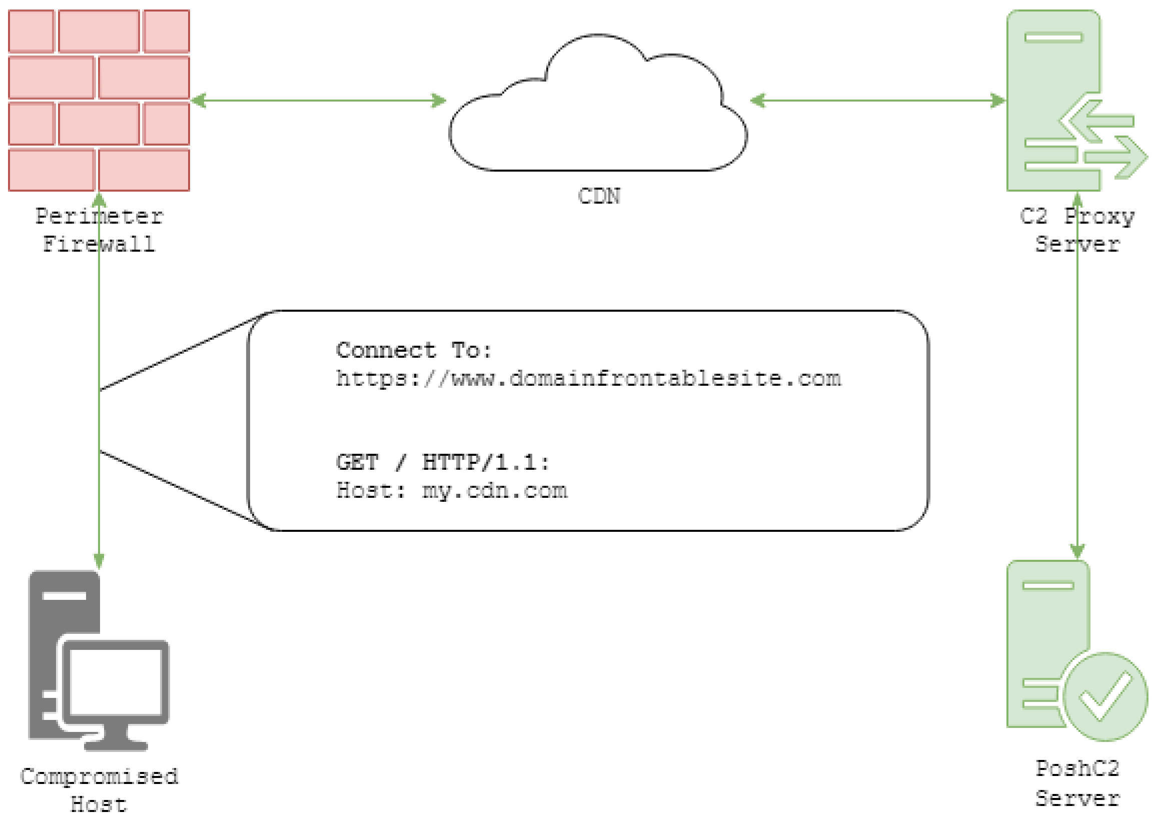
# Detecting PoshC2 – Indicators of Compromise

labs.nettitude.com/blog/detecting-poshc2-indicators-of-compromise/

Rob Bone

June 17, 2020




As a counterpart to the release of PoshC2 version 6.0 we are providing a list of some of its Indicators of Compromise (IoCs), particularly as used out-of-the-box, as well as some other effective methods for detecting it in your environment.

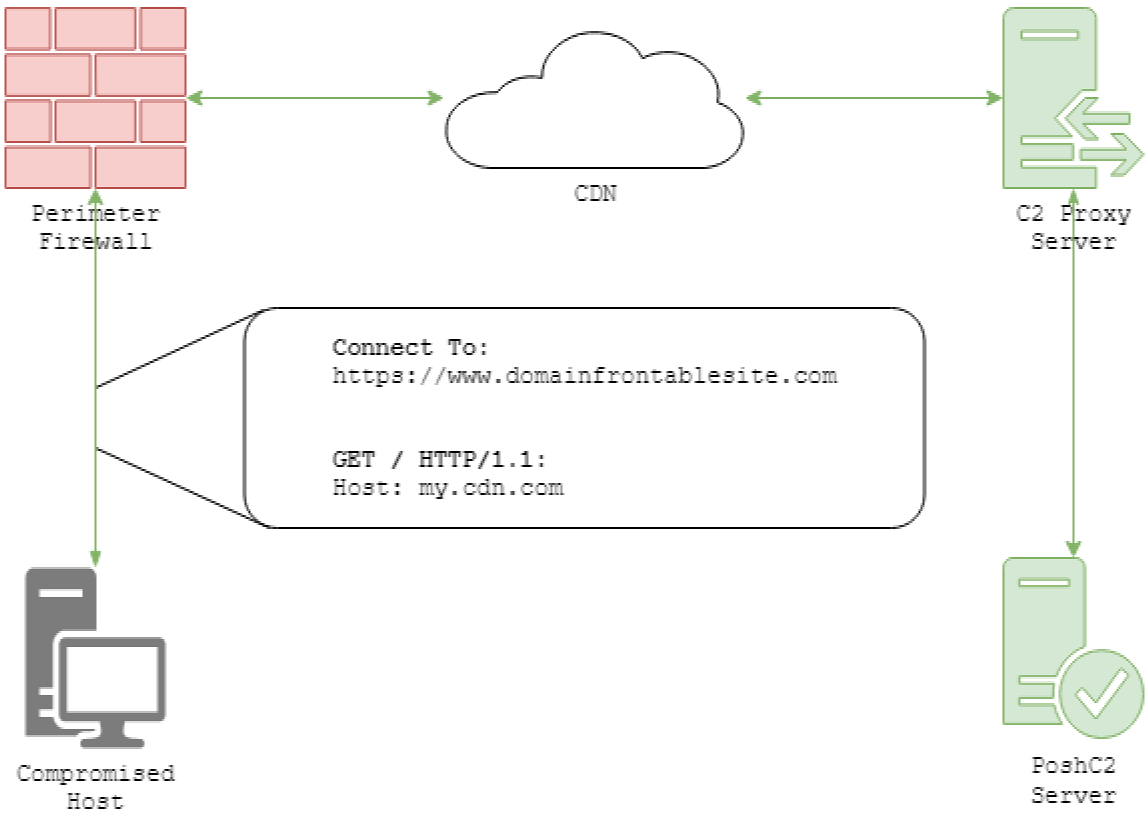
We also introduce the new PoshC2 Detections GitHub repository at https://github.com/nettitude/PoshC2_IOCs that will be continually updated as development continues, in order to assist blue teams with detecting PoshC2, particularly when used by less sophisticated attackers that do not alter or configure it to change the default IoCs. We encourage the community to contribute to and help update and improve this repository.

It is worth noting that PoshC2 is open source, so while these are IoCs for PoshC2 if used it its default state, ultimately it can be altered either through configuration or by changing the source code itself. The default configuration is subject to change, however where possible the location of that value is pointed out to the reader in order to allow these values to be monitored and updated, in addition to providing the GitHub repository.

## Communications

One way to detect PoshC2 is to focus on the communications. Compromised hosts have to communicate with the C2 server somehow in order to pick up tasks and return task output. This is unavoidable and PoshC2 can currently only do this through the use of HTTP(S) requests.

That isn't to say this this makes it easy to detect; a huge amount of HTTP traffic is present in most environments and the flexibility of the protocol allows for traffic to be hidden and routed through legitimate websites using techniques such as Domain Fronting and reverse proxies.




An example of HTTP comms when domain fronting.

Something very helpful for catching C2 communications is SSL Inspection. By being able to inspect SSL traffic leaving the perimeter, the contents of that traffic can be checked and statistics acquired for detecting C2 communications. Without this, network defenders are largely blind, particularly against domain fronted communications which travel via legitimate third-party websites.

