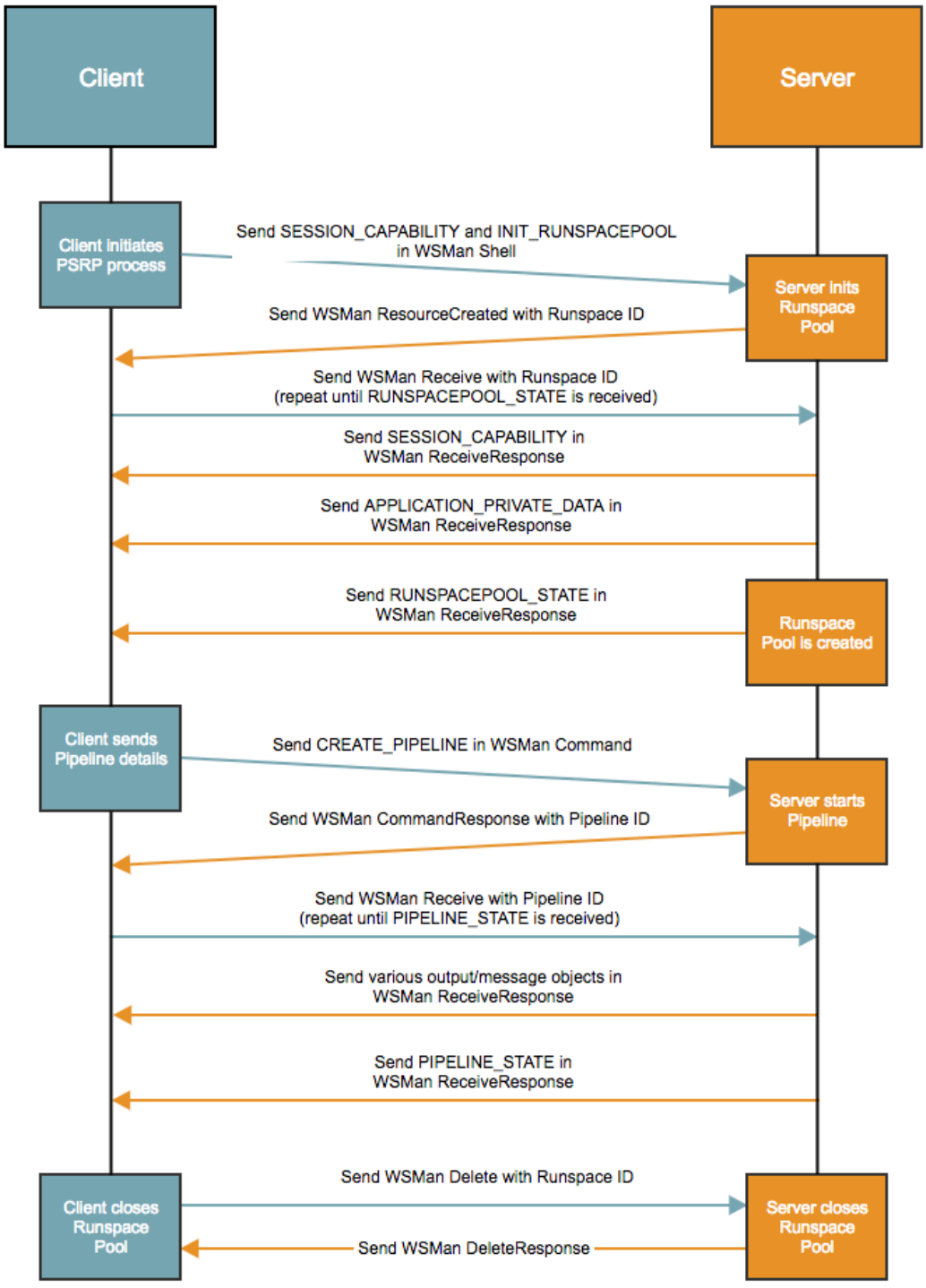
PowerShell Remoting on Python

bloggingforlogging.com/2018/08/14/powershell-remoting-on-python

August 14, 2018



One thing I am looking into everyday as part of my job is how to make the remote management of Windows servers easier. Currently the best way is through WinRM but as I've written about before, WinRM can be such a vague term. It can mean refer to different technologies and the answer to what is WinRM depends on who you ask and in what context. When talking about third party implementations, like Ansible/pywinrm; it refers to the raw Windows Remote Shell (WinRS) over the WSMan transport protocol. Whereas, a Windows admin, would probably be talking about the PowerShell remoting protocol through things like the `Invoke-Command`, `Enter-PSSession` PowerShell cmdlets.

While they both share the same transport mechanism over WSMan, they are really not the same thing. WinRS is a very basic remote shell protocol to run commands while the PowerShell Remoting Protocol (PSRP) is like WinRS but on steroids. This can lead to some confusion when trying to explain some of the differences between WinRS and PSRP, but in short these are the benefits I see with using PSRP over WinRS;

- PSRP is faster when executing PowerShell commands as it does not need to start a new PowerShell process on each invocation
- PSRP can connect to custom endpoints/configurations, enabling things like Just Enough Administration (JEA)
- PSRP deals with PowerShell objects directly, e.g. strings, ints, dicts, lists, and other objects can be serialised/deserialised as it transfers between each host
- PSRP has a mechanism for securely sharing secrets through the SecureString type

Before I continue on to talk about this a bit more, I wanted to give a shout out to Matt Wrock who wrote some posts about PSRP here and here. While I think some of the points raised in these blogs are not 100% correct, they are what inspired me to work down this path and are really helpful for anyone interested in this topic. I highly recommend you read through them when you have a chance as they are really great articles on PSRP and how it has been implemented in the Ruby WinRM library

What is PSRP and PyPSRP

So what is PSRP, PSRP is an acronym for PowerShell Remoting Protocol. As I mentioned above it is a protocol that sits on top of the WSMan/WinRM protocol that was designed for interacting with a PowerShell instance remotely. In recent years it has been expanded to also run on other transports mechanisms like SSH but this has only been a recent addition with the PowerShell Core (6.0+) releases. I'm still not fully sold on PowerShell Core and it's benefits today so I will mostly be focusing on PowerShell 2 – 5.x which only allows the WSMan transport.

PyPSRP is a Python library I've written that is designed to operate on the PSRP layer rather than just the WinRS component most other third party libraries offer. The only other third party library that I know works with PSRP is the Ruby WinRM library, but its implementation has mostly been around adapting the PSRP components to an stdio based process rather than taking full advantage of PSRP and what it offers. When I first started to look into this protocol, I went a similar route to the Ruby WinRM library and just bolted on PSRP into the pywinrm library. I was young and naive and expected excellent performance from the implementation I jammed into the library at the time. This did not pan out at all and I kept it on the backburner for a few months and focused on other projects.

Over time I kept on coming across really weird, incorrect, buggy behaviour in pywinrm and I didn't have the maintainer rights to the repo so I decided to cut my losses and start again in a branch new project. From the ashes of my failure, pypsrp was born.

What I set out to achieve when creating pypsrp was;

- Include all the same features from pywinrm but in a nicer and less confusing interface
- The ability to run commands over both the WinRS layer and PSRP layer
- Support all authentication types like `Basic`, `Certificate`, `Negotiate`, `Kerberos`, and `CredSSP`
- Support message encryption when running over HTTP with `Negotiate`, `Kerberos`, and `CredSSP` authentication
- Deal directly with the PowerShell objects rather than just strings
- Serialise secure strings so it can securely transmit secrets not in plaintext
- Create an interface that closely resembles the .NET System.Management.Automation namespace
- Create a higher level interface for people to run commands on either protocol as well as copy and fetch files

In the end, I believe I've been able to achieve these goals, and some more, with pypsrp.

Before I go into how to use pypsrp, I want to go through the core concepts of PSRP and how it all works.

Key Concepts

There are a few key concepts used in PSRP that I thought it best to mention and explain a bit more.

Runspace Pool and Runspaces

Runspaces are quite simply a new thread on an existing PowerShell process that can run a single “Pipeline” at one point in time. A Runspace Pool is a collection/pool of Runspaces that can handle the execution of multiple Runspaces in an efficient manner for you. Runspaces is mostly a developer focused topic so you usually don’t need to understand this layer. In reality, a lot of the Microsoft cmdlets that run over WinRM, like the `*-PSSession` and `*-Job` cmdlets, use Runspaces in their implementation.

