



Active Directory Security

Active Directory & Enterprise Security, Methods to Secure Active Directory, Attack Methods & Effective Defenses, PowerShell, Tech Notes, & Geek Trivia...

[Home](#)[About](#)[AD Resources](#)[Attack Defense & Detection](#)[Contact](#)[Mimikatz](#)[Presentations](#)[Schema Versions](#)[Security Resources](#)[SPNs](#)[Top Posts](#)[PowerShell Version 5 Security Enhancements](#)[Building an Effective Active Directory Lab Environment for Testing](#)

FEB

11

2016

Detecting Offensive PowerShell Attack Tools

By [Sean Metcalf](#) in [ActiveDirectorySecurity](#), [Malware](#), [Microsoft Security](#), [PowerShell](#), [Technical Reference](#)

At [DerbyCon V \(2015\)](#), I [presented on Active Directory Attack & Defense](#) and part of this included how to detect & defend against PowerShell attacks.

Update: I [presented at BSides Charm \(Baltimore\)](#) on PowerShell attack & defense in April 2016.

More information on PowerShell Security: [PowerShell Security: PowerShell Attack Tools, Mitigation, & Detection](#)

The most important take-away from this post: you want to log all PowerShell activity and get that data into a central logging system to monitor for suspicious and anomalous activity.

The Evolution of PowerShell as an attack tool

PowerShell is a built-in command shell available on every supported version of Microsoft Windows (Windows 7 / Windows 2008 R2 and newer) and provides incredible flexibility and functionality to manage Windows systems. This power makes PowerShell an enticing tool for attackers. Once an attacker can get code to run on a computer, they often invoke PowerShell code since it can be run in memory where antivirus can't see it. Attackers may also drop PowerShell script files (.ps1) to disk, but since PowerShell can download code from a website and run it in memory, that's often not necessary.

Dave Kennedy & Josh Kelley presented at DEF CON 18 (2010) on how PowerShell could be leveraged by attackers. Matt Graeber developed PowerSploit and blogged at [Exploit-Monday.com](https://exploit-monday.com) on why PowerShell is a great attack platform. Offensive PowerShell usage has been on the rise since the release of “PowerSploit” in 2012, though it wasn’t until Mimikatz was PowerShell-enabled (aka [Invoke-Mimikatz](#)) about a year later that PowerShell usage in attacks became more prevalent. PowerShell provides tremendous capability since it can run .Net code and execute dynamic code downloaded from another system (or the internet) and execute it in memory without ever touching disk. These features make PowerShell a preferred method for gaining and maintaining access to systems since they can move around using PowerShell without being seen. PowerShell Version 5 (v5) greatly improves the defensive posture of PowerShell and when run on a Windows 10 system, PowerShell attack capability is greatly reduced.

