Mandatory group, Enabled by default, Enabled group
DOMAIN\Denied RODC Password Replication Group Alias         S-1-5-21-3242954042-3778974373-1659123385-572 Mandatory group, Enabled by default, Enabled group, Local Group
Mandatory Label\Medium Mandatory Level     Label            S-1-16-8192
```

You can also see it has been denied the Administrators group

When running that same local process but now elevated (Run as Administrator) I get;

```
C:\Windows\system32>whoami /groups

GROUP INFORMATION
-----------------

Group Name                                 Type             SID                                            Attributes
========================================== ================ ============================================== ===============================================================
Everyone                                   Well-known group S-1-1-0                                        Mandatory group, Enabled by default, Enabled group
BUILTIN\Users                              Alias            S-1-5-32-545                                   Mandatory group, Enabled by default, Enabled group
BUILTIN\Performance Log Users              Alias            S-1-5-32-559                                   Mandatory group, Enabled by default, Enabled group
BUILTIN\Administrators                     Alias            S-1-5-32-544                                   Mandatory group, Enabled by default, Enabled group, Group owner
NT AUTHORITY\REMOTE INTERACTIVE LOGON      Well-known group S-1-5-14                                       Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\INTERACTIVE                   Well-known group S-1-5-4                                        Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users           Well-known group S-1-5-11                                       Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization             Well-known group S-1-5-15                                       Mandatory group, Enabled by default, Enabled group
LOCAL                                      Well-known group S-1-2-0                                        Mandatory group, Enabled by default, Enabled group
DOMAIN\Domain Admins                       Group            S-1-5-21-3242954042-3778974373-1659123385-512 Mandatory group, Enabled by default, Enabled group
Authentication authority asserted identity Well-known group S-1-18-1                                       Mandatory group, Enabled by default, Enabled group
DOMAIN\Denied RODC Password Replication Group Alias         S-1-5-21-3242954042-3778974373-1659123385-572 Mandatory group, Enabled by default, Enabled group, Local Group
Mandatory Label\High Mandatory Level       Label            S-1-16-12288
```

Now I get the Administrators group

Finally when running a WinRM command with `winrs -r:http://127.0.0.1:5985/wsman -u:Administrator -p:Password whoami /groups`, here is what I get;

```
C:\Windows\system32>winrs -r:http://127.0.0.1:5985/wsman -u:DOMAIN\vagrant-domain -p:VagrantPass1 whoami /groups

GROUP INFORMATION
-----------------

Group Name                                 Type             SID                                            Attributes
========================================== ================ ============================================== ===============================================================
Everyone                                   Well-known group S-1-1-0                                        Mandatory group, Enabled by default, Enabled group
BUILTIN\Users                              Alias            S-1-5-32-545                                   Mandatory group, Enabled by default, Enabled group
BUILTIN\Administrators                     Alias            S-1-5-32-544                                   Mandatory group, Enabled by default, Enabled group, Group owner
NT AUTHORITY\NETWORK                       Well-known group S-1-5-2                                        Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users           Well-known group S-1-5-11                                       Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization             Well-known group S-1-5-15                                       Mandatory group, Enabled by default, Enabled group
DOMAIN\Domain Admins                       Group            S-1-5-21-3242954042-3778974373-1659123385-512 Mandatory group, Enabled by default, Enabled group
Authentication authority asserted identity Well-known group S-1-18-1                                       Mandatory group, Enabled by default, Enabled group
DOMAIN\Denied RODC Password Replication Group Alias         S-1-5-21-3242954042-3778974373-1659123385-572 Mandatory group, Enabled by default, Enabled group, Local Group
Mandatory Label\High Mandatory Level       Label            S-1-16-12288
```

The same label as an elevated process

You can see in the non-elevated local process, it has a label of `Medium` and the `Administrators` group is not associated with the process, e.g. it is running with the limited access token. Compare this to the elevated local process, it contains both the `High` integrity label and the `Administrators` group is associated with the access token. Finally when looking at the WinRM process, we can see that it has the same label and groups as the elevated local process, so where is the `Access is Denied` error coming from?

In reality, all these issues stem from the logon type associated with the logon session ID on the token. When the Local Security Authority `LSA` creates a new access token for the user, it can create them with a few different logon types and Microsoft uses this internally for securing some objects. So really the only way to bypass these issues under WinRM is to create a new access token that is associated with a different logon type. But before we get into that, we would need to first clarify what a logon type is. There are numerous logon types within Windows, the core/relevant types are;

- Interactive – typical logon when logging onto the local console or through RDP
- Network
- Network Cleartext
- Batch – logon uses by task scheduler is most cases
- Service – logon that used by processes run by `LocalSystem`, `NetworkService`, and `LocalService`
- New Credentials

Network and Network Cleartext Logon

A Network or Network Cleartext logon means that the logon occurred from the network and so it makes sense that WinRM processes have a network logon type. Unfortunately a network logon has a few restrictions enforced by Windows which casuses the issues I wrote about above, I've put some more details about each restriction below.

Network Access

When LSA handles a network logon, it usually receives a hash or token of the user's password and not the password itself. This becomes a problem when the process running on that logon then tries to access a network resource. This is because this request does not have the password for the account which is usually available on interactive or batch logons. Without this password, any network requests are done under an `Anonymous` user. This is a problem as Windows will not allow anonymous access on an SMB share even if it is explicitly set in the ACL of the folder. This can be overridden by editing the local security policy but it is not recommended.

In the case of a `Network Cleartext` logon (CredSSP), this isn't the case as the password was provided to LSA during the logon and so the process is able to then use that password to authenticate with network shares. Kerberos is the similar where you can set

a delegation flag when retrieving the initial ticket and pass that along to the server. This flag means that the process that is run on the network logon is able to then use that same Kerberos ticket to authenticate with another network server.

DPAPI

DPAPI is a Microsoft interface used to provide easy access to crypto functions like managing private keys or credentials for a user account. Like the network access issue, it requires that LSA has access to the password of the user for it to work. In the case of a network logon this is not available for the reasons stated above. Because a `Network Cleartext` logon has the actual password, it's processes are able to interact with DPAPI like usual and will not error out.

While using DPAPI is not a common scenario like accessing network shares, it is used in a few key scenarios like;

- Installing SQL Server requires access to DPAPI to complete the install
- Managing private keys in the certificate store

WUA

Not much to say on this unfortunately, Microsoft restricts certain calls to the Windows Update API under a `Network` or `Network Cleartext` logon. Any attempt to install or uninstall updates using the COM API or even just `wusa.exe` will fail in these scenarios. I am not sure why Microsoft enforces this restriction so I cannot really explain this further.

New Credentials

The `New Credentials` logon is a special logon compared to the others that I have mentioned. It is designed as a way to run a process locally that needs to access a network resource with a different set of credentials. The access token of the process is a clone of the token used to logon on the user but any outbound connections are authenticated as the user specified in the logon.

For example, if I want to open `dsa.msc` and run as a higher privileged domain user I can run the following