Pipelines

In PSRP, a pipeline is an ordered collection of “Statements” to execute on the “Runspace”. There is a 1 to 1 mapping of Runspaces and Pipelines which means that a Runspace can execute one and only one pipeline. This does get confusing as a Pipeline can then create what is called a nested pipeline within itself but fundamentally, a Runspace can only have one root Pipeline. A nested pipeline is special because it interrupts the thread of the running pipeline and can be used to check things like the state of a variable or running command. Once the nested pipeline is finished executing, the parent pipeline will resume from where it was blocked.

Statements

A statement is an ordered collection of “Commands” or scripts to run on a “Pipeline”. A statement can easily be seen as a unit of work, and is most commonly represented as a line in Powershell, e.g.

```
# 2 statements
$service = Get-Service -Name winrm
$service.Status

# same 2 statements in the 1 line
$service = Get-Service -Name winrm; $service.Status
```

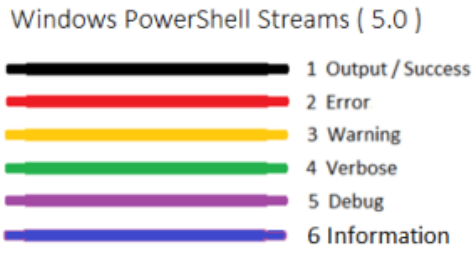
Commands

A command is an ordered collection of cmdlets or scripts that are piped together. A script is simply like a PowerShell codeblock which can contain multiple lines of code as well as functions and other properties. When there are multiple cmdlets or scripts in a command, the output of the first cmdlet/script is piped into the input of the second cmdlet/script, e.g. `Get-Service -Name winrm | Stop-Service` will pipe the result of `Get-Service` into `Stop-Service`. Each cmdlet/script can have none or multiple parameters/arguments when running.

Streams

If you aren’t new to PowerShell then you are probably already aware of the concept of streams in PowerShell. Unlike typical processes which use the stdio concept of an input, output and error stream of bytes, PowerShell contains 6 (5 pre v 5.0) streams;

I recommend reading Understanding Streams, Redirect, and Write-Host in PowerShell and it’s follow up article Welcome to the PowerShell Information Stream which go in further detail of what streams are and how to use them.



Source: https://blogs.technet.microsoft.com/heyscriptingguy/2 scripter-welcome-to-the-powershell-information-stream/

Objects

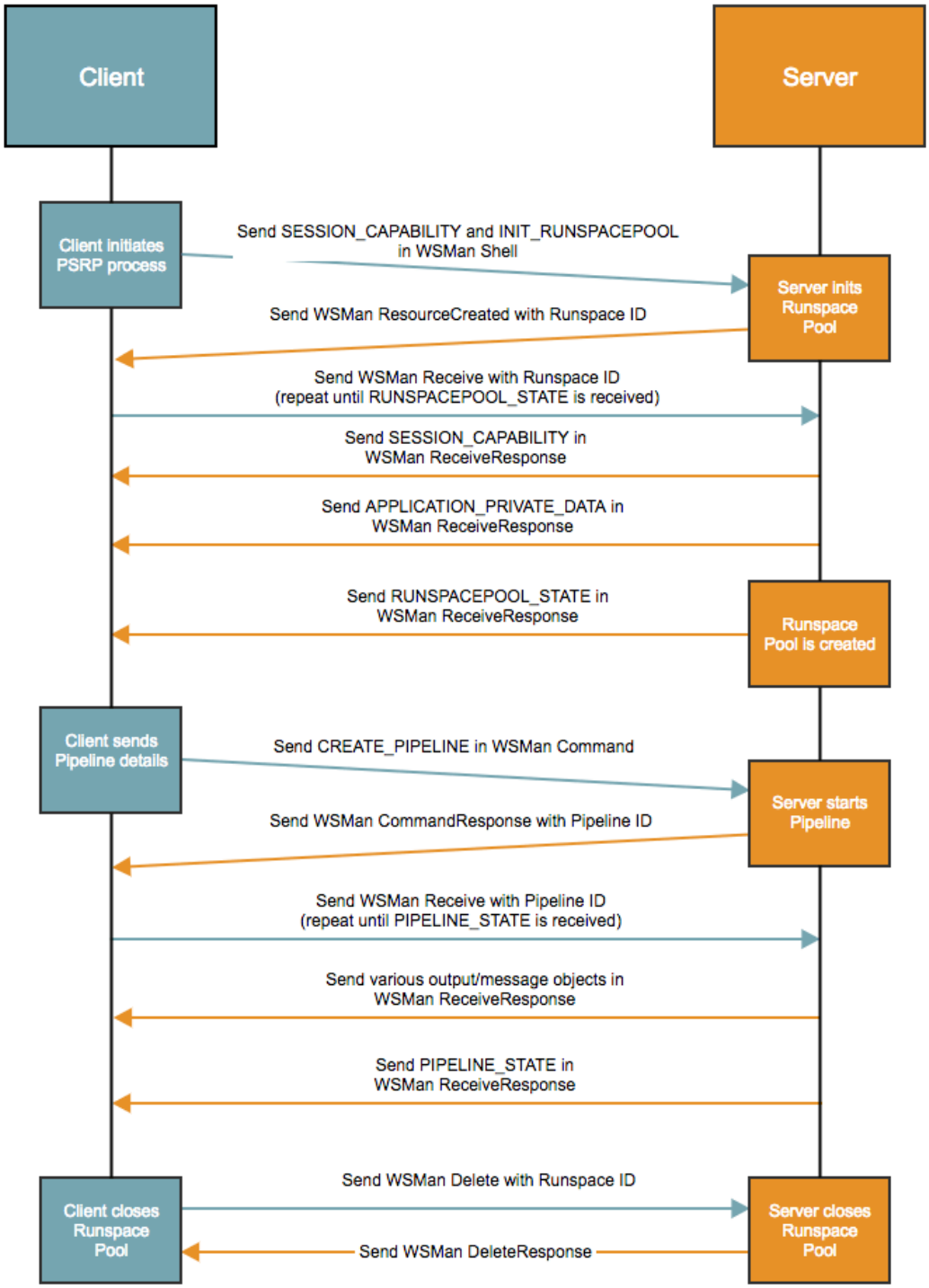
As well as having more streams, PowerShell differs from stdio where it streams objects rather than bytes. While technically at a deeper level they are represented as bytes, this is not exposed in PowerShell layer. For example, if a C program runs `printf("Hello World")`, it will send the bytes `48 65 6c 6c 6f 20 57 6f 72 6c 64` to the stdout stream. Compare that to PowerShell where `Write-Output "Hello World"` will instead send a string as a .NET object on the first stream. A problem with this approach is how to represent these objects over remote transport protocol like WSMan as these objects are not native to this layer. Microsoft decided to use CLIXML to work around this and I'll go into more details for this further in this post (hint it's not pretty).

Process Flow

So now we have a basic understanding of some of the components used in PSRP, let's walk through the steps required to run a pipeline of commands on a remote host. In this example I'll explain in more details, what pypsrp does when executing the following script;

```
from pypsrp.powershell import PowerShell, RunspacePool
from pypsrp.wsman import WSMan

wsman = WSMan("server2016.domain.local", username="vagrant",
              password="vagrant",
              cert_validation=False)

with RunspacePool(wsman) as pool:
    ps = PowerShell(pool)
    ps.add_cmdlet("Get-PSDrive").add_parameter("Name", "C")
    ps.invoke()
    # we will print the first object returned back to us
    print(ps.output[0])
```

This script effectively runs the cmdlet `Get-PSDrive -Name C` on the host `server2016.domain.local`. Here is a basic process flow of messages exchanged when executing this code;



What you are not seeing, lots and lots of XML

While the messages used above are the most common types used in the PSRP protocol, there are are currently 31 distinct PSRP message types in the base protocol.