If SSL Inspection is implemented then the HTTP traffic can be viewed, and while PoshC2 encrypts the contents of the HTTP bodies, the HTTP URLs and headers can still be viewed.

## URLs

PoshC2 has two different ways of generating URLs to use for communications. Operators can either use a static list of URLs or provide a wordlist from which random URLs will be generated, and these files are stored at `resources/urls.txt` and `resources/wordlist.txt` respectively, with the static URL list being the default option. These URLs are then loaded into the database when the project is first created, and a random URL is chosen by each implant each time it beacons. The default URL list is below:

```
/adsense/troubleshooter/1631343/
/adServingData/PROD/TMClient/6/8736/
/advanced_search?hl=en-GB&fg=
/async/newtab?ei=
/babel-polyfill/6.3.14/polyfill.min.js=
/bh/sync/aol?rurl=/ups/55972/sync?origin=
/bootstrap/3.1.1/bootstrap.min.js?p=
/branch-locator/search.asp?WT.ac&api=
/business/home.asp&ved=
/business/retail-business/insurance.asp?WT.mc_id=
/cdba?ptv=48&profileId=125&av=1&cb=
/cisben/marketq?bartype=AREA&showheader=FALSE&showvaluemarkers=
/classroom/sharewidget/widget_stable.html?usegapi=
/client_204?&atyp=i&biw=1920&bih=921&ei=
/load/pages/index.php?t=
/putil/2018/0/11/po.html?ved=
/qqzddddd/2018/load.php?lang=en&modules=
/status/995598521343541248/query=
/TOS?loc=GB&hl=en&privacy=
/trader-update/history&pd=
/types/translation/v1/articles/
/uasclient/0.1.34/modules/
/usersync/tradedesk/
/utag/lbg/main/prod/utag.15.js?utv=
/vfe01s/1/vsopts.js?
/vssf/wppo/site/bgroup/visitor/
/wpaas/load.php?debug=false&lang=en&modules=
/web/20110920084728/
/webhp?hl=en&sa=X&ved=
/work/embedded/search?oid=
/GoPro5/black/2018/
/Philips/v902/
```

While these URLs were originally copied from legitimate requests, if you see several of them being repeated to a site, particularly if they do not seem relevant to that site and if the response does not make sense, then it could be PoshC2 beacon traffic.

## HTTP Responses

PoshC2 also has static HTML responses that it responds with. The default is six HTTP 200 responses and one 404 response. These are stored in files at `resources/responses/` and also loaded into the database when the server is first created. The server responds with a random 200 response to POST requests that do not error or require a specific response, and with the single 404 response to all unexpected URLs or when the C2 server errors. Other responses return context relevant data, such as tasks, implant code and so on.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL was not found on this server.</p>
<hr>
<address>Apache (Debian) Server</address>
</body></html>
```

This static HTML file then at `resources/responses/404_response.html` is an IoC and if returned from a webserver that you are investigating is suggestive of PoshC2. Similarly, the *200_response** files in the same directory are IoCs if returned by POST requests.

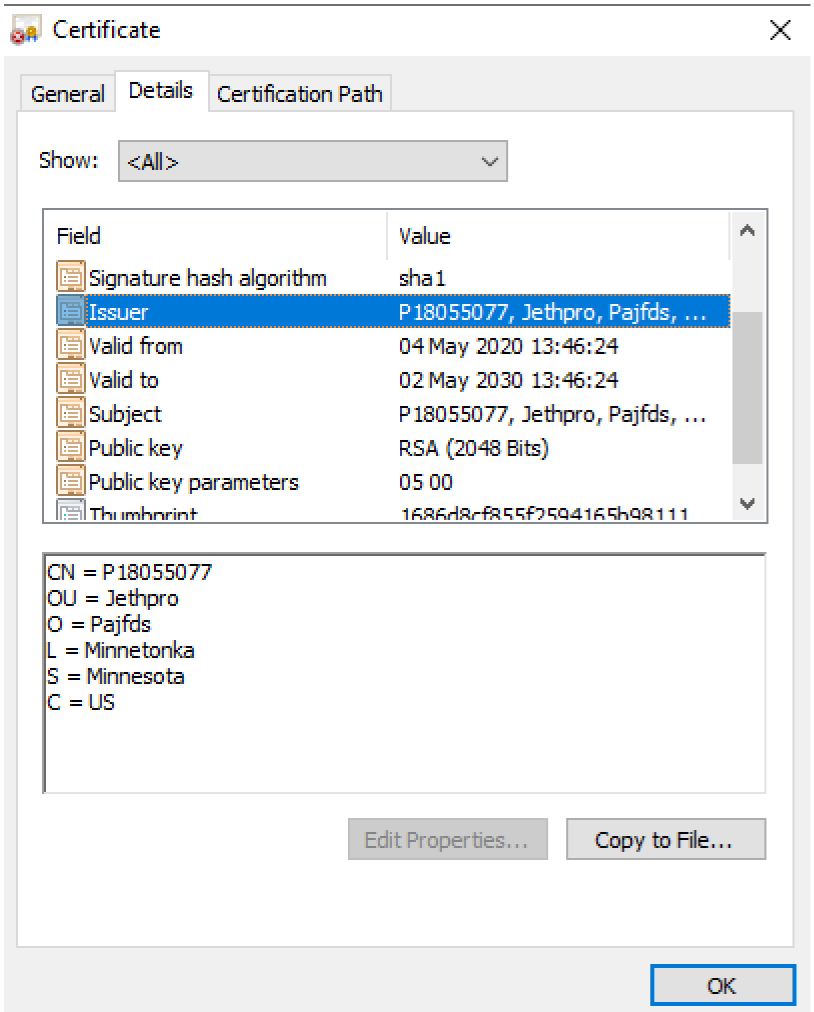
Note however, that it is recommended that operators change these files before creating a PoshC2 project, as is the use of a C2 proxy, so as with the other indicators the absence of this particular response is not unexpected for a PoshC2 installation.

## SSL Certificate

PoshC2 by default creates a self-signed certificate for its HTTP server, the values for which are stored in `poshc2/server/Config.py` file. These values are not in the 'normal' configuration file `config.yml` and are less documented and are therefore harder to change.

```
Cert_C = "US"
Cert_ST = "Minnesota"
Cert_L = "Minnetonka"
Cert_O = "Pajfds"
Cert_OU = "Jethpro"
Cert_CN = "P18055077"
Cert_SerialNumber = 1000
Cert_NotBefore = 0
Cert_NotAfter = (10 * 365 * 24 * 60 * 60)
```

An experienced operator will not expose their C2 server to the internet, but will instead use a proxy server with a valid certificate and filter firewall traffic to the C2 server that is not from that proxy, however if these steps are not taken and a certificate with the below values is presented then it is another strong indicator that PoshC2 is in use, likely by less sophisticated adversaries.

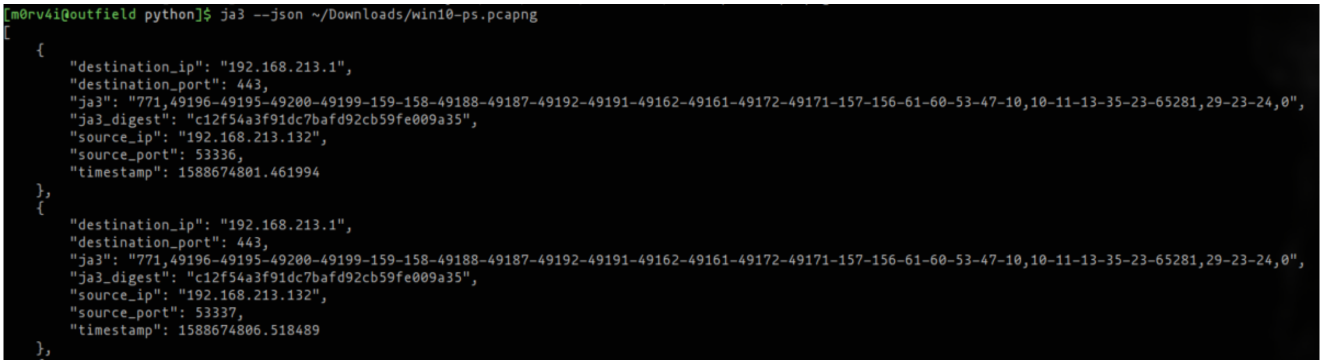



The issuer data can be viewed using Chrome and the default values suggest this is a PoshC2 server.

## JA3 Signatures

Another method for signaturing C2 traffic without SSL Inspection is to fingerprint the *Client Hello* packet that initialises the TCP connection. The specific bytes that make up the packet are dependent upon the type of connection being formed and the underlying packages and methods used to do so.

As these packets are sent before encryption has been established they are cleartext and can be intercepted and read. It turns out that the packet contents are quite unique and can be fingerprinted. Salesforce have an excellent blog post on JA3 fingerprinting, but essentially the details are extracted from the packets that make up this TCP handshake and hashed to create a quick and easy signature, that can be used to identify not only that PoshC2 is in use, but also the specific implant type.




The current Windows 10 PoshC2 signatures are below:

```
PowerShell: c12f54a3f91dc7bafd92cb59fe009a35
Sharp: fc54e0d16d9764783542f0146a98b300
```

JA3 fingerprinting has been incorporated into security products such as Splunk and Darktrace to provide quick and easy identification of C2 traffic, and can be utilised with these fingerprints. These fingerprints are available and will be maintained in the PoshC2 Detections GitHub repository.

## Telemetry

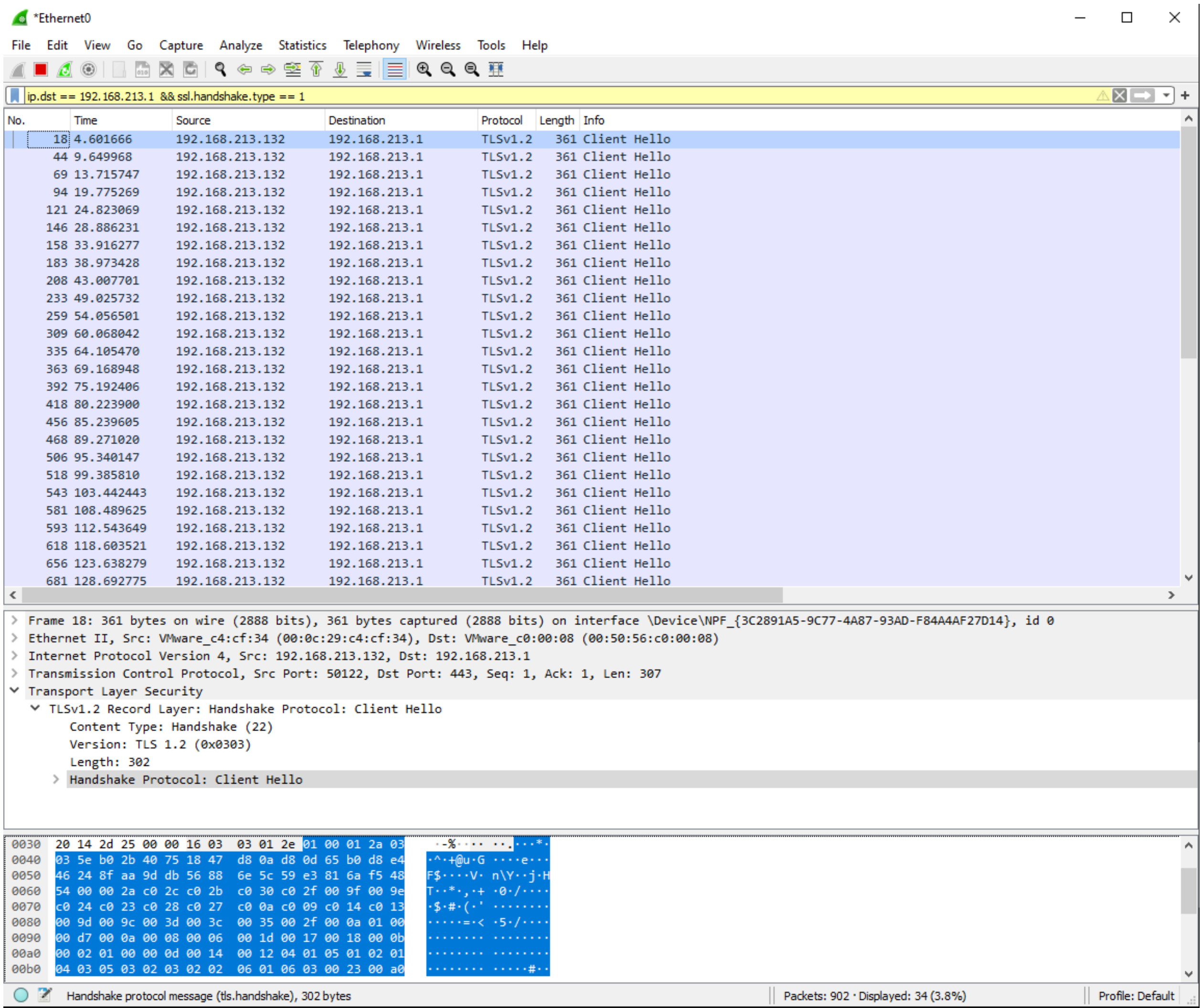
Another mechanism for detecting C2 traffic, with or without SSL Inspection, is through the use of telemetry.

PoshC2's default values for beacon time and jitter in the `config.yml` file are 5 seconds and a 20% jitter, as can be seen below, which is both a fast beacon rate and a small jitter, making it relatively easy to detect using this method.

