PS Protector: Convert your PowerShell module into a .NET assembly DLL

Tim Warner

PS Protector is a Windows utility that simplifies converting your plaintext PowerShell module into a Windows .NET assembly DLL. The use case is for businesses that need to protect their intellectual property by preventing source code inspection.

Look, you and I both know that the PowerShell team at Microsoft has gradually emphasized collaborative, open-source development. Look at these products, all of which are free, open-source, and available for community contributions:

- .NET Core
- PowerShell 6
- PowerShell documentation
- PowerShell Desired State Configuration (DSC) resources

However, your business may develop PowerShell modules either not for public consumption or for sale to paying customers. For whatever reason, you want to obfuscate your source code and prevent code inspection, with or without reverse engineering.

PS Protector to the rescue! PS Protector is a small Windows utility that simplifies converting your PowerShell .psm module file(s) into Windows .NET dynamic-link library (DLL) assemblies.

PS Protector is the work of a Swiss developer named Stefan Soller. Let's learn how to use the tool.

Protecting a PowerShell module

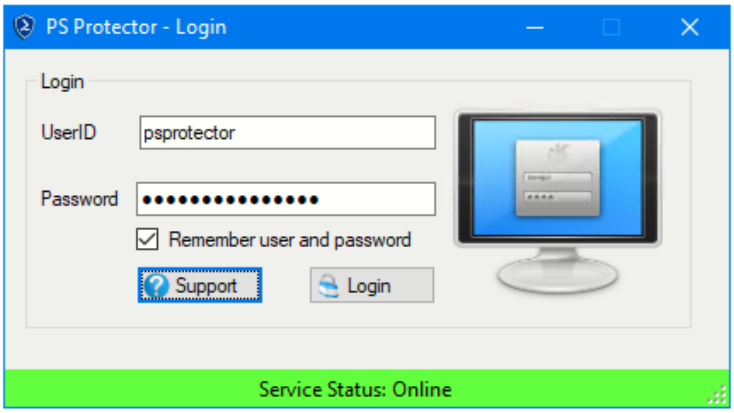
Let's begin by writing a simple test function using PowerShell 5.1 Desktop on my Windows 10 workstation:

```
Function Test-Function()

{

      Write-Output -InputObject 'If you can read this message, then the Test-
      Function function ran correctly.'

}

Export-ModuleMember -Function *
```

Next, download the PS Protector free trial. PS Protector comes down as a 400 KB standalone executable along with a simple .config file. Upon launch, you're required to sign into the PS Protector web API. Here are the trial credentials as listed on their website:

- **UserID:** demo

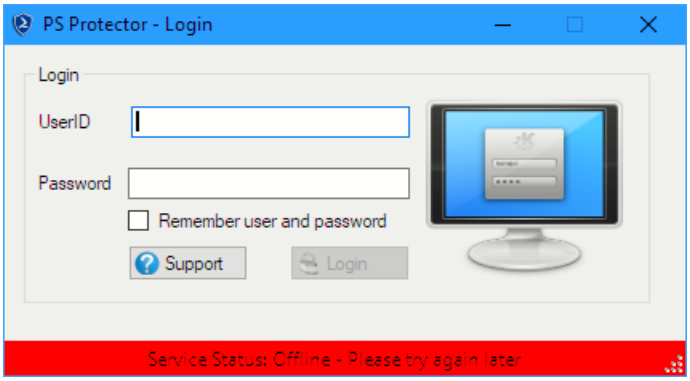
- **Password:** rWf1+ccFx!p2a0e



Sign into PS Protector

The trial license lets you protect PowerShell modules that contain no more than 200 characters.