Message Structure

While this article is about PSRP, this section will also break down the WSMan component to help you get a better understanding of the data sent over the wire. The following sections will break down the following message which is the first message sent in a standard PSRP exchange;

```
<s:Envelope xmlns:rsp="http://schemas.microsoft.com/wbem/wsman/1/windows/shell"
xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsman="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"
xmlns:wsmv="http://schemas.microsoft.com/wbem/wsman/1/wsman.xsd">
    <s:Header>
        <wsa:Action
s:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/09/transfer/Create</wsa:Action>
        <wsmv:DataLocale s:mustUnderstand="false" xml:lang="en-US"/>
        <wsman:Locale s:mustUnderstand="false" xml:lang="en-US"/>
        <wsman:MaxEnvelopeSize s:mustUnderstand="true">153600</wsman:MaxEnvelopeSize>
        <wsa:MessageID>uuid:1C74E6AE-8C7A-4C03-99D1-C4AD3DABFD6D</wsa:MessageID>
        <wsman:OperationTimeout>PT20S</wsman:OperationTimeout>
        <wsa:ReplyTo>
            <wsa:Address
s:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:Address
>
        </wsa:ReplyTo>
        <wsman:ResourceURI
s:mustUnderstand="true">http://schemas.microsoft.com/powershell/Microsoft.PowerShell</wsman:Resource
URI>
        <wsmv:SessionId s:mustUnderstand="false">uuid:C6602DD7-6096-4983-A956-
82B048821F4D</wsmv:SessionId>
        <wsa:To>http://server2016.domain.local:5985/wsman</wsa:To>
        <wsman:OptionSet s:mustUnderstand="true">
            <wsman:Option MustComply="true" Name="protocolversion">2.3</wsman:Option>
        </wsman:OptionSet>
    </s:Header>
    <s:Body>
        <rsp:Shell ShellId="5A416EA5-FB2A-4AAA-91BF-77BF51043386">
            <rsp:InputStreams>stdin pr</rsp:InputStreams>
            <rsp:OutputStreams>stdout</rsp:OutputStreams>
            <creationXml
xmlns="http://schemas.microsoft.com/powershell">AAAAAAAAAAEAAAAAAAAAAAMAAADHAgAAAAIAAQBaQW6l+ypKqpG/
d79RBDOGAAAAAAAAAAAAAAAAAAAAAADxPYmogUmVmSWQ9IjAiPjxNUz48VmVyc2lvbiBOPSJwcm90b2NvbHZlcnNpb24iPjIuMzwv
VmVyc2lvbj48VmVyc2lvbiBOPSJQU1ZlcnNpb24iPjIuMDwvVmVyc2lvbj48VmVyc2lvbiBOPSJTZXJpYWxpemF0aW9uVmVyc2lv
biI+MS4xLjAuMTwvVmVyc2lvbj48L01TPjwvT2JqPgAAAAAAAAACAAAAAAAAAADAAAC/QIAAAAEAAEAWkFupfsqSqqRv3e/UQQz
hgAAAAAAAAAAAAAAAAAAAAA8T2JqIFJlZklkPSIwIj48TVM+PEkzMiBOPSJNaW5SdW5zcGFjZXMiPjE8L0kzMj48STMyIE49Ik1h
eFJ1bnNwYWNlcyI+MTwvSTMyPjxPYmogTj0iUFNUaHJlYWRPcHRpb25zIiBSZWZJZD0iMSI+PFROIFJlZklkPSIwIj48VD5TeXN0
ZW0uTWFuYWdlbWVudC5BdXRvbWF0aW9uLlJ1bnNpYWNlcy5QU1RocmVhZE9wdGlvbnM8L1Q+PFQ+U3lzdGVtLkVudW08L1Q+PFQ+
U3lzdGVtLlZhbHVlVHlwZTwvVD48VD5TeXN0ZW0uT2JqZWN0PC9UPjwvVE4+PFRvU3RyaW5nPkRlZmF1bHQ8L1RvU3RyaW5nPjxJ
MzI+MDwvSTMyPjwvT2JqPjxPYmogTj0iQXBhcnRtZW50U3RhdGUiIFJlZklkPSIyIj48VE4gUmVmSWQ9IjEiPjxUPlN5c3RlbS5N
YW5hZ2VtZW50LkF1dG9tYXRpb24uUnVuc3BhY2VzLkFwYXJ0bWVudFN0YXRlPC9UPjxUPlN5c3RlbS5WYWx1ZVR5cGU8L1Q+
bS5WYWx1ZVR5cGU8L1Q+PFQ+U3lzdGVtLk9iamVjdDwvVD48L1ROPjxUb1N0cmluZz5VTktOT1dOPC9Ub1N0cmluZz48STMyPjI8
L0kzMj48L09iaj48T2JqIE49Ikhvc3RJbmZvIiBSZWZJZD0iMyI+PE1TPjxCIE49Il9pc0hvc3ROdWxsIj50cnVlPC9CPjxCIE49
Il9pc0hvc3RVSU51bGwiPnRydWU8L0I+PEIgTj0iX2lzSG9zdFJhd1VJTnVsbCI+dHJ1ZTwvQj48QiBOPSJfdXNlUnVuc3BhY2VI
b3N0Ij50cnVlPC9CPjwvTVM+PC9PYmo+PE5pbCBOPSJBcHBsaWNhdGlvbkFyZ3VtZW50cyIgLz48L01TPjwvT2JqPg==
</creationXml>
        </rsp:Shell>
    </s:Body>
</s:Envelope>
```

WSMan

WSMan is a SOAP based protocol that is sent over HTTP which means it's time to get down and dirty with the first XML layer used in PSRP. Each WSMan message contains a root element `<{http://www.w3.org/2003/05/soap-envelope}Envelope>` which in turn contains 2 elements;

- `<{http://www.w3.org/2003/05/soap-envelope}Header>`: Contains metadata around the message such as the action, the resource/configuration endpoint, and Runspace Pool/Shell selector each the message relates to.
- `<{http://www.w3.org/2003/05/soap-envelope}Body>`: Depending on the type/action can be empty or in a specific format, this is where the PSRP message are usually contained.

While it isn't guaranteed, most implementations I've seen use `s` as the namespace prefix for `http://www.w3.org/2003/05/soap-envelope`, so usually these elements are represented as `<s:Envelope>`, `<s:Header>`, and `<s:Body>`. When in doubt you can always look at the prefix defined in the xmlns attribute definitions for each message.

One of the key fields in the header block is the `<{http://schemas.xmlsoap.org/ws/2004/08/addressing}:Action>` element. This element is used by the client and server to determine the format the body element will be in. There are many different action types in the WSMan protocol but the following are key in PSRP;

- `Create`: Used to create the RunspacePool, only the `SESSION_CAPABILITY` and `INIT_RUNSPACEPOOL` objects are sent with a create message
- `Command`: Used to create the Pipeline, only the INIT_PIPELINE` object is sent with a command message
- `Connect`: Connect to a disconnected RunspacePool (created by another client) that is in a disconnected state
- `Delete`: Used to close the RunspacePool, no PSRP objects are sent with this message
- `Disconnect`: Disconnect from an opened RunspacePool
- `Enumerate`: Used to get a list of RunspacePools and Pipelines on a remote host
- `Fault`: Used to display error information if something went wrong in the request
- `Receive`: Used to get the output of a RunspacePool or Pipeline based on the ID specified, no PSRP objects are sent with this message
- `Reconnect`: Reconnect to a disconnected RunspacePool (created by the same client) that is in a disconnected state
- `Send`: Used to send the majority of the PSRP objects to the server
- `Signal`: Forcibly stop a running Pipeline

Looking at our example message we can see the following headers are defined;

- `Action`: This is set to `http://schemas.xmlsoap.org/ws/2004/09/transfer/Create` which tells the server the body should contain a resource object that needs to be created, in this case a Shell/Runspace Pool
- `DataLocale`: The format of the numerical data sent by the client. This is largely inconsequential to the end user and is just an implementation detail
- `Locale`: Similar to DataLocale but it specifies the language of the text in the message
- `MaxEnvelopeSize`: The maximum size of the whole WSMan message that the client can expect in a response
- `MessageID`: A unique ID for this message, used to identify responses with the request sent from the client
- `OperationTimeout`: The maximum time that the server can spend processing the request, if it exceeds this limit than it will send a timeout fault
- `ReplyTo`: Always constant as defined by the rules in MS-WSMV
- `ResourceURI`: The resource URI to connect to, by default it is set to connect to the `Microsoft.PowerShell` configuration endpoint on the server but can be changed depending on the user setup
- `SessionId`: A unique ID for the current user session, spans multiple requests and is required if the user wants to disconnect a Runspace Pool
- `To`: The endpoint this message is for, this isn't mandatory but makes it easier when debugging messages to see who it is for
- `OptionSet`: Used to define various options in by the PSRP protocol, currently only `protocolversion` is defined as an OptionSet here
- `SelectorSet`: Not used in the create message but contains the Runspace Pool ID in subsequent messages

When looking at the body of the message we can see it contains the following elements;

`Shell`: The root element with a `ShellId` attribute defined by the client. If the `ShellId` is not defined here then the server will generate a unique one
- `InputStreams`: For PSRP, this must be set to `stdin pr`, it basically says it can expect input through the stdin pipe as well as a "prompt response" input pipe
- `OutputStreams`: For PSRP, this must be set to `stdout`, `stdout` is used as the stream name when sending the `Receive` messages
- `creationXml`: Contains the the base64 encoding `SESSION_CAPABILITY` and `INIT_RUNSPACEPOOL` PSRP message fragments

Technically the `creationXml` may not contain the full `INIT_RUNSPACEPOOL` PSRP message if it exceeds the `MaxEnvelopeSize` set on the server config. I have been unable to replicate this scenario on a real Windows client and it would be quite rare for that message to exceed the max size.

