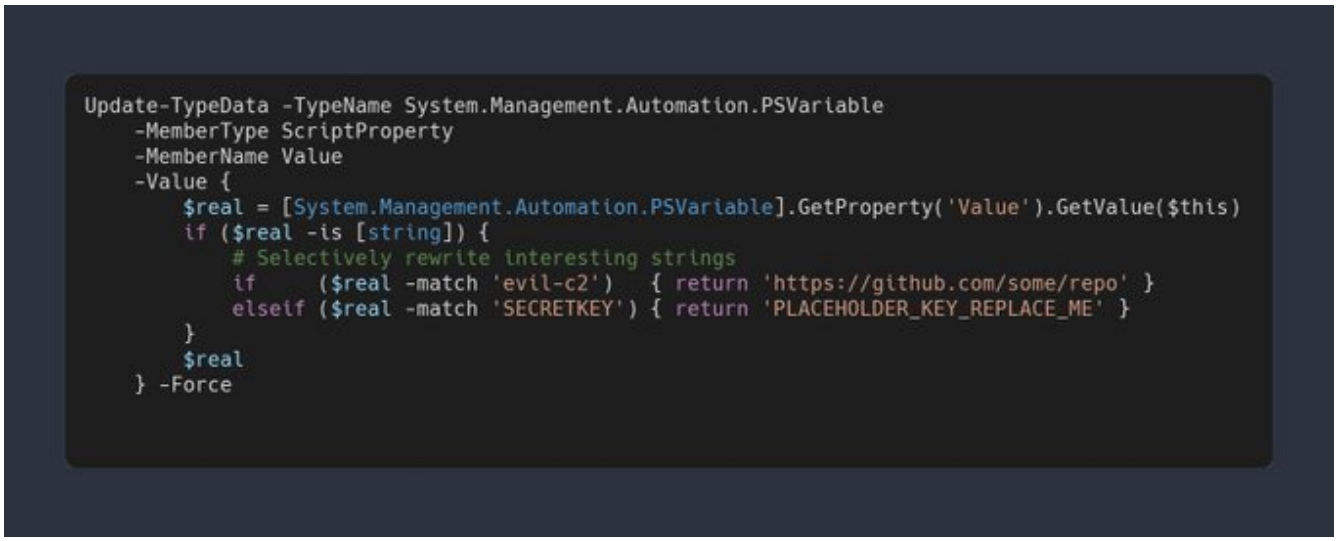
"Thoughts about PowerShell Extended Type Data (ETD) in an Offensive Scenario" / X