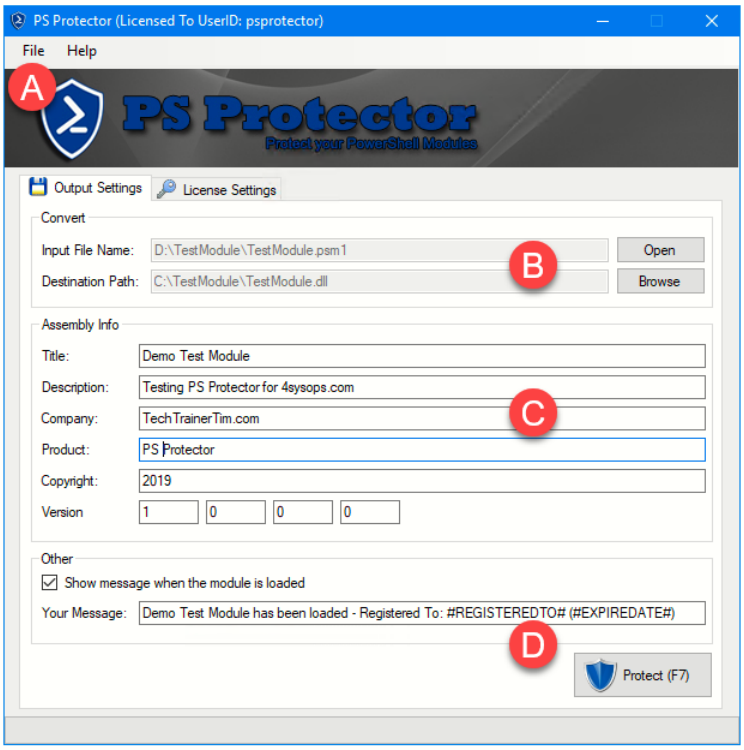
You will receive your own PS Protector credentials when you purchase a license (we'll discuss pricing at the end of this product review). Note that signing into PS Protector is mandatory; if you don't have an internet connection, or if the PS Protector web API is unavailable, you'll see the error shown in the next screenshot.



You need to be online to use PS Protector

Incidentally, the difficult-to-read text in the previous screenshot says **Service Status: Offline - Please try again later**.

Okay—now it's time to protect our test module. Fill out the **Output Settings** form to get started; notice the next screenshot, and then I'll explain the major configuration options.



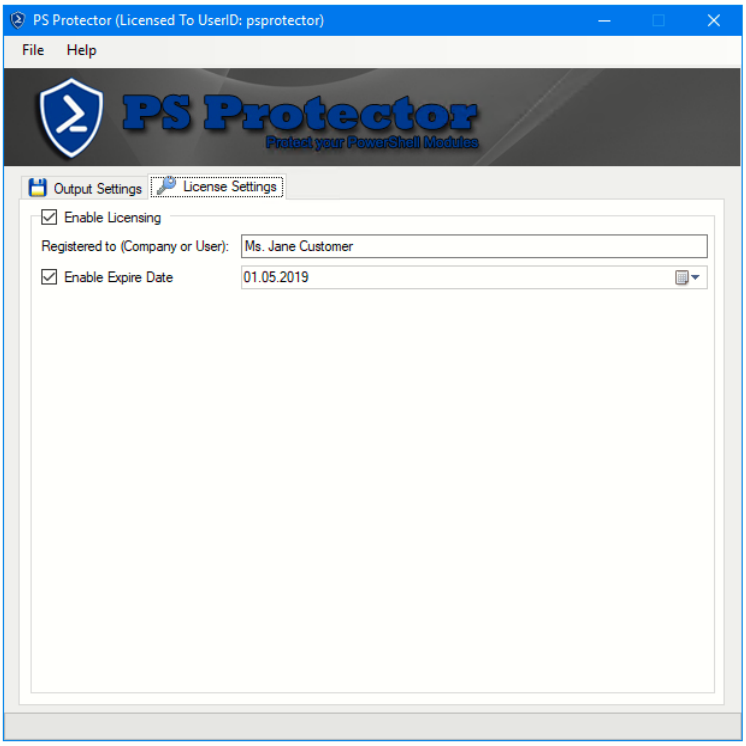
Protect a PowerShell module

- **A:** You can save your work as a project file to make protecting the same module for different customers easier.
- **B:** The input file needs to be a .psm1 PowerShell module; the output file is a .dll for which you provide a name and location.
- **C:** This is metadata attached to your new protected assembly.
- **D:** You can display a customized message when a user or customer imports the module.

