ConvertTo-Powershell - wrapping applications with PS

cyberstoph.org/posts/2020/09/convertto-powershell-wrapping-applications-with-ps

September 6, 2020

TL;DR;

The newest addition to PSArmoury is a small utility script called ConvertTo-Powershell. It allows to “convert” a C# console application into a powershell script or in other words it creates a .ps1 file from a .exe file. This can be useful for bypassing AWL or AV software and if you are a similar PS-lunatic like myself, it’s also just fun :-)

From exe to ps1

If you read one of my previous posts, you might already know that I prefer to do stuff in Powershell. One of my pet projects is the PowerShellArmoury, which allows you to store other PowerShell code in a single, encrypted .ps1-file called a “loader”. The loader, when invoked, tries to bypass the Windows Anti-Malware-Scan-Interface (AMSI) and then decrypts and loads the original Powershell code in the current session.

That works pretty well for Powershell but since pure .NET is the popular kid in town right now, more and more fancy tools are written in C# and no longer in PS. Therefore, I was looking for a way to integrate things like Ghostpack into my existing armoury and found the solution pretty much at the end of the Rubeus wiki.

```
$RubeusAssembly =
[System.Reflection.Assembly]::Load([Convert]::FromBase64String("aa..."))
```

The command above uses the `System.Reflection` namespace to load a compiled C# console application into your active Powershell session. You can then execute the methods within directly from Powershell. Pretty cool I’d say but how does it work?

A quick peak into reflection

First of all: I am not a software developer and I never learned software engineering. So if you find any conceptual mistakes or other nonsense in this post, please let me know so I can learn :) That said, let’s take a quick look at the concept of “reflection” in the .NET framework.

Reflection allows you to programmatically obtain information about .NET assemblies (.exe and .dll files written in .NET). In other words: you can write a .NET application that dynamically interacts with other, already compiled, .NET applications on your system. But the classes in the `Reflection.Assembly` namespace not only allow to query classes, methods and attributes from assemblies but also to instantiate new objects from these classes (=run them).

To get started, you need the `[System.Reflection.Assembly]::Load`-class. If you have a look at the docs, you'll see that we can load an assembly by handing a base64 encoded blob of the assembly to the `Load`-function.

Load(Byte[])	Loads the assembly with a common object file format (COFF)-based image containing an emitted assembly. The assembly is loaded into the application domain of the caller.

That's espescially useful since converting stuff into base64 means we can embed it into a PS script

```
$file = [Convert]::ToBase64String([IO.File]::ReadAllBytes(".\Rubeus.exe"))
```

Next, we create an object of type assembly

```
$Assembly =
[System.Reflection.Assembly]::Load([Convert]::FromBase64String($file))
```

Now we are able to enumerate the available types. The screenshot below is an excerpt of Rubeus.

```
$Assembly.GetTypes()
```

```
PS C:\> $Assembly.GetTypes()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    ConsoleTable                             System.Object
True     False    ConsoleTableOptions                      System.Object
True     True     Format                                   System.Enum
True     False    RubeusException                          System.Exception
True     False    KerberosErrorException                   Rubeus.RubeusException
True     False    Ask                                      System.Object
True     False    IBruteforcerReporter
True     False    Bruteforcer                              System.Object
True     False    Crypto                                   System.Object
True     False    Harvest                                  System.Object
True     False    Helpers                                  System.Object
True     False    Interop                                  System.Object
True     False    AP_REQ                                   System.Object
True     False    AS_REP                                   System.Object
True     False    AS_REQ                                   System.Object
```

Also of interest to our specific usecase is the entrypoint.

```
$Assembly.EntryPoint | select Name,ReflectedType,Module
```

```
PS C:\> $Assembly.EntryPoint | select Name,ReflectedType,Module

Name ReflectedType  Module
---- ------------  ------
Main Rubeus.Program Rubeus.exe
```

The image above tells us that the operating system would start execution of `Rubeus.exe` in the function `Main` of the namespace/class `Rubeus.Program`. If we want to run Rubeus manually, that's the information we need.

ConvertTo-Powershell

With the knowledge about the entrypoint, we've got all we need to automatically create a .ps1 wrapper for a given .NET assembly.