```
DefaultSleep: "5s"
Jitter: 0.20
```

With these values the implants are going to beacon every 4-6 seconds by default, with an average of 5 seconds.

After capturing some traffic, filtering by the C2 server host and only checking TLS Client Hello packets, we can see that the TLS connection is created roughly every five seconds, confirming what we expect.



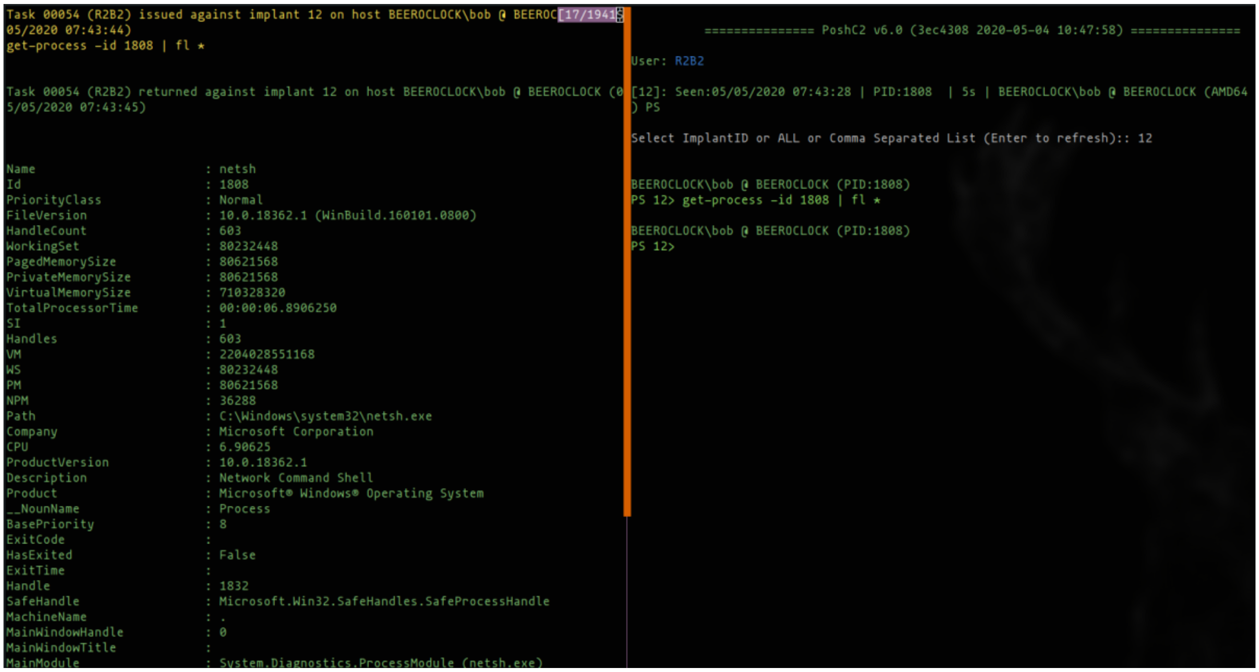
We can see the difference between each session initialisation is between 4 and 6 seconds.

By exploring this small sample of data, we can determine that the average time between the requests is 5.05 seconds, with a standard deviation of 0.81 or 16%, close to the 5 second beacon with the 20% jitter we expect from PoshC2's defaults.

Security products that can provide telemetry data, e.g. Splunk can be used to detect C2 traffic in this way by checking for repetitive beacons at a relatively fixed frequency, whether it's at the above defaults for PoshC2 or at any frequency, given that it is configurable.

## PowerShell Implant

PoshC2 has three implant types; PowerShell, C#, and Python, with the latter being a lightweight implant type mostly intended for compromising *nix hosts.
The PowerShell Implant runs by loading `System.Management.Automation.dll`, the engine behind `PowerShell.exe`, into the desired process and executing PowerShell commands directly using this DLL.



The PowerShell implant supports full PowerShell execution from any process (here netsh.exe) by loading System.Management.Automation.dll.

While this does not utilise `PowerShell.exe,` bypassing many restrictions and controls, it is still subject to PowerShell specific constraints such as Constrained Language mode and ScriptBlock logging. The latter, in particular, can be used to detect PoshC2 and any other malicious PowerShell commands, whether via `PowerShell.exe` or otherwise.

Note, however, that ScriptBlock logging was only enabled in PowerShell v5.0 and above, so if PowerShell v2.0 is available on the target then a downgrade attack can be performed and PowerShell 2.0 used, bypassing Constrained Language mode, ScriptBlock logging, AMSI and other controls added in v5.0.

## PS Logging

Enabling ScriptBlock logging and Transcript Logging for PowerShell allows logging of processes running PowerShell commands.

There are multiple IoCs here, not least that the Host Application is not `PowerShell.exe`, as should be expected, but instead `netsh.exe`.

```
**********************
Windows PowerShell transcript start
Start time: 20200504164342
Username: BEEROCLOCK\bob
RunAs User: BEEROCLOCK\bob
Configuration Name:
Machine: BEEROCLOCK (Microsoft Windows NT 10.0.18362.0)
Host Application: C:\Windows\system32\netsh.exe
Process ID: 7772
PSVersion: 5.1.18362.145
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.18362.145
BuildVersion: 10.0.18362.145
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
**********************
Command start time: 20200504164342
**********************
PS>$o = IEX $c | Out-String
**********************
```

PowerShell transcripts provide valuable information. Alerting on Host Applications that are not PowerShell.exe is a good way to find PowerShell implants from any C2 framework.

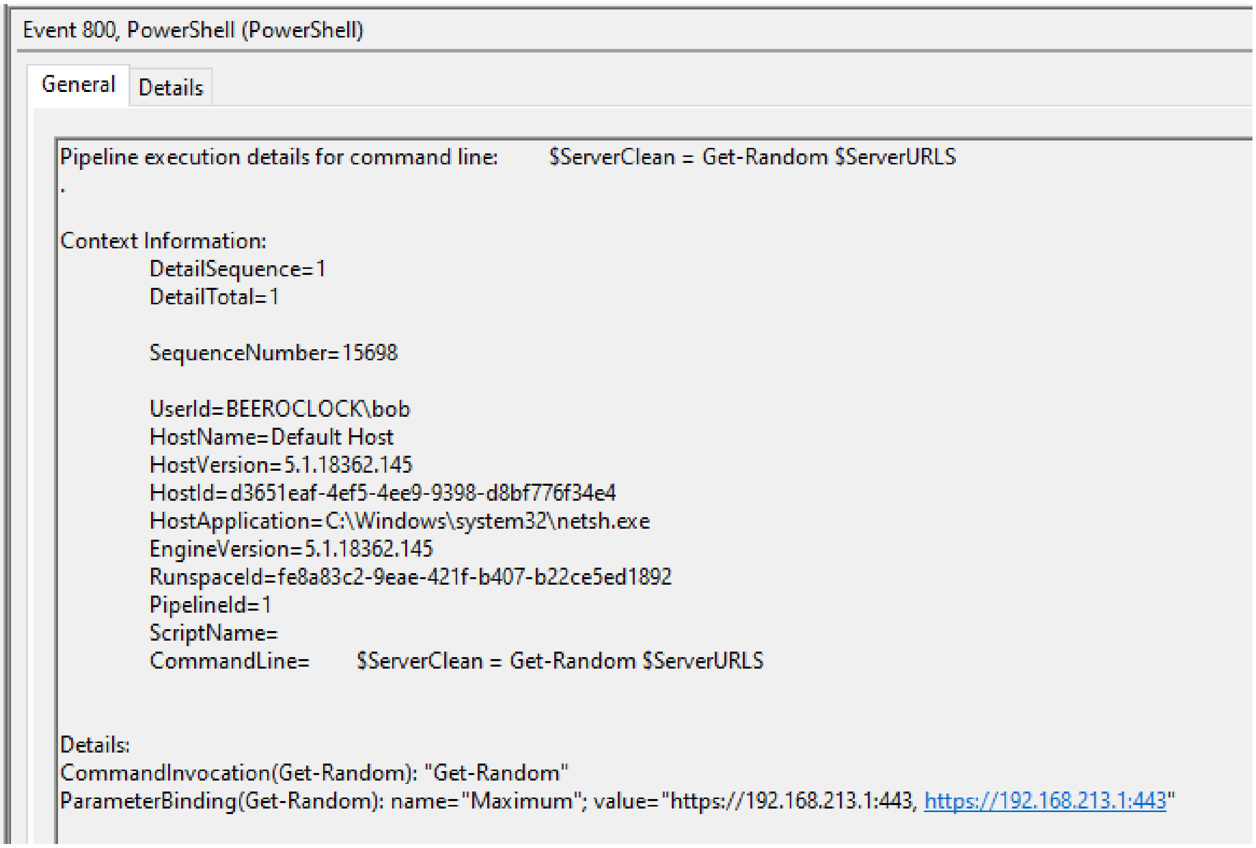
Similarly in the Event Viewer for ScriptBlock logging:

```
Event 800, PowerShell (PowerShell)
General  Details

Pipeline execution details for command line:     $RandomURI = Get-Random $URLS
.

Context Information:
        DetailSequence=1
        DetailTotal=1

        SequenceNumber=15696

        UserId=BEEROCLOCK\bob
        HostName=Default Host
        HostVersion=5.1.18362.145
        HostId=d3651eaf-4ef5-4ee9-9398-d8bf776f34e4
        HostApplication=C:\Windows\system32\netsh.exe
        EngineVersion=5.1.18362.145
        RunspaceId=fe8a83c2-9eae-421f-b407-b22ce5ed1892
        PipelineId=1
        ScriptName=
        CommandLine=     $RandomURI = Get-Random $URLS


Details:
CommandInvocation(Get-Random): "Get-Random"
ParameterBinding(Get-Random): name="Maximum"; value="adServingData/PROD/TMClient/6/8736/, qqzddddd/2018/, vfe01s/1/, babel-polyfill/6.3.14/, usersync/tradedesk/, cisben/,
```

PowerShell ScriptBlock logging can reveal a wealth of information, here including the beacon URLs for PoshC2 and again netsh.exe as a host process.

This also includes the ‘command line’ of the ScriptBlock being executed, which includes the beacon URLs from which a random element is being chosen, therefore listing all the C2 URLs that are in use by this implant.

Aside from the wealth of information ScriptBlock logging provides for any threat hunter, an event command line specific to PoshC2 that can be used to identify the PowerShell implant is below:



The $ServerClean variable is specific to PoshC2 and a clear IoC.