PowerShell is more than PowerShell.exe

Blocking access to PowerShell.exe is an “easy” way to stop PowerShell capability, at least that’s how it seems. The reality is that PowerShell is more than a single executable. PowerShell exists in the System.Management.Automation.dll dynamic linked library file (DLL) and can host different runspaces which are effectively PowerShell instances. A custom PowerShell runspace can be instantiated via code, so PowerShell can be executed through a custom coded executable (such as MyPowershell.exe). In fact there are several current methods of running PowerShell code without Powershell.exe being executed. Justin Warner (@SixDub) blogged about [bypassing PowerShell.exe on Red Team engagements in late 2014, aka PowerPick](#)). Since PowerShell code can be executed without running PowerShell.exe, blocking this executable is not an ideal solution to block attacks.

```
PS C:\temp> $PS = [PowerShell]::Create()
$PS.AddCommand("Get-Process")
$PS.Invoke()

Commands           : System.Management.Automation.PSCommand
Streams            : System.Management.Automation.PSDataStreams
InstanceId          : 57ef9f1e-be3a-43a1-a7ed-cec4e9177c76
InvocationStateInfo : System.Management.Automation.PSInvocationStateInfo
IsNested           : False
HadErrors          : False
Runspace           : System.Management.Automation.Runspaces.LocalRunspace
RunspacePool       :
IsRunspaceOwner    : True
HistoryString      :

Id      : 396
Handles : 373
CPU     :
Name    : csrss

Id      : 456
Handles : 192
CPU     :
Name    : csrss

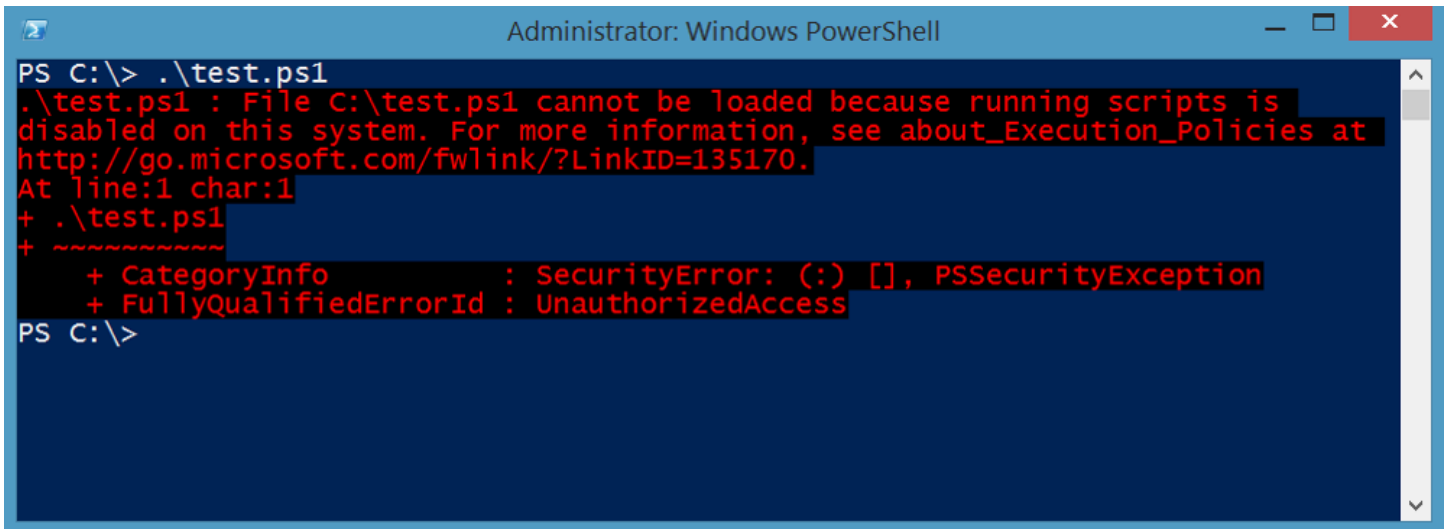
Id      : 2064
Handles : 71
CPU     : 0.015625
Name    : dwm
```

Blocking PowerShell.exe does not stop PowerShell attacks.

Since blocking PowerShell is not effective in stopping PowerShell-based attacks, a more nuanced approach is required (see PowerShell Constrained Language mode and PowerShell v5).

PowerShell Execution Policies aren't about Security (not really)

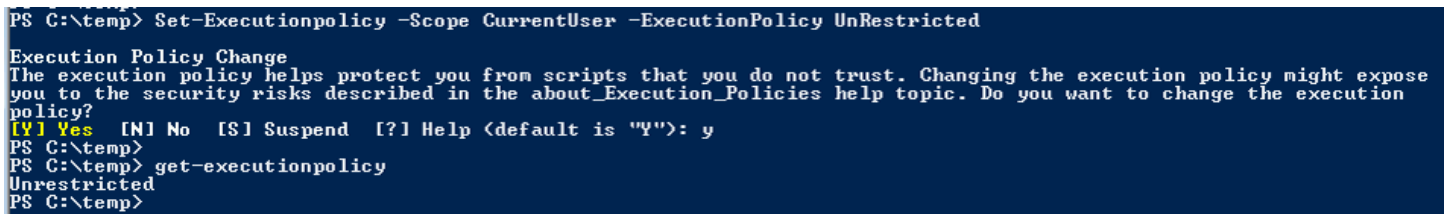
The PowerShell Execution Policy is set to restricted by default so .PS1 files don't auto-execute. Microsoft learned their lesson about allowing dynamic code to easily execute on Windows. This means all script execution is disabled by default, though one can still type in the commands by hand.



```
Administrator: Windows PowerShell
PS C:\> .\test.ps1
.\test.ps1 : File C:\test.ps1 cannot be loaded because running scripts is disabled on this system. For more information, see about_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\test.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\>
```