```
runas.exe /netonly /user:DOMAIN\admin dsa.msc
```

The `dsa.msc` process running locally is run by the user who ran `runas.exe` but when it makes a connection to the domain controller it will be with the credentials I specified. Even better is that the user specified does not need to have the logon rights on the host and it can be any user/password combination. Because the access token that is created is a clone of the one who ran `runas.exe`, any local actions are still under the same limitations of the caller logon, e.g. `runas.exe /netonly` from a `Network` logon will still not be able to interact with DPAPI or WUA.

Bypassing the Network Logon Restrictions

Now that you know more info into how these restrictions occur, getting past them is as simple as spawning a new process from the WinRM session under a different logon. There are a few ways that this can be done such as;

- Use Task Scheduler
- Use `runas.exe` with the `/profile` argument
- Use psexec

Personally I don't like Task Scheduler as it can be problematic when it comes to starting tasks and cleaning them up once finished. Saying that, it is still a relatively simple way to run processes under a different account and achieves similar results to using `psexec`. If you still want to use Task Scheduler, there are 3 main tools that you can use;

- `schtasks.exe` – Use this only if you aren't running in PowerShell
- PowerShell Scheduled Tasks cmdlets like `New-ScheduledTaskAction` – useful as a quick and easy way to build and execute tasks in PowerShell
- Task Scheduler Scripting COM objects – gives you finer control over Scheduled Tasks and execution

Using `runas.exe /profile` is a quick and easy way to run a process as another user account with an `Interactive` logon as it is a builtin tool that comes with Windows. The trouble will be passing in the password in the command, there is no `/pass:password` argument and the value must be sent in the stdin of the spawned process. PsExec is easier still as you specify the password as an argument as well as run the process as `NT AUTHORITY\SYSTEM`. The only trouble is that the program is not included with Windows and needs to be downloaded separately.

Unfortunately most of these options make it harder to read the output and return codes of processes as they run in a separate shell. You will have to implement some shell pipes to redirect the `stdout` and `stderr` to some local files in order to save the output.

As I am an Ansible user, I've been coming across these issues repeatedly as Ansible uses WinRM as the transport mechanism. I first decided to implement a Python library that added support for CredSSP support with Ansible and that solved the issues I had at then. Over time, I've come across more things where CredSSP was just not enough and I was not able to run certain processes through Ansible without resorting to the hacks above. Ansible's solution to this is to use their become `runas` implementation which handles all the logon processes internally. Not only does `become` allow me to run a process as a different user, but for Windows, it allows me to escape the `Network` logon hell.

For example, I can upgrade PowerShell with Chocolatey using the `win_chocolatey` module like so

```
---
- name: upgrade PowerShell to v5
  hosts: windows
  tasks:
  - name: upgrade PowerShell
    win_chocolatey:
      name: powershell
      state: latest
    become: yes
    become_user: SYSTEM
    become_method: runas
```

Without `become` this would fail due to the Chocolatey process being under a `Network` logon and the PowerShell install failing due to the logon type. In the past I would have had to fall back to using a scheduled task and a custom script to execute the command but in this process I loose out on the idempotency and simplicity of an Ansible module.

One other great feature with Ansible 2.5 is that you are able to use become to set network credentials like you would with the `New Credentials` logon type. This is extremely useful if I am running on a Windows host that is not part of a domain network but I needed to authenticate with a domain account. I would achieve this with

