

Wacky WSMAN on Linux

 bloggingforlogging.com/2020/08/21/wacky-wsman-on-linux

August 21, 2020

A few years ago I jumped from doing all my dev work on Windows to Linux. This migration has had a few challenges but one of the things I struggled with initially was the lack of native tooling that can be used to easily and seamlessly interact with other Microsoft products. I've typically found some great open source products that fill in that gap quite nicely, but one area that has always felt a bit lacking was support for PSRemoting. There are some fantastic WSMAN/WinRM client libraries out there which are pretty close to feature parity with WSMAN on Windows but typically lack 2 main features, being interactive and directly integrated in PowerShell.

I thought that with Microsoft joining the fray and creating an open source and cross platform release of PowerShell that this gap would be filled nicely. My main desire was being able to do `Enter-PSSession` to target some of my dev Windows boxes just like I could do from a Windows workstation. Much to my disappointment, this has not lived up to my expectations. In this post, I will talk about my journey in trying to get WSMAN as a client working on PowerShell from a Linux host, and what I've learnt along the way.

Current Landscape

PowerShell Remoting (PSRemoting) is a protocol that PowerShell uses to execute PowerShell on another PowerShell instance. This can be another PowerShell process on the same host, or even another host altogether. PSRemoting works by encapsulating .NET objects, serialised in CLIXML, inside another remoting protocol like WSMAN or SSH. If you are interested in learning more about PSRemoting, my [PowerShell Remoting on Python](#) post goes into more detail.

Historically WSMAN was the sole transport option due to it being the de facto protocol used by Windows for remote management. Over time more protocols have been added, like SSH, to take advantage of PowerShell's move to open source and the new target audience that comes with that. Ultimately I feel like SSH is the superior transport option over WSMAN but there are a few times where WSMAN is still needed like:

- Targeting Windows hosts
 - WSMAN works out of the box
 - SSH requires the SSH server and PowerShell 6+ to be installed
- The Win32-OpenSSH fork that comes with Windows does not support Kerberos authentication
 - With Kerberos authentication you can easily solve the double hop limitation without sending your actual password across the wire
 - A newer release will add support for this, negating this requirement somewhat

- Just Enough Administration (JEA) only works on WSMAN
There is talk about adding support for JEA on SSH but nothing it out as of yet
- Exchange Online and Exchange on premise only work with WSMAN, no SSH at all

This wouldn't be too much of an issue if the WSMAN client that ships with PowerShell on Linux and macOS wasn't so horribly broken and outdated. After quickly browsing the PowerShell repo for issues related to WSMAN on Linux and macOS, a common pattern emerges:

Adding more salt to the wound is the PowerShell team's stance of WSMAN in PowerShell is that it is deprecated and there are no plans on trying to fix any of the existing bugs that people are encountering. There is an issue that states OMI is deprecated and there are no plans for any future bugfixes or features.

OMI will not fix PowerShell bugs meaning PowerShell cannot offer any level of support
Known and unknown security issues will not be fixed
OMI does not load OpenSSL correctly causing segfaults

Don't get me wrong, I understand their stance on this issue. They are a small team and need to prioritise their focus based on things that bring benefits to most users. There are some scattered comments stating that there are plans on creating a new WSMAN client in .NET for PowerShell that fix a lot of these problems but so far these just seem to be plans with no concrete work.

So instead I thought "this is all open source, why don't I fix the bugs and contribute it back?". I fixed a few of the major issues like getting things to compile on some newer versions of macOS and some of the bigger GSSAPI problems. I opened PR 1 and PR 2 on the OMI repo but unfortunately these PRs were subsequently rejected by whatever team in Microsoft manages OMI, not the same as the PowerShell team, mostly on the grounds that "support" for PowerShell was dropped in 2018 and "new features" need to be internally prioritised. So ultimately even if someone wanted to try and improve OMI, any changes will be stonewalled when trying to get those fixes into the actual codebase.

So we are stuck in a situation where the library PowerShell uses has known issues and problems due to its reliance on older libraries, among other problems, and no way to get fixes for those issues into the codebase. You might be thinking, big deal, WSMAN is only used for legacy Windows products. Just use your existing Windows hosts to manage those. Unfortunately even new products in development today, like the Exchange Online V2 PowerShell Module are still based on WSMAN only. If you wanted to use this new module you are stuck with running it on Windows only. While this may not be a feature that's massively in demand, I personally find the whole situation unfortunate and just leads to more barriers of entry for the adoption of PowerShell on Linux.