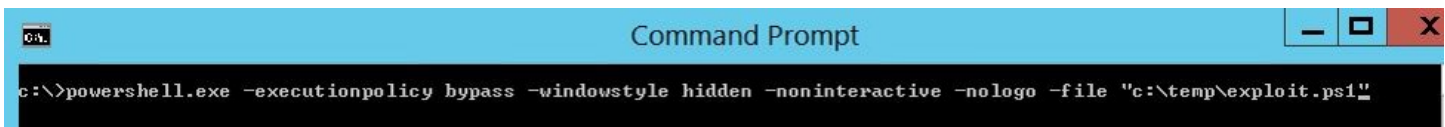
“The PowerShell Execution Policy is not a security boundary”

Bypassing the PowerShell Execution Policy is as easy as asking.



```
PS C:\temp> Set-Executionpolicy -Scope CurrentUser -ExecutionPolicy UnRestricted
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic. Do you want to change the execution policy?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y
PS C:\temp>
PS C:\temp> get-executionpolicy
Unrestricted
PS C:\temp>
```

PowerShell.exe can be instantiated with no execution policy to run the script of choice.



```
Command Prompt
c:\>powershell.exe -executionpolicy bypass -windowstyle hidden -noninteractive -nologo -file "c:\temp\exploit.ps1"
```

PowerShell as an Attack Platform

The rise of PowerShell as an attack platform has caught a number of organizations by surprise since traditional defenses aren't able to mitigate or even stop them from being successful. The use of PowerShell by an attacker is as a “post-exploitation” tool; the malicious PowerShell code is being run since the attacker has access to run code on a system already. In some attacks a user was tricked into opening/executing a file or through exploiting vulnerability. Regardless, once an attacker has access, all the operating system standard tools and utilities are available.

Expect at some point that an attacker will get their code running on a computer in your environment, how are you aligning defenses for this new reality?

Powershell is an ideal platform for attackers:

- Run code in memory without touching disk
- Download & execute code from another system
- Interface with .Net & Windows APIs
- Most organizations are not watching PowerShell activity
- CMD.exe is commonly blocked, though not PowerShell

PowerShell is often leveraged as part of an attack, often initially targeting a user on a user's workstation.

PowerShell attack code can be invoked by:

- Microsoft Office Macro (VBA)
- WMI
- HTA Script (HTML Application – control panel extensions)
- CHM (compiled HTML help)
- Java JAR file
- Other script type (VBS/WSH/BAT/CMD)
- Typically an Encoded Command

Encoded commands obfuscate attack code and can even be compressed to avoid the Windows console character limitation (8191).

Powershell.exe –WindowStyle Hidden –nopprofile –EncodedCommand <BASE64ENCODED>

Limiting PowerShell Attack Capability with Constrained Language Mode

Additionally, PowerShell supports various language modes that restrict what PowerShell can do. The PowerShell Constrained Language Mode was developed to support the Surface RT tablet device, though this mode is available in PowerShell in standard Windows as well. Constrained language mode limits the capability of PowerShell to base functionality removing advanced feature support such as .Net & Windows API calls and COM access. The lack of this advanced functionality stops most PowerShell attack tools since they rely on these methods. The drawback to this approach is that in order to configured PowerShell to run in constrained mode, an environment variable must be set, either by running a command in PowerShell or via Group Policy.

Constrained language mode is a useful interim PowerShell security measure and can mitigate many initial PowerShell attacks, though it is not a panacea. It should be considered minor mitigation method on roadmap to whitelisting. Keep in mind that bypassing Constrained PowerShell is possible and not all PowerShell “attack scripts” will be blocked – certainly the ones that use advanced functionality to reflectively load a DLL into memory like Invoke-Mimikatz will be blocked.