```
CommandLine= $ServerClean = Get-Random $ServerURLS
```

This ScriptBlock is part of the beaconing process and will be repeated frequently in the Event Viewer making it easy to identify.

## System.Management.Automation.dll

As mentioned earlier, the PowerShell implant does not function by invoking `PowerShell.exe`, but instead by loading `System.Management.Automation.dll` into the implant process.

This means that this is an IoC, and if we find a process that should not have this DLL loaded, particularly if it is an unmanaged code binary (so not .NET) then it is highly likely that this is a process that has been injected into by a PowerShell implant, PoshC2 or otherwise.

**System.Management.Automation.**

| Property | Value |
| --- | --- |
| File Name | C:\Program Files (x86)\Reference Assemblies\Microsoft\WindowsPo... |
| File Type | Portable Executable 32 .NET Assembly |
| File Info | Microsoft Visual Studio .NET |

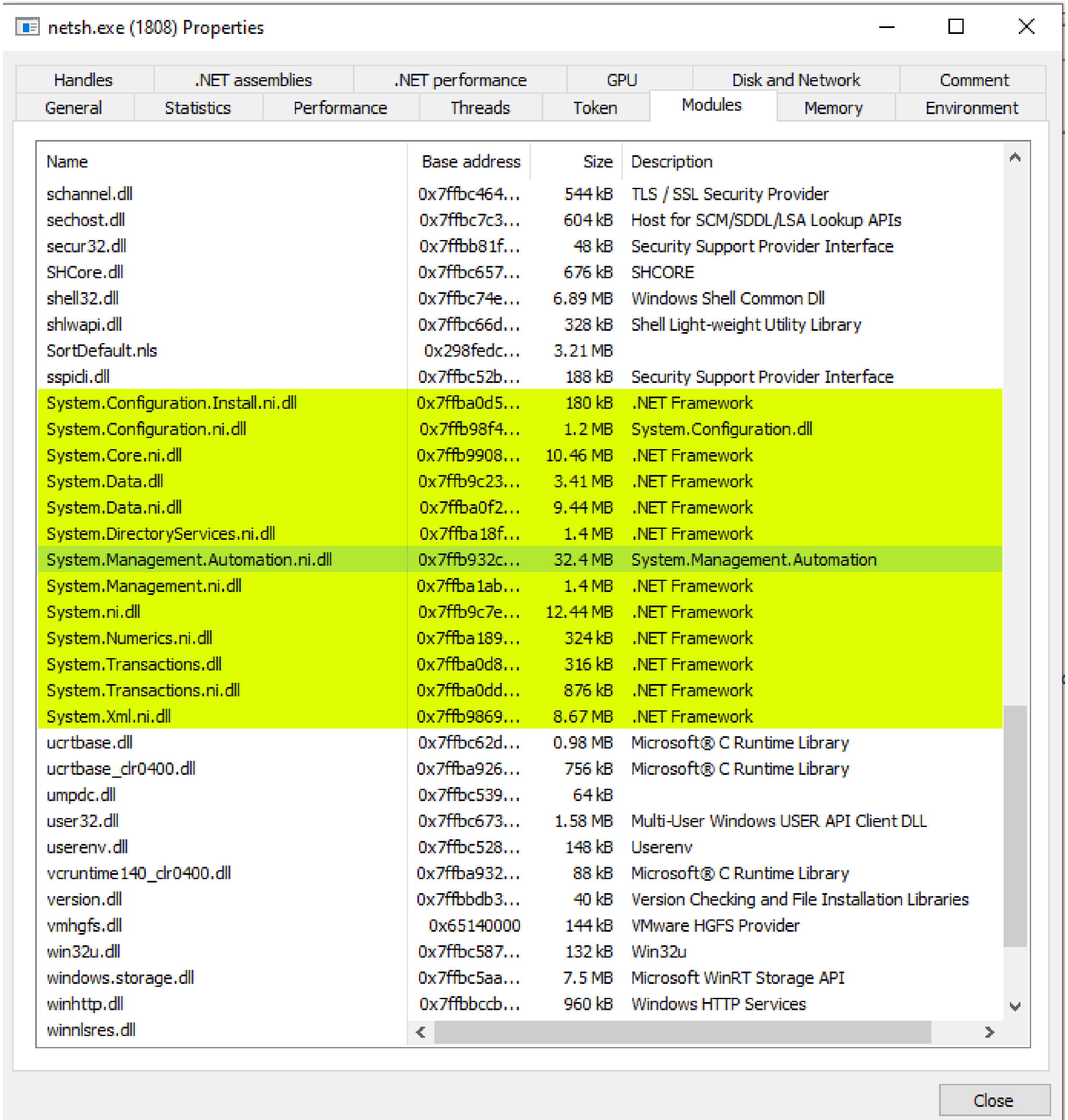
System.Management.Automation.dll, the engine behind PowerShell, is a .NET library.

**netsh.exe**

| Property | Value |
| --- | --- |
| File Name | C:\Windows\System32\netsh.exe |
| File Type | Portable Executable 64 |
| File Info | Microsoft Visual C++ 8.0 |

netsh.exe is a C++ binary and should not be loading .NET libraries.

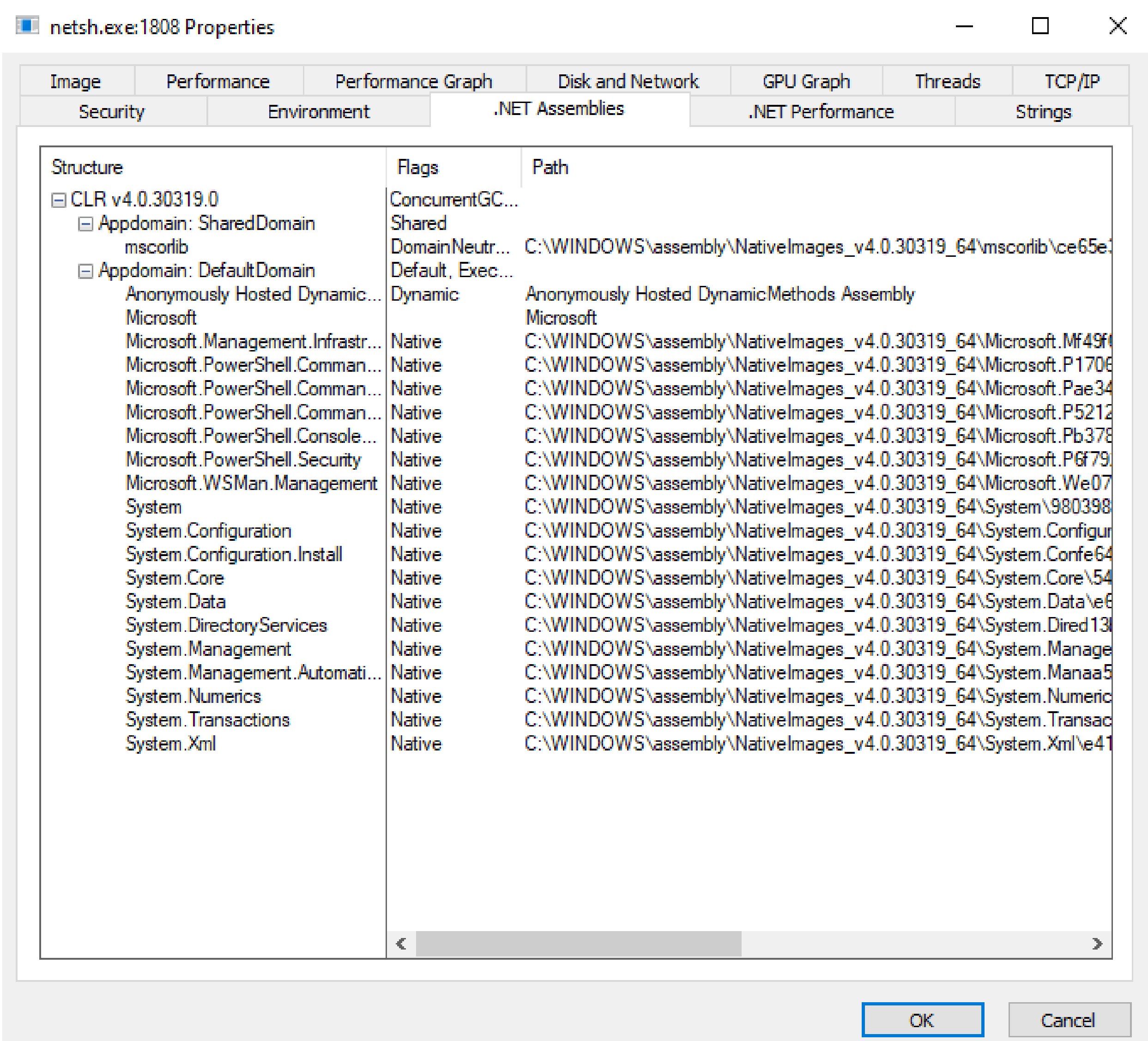
We can see this DLL loaded into an implant process using Process Hacker:




| Name | Base address | Size | Description |
|---|---|---|---|
| schannel.dll | 0x7ffbc464... | 544 kB | TLS / SSL Security Provider |