In the end, I decided to just continue on with fixing the bugs in the existing client and take advantage of the code being open sourced. I forked the OMI repo and starting fixing any of the bugs I encountered there. I'll talk more about using this fork later on, first I want to

talk about these bugs and what I did to fix them. I've found that the existing bugs in OMI can be split into 3 different categories:

1. Dynamic linking library problems
2. GSSAPI (authentication and encryption) problems
3. Client behaviour problems

Dynamic Linking

OMI is written in C and is compiled to various shared library objects for the different functionality that it offers. PowerShell is only interested in using the `mi` library that is produced as that is where the WSMAN client code is stored. If you look into your PowerShell directory on your Linux/macOS host you will see two different `lib<name>` libraries that affect WSMAN:

- `psrpclient`
- `mi`

The `psrpclient` is basically a compatibility layer to translate the public interface that `mi` exposes to the same interface as the Win32 WSMAN API calls. This compat layer mostly just allows PowerShell to call one interface for any WSMAN calls across the various platforms. We can see the actual PInvoke definitions in the [PowerShell codebase](#)

```
#if !UNIX
    internal const string WSMANClientApiDll = @"WsmSvc.dll";
    internal const string WSMANProviderApiDll = @"WsmSvc.dll";
#else
    internal const string WSMANClientApiDll = @"libpsrpclient";
    internal const string WSMANProviderApiDll = @"libpsrpomiprov";
#endif

[DllImport(WSMANNativeApi.WSMANClientApiDll, SetLastError = false, CharSet
= CharSet.Unicode)]
    internal static extern int WSMANInitialize(int flags,
        [In, Out] ref IntPtr wsManAPIHandle);
```

In this case it is calling the `WSMANInitialize` function, along with many others, that is exposed in `WsmSvc.dll` on Windows and `libpsrpclient` on other platforms. We can see in the [psrpclient repo](#) that `WSMANInitialize` is mostly a shim for `MI_Application_InitializeV1` which is the public interface exposed by `mi`.

```

MI_EXPORT MI_Uint32 WINAPI WSMANInitialize(
    MI_Uint32 flags,
    _Out_ WSMAN_API_HANDLE *apiHandle
)
{
    MI_Result miResult;

    _GetLogOptionsFromConfigFile(SHELL_LOGGING_FILE);

    LogFunctionStart("WSMANInitialize");

    (*apiHandle) = calloc(1, sizeof(struct WSMAN_API));
    if (*apiHandle == NULL)
        return MI_RESULT_SERVER_LIMITS_EXCEEDED;

    miResult = MI_Application_InitializeV1(0, NULL, NULL, &(*apiHandle)-
>application);
    if (miResult != MI_RESULT_OK)
    {
        free(*apiHandle);
        *apiHandle = NULL;
    }
    LogFunctionEnd("WSMANInitialize", miResult);
    return miResult;
}

```

So for PowerShell to be able to create WSMAN instances on Linux it needs to be able to load `psrpclient` which in turn relies on `mi` being available. If any one of those libraries, or any of their linked dependencies are not present then PowerShell will fail with the following error:

```

Enter-PSSession: This parameter set requires WSMAN, and no supported WSMAN
client library was found. WSMAN is either not installed or unavailable for this
system.

```

It's not very helpful in telling you what actually failed to load as it could be either `psrpclient`, `mi`, or one of their deps not being available. To investigate this a bit more, you can run the following to list the dynamic linked libraries of both libraries

```
PWSHDIR="$( dirname "$( readlink "$( which pwsh )" )" )"
```

```

# On macOS
otool -L "${PWSHDIR}/libpsrpclient.dylib"
otool -L "${PWSHDIR}/libmi.dylib"

```

```

# On Linux
ldd "${PWSHDIR}/libpsrpclient.so"
ldd "${PWSHDIR}/libmi.so"

```

On macOS we can see that the shipped `libmi.dylib` is linked to a custom OpenSSL path.

```
/usr/local/microsoft/powershell/7/libmi.dylib:  
  @rpath/libmi.dylib (compatibility version 0.0.0, current version 0.0.0)  
  /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version  
1238.60.2)  
  /usr/lib/libpam.2.dylib (compatibility version 3.0.0, current version 3.0.0)  
  /usr/local/opt/openssl/lib/libssl.1.0.0.dylib (compatibility version 1.0.0,  
current version 1.0.0)  
  /usr/local/opt/openssl/lib/libcrypto.1.0.0.dylib (compatibility version 1.0.0,  
current version 1.0.0)  
  /usr/lib/libz.1.dylib (compatibility version 1.0.0, current version 1.2.8)
```

There are 2 problems here:

