Constrained Language Mode

The ideal PowerShell malware mitigation.

Constrained Language Mode

- Goal: Enable users to use most PowerShell language features and only execute functions/cmdlets approved per policy*. Prevent the use of PowerShell to achieve arbitrary, unsigned code execution.
- `Get-help about_Language_Modes`
- Enforcement mechanisms (PSv5):
  - AppLocker
  - Device Guard
  - Remoting Session Configuration/Just Enough Administration (JEA)
  - __PSLockdownPolicy = 4 (not recommended in production)
- Anything approved to execute per policy runs in FullLanguage mode.
- The ideal method of mitigating against PowerShell malware!!!

Constrained Language Mode - Exercise

- Let’s figure out one of the many reasons setting __PSLockdownPolicy is not a durable defense.

- Search for “__PSLockdownPolicy” in the PowerShell source or in dnSpy and observe the code that makes an enforcement determinations.

- Is there a code path that will “fail-open” under an attacker-controllable condition?

Constrained Language Mode - Setup

You have two options to play with constrained language mode: the real way and the bad way.

- Real way - Enforce Device Guard that permits only Windows-signed code to execute:
  - ConvertFrom-CIPolicy -XmlFilePath C:\Windows\schemas\CodeIntegrity\ExamplePolicies\DefaultWindows_Enforced.xml -BinaryFilePath C:\Windows\System32\CodeIntegrity\SIPolicy.p7b; # then reboot
- Bad way - Set the system "__PSLockdownPolicy" env var to 4.
  - This is acceptable if Device Guard isn't working for some reason.

Constrained Language Mode

- Imposes the following restrictions (non-exhaustive):
  1. Add-Type cannot be called.
  2. New-Object can only be called (and type conversion) on a small set of whitelisted objects:
     - `[PSObject].Assembly.GetType('System.Management.Automation.CoreTypes').GetField('Items', [Reflection.BindingFlags] 'NonPublic, Static').GetValue($null).Value.Keys.FullName | Sort-Object -Unique`
  3. The only .NET method that can be called on non-whitelisted types is ToString().
  4. .NET property setters are not allowed. Property getters are allowed.
  5. Instantiation of a fixed set of COM objects is allowed*

Constrained Language Mode

- Extremely effective in preventing malicious PowerShell!
- But…
- Approved code runs in FullLanguage mode and may be vulnerable to injection.
- For example, a good amount of MS-signed code calls Add-Type. An attacker successfully influencing what's passed to Add-Type can bypass constrained language mode.
- Constrained language mode bypasses qualify for CVEs! Report them to secure@microsoft.com!

This would be a good time to

take a break and attempt

Lab: Constrained Language Mode

Constrained Language Mode - Injection Hunting

Lee Holmes wrote an amazing PSScriptAnalyzer plugin to automate the process of finding potentially injectable code - InjectionHunter

```
Install-Module -Name InjectionHunter

ls C:\* -Include '*.ps1', '*.psm1' -Recurse | % { Invoke-ScriptAnalyzer -Path $_.FullName -CustomizedRulePath (Get-Module -ListAvailable -Name InjectionHunter).Path -ExcludeRule PS* }
```

Windows PowerShell

```
PS C:\Users\Anon-OSX\Desktop> $Results = ls C:\Windows\diagnostics\* -Include '*.ps1', '*.psm1' -Recurse
| % { Invoke-ScriptAnalyzer -Path $_.FullName -CustomizedRulePath (Get-Module -ListAvailable -Name Inject
ionHunter).Path -ExcludeRule PS* }
PS C:\Users\Anon-OSX\Desktop> $Results | Group-Object RuleName

Count Name                      Group
----- ----                      -----
   21 InjectionRisk.StaticPr... {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...
   19 InjectionRisk.AddType     {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...
    5 InjectionRisk.InvokeEx... {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...
    2 InjectionRisk.ForeachO... {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...


PS C:\Users\Anon-OSX\Desktop>
```

Constrained Language Mode

- Having reported many CLM bypasses, the PowerShell team not only addresses injection vulns, but is also eliminating exploitation primitives:
  1. You cannot dot-source unapproved code. Import-Module is allowed though.
  2. If a module manifest (PSD1) is included, it must also be signed (and approved per policy).
  3. Module components are only exposed when explicitly exported in a module manifest or by calling Export-ModuleManifest.
  4. $PSDefaultParameterValues doesn't apply to full language mode from CLM.

Constrained Language Mode

- The PowerShell team is aware of other injection primitives (not all) and they are working to address some. An attacker must apply creativity to finding injection primitives – i.e. breaking scope assumptions.
  - Read-only global variable injection.
  - Attacker-controllable strings later interpreted as code – e.g. cast to scriptblock or passed to Add-Type
  - People doing stupid things. E.g. Add-Type wrapper function.
  - Making scope assumptions. E.g. referencing a variable set elsewhere that isn't explicitly scoped (like script scope)

Constrained Language Mode - Conclusion

- Specific CLM bypasses aside (which MSFT fixes), the most obvious CLM bypass involves running outdated PowerShell or PowerShell v2.
- PowerShell v2 doesn't implement constrained language mode.
- Unless you're running the latest version of PSv5.1 (at time of writing), you won't benefit from all CLM bypass mitigations.
- Lee Holmes' blog post is the authoritative reference for detecting and preventing PowerShell downgrade attacks.
  - http://www.leeholmes.com/blog/2017/03/17/detecting-and-preventing-powershell-downgrade-attacks/
  - tl;dr: detection - monitor the classic PowerShell event log for EngineVersion
  - prevention - use Device Guard to block all previous versions of System.Management.Automation.dll