| sechost.dll | 0x7ffbc7c3... | 604 kB | Host for SCM/SDDL/LSA Lookup APIs |
| secur32.dll | 0x7ffbb81f... | 48 kB | Security Support Provider Interface |
| SHCore.dll | 0x7ffbc657... | 676 kB | SHCORE |
| shell32.dll | 0x7ffbc74e... | 6.89 MB | Windows Shell Common Dll |
| shlwapi.dll | 0x7ffbc66d... | 328 kB | Shell Light-weight Utility Library |
| SortDefault.nls | 0x298fedc... | 3.21 MB | |
| sspicli.dll | 0x7ffbc52b... | 188 kB | Security Support Provider Interface |
| System.Configuration.Install.ni.dll | 0x7ffba0d5... | 180 kB | .NET Framework |
| System.Configuration.ni.dll | 0x7ffb98f4... | 1.2 MB | System.Configuration.dll |
| System.Core.ni.dll | 0x7ffb9908... | 10.46 MB | .NET Framework |
| System.Data.dll | 0x7ffb9c23... | 3.41 MB | .NET Framework |
| System.Data.ni.dll | 0x7ffba0f2... | 9.44 MB | .NET Framework |
| System.DirectoryServices.ni.dll | 0x7ffba18f... | 1.4 MB | .NET Framework |
| System.Management.Automation.ni.dll | 0x7ffb932c... | 32.4 MB | System.Management.Automation |
| System.Management.ni.dll | 0x7ffba1ab... | 1.4 MB | .NET Framework |
| System.ni.dll | 0x7ffb9c7e... | 12.44 MB | .NET Framework |
| System.Numerics.ni.dll | 0x7ffba189... | 324 kB | .NET Framework |
| System.Transactions.dll | 0x7ffba0d8... | 316 kB | .NET Framework |
| System.Transactions.ni.dll | 0x7ffba0dd... | 876 kB | .NET Framework |
| System.Xml.ni.dll | 0x7ffb9869... | 8.67 MB | .NET Framework |
| ucrtbase.dll | 0x7ffbc62d... | 0.98 MB | Microsoft® C Runtime Library |