1. It's linked to a hardcoded location `/usr/local/opt/lib/libssl.1.0.0.dylib` that isn't a system library
2. It's linked to OpenSSL 1.0.0 which is an old, outdated, and probably insecure library

So when you try and load `mi` on macOS it will most likely fail as you more than likely do not have the OpenSSL 1.0.0 libs at `/usr/local/opt/openssl/lib`. You can't even install this version of OpenSSL using `brew` anymore unless you find a custom formula. Even if you use a custom formula, or manually compile your own OpenSSL to this location, it's not going to change the fact the OpenSSL 1.0.0 is not an ideal library to use for creating secure backends due to its age. So to fix this problem on macOS we need to make sure we compile `mi` against a newer version of OpenSSL and hopefully with one that is easily available on most systems.

For Linux, I tested with CentOS 7, we can see that we have a similar setup.

```
[user@hostname /home/user]# ldd "${PWSHDIR}/libmi.so"
ldd: warning: you do not have execution permission for
`/opt/microsoft/powershell/7/libmi.so'
linux-vdso.so.1 => (0x00007fffd2f1eb000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fe4775bf000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007fe4773bb000)
libpam.so.0 => /lib64/libpam.so.0 (0x00007fe4771ac000)
libssl.so.1.0.0 => /opt/microsoft/powershell/7/libssl.so.1.0.0
(0x00007fe476f3a000)
libcrypto.so.1.0.0 => /opt/microsoft/powershell/7/libcrypto.so.1.0.0
(0x00007fe476ad7000)
libc.so.6 => /lib64/libc.so.6 (0x00007fe476709000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe4777db000)
libaudit.so.1 => /lib64/libaudit.so.1 (0x00007fe4764e0000)
libgssapi_krb5.so.2 => /lib64/libgssapi_krb5.so.2 (0x00007fe476293000)
libkrb5.so.3 => /lib64/libkrb5.so.3 (0x00007fe475faa000)
libcom_err.so.2 => /lib64/libcom_err.so.2 (0x00007fe475da6000)
libk5crypto.so.3 => /lib64/libk5crypto.so.3 (0x00007fe475b73000)
libz.so.1 => /lib64/libz.so.1 (0x00007fe47595d000)
libcap-ng.so.0 => /lib64/libcap-ng.so.0 (0x00007fe475757000)
libkrb5support.so.0 => /lib64/libkrb5support.so.0 (0x00007fe475547000)
libkeyutils.so.1 => /lib64/libkeyutils.so.1 (0x00007fe475343000)
libresolv.so.2 => /lib64/libresolv.so.2 (0x00007fe475129000)
libselinux.so.1 => /lib64/libselinux.so.1 (0x00007fe474f02000)
libpcre.so.1 => /lib64/libpcre.so.1 (0x00007fe474ca0000)

[user@hostname /home/user]# ls -al /opt/microsoft/powershell/7/libssl.so.1.0.0
lrwxrwxrwx 1 root root 19 Aug 19 11:16 /opt/microsoft/powershell/7/libssl.so.1.0.0
-> /lib64/libssl.so.10
```

It is still linked to OpenSSL 1.0.0 but now it's through a symlink to [/lib64/libssl.so.10](#). While it is still bad to use OpenSSL 1.0.0 if you can avoid it. It's still less difficult to source OpenSSL 1.0.0 on most Linux distributions. As time goes on this will become more of an issue on Linux as newer distributions drop OpenSSL 1.0.0 altogether so it's something to keep in mind.

Luckily the underlying OMI repo does actually compile and work against OpenSSL 1.1.0, we just need to make sure the compiled library we use was built against OpenSSL 1.1.0. By recompiling the code as is, we automatically solve this problem and the build process automatically selects the OpenSSL library that's already installed and in the [PATH](#) of the host.

GSSAPI

So once we solve the linking problem and have gotten PowerShell to load [mi](#), we move onto the next major problem with the library, the authentication process. The WSMAN protocol supports the following authentication protocols:

- Basic
- Certificate
- Negotiate – Kerberos with an NTLM fallback

- Kerberos – with no NTLM fallback
- CredSSP

If you want to learn more about how this works in WSMAN, you can have a read of my [Demystifying WinRM](#) post. WSMAN on Windows supports all these protocols out of the box but `mi` is only designed to work with Basic, Negotiate, and pure Kerberos authentication. Even if Certificate and CredSSP auth was implemented in the future in `mi`, there is a [hardcoded check](#) in the `psrpcclient` library that returns an error if another authentication option was specified.