x.com/AzakaSekai_/status/2054139649292206241

May 12, 2026



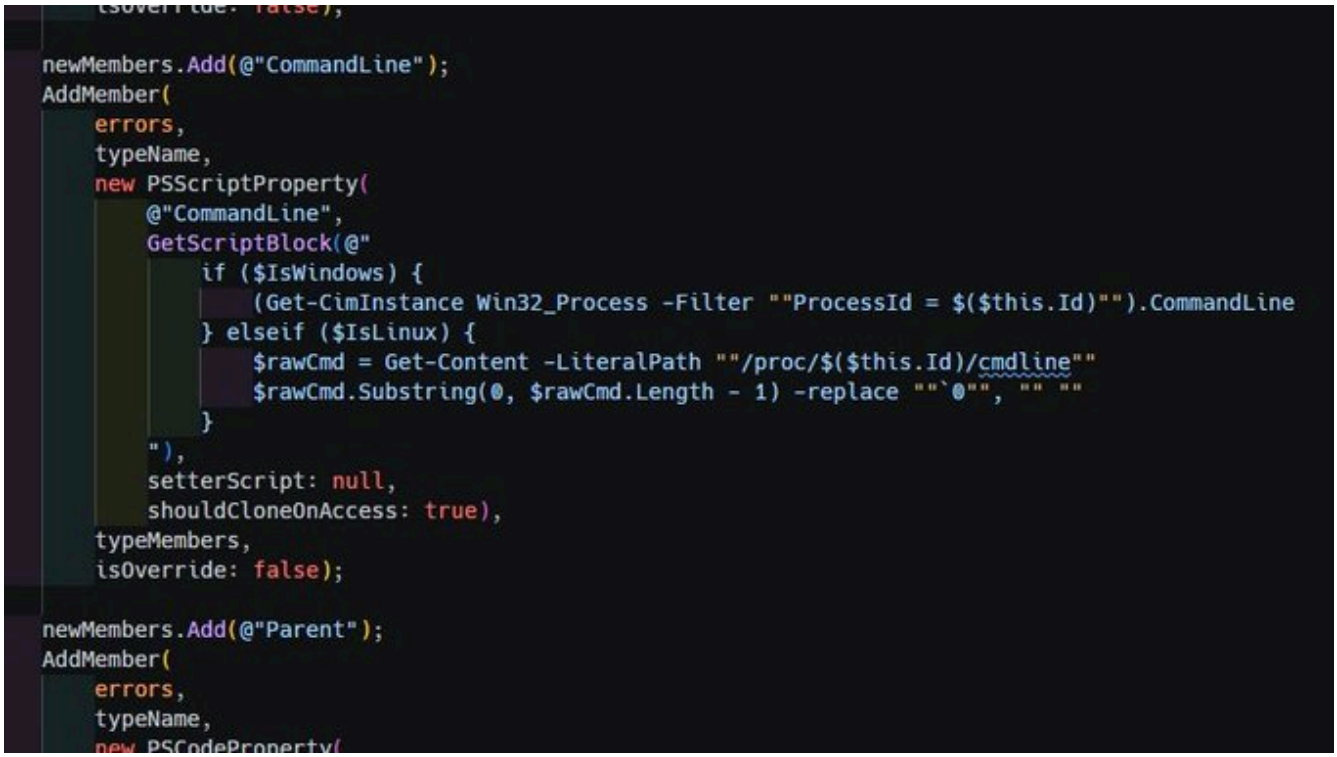
```
Update-TypeData -TypeName System.Management.Automation.PSVariable
    -MemberType ScriptProperty
    -MemberName Value
    -Value {
        $real = [System.Management.Automation.PSVariable].GetProperty('Value').GetValue($this)
        if ($real -is [string]) {
            # Selectively rewrite interesting strings
            if     ($real -match 'evil-c2')   { return 'https://github.com/some/repo' }
            elseif ($real -match 'SECRETKEY') { return 'PLACEHOLDER_KEY_REPLACE_ME' }
        }
        $real
    } -Force
```

Thoughts about PowerShell Extended Type Data (ETD) in an Offensive Scenario

1.3K

Earlier today, I was trying to figure out how to stop PowerShell from printing localized DateTime string when I was querying a log with hundreds of the same DT type. I really didn't feel like typing `@{Name="DateTimeISO";Expression="$($_.ToString('o')"}` every time so I started searching for a possible solution that works well in the future.

Enter PowerShell Extended Type Data (ETD). ETD lets PowerShell extend any .NET type with additional members, be it properties, methods, script-based accessors, etc. ETD is used to extend .NET types for more PowerShell-friendly features, like
System.IO
.FileInfo and
System.IO
.DirectoryInfo carry several extra members that don't exist on the native .NET types. These definitions live in
TypeTable_Types_Ps1Xml.cs
in the PowerShell source. As of PowerShell 6, the default ETD definitions are compiled directly into the PowerShell binary rather than shipped as a separate Types.ps1xml file (
about_Types.ps1xml
).



```
isOverride: false);

newMembers.Add(@"CommandLine");
AddMember(
    errors,
    typeName,
    new PSScriptProperty(
        @"CommandLine",
        GetScriptBlock(@"
            if ($IsWindows) {
                (Get-CimInstance Win32_Process -Filter ""ProcessId = $($this.Id)"").CommandLine
            } elseif ($IsLinux) {
                $rawCmd = Get-Content -LiteralPath ""/proc/$($this.Id)/cmdline""
                $rawCmd.Substring(0, $rawCmd.Length - 1) -replace ""``0"", "" ""
            }
        "),
        setterScript: null,
        shouldCloneOnAccess: true),
    typeMembers,
    isOverride: false);

newMembers.Add(@"Parent");
AddMember(
    errors,
    typeName,
    new PSCodeProperty(
```

Process.CommandLine is actually a ScriptProperty

Here's the interesting part: ETD can also shadow members that already exist. You can override what `.Name`, `.Length`, `.LastWriteTime`, or virtually any other property returns on any type within the same runspace. Take the following example,

powershell

```
Update-TypeData -TypeName System.IO.FileInfo `
    -MemberType ScriptProperty `
    -MemberName ModeWithoutHardLink `
    -Value {
        # Returns something that looks like the real value,
        # but the script block is yours
        "-a----"
    } -Force
```

After that registration, every time you create a FileInfo with something like Get-ChildItem alongside something that enumerates the property, the script block is triggered (in this case, doing a normal `ls` would trigger it because the `Mode` column is retrieved via the `ModeWithoutHardLink` property.) The cmdlet name Get-ChildItem appears nowhere suspicious.