```
---
- name: install program from domain share
  hosts:
  tasks:
  - name: copy the executable to a local path
    win_copy:
      src: \\192.168.1.10\programs\program.msi
      dest: C:\temp\program.msi
    become: yes
    become_method: runas
    become_flags: logon_type=new_credentials logon_flags=netcredentials_only
    vars:
      ansible_become_user: DOMAIN\username
      ansible_become_pass: Password01

  - name: install program
    win_package:
      path: C:\temp\program.msi
      state: present
```

More information on Become in Ansible can be found here.

Speed of WinRM

To put it simply, the WinRM protocol was not designed for speed, the protocol itself goes through multiple encoding processes and the numerous network packets required to setup a shell and run a command slow this down considerably. While the speed isn't abysmally slow and is tolerable, one area where WinRM really does falter is the copy speed. There is no native file transfer implementation within WinRM which means that most implementations copy files across the protocol by;

- Encoding the source file in Base64 to get an ASCII output
- Execute a command on the Windows host to create a blank file locally
- Send the Base64 string along the stdin of the WinRM pipe in small chunks
- The process will read the stdin, decode the Base64 string and append the bytes to the file

This is slow as the packet size for WinRM is a lot smaller than other implementations like SFTP so it requires more round trips. The other issue is that Base64 encoding takes time to complete and each stdin packer is then serialized in an XML packet which takes up more processing time.

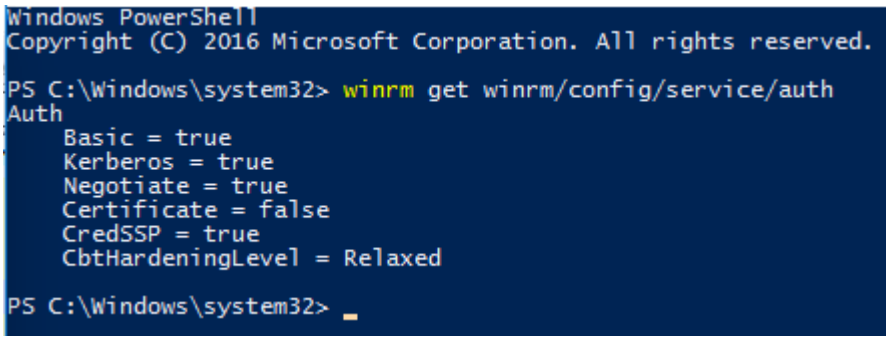
Ultimately there is not much that can be done to fix this with WinRM as it is a limitation of the protocol itself. Moving towards using SSH and SFTP would improve this scenario a lot more but currently Microsoft's SSH implementation is still in beta.

Authentication

Authentication in WinRM is a major part of the protocol and can have a big impact on how the process will run once it starts. You can use the following protocols to authenticate the user with WinRM;

- Basic
- Certificate
- Negotiate (Covers both NTLM and Kerberos but in this article will refer to NTLM)
- Kerberos
- CredSSP

If you want to find out what options are currently enabled or disabled for a WinRM service on a Windows host, run `winrm get winrm/config/service/auth` on the host.



On this server Basic, Kerberos, Negotiate and CredSSP auth has been enabled

By default both `Negotiate` and `Kerberos` are enabled for an active WinRM service and between the 2, can be used for both local and domain accounts. Here is a brief map of the options and some of the differences between them;

Auth	Local Accounts	Domain Accounts	Credential Delegation	Message Encryption	Logon Type

Auth	Local Accounts	Domain Accounts	Credential Delegation	Message Encryption	Logon Type
Basic	Yes	No	No	No	Network
Certificate	Yes	No	No	No	Network
Negotiate	Yes	Yes	No	Yes	Network
Kerberos	No	Yes	Yes (set explicitly)	Yes	Network
CredSSP	Yes	Yes	Yes (always)	Yes	Network Cleartext

Here are what each of the columns mean;

- `Local Accounts`: Whether the auth type can be used to authenticate as a local account
- `Domain Accounts`: Whether the auth type can be used to authenticate as a domain account
- `Credential Delegation`: Whether the credentials used in the authentication can be delegated to another server, like an SMB share