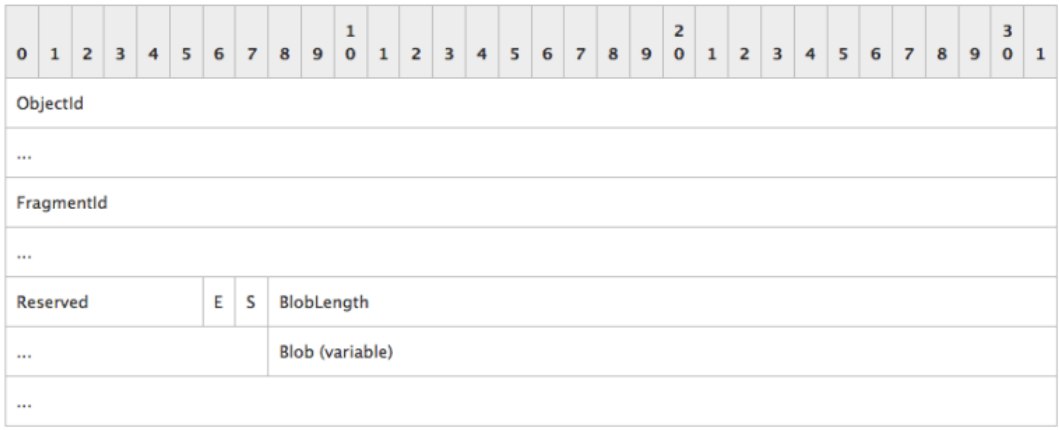
If successful, I should get a `CreateResponse` message back from the server, here is an example of one of these responses;

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:x="http://schemas.xmlsoap.org/ws/2004/09/transfer"
xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"
xmlns:rsp="http://schemas.microsoft.com/wbem/wsman/1/windows/shell"
xmlns:p="http://schemas.microsoft.com/wbem/wsman/1/wsman.xsd" xml:lang="en-US">
    <s:Header>
        <a:Action>http://schemas.xmlsoap.org/ws/2004/09/transfer/CreateResponse</a:Action>
        <a:MessageID>uuid:0880F6CA-19EA-4CE3-8309-E0512677BABD</a:MessageID>
        <a:To>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</a:To>
        <a:RelatesTo>uuid:1C74E6AE-8C7A-4C03-99D1-C4AD3DABFD6D</a:RelatesTo>
    </s:Header>
    <s:Body>
        <x:ResourceCreated>
            <a:Address>http://server2016.domain.local:5985/wsman</a:Address>
            <a:ReferenceParameters>

<w:ResourceURI>http://schemas.microsoft.com/powershell/Microsoft.PowerShell</w:ResourceURI>
                <w:SelectorSet>
                    <w:Selector Name="ShellId">5A416EA5-FB2A-4AAA-91BF-77BF51043386</w:Selector>
                </w:SelectorSet>
            </a:ReferenceParameters>
        </x:ResourceCreated>
        <rsp:Shell xmlns:rsp="http://schemas.microsoft.com/wbem/wsman/1/windows/shell">
            <rsp:ShellId>5A416EA5-FB2A-4AAA-91BF-77BF51043386</rsp:ShellId>

<rsp:ResourceUri>http://schemas.microsoft.com/powershell/Microsoft.PowerShell</rsp:ResourceUri>
            <rsp:Owner>SERVER2016\vagrant</rsp:Owner>
            <rsp:ClientIP>192.168.56.1</rsp:ClientIP>
            <rsp:ProcessId>2812</rsp:ProcessId>
            <rsp:IdleTimeOut>PT7200.000S</rsp:IdleTimeOut>
            <rsp:InputStreams>stdin pr</rsp:InputStreams>
            <rsp:OutputStreams>stdout</rsp:OutputStreams>
            <rsp:MaxIdleTimeOut>PT2147483.647S</rsp:MaxIdleTimeOut>
            <rsp:Locale>en-US</rsp:Locale>
            <rsp:DataLocale>en-US</rsp:DataLocale>
            <rsp:CompressionMode>NoCompression</rsp:CompressionMode>
            <rsp:ProfileLoaded>Yes</rsp:ProfileLoaded>
            <rsp:Encoding>UTF8</rsp:Encoding>
            <rsp:BufferMode>Block</rsp:BufferMode>
            <rsp:State>Connected</rsp:State>
            <rsp:ShellRunTime>P0DT0H0M0S</rsp:ShellRunTime>
            <rsp:ShellInactivity>P0DT0H0M0S</rsp:ShellInactivity>
        </rsp:Shell>
    </s:Body>
</s:Envelope>
```

We can see it returned a few bits and pieces about the created Runspace but the one we are interested in is the `SelectorSet` returned. This is added to the WSMan Header elements for each subsequent message sent for this Runspace Pool.

PSRP Fragment

So now that you've seen the WSMan message side, the remaining layers are all defined in PSRP. I said, in the example above, that the `creationXml` contained a base64 encoded value of the `SESSION_CAPABILITY` and `INIT_RUNSPACEPOOL` message but that's not 100% accurate. Each WSMan message has a maximum size that the client can send to the server which can be problematic when dealing with objects larger than this limit (the default limit since PSv3 is 500 KiB). To overcome this hurdle, PSRP always fragments messages before adding it to the WSMan payload, even if it is smaller than the max size. The structure of a PSRP fragment is as follows



Lets break down each of the fields;

- `ObjectId`: An 8 byte unsigned integer for the global fragment ID counter starting at 1, no 2 fragments in the same session should have the same ID
- `FragmentId`: An 8 byte unsigned integer for the sequence number that identifies where this fragment belongs in the whole message, this starts at 0
- `Reserved`: 6 bits that are set to 0, not used right now
- `E`: 1 bit that specifies if this is the last fragment from the message
- `S`: 1 bit that specifies if this is the first fragment from the message
- `BlobLength`: A 4 byte unsigned integer that is the size of the `Blob` field
- `Blob`: Either the entire PSRP message if it fits into the `MaxEnvelopeSize` or part of a fragmented PSRP message

If we were to take the value of `creationXml`, base64 decode it, we get a total of `1006` bytes. Each fragment header is a fixed size of 21 bytes so let's get the first 21 bytes in our output and parse our fragment header;

```
00 00 00 00 00 00 00 01
00 00 00 00 00 00 00 00
03
00 00 00 c7
```

- `ObjectId`: 1
- `FragmentId`: 0
- `Reserved + E + S`: In binary this is `0000 0011` which means both the E and S bit are set (the blob is the whole message)
- `BlogLength`: 199

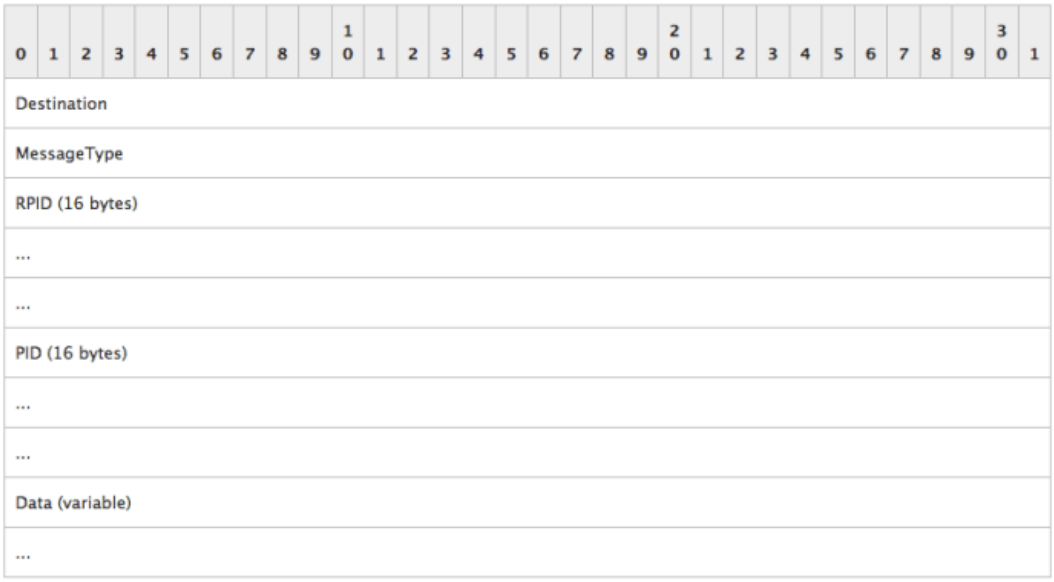
With these values we know this fragment contains the full PSRP message which is `199` bytes long, this still leaves `786` bytes left (1006 – (21 + 199)) in our payload which means there's another fragment for us to parse. If we get bytes 220 – 240 we can parse the next fragment header;