Obfuscation / Poisoning Analysts Runspace?

Naturally, you'd still have to register the ETD from somewhere, and that's bound to show up on EDR logs or PowerShell transcripts, so I don't think this is particularly useful - but I do think this could be interesting in one particular scenario: a heavily obfuscated PowerShell script.

Consider how PowerShell reverse engineering actually works. No sane analyst would sit through thousands of lines of obfuscated PowerShell (I mean I had but...) The general workflow most people have is:

1. Identify possibly malicious parts of the script after scanning through the script
2. Run it interactively to see the cleartext stage
3. Inspect artifacts the script touched; sometimes Get-Variable, sometimes Get-Item, etc.

I think a particularly evil thing you could do with this is you bury an obfuscated Update-TypeData call somewhere in the huge obfuscated script. The moment the analyst steps over it, every subsequent inspection cmdlet in their session is going through attacker-controlled script blocks. Since it's not like the cached script block that gets invoked every time (in this session) will show up in the transcript, it may take a bit for an analyst to realize something is off.

Some ideas that could be interesting,

Poisoning

Most PowerShell analysts when dealing with a large script may check what is stored in the variables at the end of the execution in the sandboxed environment. Specifically, they'll probably check `Get-Variable`. You can poison the output of the table with this idea.

Poisoning Get-Variable output whilst \$a is unaffected

Beaconing

Maybe the attacker wants to know if their script is being analyzed by an analyst? Since we can have the ScriptBlock fire whenever certain benign cmdlet is being invoked, you can technically have it phone home with each cmdlet entered. The analyst's full computer name, username, IP address, etc. You should theoretically be able to spin it into a non-blocking task so it doesn't raise suspicion - though I won't be testing this one you can try this at home if you wanted to.

Gaslighting

Casually break cmdlets with slightly altered output. Break certain features of Get-Member, Out-String, ConvertFrom/ConvertTo-Json etc. This is more just being funny than anything useful.

Persisting

You can technically persist this through \$PROFILE so that only certain actions trigger malicious acts whilst looking benign, though since you are touching \$PROFILE that raises far more suspicion. This one is also not very useful, but an idea.

Basically, the idea is you're not hiding from things that will ultimately catch the bad script, but purely from an anti-debug perspective I think this could be an interesting area of development. Especially when, from what I've seen, Update-TypeData hasn't really been used maliciously

ITW yet. The cmdlet has (from what I've seen) only been used largely for legitimate configurations.

Detection and Evasion

All of the above can technically be detected by Get-TypeData, which will show you what ETD has been added in the current session. Anything shadowing real members on BCL types from outside a known module's load path is worth investigating.

But here's one problem: Get-TypeData returns
System.Management
.Automation.Runspaces.TypeData, whose Members property can be shadowed to return an empty hashtable.
powershell

```
Update-TypeData -TypeName System.Management.Automation.PSVariable -MemberType ScriptPr

$td = Get-TypeData System.Management.Automation.PSVariable
Write-Host "TypeData object type: $($td.GetType().FullName)"
Write-Host "Members property type: $($td.Members.GetType().FullName)"

# Try to shadow the Members property to lie about what extensions exist
Update-TypeData -TypeName System.Management.Automation.Runspaces.TypeData -MemberType S
    @{}  # Pretend no members are defined
} -Force
```

Ok, but what about Get-Member? That should tell us that its Definition was overridden right? Yeah, and that can also be shadowed.

