Inhalt

Beschreibung der Schutzkomponenten

Worum geht es?

Die Powershell ist ein mächtiges, administratives Werkzeug. Viele Tätigkeiten lassen sich automatisieren. Fast alle Bereiche der Microsoft-Produkte lassen sich so steuern. Und die Powershell ist nativ mit an Bord. Jedes moderne Betriebssystem hat sie dabei.

Da ist es nur natürlich, dass auch unsere Hacker-Kollegen seit Langem einen Gefallen an dieser Plattform gefunden haben. Es existieren etliche Schadcodes für eine Vielzahl von Angriffen…

Doch auch Microsoft hat nicht (mehr) geschlafen! In der aktuellen PowerShell-Version gibt es einige interessante Schutzmechanismen. Diese möchte ich nun einmal vorstellen.

Voraussetzungen

Für einige der neuen Schutzmöglichkeiten wird die PowerShell-Version 5 benötigt. Diese gehört zu Windows 10 und Windows Server 2016 standardmäßig zur Grundausstattung. Auf älteren Systemen kann die PowerShell natürlich aktualisiert werden. Davon halte ich persönlich nicht viel, denn zu oft habe ich danach Probleme auf Clients und Servern beobachten dürfen. Aber auch für diese Systeme gibt es ein paar Einstellungsmöglichkeiten.

Generell halte ich es nicht für sinnvoll, die PowerShell einfach zu deaktivieren bzw. zu blockieren. Sie ist so tief im System verankert, dass ein geübter Angreifer es dennoch schaffen wird, auf einen PS-Prozess zuzugreifen. Und zudem ist sie für das Daily-Business einfach zu wertvoll!

Für die Angriffe habe ich eines meiner PSHacking-Scripte ausgewählt.

Und um einen Enterprise-Charakter darzustellen werde ich die Konfiguration natürlich über Gruppenrichtlinien vornehmen. Dazu steht in meiner LAB-Umgebung ein kleiner DomainController bereit. 😊

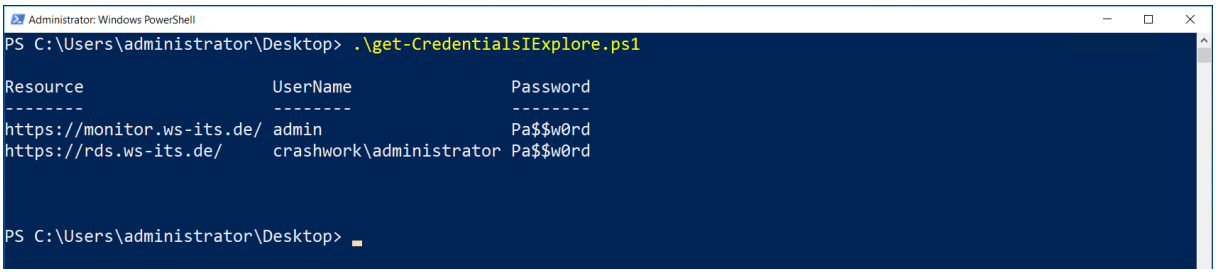
mögliche Angriffsszenarien

Die Szenarien sind extrem vielfältig. Da die PowerShell auch auf das gesamte .net-Framework und alle WMI-Repositories zugreifen kann ist hier der Fantasie kaum eine Grenze gesetzt. Einige interessante Beispiele kommen gleich in meinen Szenarien vor.

In meinem Beispiel verwende ich einen Code, mit dem die Internet-Explorer-Passworte aus dem Credential-Manager ausgelesen werden. Das funktioniert für Benutzer mit und ohne administrative Berechtigungen und der Code ist nicht besonders lang:

```
$ClassLoader   =
[Windows.Security.Credentials.PasswordVault,Windows.Security.Credentials,ContentType=WindowsRuntime]
$PasswordVault = new-object Windows.Security.Credentials.PasswordVault
$PasswordVault.RetrieveAll() |
    ForEach-Object { $_.RetrievePassword() ; $_ } |
        Select-Object -Property Resource, UserName, Password |
            Sort-Object Resource |
                Format-Table -AutoSize
```