```
00 00 00 00 00 00 00 02
00 00 00 00 00 00 00 00
03
00 00 02 fd
```

We can see the `ObjectId` for this second fragment has been incremented to 2 and the `FragmentId` has been reset back to 0. This fragment also has the `E` and `S` bits set so we know it also contains the full PSRP message which is `765` bytes long. This brings us to the full `1006` bytes in our payload so there's no more fragments to parse.

PSRP Message

We've parsed the payload and found two fragments, one at `199` bytes long and another at `765` bytes and are now ready to look at the message structure itself. Each message is structured like the following;



Lets break down each of the fields;

- `Destination`: A 4 byte field that tells us whether the message is targeted towards the client `01 00 00 00` or the server `02 00 00 00`
- `MessageType`: A 4 byte field that defines the type of message contained in the data field
- `RPID`: A 16 byte field of the Runspace Pool/Shell ID returned by the server in the `CreateResponse` message
- `PID`: A 16 byte field of the Pipeline ID of the Pipeline the message is targeted to, this is no set if it isn't targeted to a pipeline
- `Data`: The actual PSRP data of the PSRP message defined by `MessageType`

We looking at the first fragment we get;

```
02 00 00 00
02 00 01 00
5a 41 6e a5 fb 2a 4a aa 91 bf 77 bf 51 04 33 86
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

# I have conveniently converted these bytes to the utf-8 form
<Obj RefId="0">
    <MS>
        <Version N="protocolversion">2.3</Version>
        <Version N="PSVersion">2.0</Version>
        <Version N="SerializationVersion">1.1.0.1</Version>
    </MS>
</Obj>
```

- `Destination`: This is targeted towards the server
- `MessageType`: This type is `SESSION_CAPABILITY`
- `RPID`: The Runspace Pool/Shell ID is actually a GUID, this value is the bytes representation of `5A416EA5-FB2A-4AAA-91BF-77BF51043386`
- `PID`: This is not set as this message isn't targeted towards a Pipeline
- `Data`: The remaining bytes in the fragment is the data, as you can see, we have another XML string in CLIXML format

Now lets have a look at the second fragment in our payload

```
02 00 00 00
04 00 01 00
5a 41 6e a5 fb 2a 4a aa 91 bf 77 bf 51 04 33 86
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

<Obj RefId="0">
    <MS>
        <I32 N="MinRunspaces">1</I32>
        <I32 N="MaxRunspaces">1</I32>
        <Obj N="PSThreadOptions" RefId="1">
            <TN RefId="0">
                <T>System.Management.Automation.Runspaces.PSThreadOptions</T>
                <T>System.Enum</T>
                <T>System.ValueType</T>
                <T>System.Object</T>
            </TN>
            <ToString>Default</ToString>
            <I32>0</I32>
        </Obj>
        <Obj N="ApartmentState" RefId="2">
            <TN RefId="1">
                <T>System.Management.Automation.Runspaces.ApartmentState</T>
                <T>System.Enum</T>
                <T>System.ValueType</T>
                <T>System.Object</T>
            </TN>
            <ToString>UNKNOWN</ToString>
            <I32>2</I32>
        </Obj>
        <Obj N="HostInfo" RefId="3">
            <MS>
                <B N="_isHostNull">true</B>
                <B N="_isHostUINull">true</B>
                <B N="_isHostRawUINull">true</B>
                <B N="_useRunspaceHost">true</B>
            </MS>
        </Obj>
        <Nil N="ApplicationArguments"/>
    </MS>
</Obj>
```

The only difference in this message is the `MessageType` field which is each to `INIT_RUNSPACEPOOL` and the `Data` field. Once again I've conveniently converted the byte values of `Data` to the clixml value of the message.

CLIXML

We've seen the XML data in the WSMan messages but to make matters even worse, there's another layer of XML hiding behind the base64 encoded text, enter CLIXML. I personally think this is the most complex component of PSRP and I'm not 100% happy with how I've implemented it in pypsrp. I may revisit the implementation at some point in the future but for now it is workable. Ultimately, CLIXML is a way of serialising objects like strings, lists, .NET classes, and more into a format that can be sent over another channel. There are two types of objects in CLIXML;

- `Primitive`: the somewhat easy stuff like strings, integers, see here for a full list
- `Complex`: the hard/ugly stuff like lists, dictionaries, pretty much the rest of the .NET objects. Objects used by PSRP are found here but this does not include all the objects returned in the PowerShell output which can be anything

Starting with a string, primitive object, if I was to serialise

```
Hello World\nXML is ? when dealing with things like _x000A_\n
```

to CLIXML, it would become

```
<S>Hello World_x000A_XML is _xD83D__xDCA9_ when dealing with things like
_x005F_x000A__x000A_</S>
```

Normal unicode values are fine but when dealing with control codes, surrogate chars and escaping string that match the regex `_x([a-fA-F0-9]{4})_` it becomes a massive mess. We can see control codes are escaped in the format `_x{utf-16-be hex value}_` and values that are already like this pattern have the leading `_` replaced with `_x005F_`. Surrogate chars are a bit more complex but the same fundamental rules apply, for example Pile of Poo Emoji has a UTF-16 hex representation of `D8 3D DC A9` and this will be serialised like 2 code points in clixml `_xD83D__xDCA9_`.

Complex objects are a whole other ball game, it makes escaping strings look like child's play. You can see that each PSRP message has its own object structure and using the `SESSION_CAPABILITY` we send we can see it is formatted like this;

```
<Obj RefId="0">
    <MS>
        <Version N="protocolversion">2.3</Version>
        <Version N="PSVersion">2.0</Version>
        <Version N="SerializationVersion">1.1.0.1</Version>
    </MS>
</Obj>
```

We can see this is a relatively simple "complex object" with three extended properties; `protocolversion`, `PSVersion`, and `SerializationVersion` all of the primitive `Version` type. If we look at the `INIT_RUNSPACEPOOL` it starts to be a bit more daunting

```
<Obj RefId="0">
    <MS>
        <I32 N="MinRunspaces">1</I32>
        <I32 N="MaxRunspaces">1</I32>
        <Obj N="PSThreadOptions" RefId="1">
            <TN RefId="0">
                <T>System.Management.Automation.Runspaces.PSThreadOptions</T>
                <T>System.Enum</T>
                <T>System.ValueType</T>
                <T>System.Object</T>
            </TN>
            <ToString>Default</ToString>
            <I32>0</I32>
        </Obj>
        <Obj N="ApartmentState" RefId="2">
            <TN RefId="1">
                <T>System.Management.Automation.Runspaces.ApartmentState</T>
                <T>System.Enum</T>
                <T>System.ValueType</T>
                <T>System.Object</T>
            </TN>
            <ToString>UNKNOWN</ToString>
            <I32>2</I32>
        </Obj>
        <Obj N="HostInfo" RefId="3">
            <MS>
                <B N="_isHostNull">true</B>
                <B N="_isHostUINull">true</B>
                <B N="_isHostRawUINull">true</B>
                <B N="_useRunspaceHost">true</B>
            </MS>
        </Obj>
        <Nil N="ApplicationArguments"/>
    </MS>
</Obj>
```

Here we still have a few extended properties; `MinRunspaces`, `MaxRunspaces`, `PSThreadOptions`, `ApartmentState`, `HostInfo`, and `ApplicationArguments`. The values for each of these properties are different types with some being another complex object. Once again, this is not too bad and somewhat simple to parse. It becomes highly complex when dealing with lists/dicts of complex objects with nested complex objects. Just have a look at ERROR_RECORD example in the MS-PSRP documentation.

Finally if we were to look at the output from our example pipeline `Get-PSDrive -Name C`, we would get the following returned back to us as a `PIPELINE_OUTPUT` message;