Step 1 - Load the assembly and get the entrypoint

```
function Get-EntryPoint
{
    [CmdletBinding()]
    Param (
    [Parameter(Mandatory = $true)]
    [ValidateScript({Test-Path $_})]
    [String]
    $Path)

    $item = Get-Item -Path $Path
    $file = [Convert]::ToBase64String([IO.File]::ReadAllBytes($item.FullName))
    $Assembly =
[System.Reflection.Assembly]::Load([Convert]::FromBase64String($file))
    $Assembly.EntryPoint
}
```

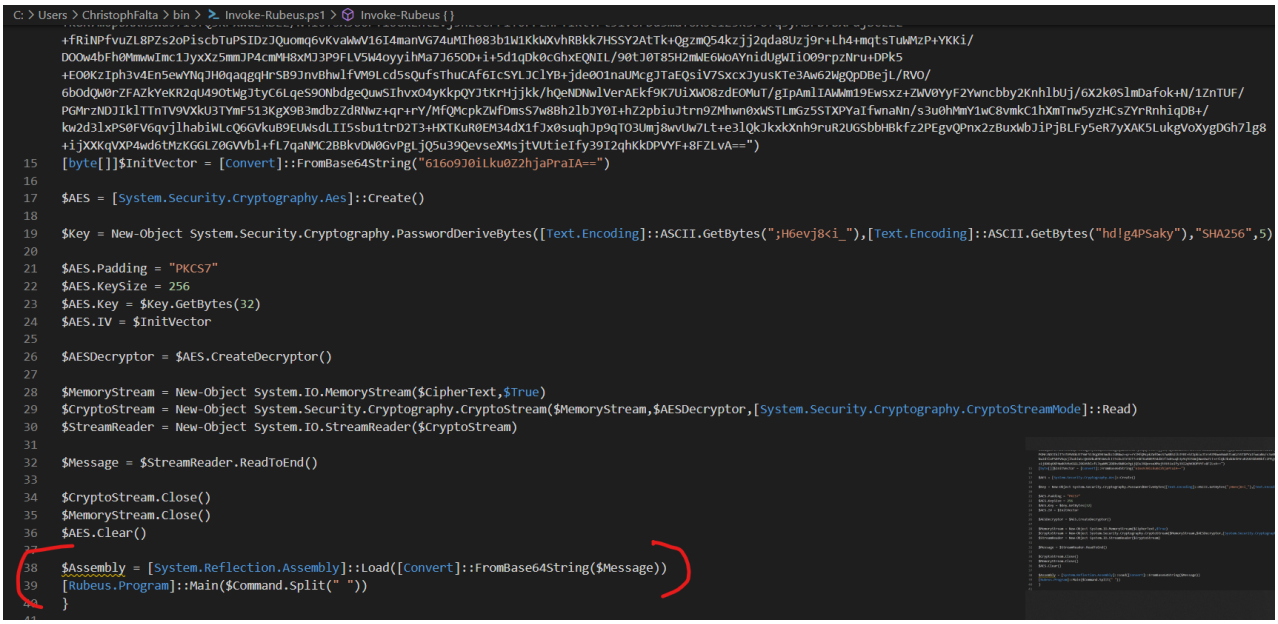
Step 2 - Build the command that will execute the entrypoint in the wrapper script

```
$ep = Get-EntryPoint -Path C:\path\yourfile.exe
$ldrcommand = "[" + $ep.reflectedtype.namespace + "." + $ep.reflectedtype.name +
"]::" + $ep.name + '($Command.Split(" "))'
```