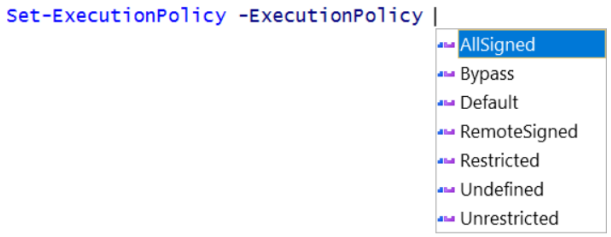
Es ist also ideal für eine Demonstration. Die erforderliche Klasse existiert aber erst seit Windows 8.x. Daher wird das auf Windows 7 nicht funktionieren. Egal, denn ich verwende ein aktuelles Windows 10. 😊



Szenario 1 - PowerShell Script Execution

Angriff ohne Schutz

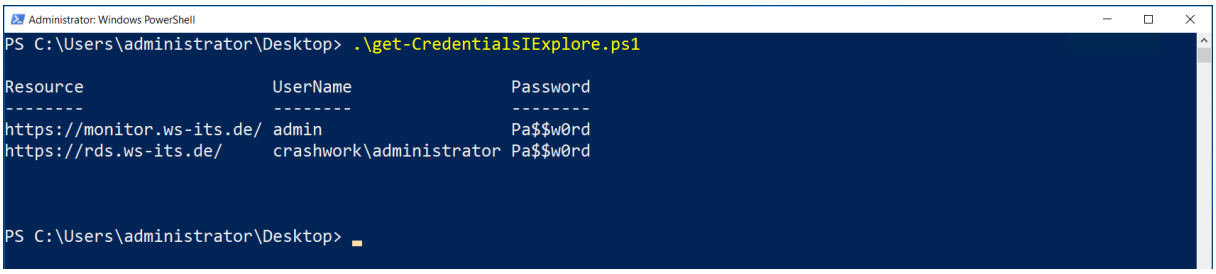
Die PowerShell verwendet seit langer Zeit eine ExecutionPolicy. Mit dieser kann gesteuert werden, ob Scripte für die Ausführung digital signiert sein müssen. Dabei werden verschiedene Ebenen unterschieden:



Zusätzlich kann dabei ein Speicherort ausschlaggebend sein: RemoteSigned setzt für die Scriptausführung die digitale Signatur nur voraus, wenn das Script nicht auf dem lokalen Rechner gespeichert ist.

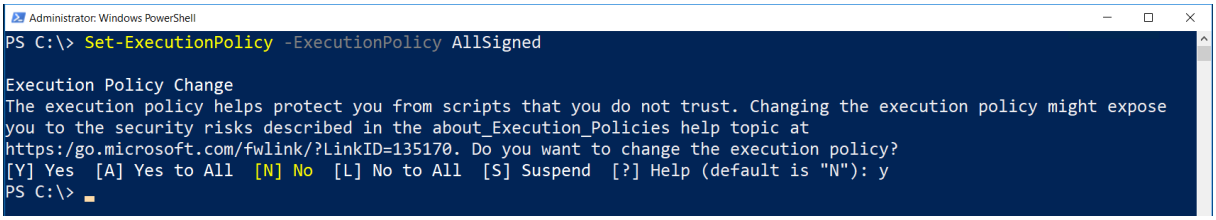
Die digitale Signatur soll sicherstellen, dass der Code unverändert von einem vertrauenswürdigen Programmierer stammt. Sollte der Code (Text) manipuliert worden sein, so ist die digitale Signatur nicht länger gültig.