```
<Obj RefId="0">
    <TN RefId="0">
        <T>System.Management.Automation.PSDriveInfo</T>
        <T>System.Object</T>
    </TN>
    <ToString>C</ToString>
    <Props>
        <S N="CurrentLocation">Users\vagrant\Documents</S>
        <S N="Name">C</S>
        <S N="Provider">Microsoft.PowerShell.Core\FileSystem</S>
        <S N="Root">C:\</S>
        <S N="Description">Windows 2016</S>
        <Nil N="MaximumSize"/>
        <Obj N="Credential" RefId="1">
            <TN RefId="1">
                <T>System.Management.Automation.PSCredential</T>
                <T>System.Object</T>
            </TN>
            <ToString>System.Management.Automation.PSCredential</ToString>
            <Props>
                <Nil N="UserName"/>
                <Nil N="Password"/>
            </Props>
        </Obj>
        <Nil N="DisplayRoot"/>
    </Props>
    <MS>
        <U64 N="Used">29512912896</U64>
        <U64 N="Free">12061024256</U64>
    </MS>
</Obj>
```

If we were running on a .NET language, the tools to deserialise this back into a `PSDriveInfo` object would be builtin but with pypsrp I had to try and implement this myself. What I ended up doing was having a predefined “known types” list which mapped known types to an existing Python object defined here. These known types are mostly types used within PSRP itself such as `PSThreadOptions`, `ErrorRecord`, `PSCredential` and more. For types that are not pre-defined, like `PSDriveInfo`, it would be encapsulated into a `GenericComplexObject`. This object is able to convert the raw CLIXML string into the following Python object properties;

- `types`: A list of types as defined by the `TN` element
- `to_string`: If the object contains a `ToString` element, this stores that value, using str(obj) would also return that value
- `adapted_properties` Any properties contained in the `Props` element are stored here, this is dict where the `N` element attribute value is the key
- `extended_properties`: Any properties contained in the `MS` element are stored here, the key is also the value of the `N` element attribute
- `property_sets`: Any other elements that are not in the `Props` or `MS` elements

If for any reason this failed, then the raw CLIXML string is returned instead.

When looking at the `PSDriveInfo` example, here is some ways of accessing each of the objects returned in Python;

```
assert output.types == ["System.Management.Automation.PSDriveInfo", "System.Object"]

# get the string representation <ToString>
assert str(output) == "C"

assert output.adapted_properties['CurrentLocation'] == r"Users\vagrant\Documents"

# PSCredentials is a known type so we can access the props based on the object definition
assert output.adapted_properties['Credential'].username is None

assert output.extended_properties['Used'] == 29512912896
```

I'm hoping the current implementation makes it somewhat easier to deal with objects but I can see there's always room for improvement. If worst comes to worst and you want to go back to a more stdio way of working and pipe the output as a string so PowerShell does all the heavy lifting for you. The best way of doing this is to run your whole command/script under `Invoke-Expression -Command "..." | Out-String Stream` like so.

```
from pypsrp.powershell import PowerShell, RunspacePool
from pypsrp.wsman import WSMan

wsman = WSMan("server2016.domain.local", username="vagrant", password="vagrant",
              cert_validation=False)

with RunspacePool(wsman) as pool:
    ps = PowerShell(pool)
    ps.add_cmdlet("Invoke-Expression").add_parameter("Command", "Get-PSDrive -Name C")
    ps.add_cmdlet("Out-String").add_parameter("Stream")
    ps.invoke()
    print("\n".join(ps.output))
```

This will produce the following output

```
Name           Used (GB)     Free (GB) Provider      Root
CurrentLocation
----           ---------     --------- --------      ----
---------------
C                  27.49         11.23 FileSystem    C:\
Users\vagrant\Documents
```

So in the end it is up to your how you want to handle objects, you can get PowerShell on the remote host to create the output string or you can manually parse the objects yourself and create the output in whatever format you desire.

Using PyPSRP

I've spoken a lot about PSRP and hope you have a better understanding of what goes on in the background. I've put in a lot of hours to get it all working and I hope some people find a use for it.

How to install

The first step to get this all working is actually installing pypsrp, the simplest way to do this is by running `pip install pypsrp`. This will download all the dependencies and get the package setup and ready to do. Out of the box you can do pretty much anything but authenticate with Kerberos or CredSSP authentication. If you wanted to add support for that, just run;

```
# Ubuntu Python 2
apt-get install gcc python-dev libkrb5-dev

# Ubuntu Python 3
apt-get install gcc python3-dev libkrb5-dev

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

# Fedora
dnf install gcc python-devel krb5-devel

pip install pypsrp[kerberos,credssp]
```

PSRP Examples

I'll first give you an example of how to run some code through the PSRP layer. The high level API is designed to wrap a lot of the work that goes on behind the scenes with Runspaces and Pipelines into a simple function that runs a script and returns the output. Here is a very simple example of how you can run a cmdlet like `New-Item` through this interface;

```
from pypsrp.client import Client

client = Client("server", username="user",
                password="password", ssl=False)

script = r"New-Item -Path C:\temp\folder -ItemType Directory -Verbose"
output, streams, had_errors = client.execute_ps(script)

print("HAD ERRORS: %s" % had_errors)
print("OUTPUT:\n%s" % output)
print("ERROR:\n%s" % "\n".join([str(s) for s in streams.error]))
print("DEBUG:\n%s" % "\n".join([str(s) for s in streams.debug]))
print("VERBOSE:\n%s" % "\n".join([str(s) for s in streams.verbose]))
```

We see instead of the standard `stdout`, `stderr`, and `rc` outputs, we get the following;

- `output`: A string that contains all the output records from the execution
- `streams`: A object that contains each of the streams (`error`, `verbose`, `debug`, `warning`, `information`) which has a list of each stream object created by the script
- `had_errors`: A boolean that indicates whether a terminating error was thrown, *note: this is different from a normal error*

The output of the above command is

```
(ansible-py36) jborean:~/dev/module-tester$ python pypsrp_psrp.py
HAD ERRORS: False
OUTPUT:


    Directory: C:\temp


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         8/14/2018   3:37 AM            folder


ERROR:

DEBUG:

VERBOSE:
Performing the operation "Create Directory" on target "Destination: C:\temp\folder".
```

If we were to run it one more time we would get

```
(ansible-py36) jborean:~/dev/module-tester$ python pypsrp_psrp.py
HAD ERRORS: False
OUTPUT:

ERROR:
An item with the specified name C:\temp\folder already exists.
DEBUG:

VERBOSE:
```

We can see that there is an error entry but `had_errors` is still `False`. This is because of the way `execute_ps` runs the script, only terminating errors, e.g. `throw` will set this to `True`.

Converting this to the low level API would look like

```
from pypsrp.powershell import PowerShell, RunspacePool
from pypsrp.wsman import WSMan

wsman = WSMan("server", username="user",
              password="password", ssl=False)

with RunspacePool(wsman) as pool:
    ps = PowerShell(pool)
    ps.add_script("New-Item -Path C:\\temp\\folder -ItemType Directory -Verbose")
    output = ps.invoke()

print("HAD ERRORS: %s" % ps.had_errors)
print("OUTPUT:\n%s" % "\n".join([str(s) for s in output]))
print("ERROR:\n%s" % "\n".join([str(s) for s in ps.streams.error]))
print("DEBUG:\n%s" % "\n".join([str(s) for s in ps.streams.debug]))
print("VERBOSE:\n%s" % "\n".join([str(s) for s in ps.streams.verbose]))`
```

```
(ansible-py36) jborean:~/dev/module-tester$ python pypsrp_psrp_low.py
HAD ERRORS: False
OUTPUT:
C:\temp\folder
ERROR:

DEBUG:

VERBOSE:
Performing the operation "Create Directory" on target "Destination: C:\temp\folder".
```

We can see that output is now a list of stream objects rather than a single string. You can always do what the example does and just cast each entry to to a string but the output will look a bit different to what you are used to on a native PowerShell console. Some of the extra features you can play with when using this lower level interface are;

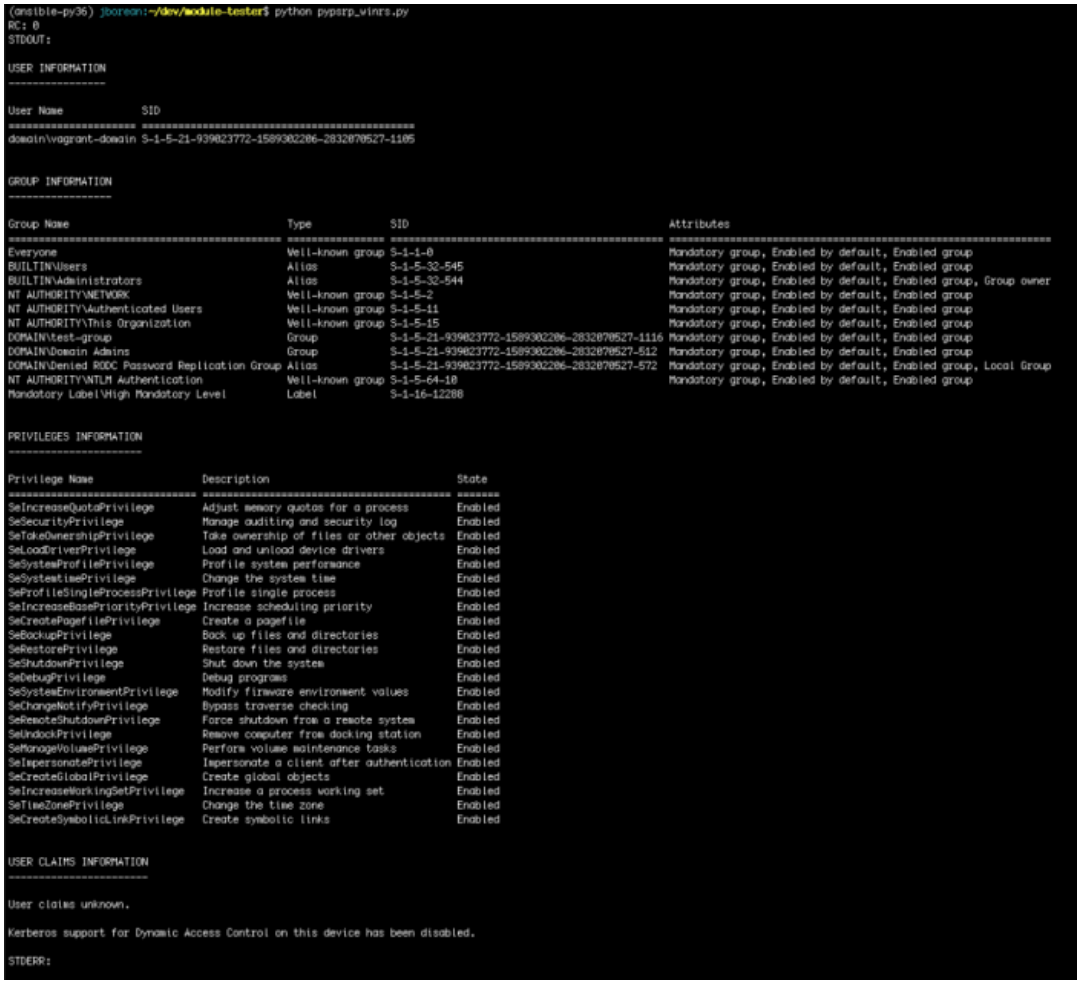
- You can specify the PSRP configuration name to connect to instead of using the default `Microsoft.PowerShell`
- You can run multiple Runspaces/Pipelines at the same time compared to just the one
- You can define a pseudo Host to handle host methods like WriteLine, Prompt, ReadLine and so on
- You can run the PowerShell script in the background without blocking, use the `begin_invoke()`, `poll_invoke()`, and `end_invoke()` methods for this
- Lots and lots more, this only scratches the surface.

The low level interface is designed to replicate the .NET API for dealing with Runspace Pools and Pipelines. See the RunspacePool Class and PowerShell Class for details on the methods.

WinRS Examples

I spoke at the start I wanted to make sure I implemented all the features currently present in pywinrm and that includes being able to run a command through WinRS. Like with the PSRP side, there is a high level implementation to make it easy for someone new to the library as well as a lower level interface if you need some of the more advanced features. Let's show an example of both.

```
from pypsrp.client import Client

client = Client("server", username="user",
                password="password", ssl=False)

stdout, stderr, rc = client.execute_cmd("whoami.exe /all")
print("RC: %d" % rc)
print("STDOUT:\n%s" % stdout)
print("STDERR:\n%s" % stderr)
```



Now comparing it to the lower level API;

```
from pypsrp.shell import Process, SignalCode, WinRS
from pypsrp.wsman import WSMan

wsman = WSMan("server", username="user", password="password",
              ssl=False)

with WinRS(wsman) as shell:
    process = Process(shell, "whoami.exe", ["/all"])
    process.invoke()
    process.signal(SignalCode.CTRL_C)

print("RC: %d" % process.rc)