Enable Constrained Language Mode:

```
[Environment]::SetEnvironmentVariable('__PSLockdownPolicy', '4', 'Machine')
```

Enable via Group Policy:

Computer Configuration\Preferences\Windows Settings\Environment

Once Constrained Language Mode is enabled, many PowerShell attack tools don't work since they rely on components blocked by constrained language.

This environment variable can be modified by an attacker once they have gained control of the system. Note that they would have to spawn a new PowerShell instance to run code in full language mode after changing the environment. These changes would be logged and could help the defender in identifying unusual activity on the system.

Remove Constrained Language Mode:

Remove-Item Env:__PSLockdownPolicy

Check Language Mode:

\$ExecutionContext.SessionState.LanguageMode

Enabling PowerShell Constrained Language mode is another method that can be used to mitigate PowerShell attacks.

Pairing PowerShell v5 with AppLocker – Constrained Language Mode No Longer Easily Bypassed.

PowerShell v5 also supports automatic lock-down when AppLocker is deployed in “Allow” mode. Applocker Allow mode is true whitelisting and can prevent any unauthorized binary from being executed. PowerShell v5 detects when Applocker Allow mode is in effect and sets the PowerShell language to Constrained Mode, severely limiting the attack surface on the system. With Applocker in Allow mode and PowerShell running in Constrained Mode, it is not possible for an attacker to change the PowerShell language mode to full in order to run attack tools. When AppLocker is configured in “Allow Mode”, PowerShell reduces its functionality to “Constrained Mode” for interactive input and user-authored scripts. Constrained PowerShell only allows core PowerShell functionality and prevents execution of the extended language features often used by offensive PowerShell tools (direct .NET scripting, invocation of Win32 APIs via the Add-Type cmdlet, and interaction with COM objects).

Note that scripts allowed by AppLocker policy such as enterprise signed code or in a trusted directory are executed in full PowerShell mode and not the Constrained PowerShell environment. This can't be easily bypassed by an attacker, even with admin rights.

Log all PowerShell Activity

Detection of PowerShell attack activity on your network (including [PowerShell Empire](#) and [PowerSploit](#)) begins with logging PowerShell activity. Enabling PowerShell logging requires PowerShell v3 and newer and PowerShell v4 adds some additional log detail (Windows 2012 R2 & Windows 8.1 with November 2014 roll-up [KB300850](#)) useful for discovering and tracking attack activity.

PowerShell logging can be enabled via Group Policy for PowerShell modules:

- Microsoft.PowerShell.* – Log most of PowerShell's core capability.
- ActiveDirectory – Log Active Directory cmdlet use.
- BITSTransfer – Logs use of BITS cmdlets.
- CimCmdlets (2012R2/8.1) – Logs cmdlets that interface with CIM.
- GroupPolicy – Log Group Policy cmdlet use.
- Microsoft.WSMan.Management – Logs cmdlets that manage Web Services for Management (WS-Management) and Windows Remote Management (WinRM).
- NetAdapter/NetConnection – Logs Network related cmdlets.

- PSScheduledJob/ScheduledTasks (PSv5) – Logs cmdlets to manage scheduled jobs.
- ServerManager – Log Server Manager cmdlet use.
- SmbShare – Log SMB sharing activity.

PowerShell logging can also be configured for all PowerShell modules (“*”), my preference, which logs all PowerShell cmdlets – useful if the attacker has imported custom module for offensive PowerShell tools.

Group Policy:

Computer Configuration\Policies\Administrative Template\Windows Components\Windows PowerShell

In order for these logs to be useful, they need to be fed into a central logging system with alerts configured for known attack methods.

Interesting Activity:

- Downloads via .Net □ (New-Object Net.WebClient).DownloadString)
- Invoke-Expression (& derivatives: “iex”).
- BITS activity

- Scheduled Task creation/deletion.
- PowerShell Remoting.

This screenshot shows how many PowerShell attacks start: Invoke-Expression (IEX) which calls the .Net Web Client download functionality to download internet PowerShell code and execute it.

PowerShell v5 provides enhanced security and improved logging.

PowerShell v5 Script Block Logging

Script block logging provides the ability to log de-obfuscated PowerShell code to the event log. Most attack tools are obfuscated, often using Base64 encoding, before execution to make it more difficult to detect or identify what code actually ran. Script block logging logs the actual code delivered to the PowerShell engine before execution which is possible since the script code needs to be de-obfuscated before execution.