Ein Angreifer könnte also ohne eine Signatur sein Script auf einem System ausführen:

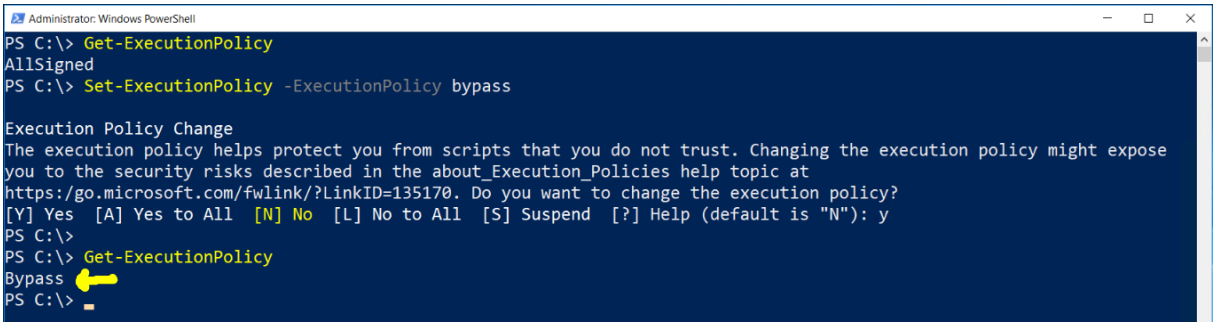


Konfiguration der Schutzkomponente

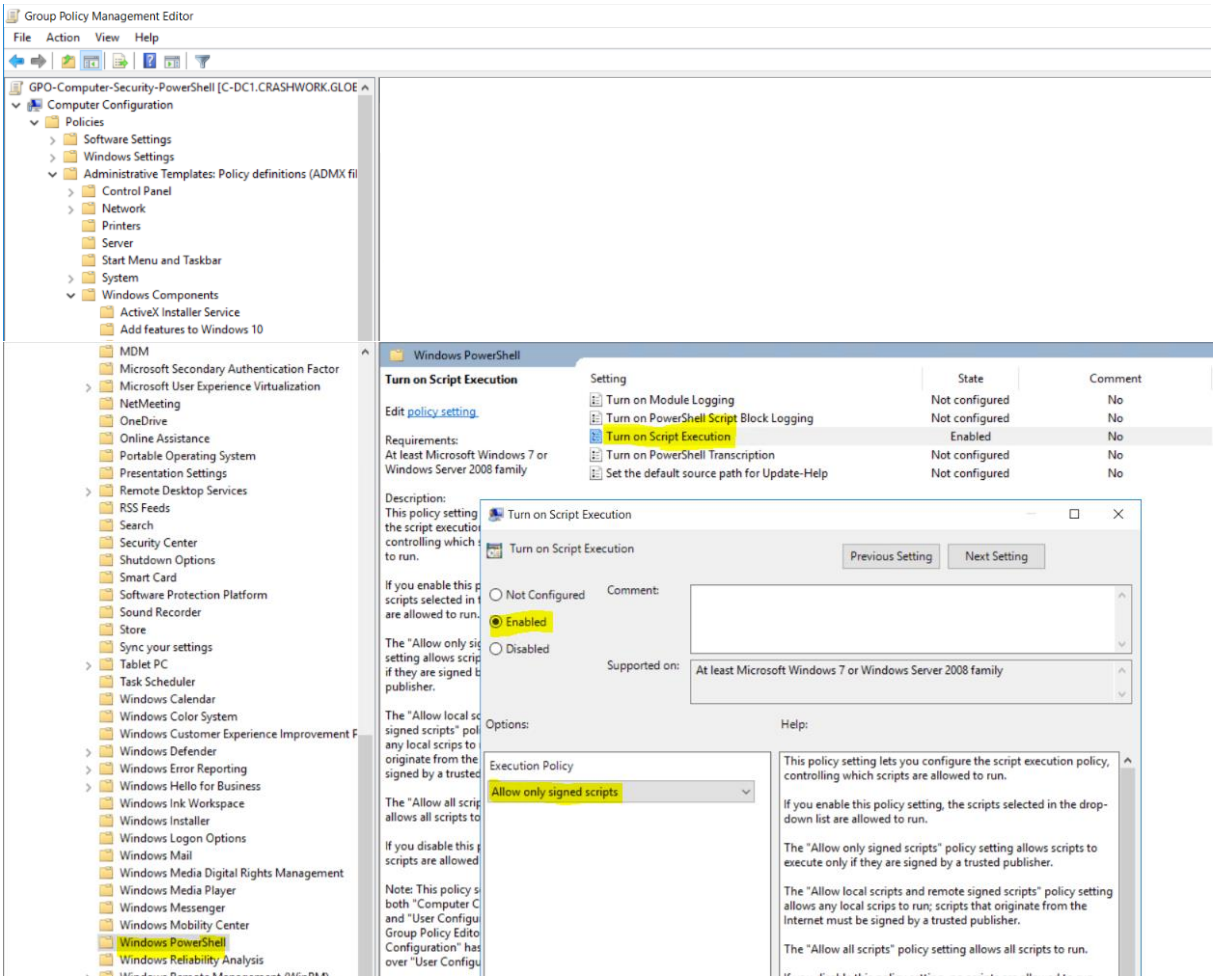
Mit einer erforderlichen Scriptsignierung sollte das nach der Idee von Microsoft verhindert werden. Die Signierung kann lokal festgelegt werden:



Dennoch kann ein Angreifer mit den erforderlichen Rechten die lokal definierte Signierungsanforderung wieder genauso rückgängig machen:

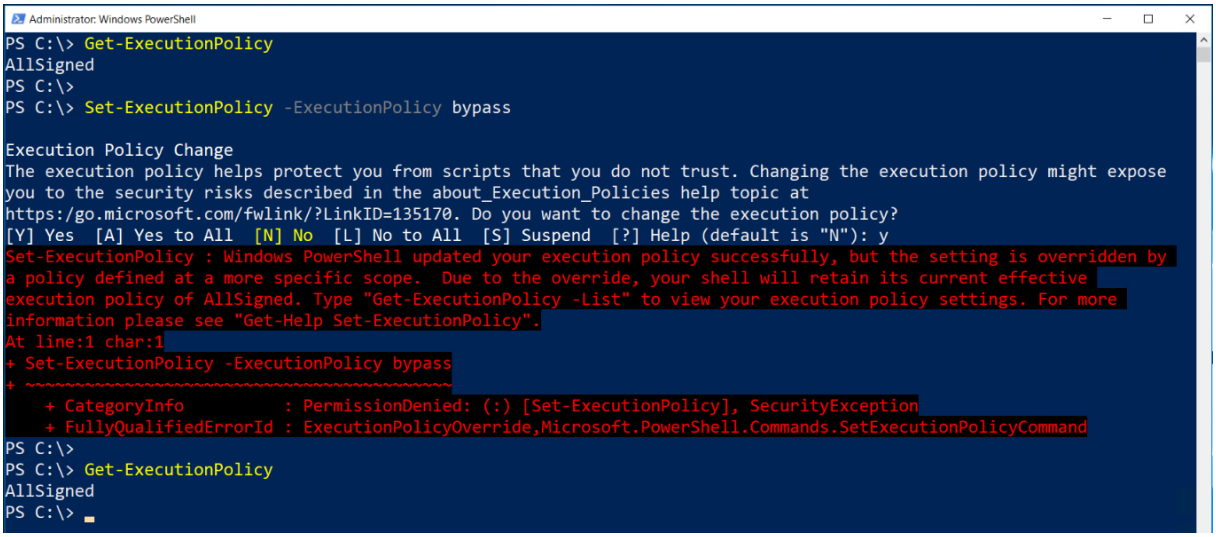


Besser und vor allem effizienter wäre die zentrale Konfiguration mit einer Gruppenrichtlinie:



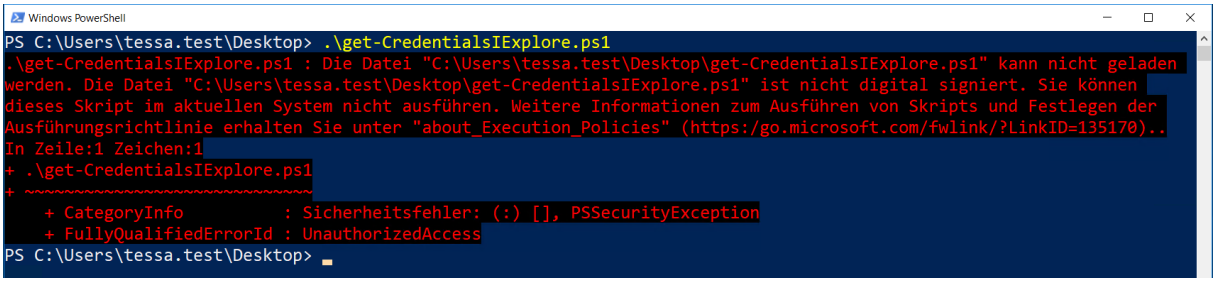
Nach dem Anwenden der GPO werden neue PowerShell-Prozesse entsprechend konfiguriert sein:

Wenn nun der Admin die Regel lokal außer Kraft setzen will, gewinnt die GPO:



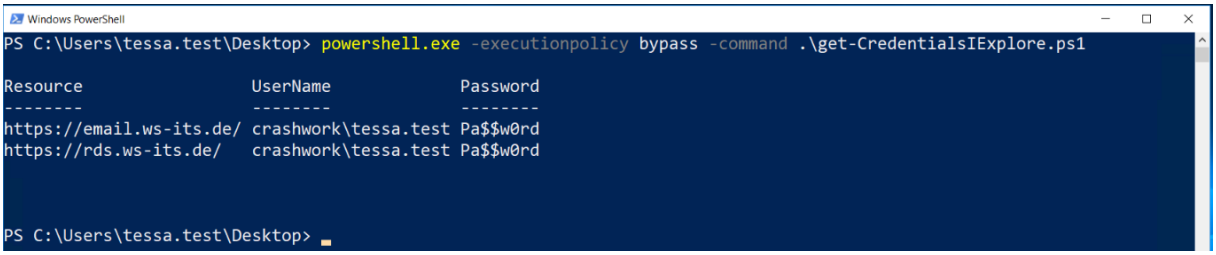
Angriff eines geschützten Systems

Wie wirkt nun die Einstellung? Benutzer, die nun ein Script ohne Signratur starten wollen werden abgewiesen:

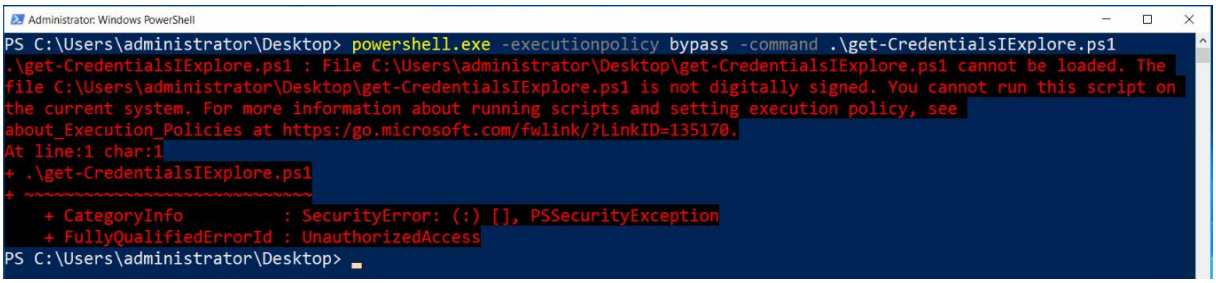


Nur leider existieren einige Optinen, die Richtlinie zu umgehen:

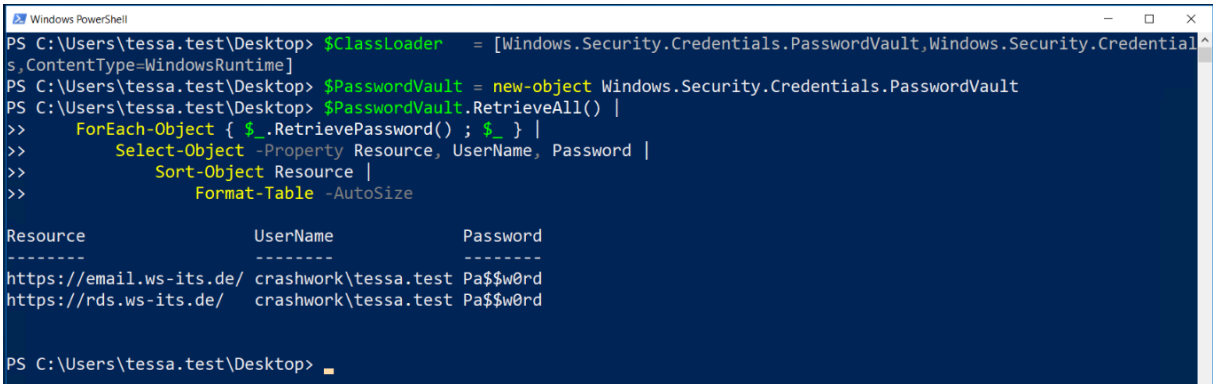
OHNE die GPO (also wenn die Executionpolicy von einem Admin lokal definiert wurde) kann sogar ein Benutzer die Einstellung aushebeln ☹. Denn die PowerShell lässt sich auch mit dem Parameter -ExecutionPolicy starten. Und dieser überschreibt die lokalen settings für den neuen Prozess:



Wird die Regel dagegen über eine GPO definiert, dann ist dies sogar für den Administrator unmöglich:

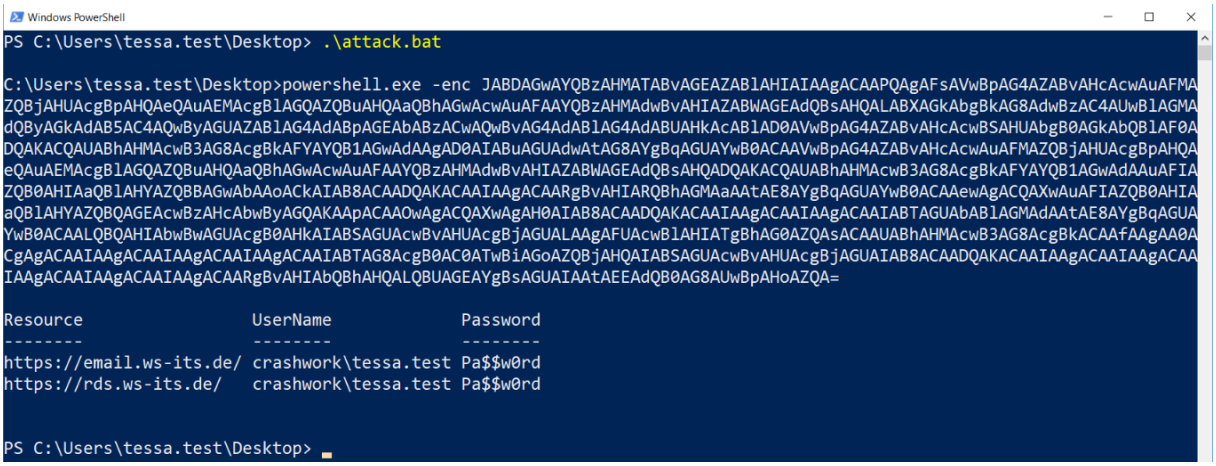


Aber der Angreifer könnte den Code auch einfach „tippen“. Dann ist es eben kein Script mehr :



Oder der Code wird zu BASE64 konvertiert und z.B. als Batch-Script gestartet:

```
$code = @'
$ClassLoader = [Windows.Security.Credentials.PasswordVault,Windows.Security.Credentials,ContentType=WindowsRuntime]
$PasswordVault = new-object Windows.Security.Credentials.PasswordVault
$PasswordVault.RetrieveAll() |
    ForEach-Object { $_.RetrievePassword() ; $_ } |
        Select-Object -Property Resource, UserName, Password |
            Sort-Object Resource |
                Format-Table -AutoSize
'@

$code = $code -join "`r`n"
$bytes = [System.Text.Encoding]::Unicode.GetBytes($code)
$enc = [Convert]::ToBase64String($bytes)

"powershell.exe -enc $enc" | out-file -FilePath .\attack.bat -Encoding default
```