Typically `Basic` authentication is used for connections to Exchange Online and is the simplest protocol overall. Negotiate/Kerberos is used for actual Windows endpoints and is required if you are authenticating with a domain account. Windows users are spoiled with a nice security provider called `SSPI` which does a wonderful job of abstracting all the complexities of Negotiate authentication. It is practically invisible to the end user and essentially just works. If they want to access a file on a file share, Windows will typically automatically authenticate using the user's credentials without any further prompts.

On Linux the equivalent to SSPI is something called GSSAPI. GSSAPI typically comes in 2 different implementations, one called [MIT krb5](#) and another called [Heimdal](#). Typically on Linux you would find MIT whereas BSD based distributions use Heimdal, macOS uses a forked version of Heimdal with some Apple specific changes made to it. While GSSAPI can provide both Kerberos and NTLM authentication it is not included out of the box, requires domain specific config files to be created, DNS working properly, and probably most importantly it doesn't normally integrate with your logon account. While these problems can be an obstacle for users who are new to GSSAPI on Linux, they are not insurmountable and once solved you can have GSSAPI act in a similar way to SSPI on your Linux host.

When trying to figure out why I was getting `MI_RESULT_ACCESS_DENIED` errors on Linux I came across a the following problems with how authentication was implemented in `mi`:

- There is a hardcoded password length of 1024 bytes
 - This is a problem with modern auth for Exchange Online
 - Modern auth uses JSON Web Tokens (JWT) which can exceed this length causing a failure
 - The limitation has been increased to 8 KiB
- OMI tries to import the wrong symbols when targeting Heimdal
 - This breaks Negotiate authentication completely on macOS
 - By using the correct symbol names, GSSAPI will load properly and can be used on macOS

- OMI constructs the Service Principal Name (SPN) in a very strict fashion
 - This makes Kerberos auth even harder to be used on Linux
 - When Kerberos fails, Negotiate auth is designed to fallback to NTLM
 - NTLM is not provided by MIT krb5 and is another optional package to install
 - Now the SPN is constructed by passing in the name type `GSS_C_NT_HOSTBASED_SERVICE` with the value `http@<hostname>`
- No support for using an implicit Kerberos credential, you always have to provide explicit credentials
 - The hardcoded check that makes sure a username was set has now been removed
- The `GSS_C_DELEG_POLICY_FLAG` was not in the `req_flags`
 - This meant that Kerberos delegation was never enabled even when the SPN was trusted for delegation
 - By adding that flag, the credential will be delegated if the SPN is trusted for delegation
- OMI resolves the target hostname to the Fully Qualified Domain Name (FQDN) of the target
 - This invalidates the Kerberos server authentication model
 - The code even causes a segfault if the hostname was not resolvable
 - This whole process was removed due to it breaking the security model for Kerberos

Essentially this means that `mi` as provided by PowerShell works only in a very specific setup, namely on Linux where Kerberos is configured in a very specific way. Fixing these problems were relatively simple and the end result is that modern auth is now working and Negotiate/Kerberos auth works in more situations than before. The only remaining holdout is supporting the NTLM fallback on macOS when connecting over HTTP. There is a bug in the macOS' GSSAPI implementation that causes an interop failure and this issue sits in a place that I cannot touch. Luckily NTLM auth is old and insecure and you should really get Kerberos working. If you really want to use NTLM auth on macOS you are stuck with connecting over HTTPS with `-UseSSL`.

Client Behaviour

So now that we've been able to authenticate with our servers over WSMAN the last remaining issue is dealing with some invalid client behaviour assumptions. The biggest problem I encountered here was dealing with the message encryption payload that WSMAN uses when connecting over HTTP. Message encryption in WSMAN encrypts the raw WSMAN payload and encodes it as a MIME multipart message. This MIME payload follows the format

```
--EncryptedBoundary
Content-Type: application/HTTP-SPNEGO-session-encrypted
OriginalContent: type=application/soap+xml;charset=UTF-8;Length=<plaintext length>
--EncryptedBoundary
Content-Type: application/octet-stream
<header length><header><encrypted payload>--EncryptedBoundary--\r\n
```


Some of these values depend on the authentication method chosen or how the client actually formats the data but the basic structure stays the same. On premise Exchange hosts changes the boundary to -- EncryptedBoundary (with the space) which is technically against the spec and `mi` was never able to handle that change breaking connections on those endpoints.

The decryption code in OMI was very temperamental in whether it worked or not. I was able to get it working from my main dev host but as soon as I was testing on a container host it was failing with the very unhelpful error:

```
MI_RESULT_FAILED For more information, see the
about_Remote_Troubleshooting Help topic.
```

When stepping through the code I found that the `HttpClient_DecryptData` function was failing to find the encrypted payload in the MIME data causing that failure. To solve this issue I decided to refactor the MIME parser logic in this function and came up with a solution that works everytime I run it and even handles the -- Encrypted Boundary setup that on premise Exchange endpoints send back.