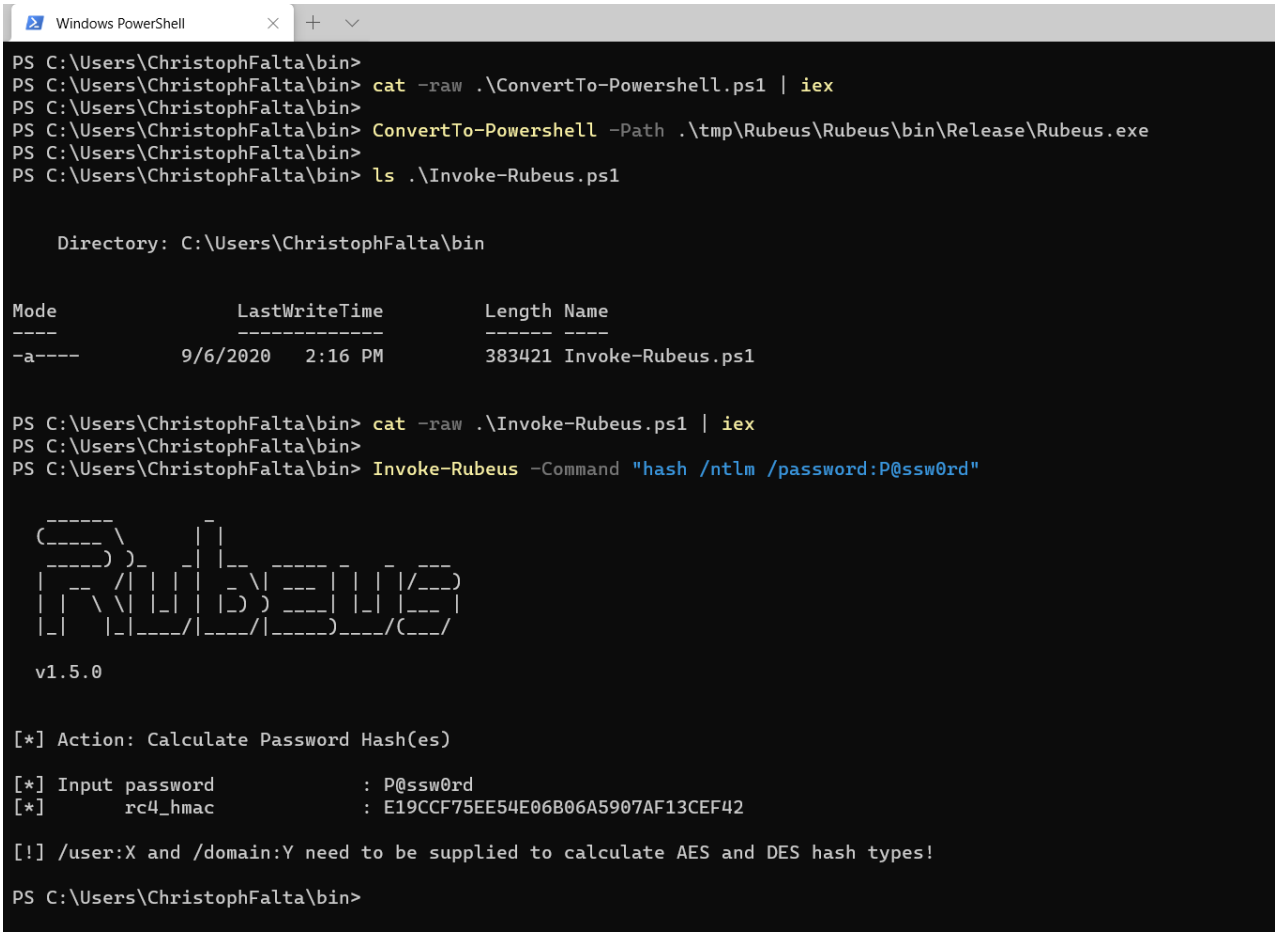
Note that we'll also pass commandline parameters to `$ep.name` through the use of the `$Command` parameter. The later will be available in the wrapper script.

Step 3 - combine

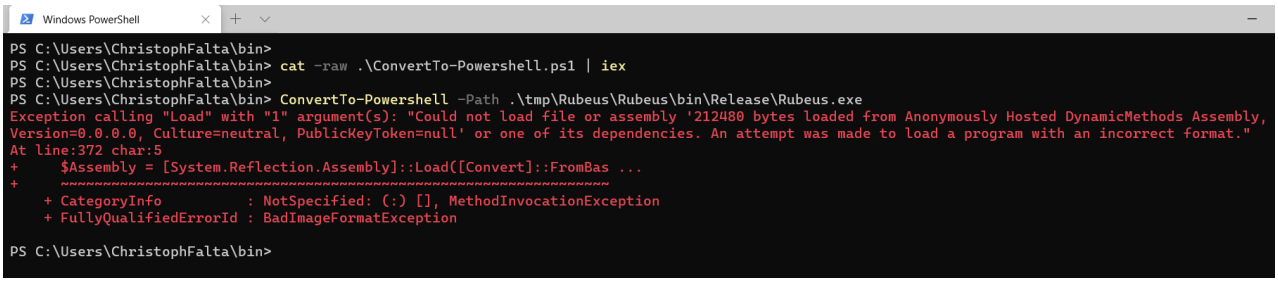
Finally, we'll put these two things together in addition to some standard PSArmoury routines for encryption/decryption and AV bypass as needed. The result will be a .ps1 file, which contains the base64-encoded (and encrypted) assembly as well as the information on how to invoke it directly from PowerShell as described above. Have a look at the last two lines in the screenshot below.



And that's it. Feel free to try and let me know what I missed (pretty sure I missed something ;-)). Here's an example for how to use it.



One last note: if you get an error message like the one below when running ConvertTo-Powershell, then that's probably AMSI complaining about the file you want to convert. In my example, thats Rubeus.exe.



So don’t forget to disable or bypass AMSI on the machine you use to convert. The AMSI bypass that comes with ConvertTo-Powershell is only executed in the .ps1 files you create but not in the builder function. You can find a standalone version of that one over here

---