Nebenwirkungen und nützliche Hinweise

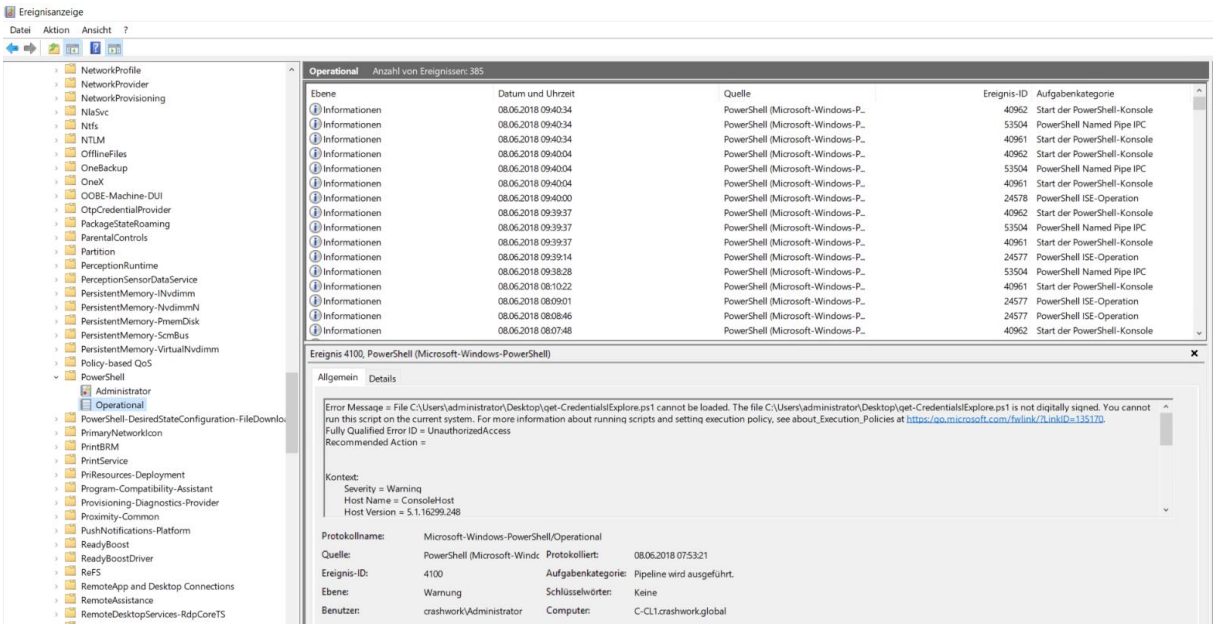
Die Erzwingung einer Scriptsignatur wird Schadcode nicht effektiv aufhalten! Ich persönlich nutze die Scriptsignatur eigentlich nur noch, um Manipulationen an meinen Scripten durch andere Admins zu erkennen.

Gleiches gilt natürlich auch, wenn die Scriptsignatur-Überwachung durch Applocker realisiert wird. Selbst diese lässt sich mit EncodedCommands oder durch „Tippen“ umgehen…