Solution

So I've identified the bugs and have fixed them, how do I share the work I've done for others to use. I eventually decided on creating a [fork](#) of the OMI repository and contributing my changes back there. This repo had to satisfy the following criteria I set out for myself:

- Make it easy to merge any upstream changes, if any are done in the future
- Document the reasons for the fork and what was changed
- Provide an easy way to build the library for various Linux distributions
- Provide a compiled "release" version for those distributions for easier consumption
- Actually test out the changes with PowerShell targeting both Windows endpoints and Exchange Online
- Add CI integration to run on every change

I've mostly succeeded in achieving these goals, any PR will automatically be run in Azure Pipelines to make sure the code will still build for the distributions. While I can't run the actual integration tests in CI I did create some scripts that will set up an environment and test out the various distributions with the libraries built in CI. The [tests](#) that are run just cover a basic set of scenarios and ensure the code can connect using Kerberos and optionally with Exchange Online. With this setup I can make a change in a PR then subsequently check if it still works when running from PowerShell itself.

Building

If you are wanting to build the code, you can either compile it manually just like you would with the upstream OMI repo with the following:

```
cd Unix
./configure --outputdirname=build --prefix=/opt/omi
make
```

You can then find the `libmi` file in `Unix/build/lib/libmi.so`. Building the code is quite simple, the hardest part is getting all the dependencies required to build OMI. I decided to write a Python script `build.py` that reads the metadata for a “known” distribution and then generate the build code for you. You can even use `build.py` to build the code in a Docker container for the distribution of your choice and not pollute your local environment. For example if you wanted to compile the code for CentOS 8 in an ephemeral Docker container, you can run the following:

```
./build.py centos8 --docker
```

The `build.py` script will read the `centos8.json` distribution meta file and create a bash script on the fly that will handle the deps for you. If you use the `--output-script` argument, the process will output the bash script that will build `mi` for you.

```

$> /build.py centos8 --output-script
#!/usr/bin/env bash

set -o pipefail -eu
echo ""
echo "-----"
echo "| Installing build pre-requisite packages |"
echo "-----"
echo ""
yum install -y -q \
    bind-utils \
    gcc \
    gcc-c++ \
    krb5-devel \
    make \
    openssl \
    openssl-devel \
    pam-devel \
    redhat-lsb-core \
    rpm-build \
    rpm-devel \
    which

echo ""
echo "-----"
echo "|           Clearing build folder           |"
echo "-----"
echo ""
if [ -d "build-centos8" ]; then
    rm -rf "build-centos8"
fi

echo ""
echo "-----"
echo "|           Running configure           |"
echo "-----"
echo ""
./configure \
    --outputdirname="build-centos8" \
    --prefix="/opt/omi"

echo ""
echo "-----"
echo "|           Running make           |"
echo "-----"
echo ""
make

```

When you run the `build.py` script, the compiled library is stored at `Unix/build-{distribution}/lib/libmi.so` (or `libmi.dylib` for macOS). If one of your favourite distributions is missing you can add your own `distribution_meta/{distribution}.json` file with the relevant details. I'm happy to add any more distributions to the list of ones built in CI.

Installation

If you haven't compiled it yourself you can even get a prebuilt copy from the [releases page](#). Make sure you select the correct distribution you want to use it on. Now that you've gotten a copy of the `libmi` file, you simply need to copy it into the PowerShell directory. I recommend you copy the existing file in case you need to revert back to what was shipped with PowerShell at some point in the future.

```
PWSHDIR="$( dirname "$( readlink "$( which pwsh )" )" )"
LIBMI_NAME="$( basename "$( ls "${PWSHDIR}/libmi.*" | head -n1 | awk '{print $1;}' )" )"
)" )"
```

Will probably need sudo for these 2 commands

```
cp "${PWSHDIR}/${LIBMI_NAME}" "${PWSHDIR}/${LIBMI_NAME}.bak"
cp "Unix/build-{distribution}/lib/${LIBMI_NAME}" "${PWSHDIR}/"
```

Once copied, any new PowerShell processes will now load the new library unlocking WSMAN on Linux to it's close to full potential. If you are wanting to make sure you have done this correctly, simply run the [Get-OmiVersion.ps1](#) script that is in the repo.