Since many PowerShell attacks obfuscate the attack code, it is difficult to identify what the script code does. Script block logging de-obfuscates the code and logs the code that is executed. Since this code is logged, it can be alerted on when seen by a central logging system.

One key challenge with identifying offensive PowerShell code is that most of the time it is obfuscated (Base64, Base64+XOR, etc). This makes real-time analysis nearly impossible since there is no keyword to trigger alerts on.

Script Block Logging records the content of the script blocks it processes as well as the generated script code at execution time.

Microsoft-provided example of obfuscated command code

```
## Malware
function SuperDecrypt
{
param($script)
$bytes = [Convert]::FromBase64String($script)
## XOR “encryption”
$xorKey = 0x42
for($counter = 0; $counter -lt $bytes.Length; $counter++)
{
$bytes[$counter] = $bytes[$counter] -bxor $xorKey
}
[System.Text.Encoding]::Unicode.GetString($bytes)
}
$decrypted = SuperDecrypt
“FUlwQitCNklnQm9CCkltQjFCNkJiQmVCEkl1QixCJkJlQg==”
Invoke-Expression $decrypted
```

The original script block passed to PowerShell for processing is logged (what you see above) as well as the actual command that PowerShell executed.

Note that Script Block Logging is enabled by default.

PowerShell v5 System-wide transcripts

System-wide transcripting can be enabled via Group Policy and provides an “over the shoulder” transcript file of every PowerShell command and code block executed on a system by every user on that system. This transcript can be directed to a write-only share on the network for later analysis and SIEM tool ingesting.

PowerShell has the ability to save text written to the console (screen) in a “transcript” file which requires the user (or script) to run “start-transcript \$FileName”. This provides a simple script log file. The drawback to this method is that only one transcript could be active at a time, the PowerShell ISE editor didn’t support transcripts, and that Start-Transcript would have to be added to every user’s PowerShell profile in order for a record of run commands to be saved.

System-wide transcripts provides a simple method to write all PowerShell commands (including those run inside scripts or downloaded from another location) into a computer-specific transcript file that is stored in a network share. This enables rapid analysis of near-real time PowerShell usage as well as identification of “known-bad” activity.

The system-wide transcript can be enabled via Group Policy and includes the following information in the header:

- Start time
- User Name
- RunAs User
- Machine (Operating System)
- Host Application
- Process ID

Parameters:

- IncludeInvocationHeader – includes start time headers for every command run.
- OutputDirectory – enables writing transcripts to a central location, such as a network share.

Enabling Script Block Logging and log script block invocation header via GPO:

Microsoft-provided example PowerShell script to configure ACL on the central transcripts share:

```
md c:\Transcripts
```

Kill all inherited permissions

\$acl = Get-Acl c:\Transcripts

\$acl.SetAccessRuleProtection(\$true, \$false)

Grant Administrators full control

\$administrators = [System.Security.Principal.NTAccount]

“Administrators”

\$permission =

\$administrators, “FullControl”, “ObjectInherit, ContainerInherit”, “None”, “Allow”

\$accessRule = New-Object

System.Security.AccessControl.FileSystemAccessRule

\$permission

\$acl.AddAccessRule(\$accessRule)

Grant everyone else Write and ReadAttributes. This prevents users from listing

transcripts from other machines on the domain.

\$everyone = [System.Security.Principal.NTAccount] “Everyone”

\$permission =

\$everyone, “Write, ReadAttributes”, “ObjectInherit, ContainerInherit”, “None”, “Allow”

\$accessRule = New-Object

System.Security.AccessControl.FileSystemAccessRule

\$permission

\$acl.AddAccessRule(\$accessRule)

```
## Deny "Creator Owner" everything. This prevents users from
## viewing the content of previously written files.
$creatorOwner = [System.Security.Principal.NTAccount] "Creator
Owner"
$permission =
$creatorOwner,"FullControl","ObjectInherit,ContainerInherit","Inheri
tOnly","Deny"
$accessRule = New-Object
System.Security.AccessControl.FileSystemAccessRule
$permission
$acl.AddAccessRule($accessRule)
```

```
## Set the ACL
$acl | Set-Acl c:\Transcripts\
```

```
## Create the SMB Share, granting Everyone the right to read and
write files. Specific
## actions will actually be enforced by the ACL on the file folder.
New-SmbShare -Name Transcripts -Path c:\Transcripts -
ChangeAccess Everyone
```