- `Message Encryption`: When not running on a HTTPS endpoint, whether the auth type can encrypt the payload using a provider specific protocol
- `Logon Type`: The type of logon that was used in the logon process, see the `Restrictions` section for more info

As WinRM is run over the HTTP protocol, the authentication process is done through HTTP headers, in specific the `WWW-Authenticate` and `Authorization` headers. Depending on the auth type chosen, this can either occur in just one message or as a result of a series of challenge/response messages sent by the client to the server.

Basic Auth

Basic authentication is pretty much what the name describes, it is a very simplistic method used to encode a credential over a HTTP request. It encodes the username and password for the account with Base64 encoding and sets that to the HTTP headers of the request. For example;

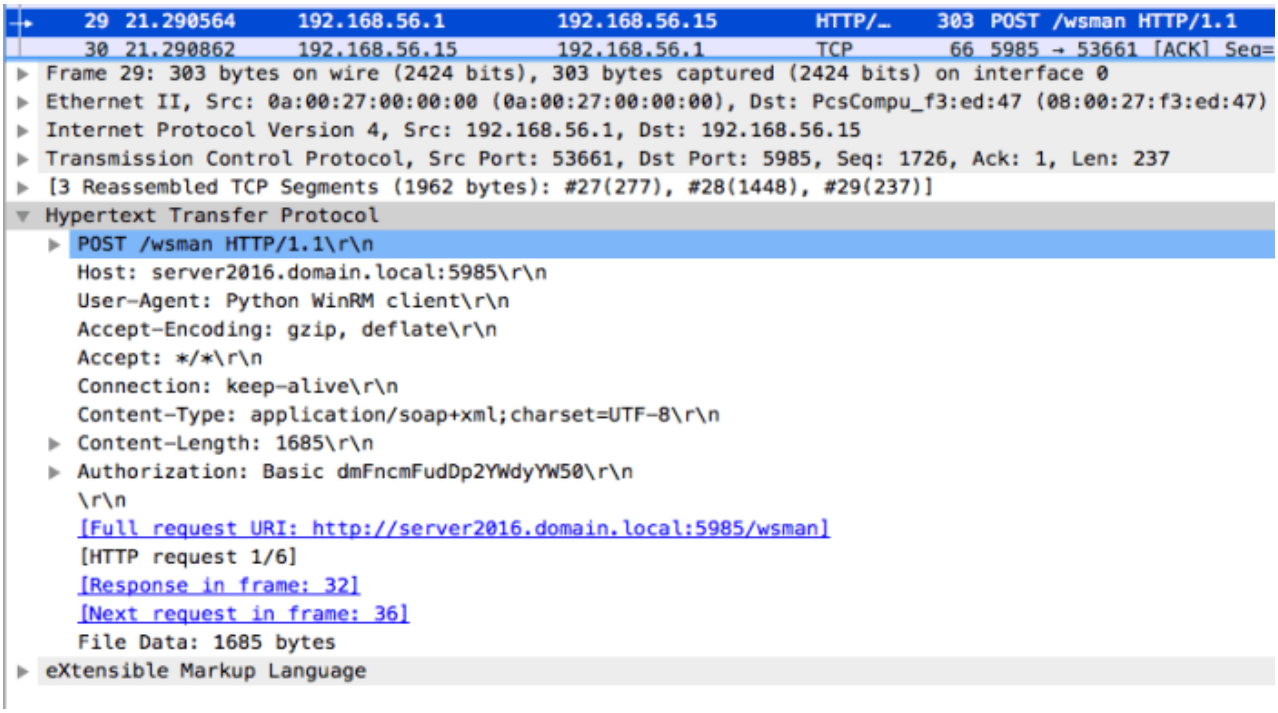
`username:password`

would become

`dXNlcm5hbWU6cGFzc3dvcmQ=`

This seems simple enough, but unfortunately its simplicity is its downfall. Anybody would be able to decode these credentials and be able to see the plaintext username and password. This can be mitigated by sending the requests over HTTPS as the headers are

encrypted using the cipher negotiated in the TLS connection. Using pywinrm and Wireshark, here is an example request that is sent to the WinRM service



I'll let you see how easy it is to find my password from this capture

Ultimately, the only advantage I see with Basic auth is that it only requires a single HTTP request and so would be one of the fastest options when network latency is an issue. In my opinion, the security concerns and lack of extra functionality like message encryption largely outweighs this positive. My recommendation is to avoid using Basic auth wherever possible and if it is needed, always run it with a HTTPS endpoint!

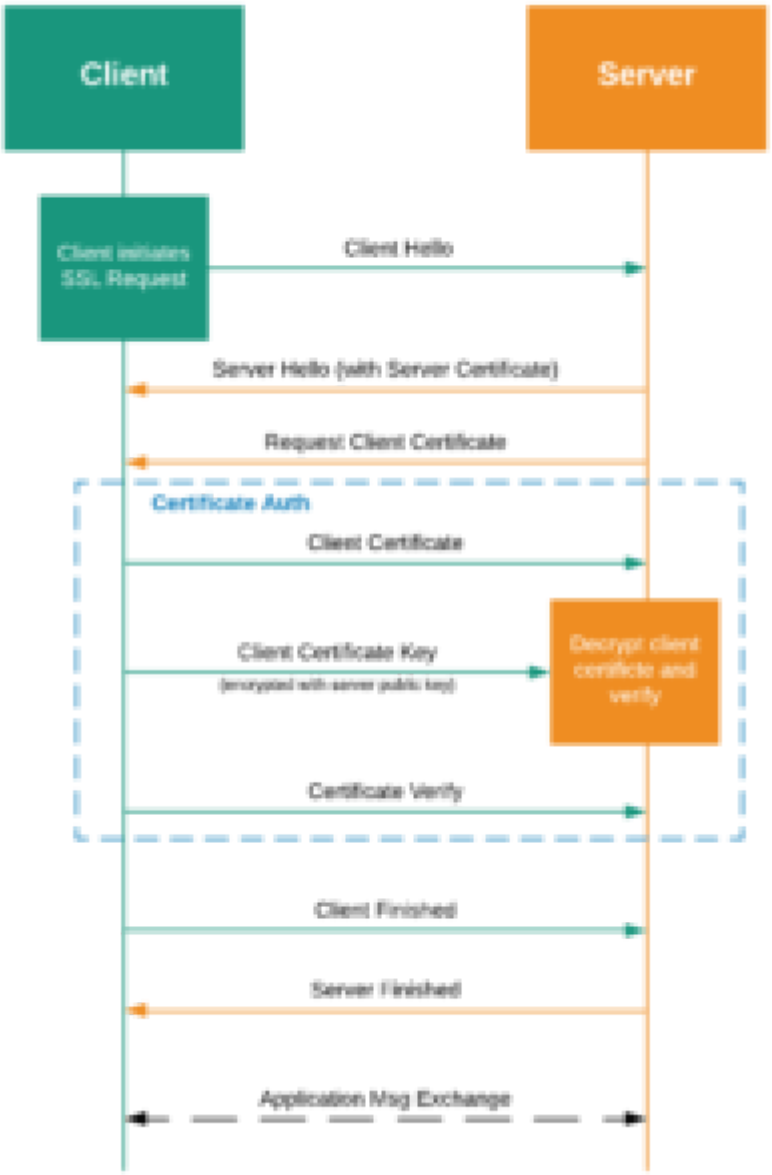
More details can be found in the RFC 7617 standard for Basic auth.

Certificate Auth

Let me just quickly sum this up in one sentence, Certificate auth is really fun to use…. not!

The name may give you false hopes that you can use SSH keys and just create an `authorized_keys` file with a list of authorized keys for a user but unfortunately this is not the case. Certificate auth for WinRM is the use of TLS with Client Authentication which uses X509 certificates as part of the TLS handshake process to authenticate a user.

Here is a basic flow of what the TLS process looks like with client authentication



Now the hard part is actually generating and mapping these certificates to a local user account. As my usual test bed is focused around Ansible and the pywinrm library, I have not tested the examples with the PowerShell native client and have heard that some of these steps may be incompatible. Unfortunately the lack of documentation makes this very hard to get right and I may revisit this at some point in the future.

This is the high level process that you would need to follow to successfully set up certificate authentication;

1. Enable Certificate auth for the WinRM service: `Set-Item -Path WSMan:\localhost\Service\Auth\Certificate -Value $true`
2. Generate key and certificate using OpenSSL
3. Copy the certificate key to the Windows host
4. Import the certificate to the `Trusted People` and `Trusted Root Certificate Authorities` store
5. Map the certificate to the local account

To generate a certificate with OpenSSL, run the following;