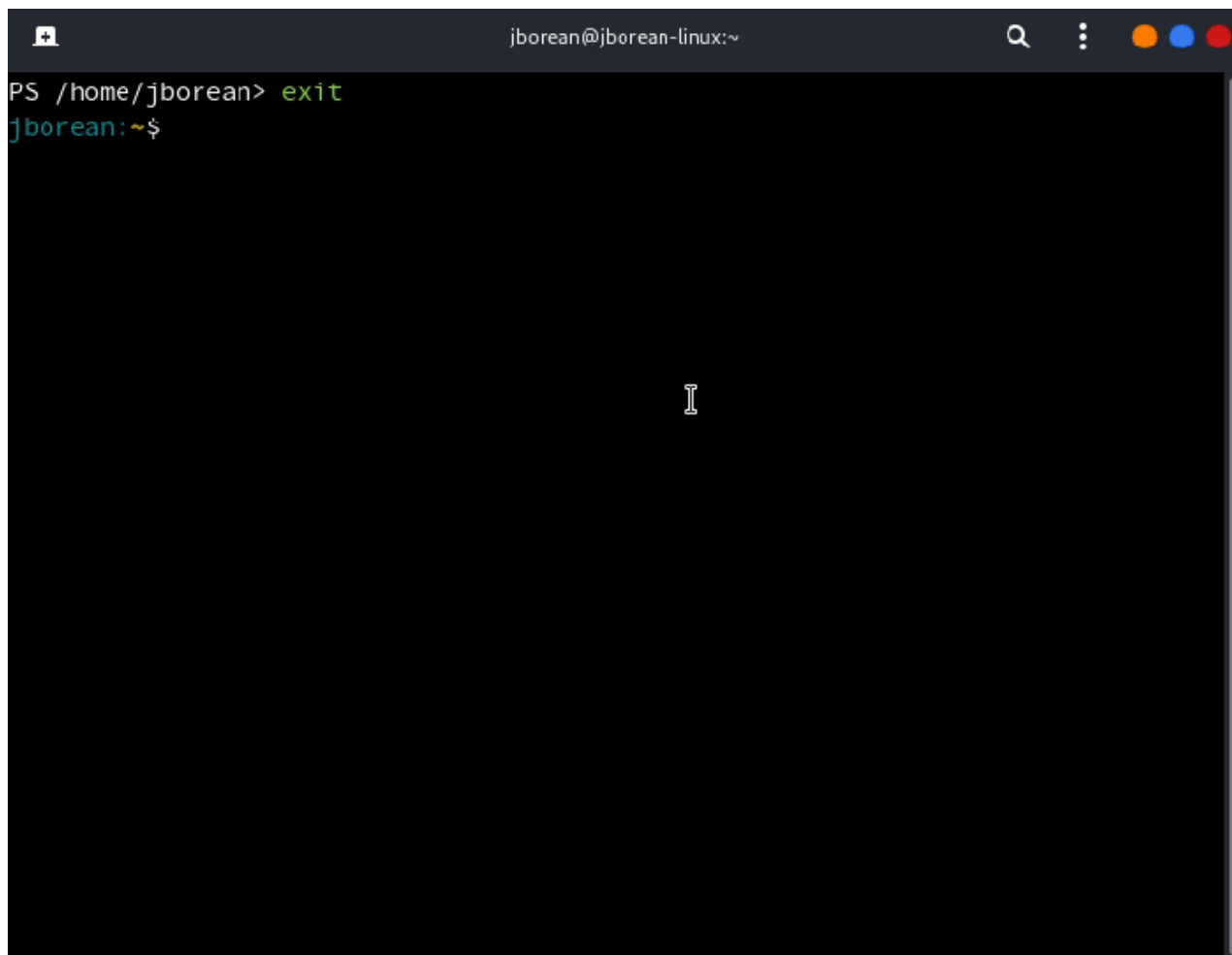
```
$> pwsh -File tools/Get-OmiVersion.ps1
```

Major	Minor	Build	Revision
-----	-----	-----	-----
1	0	1	0

If that fails then you are still using the builtin version without any of the changes in the fork.

In Action

So now that I've got the new `mi` library installed it's time to use it in PowerShell. I don't want to bore you with the details so here is it in action.

A terminal window with a dark background. The title bar shows 'jborean@jborean-linux:~'. The prompt is 'PS /home/jborean>'. The user has entered 'exit' and the prompt has changed to 'jborean:~\$'. A cursor is visible on the line below the prompt.