| ucrtbase_clr0400.dll | 0x7ffba926... | 756 kB | Microsoft® C Runtime Library |
| umpdc.dll | 0x7ffbc539... | 64 kB | |
| user32.dll | 0x7ffbc673... | 1.58 MB | Multi-User Windows USER API Client DLL |
| userenv.dll | 0x7ffbc528... | 148 kB | Userenv |
| vcruntime140_clr0400.dll | 0x7ffba932... | 88 kB | Microsoft® C Runtime Library |
| version.dll | 0x7ffbbdb3... | 40 kB | Version Checking and File Installation Libraries |
| vmhgfs.dll | 0x65140000 | 144 kB | VMware HGFS Provider |
| win32u.dll | 0x7ffbc587... | 132 kB | Win32u |
| windows.storage.dll | 0x7ffbc5aa... | 7.5 MB | Microsoft WinRT Storage API |
| winhttp.dll | 0x7ffbbccb... | 960 kB | Windows HTTP Services |
| winnlsres.dll | | | |

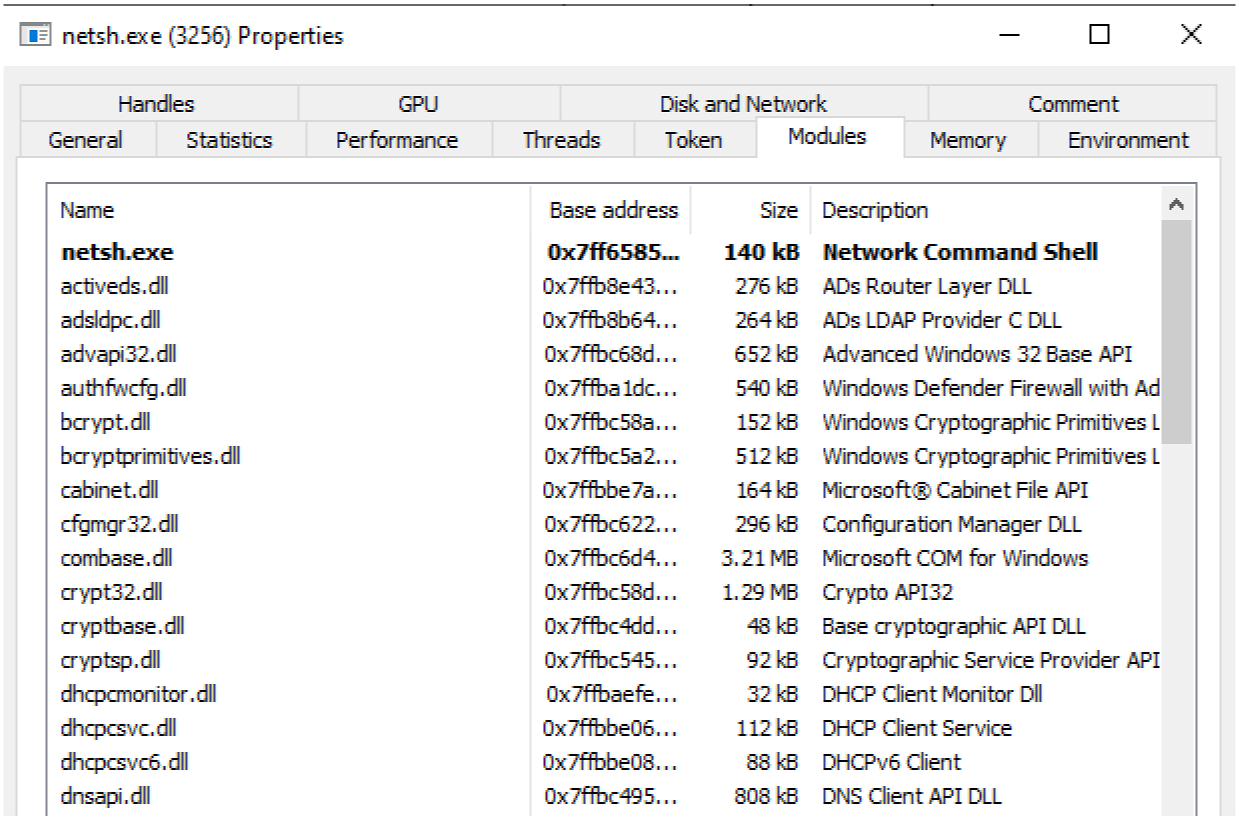
System.Management.Automation.dll loaded into netsh.exe.

Similarly we can view the .NET Assemblies in the process:



We can view the .NET Assemblies in the netsh.exe process and see PowerShell is loaded.

This tab shouldn't even be present for a genuine `netsh.exe` process as it is unmanaged code!



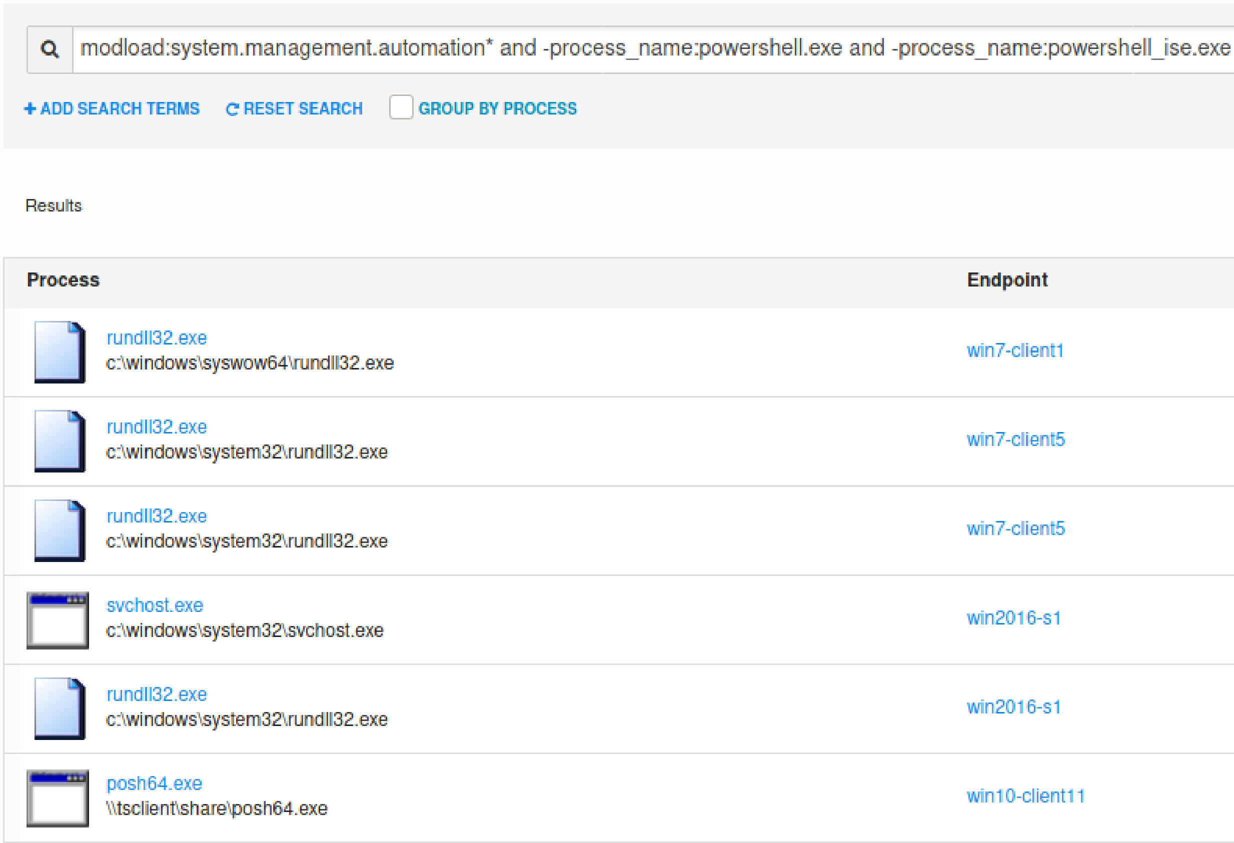

The module is not loaded in a genuine netsh process and the .NET tabs are not available.

An operator can be smarter about their migration by injecting into .NET processes, however it is still unlikely that a legitimate process (that isn't `PowerShell.exe`) would load `System.Management.Automation.dll`, so this is a great IoC to look out for in your environment.

This can be implemented at scale by searching for loaded modules in an EDR product, for example in CarbonBlack with a query of:

```
modload:System.Management.Automation* AND -process_name:powershell.exe AND -process_name:powershell_ise.exe
```

This searches for process with `System.Management.Automation.dll` or `System.Management.Automation.ni.dll` (the Native Image version) loaded into a process when that process is *not* `PowerShell.exe` or `PowerShell_ISE.exe`. Other legitimate exclusions may need to be added for your particular environment.



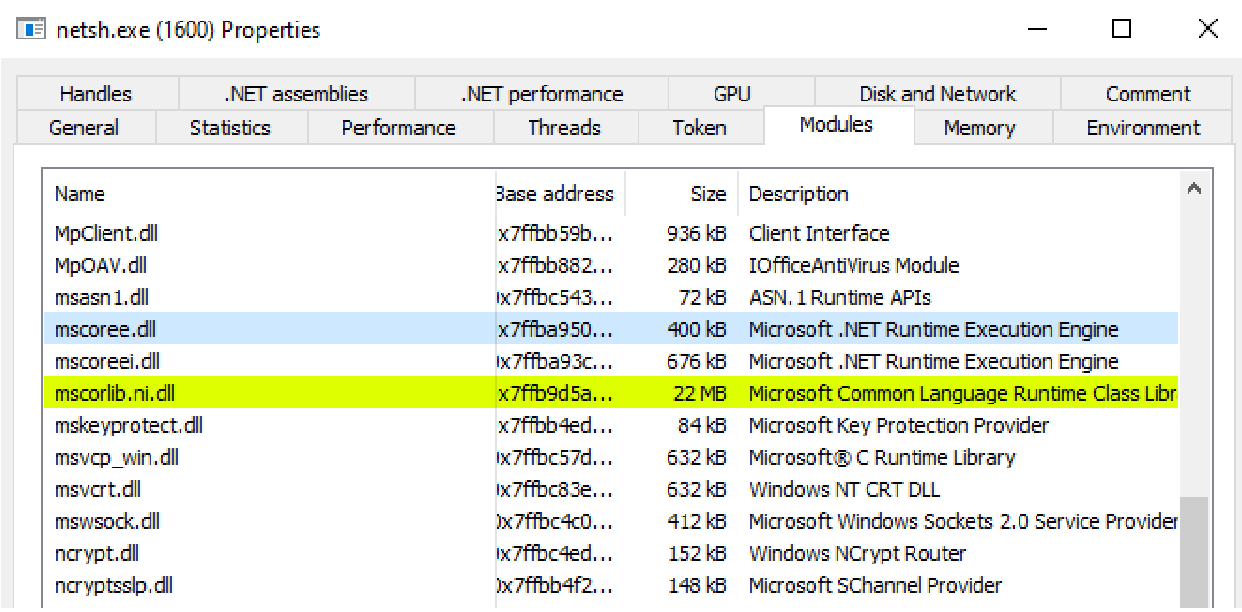

Here we can see various processes that have loaded System.Management.Automation.dll into memory that are likely implants.

## C# Implant

The C# or Sharp implant is PoshC2's 'next gen' implant, written (unsurprisingly) in C#. The key difference from a detection perspective is that this implant does not require loading `System.Management.Automation.dll` in order to function. Most of the functionality of the C# implant is custom-written and while it *can* load `System.Management.Automation.dll` in order to execute PowerShell, this is an operator decision and is by no means necessary.

A similar process to the above can be applied, but is a little harder to implement. .NET processes load the `mscoree.dll` library which is the core library behind the .NET CLR (Command Language Runtime), so again any unmanaged code processes that are loading this library could have been migrated into. The issue here is finding these processes, as it's not as simple as searching for just 'module loaded and process name is not powershell.exe'.

A blacklist can be created of common target processes that are unmanaged, always available and should not be loading the .NET runtime, and these can be monitored, as well as noting this during manual triage.

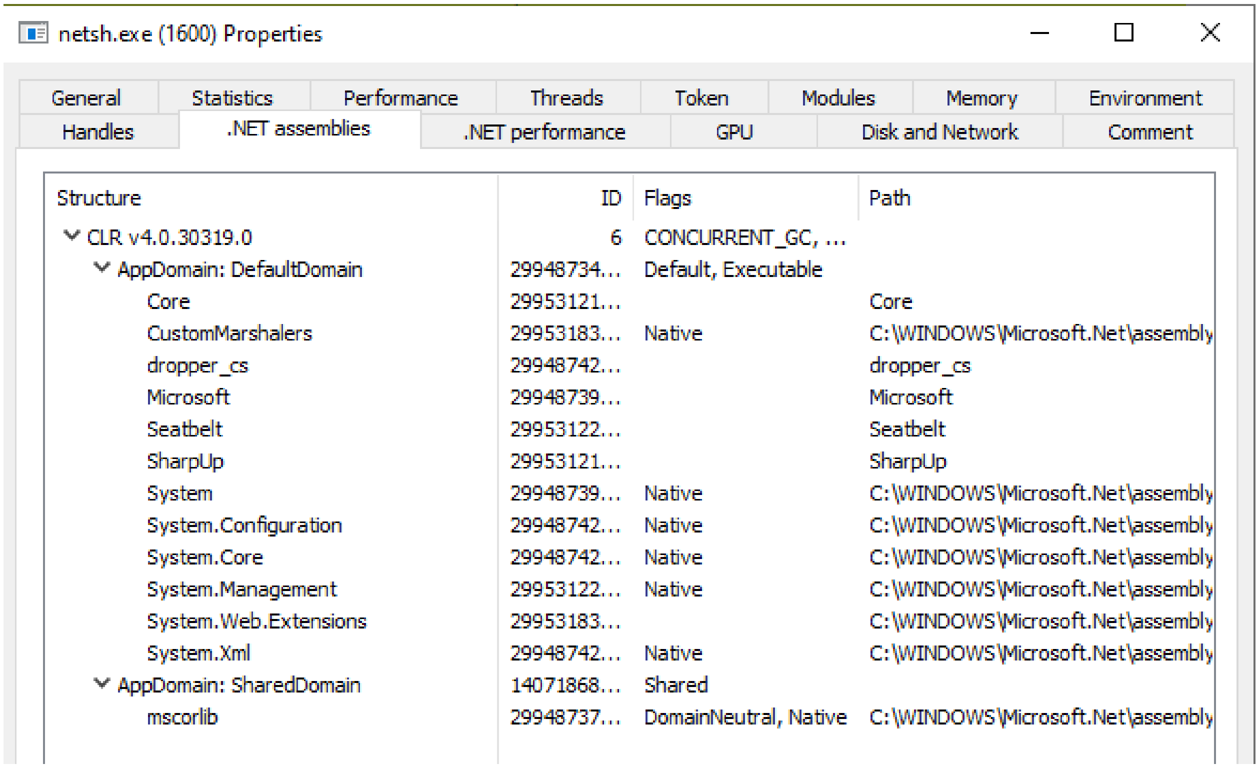



A C# implant loads mscoree.dll if it is not already present in the process – another IoC if the process is supposed to be unmanaged.

Note that the PowerShell implant will also load this library as it is also a .NET process, however the presence of `System.Management.Automation.dll` marks it as an implant that can run PowerShell.

The C# implant has the ability to load compiled binaries into memory over C2 and run them, which is extremely powerful. There are indicators of this however, as a new virtual runspace is created for each module and their namespace is listed in the AppDomain to which they are loaded.

Viewing the .NET Assemblies in Process Explorer or Process Hacker for example then reveals what modules have been run.




In the C# implant loaded modules have their namespace visible in the AppDomain, making it clear what has been loaded into the implant.

Above we can see the *Core* and *dropper_cs* modules that make up the core functionality of the C# implant, as well as some native modules from Microsoft that are required to run. These modules will always be present in the PoshC2 C# implant and are a clear IoC. We also see Seatbelt and SharpUp, two common C# offensive modules from SpectreOps, and we can surmise that they have been on the target.

## In General

### Migration

We have used `netsh.exe` as the example implant process in this post and that is for a good reason. The default migration process for PoshC2 in the C# and PowerShell implants is `C:\Windows\System32\netsh.exe`, so when the `migrate` or `inject-shellcode` commands are used and a specific process ID or name is not set, then a new `netsh.exe` process is spawned and migrated into.

This itself is a common IoC for PoshC2, as `netsh.exe` is not typically frequently run in environments, and certainly not on most end user's hosts. Therefore, if there is a sudden uptick in the number of these processes being run in the environment or if several are running on a host then it could be worth investigating.

### Persistence

PoshC2 has three quick persistence commands available to the PowerShell implant. Each of these installs a `PowerShell.exe` one liner payload to the registry in the key at `HKCU\Software\Microsoft\Windows\CurrentVersion\themes` with a name *Wallpaper777*, *Wallpaper555* or *Wallpaper666*, depending on the command being run.

This payload is then triggered by either:

- A registry key at `HKCU\Software\Microsoft\Windows\CurrentVersion\run` with the name `IEUpdate`
- A Scheduled Task, also with the name `IEUpdate`
- A shortcut file placed at `%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\IEUpdate.lnk`

All of these can be alerted upon and used to determine that the adversary using PoshC2, in addition to alerting on the invocation of `PowerShell.exe` with encoded parameters, and so on.

## Binary payloads

Some of the more common payloads that are dropped on targets are the PoshC2 executables and DLLs that can be run using `rundll32.exe`.

For the DLLs, there are different versions for PowerShell and Sharp implants across versions 2 and 4 of PowerShell and x86 and x64 bit architectures, however all the DLLs have a single entry-point common to all: `VoidFunc`.

**Sharp_v4_x64.dll**

| Member | Offset | Size | Value |
|---|---|---|---|
| Characteristics | 00013A20 | Dword | 00000000 |
| TimeDateStamp | 00013A24 | Dword | FFFFFFFF |
| MajorVersion | 00013A28 | Word | 0000 |
| MinorVersion | 00013A2A | Word | 0000 |
| Name | 00013A2C | Dword | 00014852 |
| Base | 00013A30 | Dword | 00000001 |
| NumberOfFunctions | 00013A34 | Dword | 00000001 |
| NumberOfNames | 00013A38 | Dword | 00000001 |
| AddressOfFunctions | 00013A3C | Dword | 00014848 |

| Ordinal | Function RVA | Name Ordinal | Name RVA | Name |
|---|---|---|---|---|
| | | | | |
| (nFunctions) | Dword | Word | Dword | szAnsi |
| 00000001 | 0000175C | 0000 | 0001485F | VoidFunc |

All the DLL payloads have the same single entry point of VoidFunc.

This entry-point is hard-coded in PoshC2 and cannot be changed without hacking the compiled binary itself.

For the common executable and DLL payloads we've also added Yara rules for detecting them. These are based on signaturable parts of the binaries that will not change across different installs of PoshC2, for example with different comms options. These are also available in the new PoshC2 Detections GitHub repository.

## In Summary

We've looked at a few different detections for catching PoshC2 when used out-of-the-box. Using these in your environment will help protect against the less sophisticated users of PoshC2 in addition to further understanding how the tool works.

Any new detections or amendments can be added to the https://github.com/nettitude/PoshC2_IOCs repository and we encourage the community to add their own detections or rules and configurations for other security tools to help build a centralised data store for everyone.