```
# set this variable to the value of the username the cert will map 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 v3_req_client
unset OPENSSL_CONF
rm openssl.conf
```

Once copied to the Windows server, run the following to import and map the certificate;

```
$ErrorActionPreference = "Stop"

# set the username and password for the local account to map here
$username = "username"
$password = "password"
$password = ConvertTo-SecureString -String $password -AsPlainText -Force
$credential = New-Object -TypeName System.Management.Automation.PSCredential -
ArgumentList $username, $password

$cert = New-Object -TypeName
System.Security.Cryptography.X509Certificates.X509Certificate2
$cert.Import("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()

$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()

$thumbprint = $cert.Thumbprint

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

If everything goes to plan, you can now authenticate with certificate auth, both the certificate and private key must be accessible by the process running the WinRM command. For Ansible or pywinrm, this is easy as OpenSSL has already exported the key and cert as separate files.

I have heard on the grape vine that you can set up certificates using AD CS (Active Directory Certificate Services) and GPO to issue those certs to servers and map them to user accounts. This would make the implementation on servers quite simple and painless once AD CS and GPO part has been set up. Unfortunately there is little documentation around this that I can find and I have yet to try it. I may create another blog post one day to go through this process in greater detail but for now I can’t give you any examples.

Negotiate Auth

Negotiate auth is not a specific auth protocol but rather a Microsoft provider that is used to "negotiate" a protocol to use based on the input. Currently it can be used to select either `NTLM` or `Kerberos` in the authentication process depending on the environment and server requirements. This is usually all transparent to the end user when using Microsoft tools but some third party tools, like Ansible or pywinrm, it is explicitly split between `NTLM` and `Kerberos`. For the sake of this section I'll talk about `NTLM` authentication as that is what people sometimes refer to when mentioning Negotiate.