PS Protector also provides command line support. You can pass all information as command line arguments to fully automate the creation of the assembly. In case of success or errors error codes are returned.

The PS Protector FAQ offers information how the tool protects the assembly against the use of .NET Decompilers such as Jetbrains dotPeek, Redgate .NET Reflector and ILSpy. However, the company provides no details how the code is encrypted.

Anyway, you can optionally include licensing information, as shown in the next interface screenshot.



Add license details for your customer

The idea here is you can put a license timeframe and personalization when you sell your protected assembly to customers. Well, let's test!

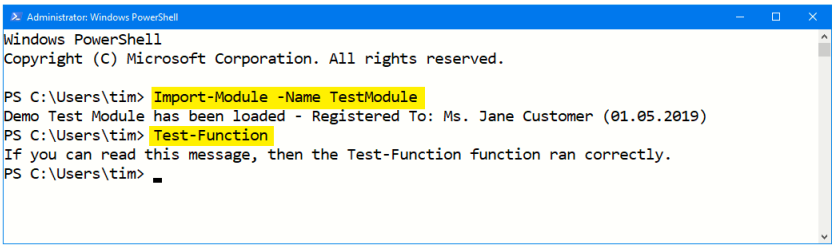
Testing the module protection

Put your new .dll and any related assets into a folder, and place that folder in a known PowerShell module path. To get these paths in Windows 10, run the following statement from an elevated PowerShell session:

```
$env:PSModulePath -split (';')
```

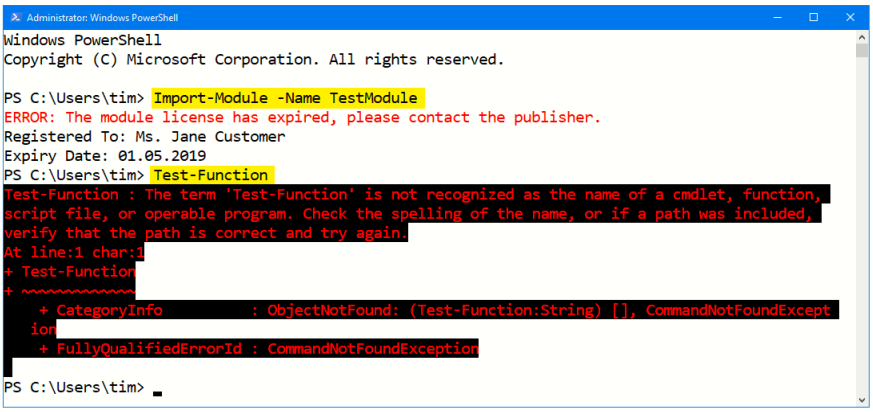
```
C:\Users\tim\Documents\WindowsPowerShell\Modules

C:\Program Files\WindowsPowerShell\Modules

C:\windows\system32\WindowsPowerShell\v1.0\Modules
```

You can then run *Import-Module* to load the assembly's contents into your runspace. For example, you can see in the next figure, I successfully imported my test module and ran its exported test function.



Test the protected module

Note also that PS Protector lists the licensee and expiration date because I chose those options during the protection operation. If users attempt to access the protected assembly after the license period expires, they see output shown in the next screenshot.



Test an expired license

Pricing and wrap-up

A PS Protector single-developer license costs approximately $55 USD. The subscription-based license is valid for one year, after which your protected modules continue to function. However, you cannot use the tool to apply protection to any more PowerShell modules until you renew.

PS Protector offers you free upgrades and technical support over your one-year license term.

In summary, the tool is simple to use and appears to work as advertised. I suppose PS Protector is a good fit for businesses whose business model, compliance regulations, and security policies require obfuscated source code.

My suggestion to the developer is to flesh out his website to explain how PS Protector works and make the trial product request involve user account creation (I found it unintuitive to have to fetch the shared trial credentials from their website).

Also, PowerShell is cross-platform now, so it would be nice if PS Protector took .NET Core and non-Windows environments into account.

However, give PS Protector a whirl and see if it suits your use case. The software has a competitive price given how much tedious development effort might otherwise be necessary to obfuscate your PowerShell source code, especially if you aren't a PowerShell developer.