Turn on System-wide Transcription via Group Policy:

Windows Components -> Administrative Templates -> Windows PowerShell -> Turn on PowerShell Transcription

This Group Policy setting configures the registry key:

HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription

Invoke-Mimikatz Capability

Read more about Mimikatz & Invoke-Mimikatz on the ADSecurity.org [Unofficial Guide to Mimikatz & Command Reference](#).

The majority of Mimikatz functionality is available in [PowerSploit](#) (PowerShell Post-Exploitation Framework) through the “[Invoke-Mimikatz](#)” PowerShell script (written by [Joseph Bialek](#)) which “leverages Mimikatz 2.0 and Invoke-ReflectivePEInjection to reflectively load Mimikatz completely in memory. This allows you to do things such as dump credentials without ever writing the Mimikatz binary to disk.” Note that the PowerSploit framework is now hosted in the “[PowerShellMafia](#)” [GitHub repository](#).

What gives Invoke-Mimikatz its “magic” is the ability to reflectively load the Mimikatz DLL (embedded in the script) into memory. The Invoke-Mimikatz code can be downloaded from the Internet (or intranet server), and executed from memory without anything touching disk. Furthermore, if Invoke-Mimikatz is run with the appropriate rights and the target computer has PowerShell Remoting enabled, it can pull credentials from other systems, as well as execute the standard Mimikatz commands remotely, without files being dropped on the remote system.

Invoke-Mimikatz is not updated when Mimikatz is, though it can be (manually). One can swap out the DLL encoded elements (32bit & 64bit versions) with newer ones. Will Schroeder (@HarmJ0y) has [information on updating the Mimikatz DLLs in Invoke-Mimikatz](#) (it’s not a very complicated process). The [PowerShell Empire version of Invoke-Mimikatz](#) is usually kept up to date.

- Use mimikatz to dump credentials out of LSASS: *Invoke-Mimikatz -DumpCreds*
- Use mimikatz to export all private certificates (even if they are marked non-exportable): *Invoke-Mimikatz -DumpCerts*
- Elevate privilege to have debug rights on remote computer: *Invoke-Mimikatz -Command “privilege::debug exit” -ComputerName “computer1”*

The Invoke-Mimikatz “Command” parameter enables Invoke-Mimikatz to run custom Mimikatz commands.

Defenders should expect that any functionality included in Mimikatz is available in Invoke-Mimikatz.

Detecting Invoke-Mimikatz & Other PowerShell Attack Tools?

Most advice on how to detect attack tools like Invoke-Mimikatz involves tracking the wrong “signature” type words/phrases (this is often the AV approach) in order to have a high success/ low false positive rate. A nice goal, but not a great approach.

These “signatures” often include:

- “mimikatz”
- “gentilkiwi”
- “Invoke-Mimikatz”

The issue is that this is PowerShell code, so it’s trivial to open in notepad and modify these signatures.

So, what does this mean?

The best method to detect PowerShell attack code is to look for key indicators – code snippets required for the code to run correctly.

Invoke-Mimikatz Event Log Keywords:

- “System.Reflection.AssemblyName”
- “System.Reflection.Emit.AssemblyBuilderAccess “
- “System.Runtime.InteropServices.MarshalAsAttribute”
- “TOKEN_PRIVILEGES”
- “SE_PRIVILEGE_ENABLED“

Searching the PowerShell Operational Log for “System.Reflection’ returns lots of events after a PowerShell attack tool like Invoke-Mimikatz is run:

Searching the PowerShell Operational Log for “TOKEN.PRIVILEGES” also returns several events after a PowerShell attack tool like Invoke-Mimikatz is run:

PowerShell Attack Tool Detection

Many PowerShell attack tools can be detected by monitoring PowerShell Operational log for the following indicators. These are specific to PowerSploit tools, but many other PowerShell attack tools use the same methods.