NTLM is an older security protocol that has evolved over time from the old LAN Manager days and Microsoft recommends that Kerberos is used instead in modern environments. Unlike `Basic` auth, the password is a hash and not just an encoded form. The strength of this hash is dependent on the version of NTLM that is being used, currently these are the main 3 permutations;

- NTLMv1
- NTLMv1 with Extended Session Security (otherwise referred to as NTLM2)
- NTLMv2

If you are still using NTLM, please make sure NTLMv2 is in use as it is relatively easy to crack NTLM hashes and using NTLMv2 helps to avoid some of the existing exploits. One other issue with NTLM is that the strength of the session key and encryption process is based on the 128-bit RC4 cipher which is mostly considered broken these days. This means that message encryption used by WinRM can technically be cracked and the plaintext of the messages can be decrypted through some difficulty.

Microsoft's stance on NTLM is to use Kerberos if available and fall back to NTLM if it is required. Unfortunately due to the nature of Kerberos, this can be a common occurrence and NTLM can't be fully avoided in some scenarios. When using NTLM I would recommend that you;

- Make sure your password is of a decent length
- When on a Microsoft environment, ensure LmCompatibilityLevel is set to `3` for the client
- When not on a Microsoft environment, ensure the library that is generating the NTLM hashes supports NTLMv2
- Run over HTTPS which encrypts the messages with a stronger cipher suite than what NTLM can do (AES is considered secured compared to RC4)

Kerberos Auth

Kerberos authentication is the best option to use when in a domain environment. It is based on the MIT Kerberos v5 protocol and is mostly interchangeable with the GSSAPI implementations on most Unix systems. Kerberos is a good choice in most circumstances as;

- The encryption mechanism can use strong ciphers like AES
- The password is not sent to the server, only a short lived token is sent
- Kerberos supports mutual authentication by default
- Kerberos usually send just 1 request to commplete the auth process
- Kerberos can support credential delegation allowing the spawned process to access network resources

While it is technically used under the `Negotiate` auth provider, it can be explicitly used to ensure the process does not fall back to NTLM if it fails. Unfortunately the Achilles heel of Kerberos is that it requires a specific setup to work properly, you would need to have the following setup;

- A domain environment, Kerberos does not work with local account
- DNS is required, IP addresses cannot be used
- The client is configured to talk to the same realm/domain as the server so that is can acquire Kerberos tickets

When on a Microsoft operating system, the client setup is as simple as joining the workstation to the domain but for non Microsoft operating systems it requires some system packages to be installed and then configured. The packages that need to be installed differ based on the distribution that is being used but these are the ones for Debian and RedHat based Distros;

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

# Centos/RHEL
sudo yum install gcc python-devel krb5-devel krb5-workstation python-devel

# Fedora
sudo dnf install gcc python-devel krb5-devel krb5-workstation python-devel
```

Once installed, the client Kerberos configuration is set through the file `/etc/krb5.conf`. The the example below configured the server to understand the realm `DOMAIN.LOCAL` and configure the KDC (domain controller) to be `dc01.domain.local`.

```
[libdefaults]
    default_realm = DOMAIN.LOCAL

[realms]
    DOMAIN.LOCAL = {
      kdc = dc01.domain.local
      admin_server = dc01.domain.local
    }