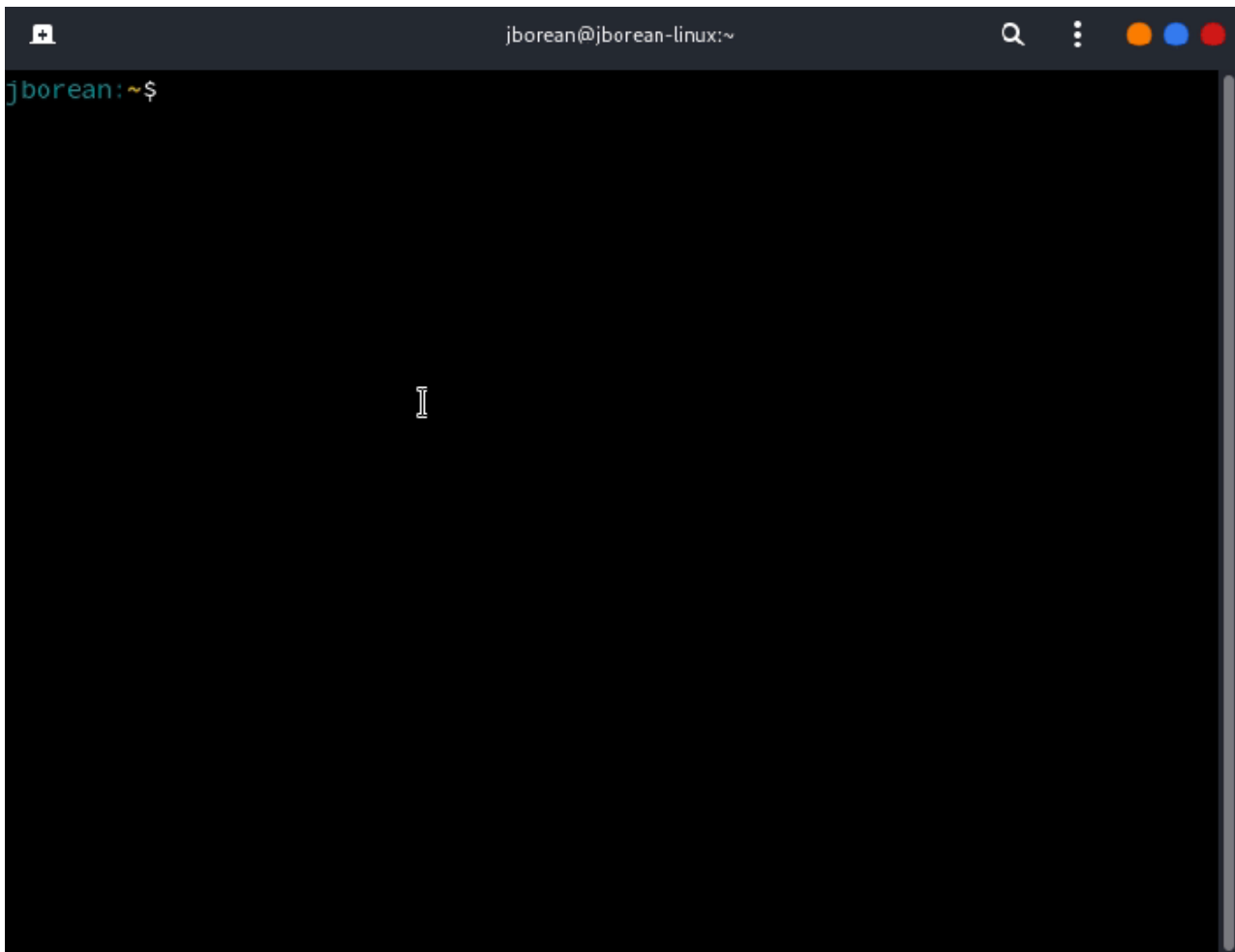
```
PS /home/jborean> exit
jborean:~$
```

In this example I am connecting using explicit credentials and have specified the `Negotiate` protocol. The WSMAN client will attempt to get a Kerberos ticket for those credentials and then start the Kerberos authentication process for SPN `HTTP/server2019.domain.local`. If any of those steps fail it will automatically fallback to using NTLM for the authentication exchange if the NTLM library is installed, otherwise it will fail. I could have also set `-Authentication Kerberos` instead of `Negotiate` if I wanted to ensure only Kerberos was used without the NTLM fallback. This could be ideal in your environments as NTLM is an old protocol and should be avoided where possible.

For example if I tried to connect using an IP address and Kerberos with `Enter-PSSession 192.168.56.15 -Authentication Kerberos -Credential $cred`, it will fail to find the target SPN in the domain.