Invoke-TokenManipulation:

- “TOKEN_IMPERSONATE”
- “TOKEN_DUPLICATE”
- “TOKEN_ADJUST_PRIVILEGES”

Invoke-CredentialInjection:

- “TOKEN_PRIVILEGES”
- “GetDelegateForFunctionPointer”

Invoke-DLLInjection

- “System.Reflection.AssemblyName”
- “System.Reflection.Emit.AssemblyBuilderAccess”

Invoke-Shellcode

- “System.Reflection.AssemblyName”
- “System.Reflection.Emit.AssemblyBuilderAccess”
- “System.MulticastDelegate”
- “System.Reflection.CallingConventions”

Get-GPPPassword

- “System.Security.Cryptography.AesCryptoServiceProvider”
- “0x4e,0x99,0x06,0xe8,0xfc,0xb6,0x6c,0xc9,0xfa,0xf4”
- “Groups.User.Properties.cpassword”

- “ScheduledTasks.Task.Properties.cpassword”

Out-MiniDump

- “System.Management.Automation.WindowsErrorReporting”
- “MiniDumpWriteDump”

(Visited 108,376 times, 6 visits today)

» Bypass PowerShell ExecutionPolicy, Detect Invoke-Mimikatz, Detect offensive PowerShell, detect PowerShell attack tools, ExecutionPolicyBypass, Invoke-Expression, Invoke-Mimikatz, InvokeMimikatz, New-Object Net.WebClient DownloadString, Offensive PowerShell, PowerShell Attack Tool, PowerShell Attacks, PowerShell Constrained Language Mode, PowerShell Execution Policy, PowerShell GPO, PowerShell Logging Group Policy, PowerShell Mimikatz, PowerShell.exe, PowershellLogging, PowerShellMafia, PowerShellSecurity, PowerShellv5, PowerSploit, script block logging, System-wide transcript, System.Management.Automation.dll, Windows10



Sean Metcalf

I improve security for enterprises around the world working for TrimarcSecurity.com

Read the About page (top left) for information about me. :)

https://adsecurity.org/?page_id=8



RECENT POSTS

[BSides Dublin – The Current State of Microsoft Identity Security: Common Security Issues and Misconfigurations – Sean Metcalf](#)

[DEFCON 2017: Transcript – Hacking the Cloud](#)

[Detecting the Elusive: Active Directory Threat Hunting](#)

[Detecting Kerberoasting Activity](#)

[Detecting Password Spraying with Security Event Auditing](#)

TRIMARC ACTIVE DIRECTORY SECURITY SERVICES

Have concerns about your Active Directory environment? Trimarc helps enterprises improve their security posture.

[Find out how...](#) TrimarcSecurity.com

POPULAR POSTS

[PowerShell Encoding & Decoding \(Base64\)](#)

[Attack Methods for Gaining Domain Admin Rights in...](#)

[Kerberos & KRBTGT: Active Directory's...](#)

[Finding Passwords in SYSVOL & Exploiting Group...](#)

[Securing Domain Controllers to Improve Active...](#)

[Securing Windows Workstations: Developing a Secure Baseline](#)

[Detecting Kerberoasting Activity](#)

[Mimikatz DCSync Usage, Exploitation, and Detection](#)

[Scanning for Active Directory Privileges &...](#)

Microsoft LAPS Security & Active Directory LAPS...

CATEGORIES

ActiveDirectorySecurity

Apple Security

Cloud Security

Continuing Education

Entertainment

Exploit

Hacking

Hardware Security

Hypervisor Security

Linux/Unix Security

Malware

Microsoft Security

Mitigation

Network/System Security

PowerShell

RealWorld

Security

Security Conference Presentation/Video

Security Recommendation

Technical Article

[Technical Reading](#)

[Technical Reference](#)

[TheCloud](#)

[Vulnerability](#)