# the stdout and stderr streams come back as bytes, this decodes them with the 437 codepage (default
on my Windows host)
print("STDOUT:\n%s" % process.stdout.decode('437'))
print("STDERR:\n%s" % process.stderr.decode('437'))
```

You can see we are manually creating the `Process` object with the executable and a list of argument(s), invoking that and sending the stop signal once finished. This is a lot more verbose than the other code but some of the things you can do with the low level interface are;

- You can run multiple commands in the same WinRS shell which saves some time
- The process object has a `begin_invoke()`, `poll_invoke()` and `end_invoke()` to effectively run the command in the background and not block Python until it is finished
- The `Process` object has a `send()` method to send bytes to the stdin pipe of the remote process
- Lots more configuration options around the WinRS shell and Process object, like environment, working directory, codepage, etc

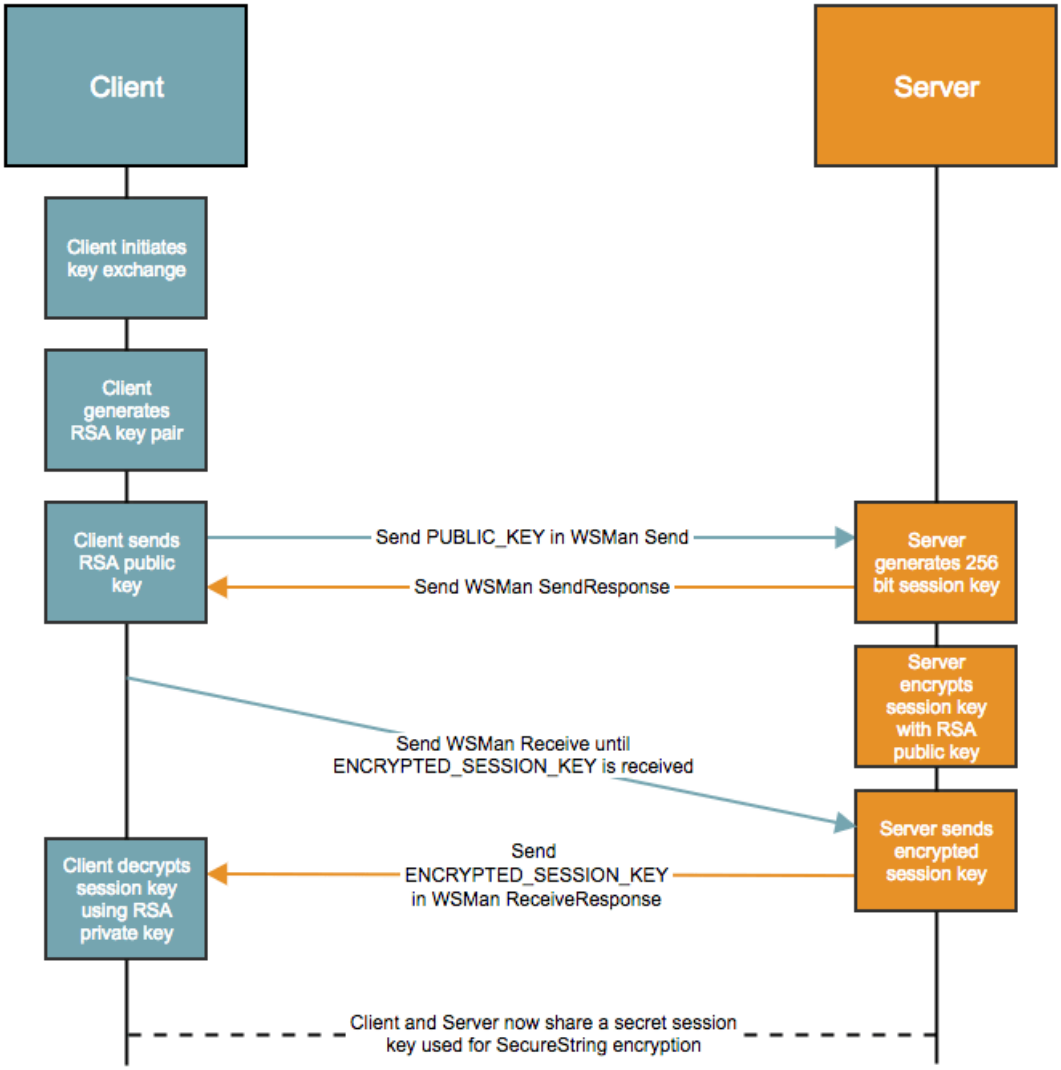
Interop with Secure Strings

This is one feature I thought was important enough to talk about a bit more and was probably one of the more complex ones to implement in Python. When creating a SecureString in PowerShell the string is encrypted using a key baked by DPAPI based on the current user's credentials. This effectively means only the user on the host that created the string can decrypt and get the plaintext of that value. When using plain WinRM, you loose the ability to create a secure string and send it over the wire to the remote host, any attempt to do so would involve manually encrypting and decrypting the string with some out of band mechanism.

PSRP offers an in built mechanism to serialise a string as a SecureString object and send that across the wire using AES256 encryption.

How it Works

Unlike WinRM message encryption which is dependent on either TLS to encrypt the entire transport payload or using the authentication context wrapping methods to encrypt the WSMan payload, PSRP uses another layer of encryption when dealing with SecureStrings. In the current protocols each key is unique per WSMan session and is based on the AES256 algorithm in CBC mode. Here is a basic process flow of what happens when the client starts the key exchange process;



Some things to note;

- The RSA key pair generated by the client, MUST never be reused
- PyPSRP will generate the key pair with a public exponent value of `65537` and a key size of `2048` bits
- The PUBLIC_KEY message contains a pre set header as defined at MS-PSRP 2.2.2.3 PUBLIC_KEY Message
- The server will generate the session key using CryptGenKey with;
  - `Algid`: CALG_AES_256
  - `dwFlags`: `0x0100000` (256 bit length) | `CRYPT_EXPORTABLE` | `CRYPT_CREATE_SALT`
- The generated key is exported with CryptExportKey with `hExpKey` set to the handle of the RSA public key sent by the client
- The encrypted session key is padded based on the RSAES-PKCS-v1_5 algorithm before sending to the client
- Due to the asymmetric nature of the RSA algorithm, only the host that generated the key (the server) and the holder of the RSA private key (the client) now know what the session key is
- When generating the AES256 CBC cipher, Windows defaults to having an IV of 16 bytes of `\x00`

Finally when it comes to serialising a string as a SecureString, we need to get the `UTF-16-LE` encoded bytes of the string, pad it iwth the PKCS7 algorithm and then encrypt that with the session key from the server.

This process may change sometime in the future, especially if PowerShell Core adds support for it, but for now this works on PowerShell 2.0 to 5.x.

Issues with Python Interop

The documentation around how this all works is pretty minimal so I found implementing this in Python was quite difficult. The issues I came across were;

- Getting the public key modulus in the format expected was difficult, cryptography's RSAPublicNumbers object contains the modulus as an int but we needed it as bytes. Being a really large number meant I couldn't use struct to do this for me
- The encrypted session key returned by the server had to have it's bytes reversed for it to work with Python cryptography, never really found out why but it needed to be done
- The docs do state that `RSAES-PKCS-v1_5` padding is used on the session key, but finding out how to implement that in Python cryptography to work with the Windows crypto libraries took a lot of trial and error
- The process to encrypt the secure strings MUST be done on the UTF-16-LE encoded bytes of the string, this is fine for Python 3 which defaults to unicode strings but Python 2's default text is already a byte string which can be problematic as people usually have UTF-8/ASCII bytes in their Python 2 strings
- PSRP would not reply with helpful error messages if the message format was incorrect, leading to lots of different trial and error attempts

Using SecureStrings in PyPSRP

Let's show an example of this in action.

```
from pypsrp.complex_objects import ObjectMeta, PSCredential
from pypsrp.powershell import PowerShell, RunspacePool
from pypsrp.wsman import WSMan

wsman = WSMan("server2016.domain.local", username="vagrant",
              password="vagrant", cert_validation=False)

with RunspacePool(wsman) as pool:
    pool.exchange_keys()

    secure_string = pool.serialize(u"My secret", ObjectMeta("SS"))
    ps_credential = PSCredential(username="Username", password="Password")

    ps = PowerShell(pool)
    # send the Python objects across and store as a variable in our Pipeline
    ps.add_cmdlet("Set-Variable").add_parameters({"Name": "secure_string", "Value": secure_string})
    ps.add_statement()
    ps.add_cmdlet("Set-Variable").add_parameters({"Name": "ps_credential", "Value": ps_credential})

    # assert it was serialized as a secure string and we can decrypt/return it back to us
    ps.add_statement().add_script('''
$sec_ptr = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($secure_string)
[Runtime.InteropServices.Marshal]::PtrToStringAuto($sec_ptr)

$sec_ptr = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($ps_credential.Password)
[Runtime.InteropServices.Marshal]::PtrToStringAuto($sec_ptr)
''')
    ps.invoke()

    assert ps.output[0] == u"My secret"
    assert ps.output[1] == u"Password"
```

In the script above we added the extra step to setup the encryption keys with `.exchange_keys()`. This is done after the RunspacePool is opened and before we go to serialise any SecureString objects. Once that's done, we can serialise any unicode string with the `pool.serialize()` method or any known complex object that uses SecureStrings like `PSCredential. Having a look at the actual PSRP objects sent in this exchange we can see the serialised form of both these variables;

```
<SS N="V">Zs+jIsTFR1jsAoq68slZIO3SFaKmOuFinKWOq89mXwk=</SS>

<Obj N="V" RefId="26">
    <TN RefId="4">
        <T>System.Management.Automation.PSCredential</T>
        <T>System.Object</T>
    </TN>
    <ToString>System.Management.Automation.PSCredential</ToString>
    <Props>
        <S N="UserName">Username</S>
        <SS N="Password">oqIE37i0OiC6Qg8TO9a931DlrqoEFaRxYWUQ/eja+5I=</SS>
    </Props>
</Obj>
```

Awesome, so anyone who manages to snoop over the wire will only see the encrypted value.

What's Next

So far I've created an Ansible connection plugin that uses pypsrp to speed up the execution a bit more. The gains are nothing dramatic but I found it saved around 45 seconds off a 5 minute playbook which is better than nothing :). If you wish to test it out, either wait until Ansible 2.7 is released or add the changes here to your install of Ansible. I want to look into setting it up as a persisted connection within Ansible to get even more of a performance gain but came across some problems with the current persisted connection framework within Ansible that needed to be solved first before moving ahead.

As for pypsrp itself, I am hoping to get the following working at some point in the future;

- Support for SSH as a transport mechanism
- Create an interactive console so you can connect to a host and run commands interactively
- Create a readthedocs site to help people use the library

SSH is probably the biggest feature I would like to implement but the only documentation I can find around this is the code itself in the PowerShell repo on GitHub. It's not impossible to get working but without some reference docs to explain some of the complex steps in the code it will take a bit more time than normal.

What I've Learnt

This project has been quite an illuminating one for me. I've always been interested in getting PSRP working with Python ever since I read Matt Wrock's article about it. The allure of a faster API than what WinRS offered was definitely a big reason for this but ultimately I found the performance gains a bit disappointing. I found that yes it was faster than WinRS, due less overhead in creating each PowerShell process, but some of the other benefits like file transfer speeds weren't actually realised.

I think that the complexities involved with PSRP, like serialisation of objects, probably outweigh the advantages for someone wanting to just run a PowerShell command which is why not many third party libraries have embraced PSRP over vanilla WinRM. In saying that there are definitely some advantages of using PSRP that make this library a good option for some. These features would be things like;

- Connecting to a custom configuration endpoint, used in tools like Just Enough Administration or things like Office 365 management consoles
- You want to deal with PowerShell objects directly instead of parsing text
- You want to utilise SecureString remotely to add extra confidentiality to the data being sent
- You want finer control over executing PowerShell commands and really enjoy the .NET Runspace interface

At the end of the day, I learnt a hell of a lot about the PowerShell/WinRM ecosystem that I didn't know before which is what I call a success. If a by product of this work means other people can benefit from it, then that's just icing on the cake.