Enter-PSSession: Connecting to remote server 192.168.56.15 failed with the following error message : Authorization failed Unspecified GSS failure. Minor code may provide more information Server not found in Kerberos database For more information, see the about_Remote_Troubleshooting Help topic

Finally, a new addition in my `mi` fork is the ability to utilise the existing Kerberos ticket cache and credential delegation. If you've already gotten a Kerberos ticket for a user account using something like `kinit`, you can omit the credential altogether when creating your PSRemoting session like so.



You can even bypass the `kinit` step yourself and automatically get a Kerberos ticket as part of the login process for your Linux account using the `pam-krb5` package. In this last example I also made sure I requested a forwardable ticket using the `-f` flag to `kinit`. When you have a forwardable ticket and the SPN you are targeting is set as trusted for delegation in your domain then the PSRemoting session will be able to delegate your credentials. This means my credentials were delegated to the remote process allowing me to re-authenticate as that user for any outbound connections in that PSRemoting session. You can't do that with SSH public key authentication :).

While you still cannot use the new Exchange Online v2 Modules you can still use something like New-EXOPSSession to connect to Exchange Online using modern auth. With that script you can import that Exchange Online PSSession and use many of the cmdlets to manage your Exchange instance.

Limitations

While the changes I've made in the fork solve a lot of the limitations with WSMAN on Linux, there are a few problems that cannot be reasonably fixed. Some of the limitations are:

- Basic auth over HTTP will always be disabled
 - There is a hard coded check in PowerShell that disables this
 - Honestly this is a good thing, using Basic auth over HTTP has no encryption so everything would be in plaintext
- HTTPS connections have no certificate verification, reducing the effectiveness the protocol brings
 - This is another hard coded check in PowerShell
 - Even if cert verification was implemented in OMI we cannot remove that check in PowerShell
 - This is unfortunate but also understandable, better to make people explicitly opt into disabling cert verification than think it is working
 - Edit: This has been fixed since v1.2.0 of the fork release where cert verification is now enabled by default
 - Edit2: Since the v2.0.0 release when used in conjunction with the PowerShell 7.2.0 release, it is no longer required to set the skip check options
- No CredSSP authentication

While you could add this to `mi` there is another check in `psrpclient` that fails when `-Authentication` is not `Basic`, `Negotiate`, or `Kerberos`
- You always need to set `-Authentication Basic|Negotiate|Kerberos` unlike Windows where omitting the parameter uses `Negotiate` auth
 - Due to the same check above, only those 3 will work
 - Edit: Since the v2.0.0 release, `libpsrpclient` has also been forked and changed to fix this problem, now the default auth is `Negotiate` removing the need to set this for default scenarios.
- NTLM auth on macOS only works over HTTPS
 - This is due to a problem in macOS' GSSAPI implementation which doesn't implement NTLM wrapping when used in SPNEGO properly
 - Once again nothing we can do about this

So even if I implement the features in OMI, PowerShell or `psrpclient` will still fail due to those hardcoded checks.

Future

The MVP that I wanted out of this whole process was a library I can use to connect over WSMAN to Windows targets and to Exchange Online. I think what I have right now meets that requirements but there is always room to improve things in the future. Some of the things that I think would be nice to add in the future are

- More distributions, maybe even a “universal” release like the one that OMI provides
- Improve the error messages for some known problems, right now failures can be quite vague in the error messages it reports
- Improve the logging situation, right now it requires a config file at `/opt/omi/etc/omicli.conf` to configure the logging details

If you decide to try out these changes, I'm happy to help as best as I can with any issues you find. Feel free to [open a new issue](#) but keep in mind this is work I do in my spare time. I cannot guarantee I will look at the issue in a timely fashion or even come up with a fix.

As for trying to get these changes included in PowerShell itself, I doubt there would be any appetite in the PowerShell team to do this. There is no guarantee of support and the general consensus is that WSMAN is deprecated on Linux and what it ships with will be it. If, in the future, they would like to pursue this further and look into getting these changes in their version of `mi` then I'm more than happy to work together to achieve that goal.

Edit: Since publishing this blog and officially releasing these changes there's been a few changes and additions made to my fork. These changes are

- The forked libraries have been published under the `PSWSMan` module on the PowerShell Gallery
 - People wishing to use my fork can install this module and run (as root) `Install-WSMan` to install my changes
- HTTPS cert validation is enabled by default
 - To disable cert validation you can use the `Disable-WSManCertVerification` cmdlet included with `PSWSMan`
 - When PowerShell 7.2.0 is released, you will also be able to disable cert verification per session using the `-SessionOption` (`New-PSSessionOption -SkipCACheck -SkipCNCheck`) options
- More distributions like Arch Linux and Alpine have been added
 - With `libpsrpcclient` being shipped with `PSWSMan` it will now be easier to add more distributions not supported by PowerShell directly