TAGS

[ActiveDirectory](#) [Active Directory](#) [Active Directory Security](#) [ActiveDirectorySecurity](#) [ADReading](#) [AD Security](#) [ADSecurity](#) [Azure](#) [AzureAD](#) [DCSync](#) [DomainController](#) [GoldenTicket](#) [GroupPolicy](#) [HyperV](#) [Invoke-Mimikatz](#) [KB3011780](#) [KDC](#) [Kerberos](#) [KerberosHacking](#) [KRBTGT](#) [LAPS](#) [LSASS](#) [MCM](#) [MicrosoftEMET](#) [MicrosoftWindows](#) [mimikatz](#) [MS14068](#) [PassTheHash](#) [PowerShell](#) [PowerShellCode](#) [Pow erShellHacking](#) [Pow erShellv5](#) [PowerSploit](#) [Presentation Security](#) [SilverTicket](#) [SneakyADPersistence](#) [SPN](#) [TGS](#) [TGT](#) [Window s7](#) [Windows10](#) [WindowsServer2008R2](#) [WindowsServer2012](#) [WindowsServer2012R2](#)



RECENT POSTS

[BSides Dublin – The Current State of Microsoft Identity Security: Common Security Issues and Misconfigurations – Sean Metcalf](#)

[DEFCON 2017: Transcript – Hacking the Cloud](#)

[Detecting the Elusive: Active Directory Threat Hunting](#)

[Detecting Kerberoasting Activity](#)

[Detecting Password Spraying with Security Event Auditing](#)

RECENT COMMENTS

Derek on [Attacking Read-Only Domain Controllers \(RODCs\) to Own Active Directory](#)

Sean Metcalf on [Securing Microsoft Active Directory Federation Server \(ADFS\)](#)

Brad on [Securing Microsoft Active Directory Federation Server \(ADFS\)](#)

Joonas on [Gathering AD Data with the Active Directory PowerShell Module](#)

Sean Metcalf on [Gathering AD Data with the Active Directory PowerShell Module](#)

ARCHIVES

[June 2024](#)

[May 2024](#)

[May 2020](#)

[January 2020](#)

[August 2019](#)

[March 2019](#)

[February 2019](#)

[October 2018](#)

[August 2018](#)

[May 2018](#)

[January 2018](#)

[November 2017](#)

[August 2017](#)

[June 2017](#)

[May 2017](#)

February 2017

January 2017

November 2016

October 2016

September 2016

August 2016

July 2016

June 2016

April 2016

March 2016

February 2016

January 2016

December 2015

November 2015

October 2015

September 2015

August 2015

July 2015

June 2015

May 2015

April 2015

March 2015

February 2015

January 2015

[December 2014](#)

[November 2014](#)

[October 2014](#)

[September 2014](#)

[August 2014](#)

[July 2014](#)

[June 2014](#)

[May 2014](#)

[April 2014](#)

[March 2014](#)

[February 2014](#)

[July 2013](#)

[November 2012](#)

[March 2012](#)

[February 2012](#)

CATEGORIES

[ActiveDirectorySecurity](#)

[Apple Security](#)

[Cloud Security](#)

[Continuing Education](#)

[Entertainment](#)

[Exploit](#)

[Hacking](#)

[Hardware Security](#)

[Hypervisor Security](#)

[Linux/Unix Security](#)

[Malware](#)

[Microsoft Security](#)

[Mitigation](#)

[Network/System Security](#)

[PowerShell](#)

[RealWorld](#)

[Security](#)

[Security Conference Presentation/Video](#)

[Security Recommendation](#)

[Technical Article](#)

[Technical Reading](#)

[Technical Reference](#)

[TheCloud](#)

[Vulnerability](#)

META

[Log in](#)

[Entries feed](#)

[Comments feed](#)

WordPress.org

COPYRIGHT

Content Disclaimer: This blog and its contents are provided "AS IS" with no warranties, and they confer no rights. Script samples are provided for informational purposes only and no guarantee is provided as to functionality or suitability. The views shared on this blog reflect those of the authors and do not represent the views of any companies mentioned. Content Ownership: All content posted here is intellectual work and under the current law, the poster owns the copyright of the article. Terms of Use Copyright © 2011 - 2020.

Content Disclaimer: This blog and its contents are provided "AS IS" with no warranties, and they confer no rights. Script samples are provided for informational purposes only and no guarantee is provided as to functionality or suitability. The views shared on this blog reflect those of the authors and do not represent the views of any companies mentioned.

Made with ❤ by Graphene Themes.