[domain_realm]
.domain.local = DOMAIN.LOCAL
```

While this is default configuration, there are more options that can be set to control things like the ciphers that are allowed to be used for encryption and so on. Once configured, you can run `kinit user@realm.com` to get a Kerberos ticket for a particular domain account. This ticket is then used in the authentication process with the service meaning that that user's password is never sent in the authentication process.

CredSSP Auth

CredSSP (Credential Security Support Provider) is a Microsoft protocol that is designed to pass the user's credentials to a server in a secure way. This is unlike most other authentication protocols, as the username and password is provided to the logon process itself (Basic sends the encoded credentials but they are not available to the logon process). Because of this, a process that was created with CredSSP authentication is able to connect to a network with it's credentials.

Summed up the basic process flow for CredSSP is;

- The initial response returns a HTTP 401 error with `CredSSP` in the `WWW-Authenticate` header
- The client sets up a TLS connection and starts the TLS Handshake which includes things like cipher suite negotiation
- Once the handshake is complete, the client will send either an NTLM or Kerberos token to authenticate the user
- After authenticating the user, the client will encrypt the server's CredSSP public key with the authentication wrap function and send that to the server
- The server validates that the correct public key was used and there is no middle man in between the client and the server
- The server then sends it's public key again with the first bit set to 1 also encrypted with the authentication wrap function
- The client will verify the public key and verify the first bit was set to 1
- Once both the client and server have verified each other, the client will then encrypt the username and password with the authentication wrap function and send that to the server

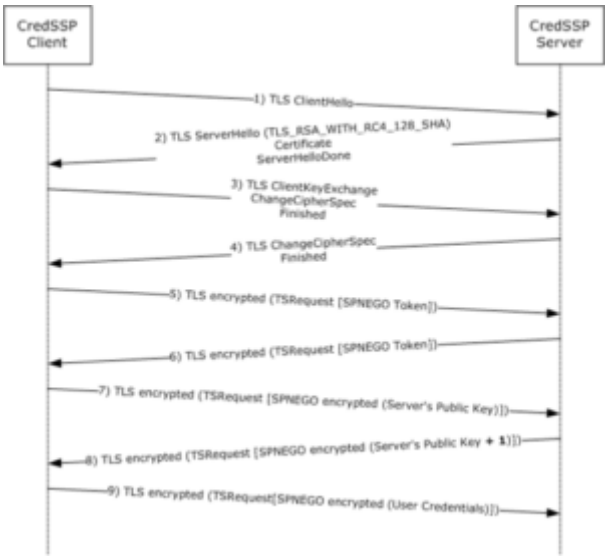
Here is Microsoft's diagram of the process flow

Except for the initial TLS handshake, all messages sent to and from the client are encrypted with the TLS protocol and messages that contain more sensitive info are doubly encrypted with the authentication wrap function as well.

As I spoke about in the logon type section, CredSSP is different from the other authentication protocols as the logon type spawned from CredSSP auth is `Network Cleartext`. This is what enables the process to access a network resource or protected internal resource like DPAPI work under CredSSP and fail on most other auth protocols.

Unfortunately CredSSP does have its downsides as there are numerous requests that are required to set up the authentication process compared to Kerberos or even NTLM which has 1 or 3 requests respectively. It also is not fully supported by third party libraries due to the complexity of the protocol but there are cases where there is third party integration.

Ultimately I believe it is an ok auth to use if Kerberos is not available in your environment and credential delegation is needed but the user should understand the implications that CredSSP creates before moving ahead.



Source: https://msdn.microsoft.com/en-us/library/cc226794.aspx

Authorization

Once the user has been authenticated, the authorization process will occur and checks whether the user is authorized to access the endpoint it is connecting to. By default, all members of the local Administrators group is able to connect on the WinRM endpoint, while `PSRP` is accessibly by both the local administrators and the `BUILTIN\Remote Management Users` group.

This gets somewhat confusing as some people are under the assumption that if you are a member of the `Remote Management Users` you automatically have the rights to run commands over WinRM. This is only the case for PowerShell Remoting, e.g. `Invoke-Command`, `Enter-PSSession` and not for commands run with `winrs` or most third party tools like Ansible.

The base WinRM process gets its rights from the permission set on the default root SDDL for WinRM, this can be modified by running

```
winrm configSDDL default
```

The command will open up a security permissions window like the below

As you can see, the `Administrators` group has full control over the endpoint, and to allow a non administrator to connect, they would need the `Read` and `Execute` rights. This can also be set programmatically but it can get a bit complex due to how Microsoft deals with Security Objects. When viewing the permissions, they are usually expressed in the SDDL form which is a “human readable” string of the Security Object. To get the SDDL of the WinRM endpoint, you can run `(Get-Item -Path WSMan:\localhost\Service\RootSDDL).Value` in PowerShell. The default SDDL for WinRM is;

```
O:NSG:BAD:P(A;;GA;;;BA)
(A;;GR;;;IU)S:P(AU;FA;GA;;;WD
)(AU;SA;GXGW;;;WD)
```

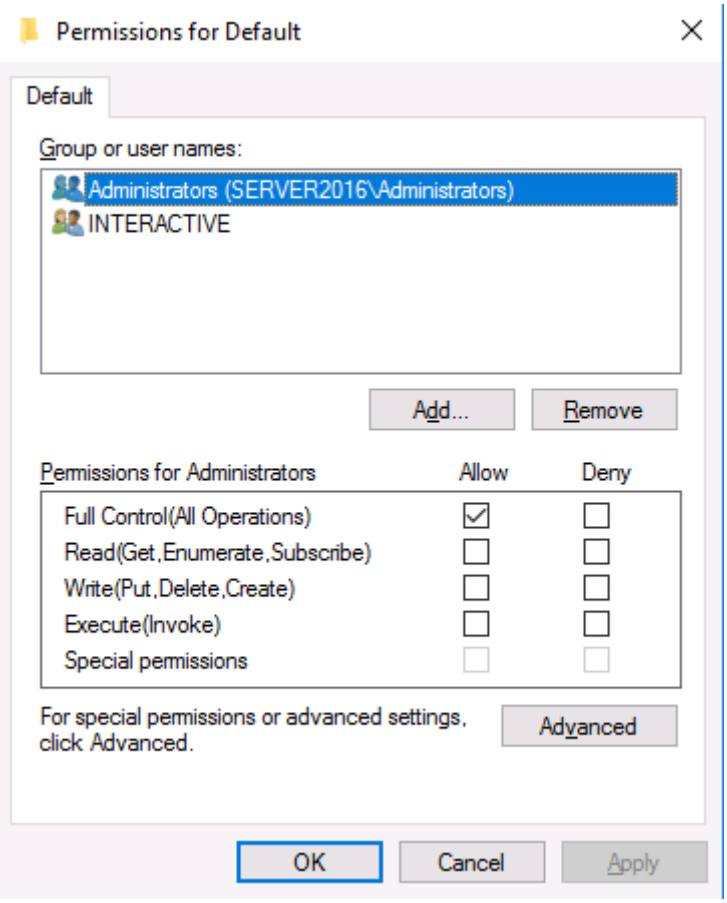
When I added a single user Read and Execute rights, it changes to;