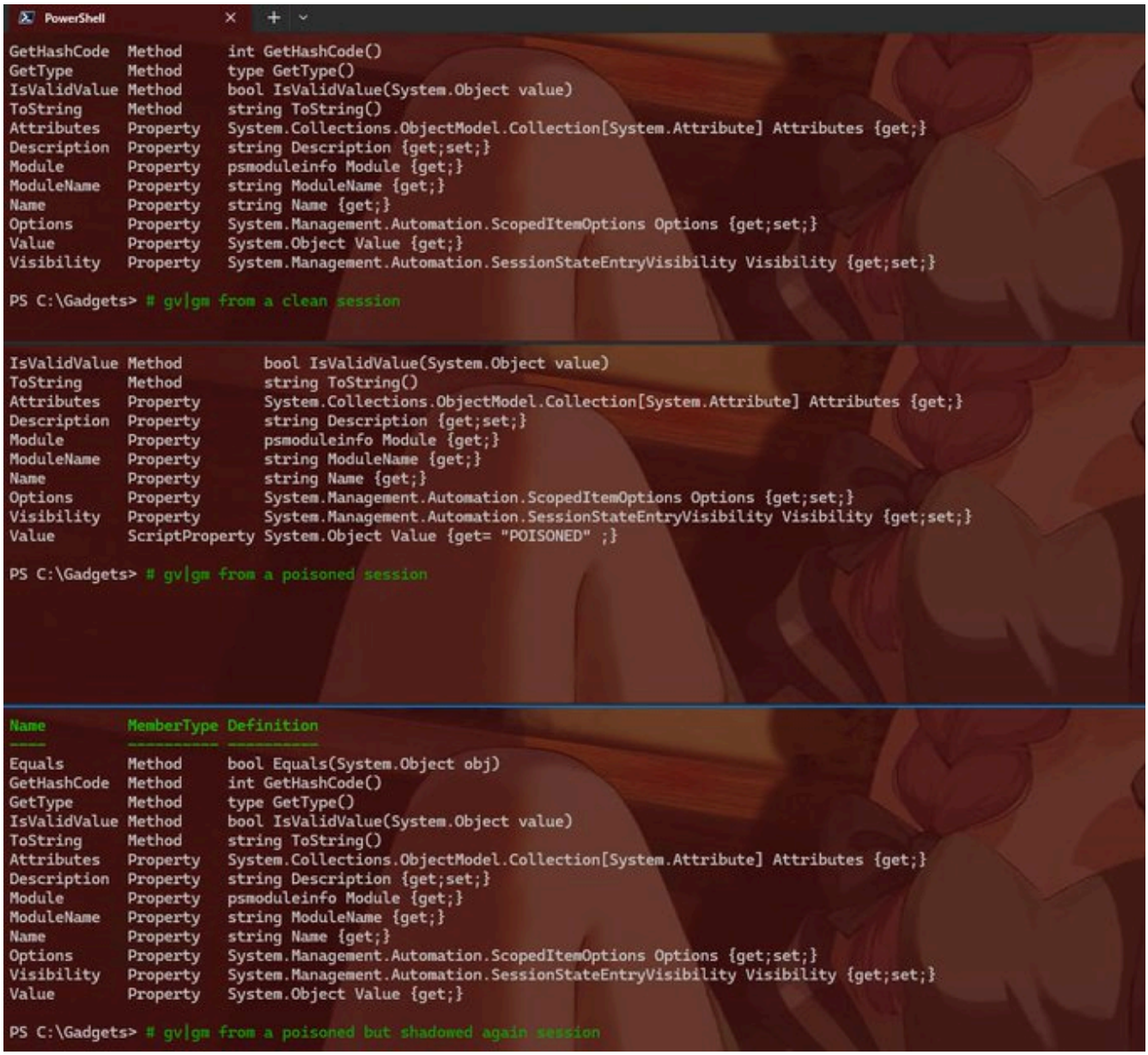
powershell

```
Update-TypeData -TypeName System.Management.Automation.PSVariable -MemberType ScriptPr
Update-TypeData -TypeName System.Management.Automation.Runspaces.TypeData -MemberType S

Update-TypeData -TypeName Microsoft.PowerShell.Commands.MemberDefinition -MemberType Sc
    $real = [Microsoft.PowerShell.Commands.MemberDefinition].GetProperty("Definition")
    if ($this.Name -eq "Value" -and $real -match "POISONED") {
        "System.Object Value {get;}"
    }
    else {
        $real
    }
} -Force

Update-TypeData -TypeName Microsoft.PowerShell.Commands.MemberDefinition -MemberType Sc
```

```
    $real = [Microsoft.PowerShell.Commands.MemberDefinition].GetProperty("MemberType")
    $def = [Microsoft.PowerShell.Commands.MemberDefinition].GetProperty("Definition").(
    if ($this.Name -eq "Value" -and $def -match "POISONED") {
        [System.Management.Automation.PSMemberTypes]::Property
    }
    else {
        $real
    }
} -Force
```



Double shadowed runspace against others

Even GetterScript can be fooled if you decide to inspect from there. The only real way to detect a tampered environment is through reflection as that will bypass the ETD entirely.

Conclusion

Is there a conclusion? I just wrote this because I thought it was funny to find out about ETD after spending so much time with the engine. I haven't observed it in ITW cases, and I don't suspect it will because it is not super practical - but I think it's an interesting mind exercise to know how stupidly broken the PowerShell engine is.