The Evolution of Offensive PowerShell Invocation

By By Lee Christensen

Published: 2015-12-28 · Archived: 2026-04-05 21:34:58 UTC

By now, PowerShell should be in every offensive security person's arsenal. There are a plethora of PowerShell projects now that penetration testers and red teams can use when testing Windows networks. For privilege escalation, we have PowerUp. For Active Directory enumeration and exploitation, we have PowerView. Want to quickly run Mimkatz in memory on a remote box? Invoke-Mimikatz to the rescue. Need some creative comm channels? No problem, PowerShell can help there too. We even have fully-featured remote access tools running written completely in PowerShell! And given Microsoft's push to make everything administrable via PowerShell, I think it's safe to say PowerShell isn't going anywhere anytime soon.

Weaponization of PowerShell begins by invoking a PowerShell script in some way. Traditionally, offensive people use some combination of the following commands to execute PowerShell on a host:

- Execute a script directly: powershell.exe -ExecutionPolicy Bypass -File script.ps1
- Execute a script via the command line:
    - powershell.exe -ExecutionPolicy Bypass -C <command>
    - powershell.exe -ExecutionPolicy Bypass -EncodedCommand <Base64 Encoded Command>

While these techniques work in most cases, skilled defenders can easily detect these methods through process creation monitoring. There are a few ways to get around the need to spawn powershell.exe to execute your PowerShell. At its core, powershell.exe loads the System.Management.Automation assembly and uses the functions therein to invoke PowerShell commands. Likewise, you too can create your own C# application that references the Automation assembly and then uses the Automation assembly's functions to execute your PowerShell (see SharpPick for a great example). This, however, requires you to upload a custom application to a target's machine (an action which is not always ideal).

To get around this, earlier this year I developed UnmanagedPowerShell. UnmanagedPowerShell is a C++ project that starts up the .NET framework and loads (in memory) a custom C# assembly that executes PowerShell. This is useful because, assuming you already have some form of code execution on the box, then you can use this code to inject PowerShell into any process you'd like, eliminating the need to spawn powershell.exe. Public examples of this can be seen in ReflectivePick and PowerShell Empire's psinject command (which uses ReflectivePick underneath).

While UnmanagedPowerShell removed the need to initially start powershell.exe, it is not exactly ideal because it is single threaded and therefore cannot handle executing multiple PowerShell scripts at once. To get around this, you could use PowerShell jobs, but those have the unfortunate side effect of executing each job in a new instance of powershell.exe. You could load up UnmanagedPowerShell/ReflectivePick each time you want to execute PowerShell, but this has the potential downsides of excessive network traffic and process injection. You could also solve this by creating a new PowerShell runspace for each job. This may also lead to problems, however, due to

PowerShell scripts in different runspaces trying to PInvoke the same functions (If you have multiple runspaces, they all execute in the same app domain. Any PInvoke'd functions are bound to the AppDomain as a whole, not just in the current runspace).

To solve these problems, I developed the library PowerShellTasking. PowerShellTasking solves these problems by executing each new PowerShell script in a new AppDomain (an isolation mechanism in the .NET framework). Each PowerShell script runs in the background in its own AppDomain until completion (solving the single-threaded only problem), upon which the main thread receives any output and then unloads the AppDomain (preventing the PInvoke-ing problems). PowerShellTasking is compiled as a class library and anyone can easily use it by referencing the library in their own project (see the "Example" program in the GitHub project for an example). While PowerShellTasking will not prevent scripts that use PowerShell jobs from spawning powershell.exe, it does provide operators a safer way (from an opsec perspective) to execute multiple PowerShell scripts simultaneously on a host.

Hopefully this gives you a better idea of how PowerShell invocation has evolved over time and gives you some insights into the challenges of PowerShell weaponization. For resources on how to detect and protect yourself from malicious PowerShell usage in your network, see Lee Holmes's excellent blog post and DerbyCon talk.