```
O:NSG:BAD:P(A;;GA;;;BA)
(A;;GR;;;IU)(A;;GXGR;;;S-1-5-
21-4043990918-2312884405-
1850620780-
1003)S:P(AU;FA;GA;;;WD)
(AU;SA;GXGW;;;WD)
```

Breaking down the string, an SDDL is comprised of the following

- Owner `O`
- Group `G`
- DACL entries `D`
- SACL entries `S`

Let's break down the SDDL for the modified WinRM entry into each group



Default permissions on WinRM

```
Owner: O:NS
Group: G:BA
DACL: D:P(A;;GA;;;BA)(A;;GR;;;IU)(A;;GXGR;;;S-1-5-21-4043990918-2312884405-
1850620780-1003)
SACL: S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)
```

Right off the bat, we can see that the owner is `NS` which represents the `Network Service` SID and the primary group is `BA` which represents the `BUILTIN\Administrators` SID. The DACL has 3 protected entries as designated by the `P`;

- `(A;;GA;;;BA)`
- `(A;;GR;;;IU)`
- `(A;;GXGR;;;S-1-5-21-4043990918-2312884405-1850620780-1003)`

Each of the entries follow the format `ace_type;ace_flags;rights;object_guid;inherit_object_guid;account_sid;(resource_attribute)` where `resource_attribute` is an optional field. Splitting up each of the entries we can see what each entry represents

```
(A;;GA;;;BA)
Type: A = Allow
Flags: none
Rights: GA = Generic All (Full Control)
Object Guid: None
Inherit Object Guid: None
Sid: BA = BUILTIN\Administrators

(A;;GR;;;IU)
Type: A = Allow
Flags: none
Rights: GR = Generic Read
Object Guid: None
Inherit Object Guid: None
Sid: IU = Interactive Users

(A;;GXGR;;;S-1-5-21-4043990918-2312884405-1850620780-1003)
Type: A = Allow
Flags: none
Rights: GXGR = Generic Execute and Generic Read
Object Guid: None
Inherit Object Guid: None
Sid: S-1-5-21-4043990918-2312884405-1850620780-1003 = The SID for the "user"
account
```

The same applies to the SACL entries but I won't go into them as they define the auditing rules and not access rules that this is about.

While it is straightforward to manipulate the permissions when you are able to use a GUI, there are definitely cases where doing this under a script is preferable. You can always just manually manipulate the string but this would be a headache inducing process and luckily there is a better way. Using PowerShell and .NET 4.5, you can import the SDDL to a `CommonSecurityDescriptor` object and manipulate it from there in a more programmatic fashion.

Here is an example of taking in the existing security object and adding the user read and execute rights, note this would not be idempotent and more checks are required to ensure the ACE is not already there;

```
$GENERIC_READ = 0x80000000
$GENERIC_WRITE = 0x40000000
$GENERIC_EXECUTE = 0x20000000
$GENERIC_ALL = 0x10000000

# get SID of user/group to add
$user = "user"
$user_sid = (New-Object -TypeName System.Security.Principal.NTAccount -
ArgumentList $user).Translate([System.Security.Principal.SecurityIdentifier])

# get the existing SDDL of the WinRM listener
$sddl = (Get-Item -Path WSMan:\localhost\Service\RootSDDL).Value

# convert the SDDL string to a SecurityDescriptor object
$sd = New-Object -TypeName System.Security.AccessControl.CommonSecurityDescriptor
-ArgumentList $false, $false, $sddl

# apply a new DACL to the SecurityDescriptor object
$sd.DiscretionaryAcl.AddAccess(
    [System.Security.AccessControl.AccessControlType]::Allow,
    $user_sid,
    ($GENERIC_READ -bor $GENERIC_EXECUTE),
    [System.Security.AccessControl.InheritanceFlags]::None,
    [System.Security.AccessControl.PropagationFlags]::None
)

# get the SDDL string from the changed SecurityDescriptor object
$new_sddl =
$sd.GetSddlForm([System.Security.AccessControl.AccessControlSections]::All)

# apply the new SDDL to the WinRM listener
Set-Item -Path WSMan:\localhost\Service\RootSDDL -Value $new_sddl -Force
```

SDDL strings can get a lot more complex than this and you can learn more about SDDL string formats at Security Descriptor String Format.

The Future

Looking to the future, I don't see anything major changing with WinRM and we will still have some of the issues we see today around complex configuration and annoying restrictions. I do have some hope around Microsoft's Win32-OpenSSH port as that solves the speed and some authentication complexity issues. Unfortunately the `Network` logon type problem is still an issue there as those restrictions are limitations set by Windows and isn't just a WinRM thing.

It seems like Microsoft is starting to embrace some of the good standards that are in place in the Unix land which is great for everybody but let's just hope they don't lapse into their previous cycle of embrace, extend, extinguish like in the past.