Szenario 2 - PowerShell Script Block Logging

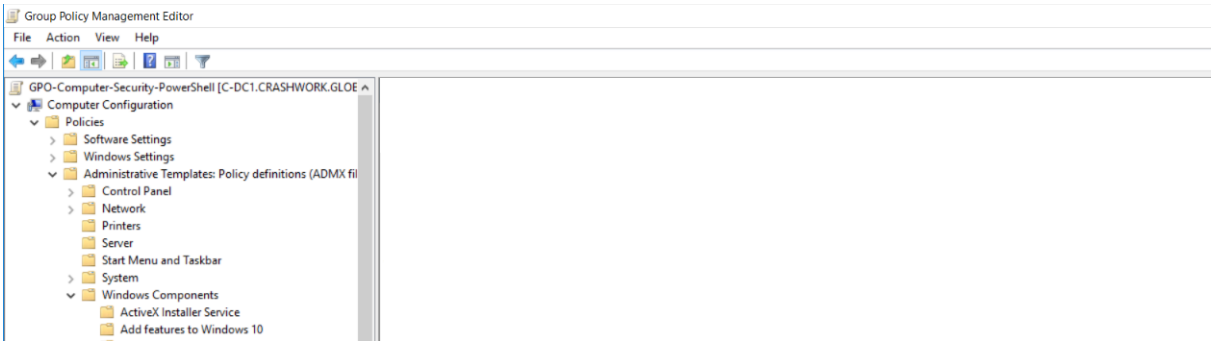
Angriff ohne Schutz

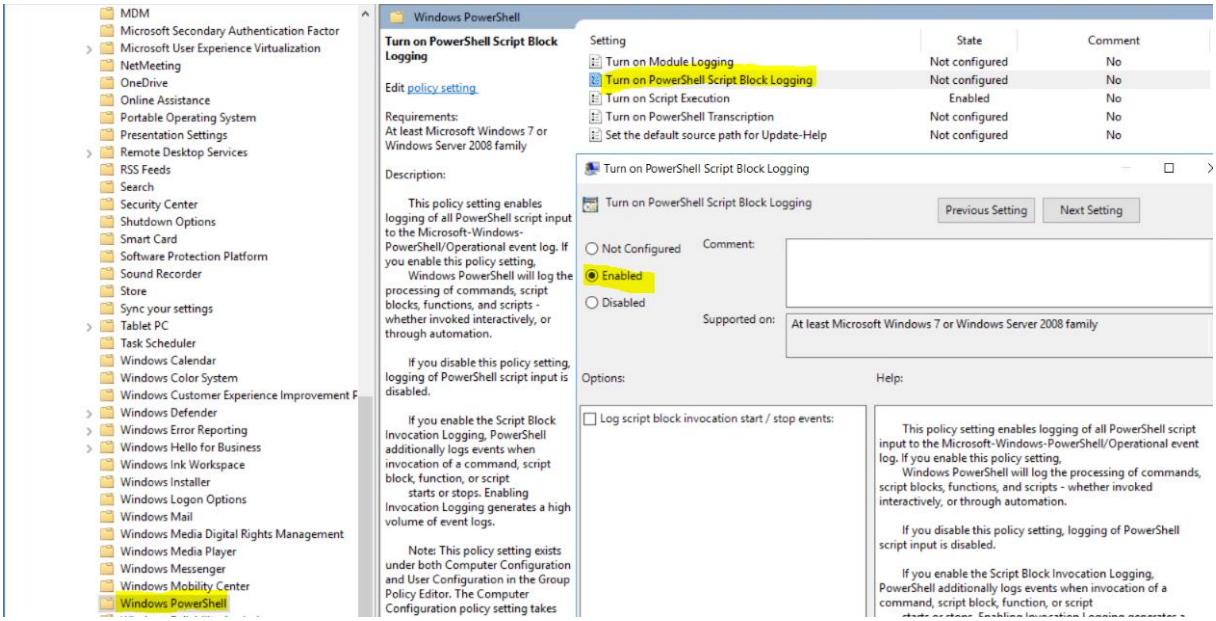
Es ist nicht praktikabel, alle Aufrufe der PowerShell zu verbieten, denn für administrative Zwecke ist sie ein unverzichtbares Werkzeug geworden. Sollte es aber einmal zu einem Angriff gekommen sein, dann sollte zumindest die forensische Analyse gewährleistet sein. Dafür ist eine Script-Protokollierung erforderlich. Durch Script Block Logging wird dies über eine GPO sehr einfach erreicht. Ohne ist der ausgeführte Code nach dem Beenden des Powershell-Prozesses nicht mehr verfügbar. Und das Eventlog bietet keine gespeicherten Infos:



Konfiguration der Schutzkomponente

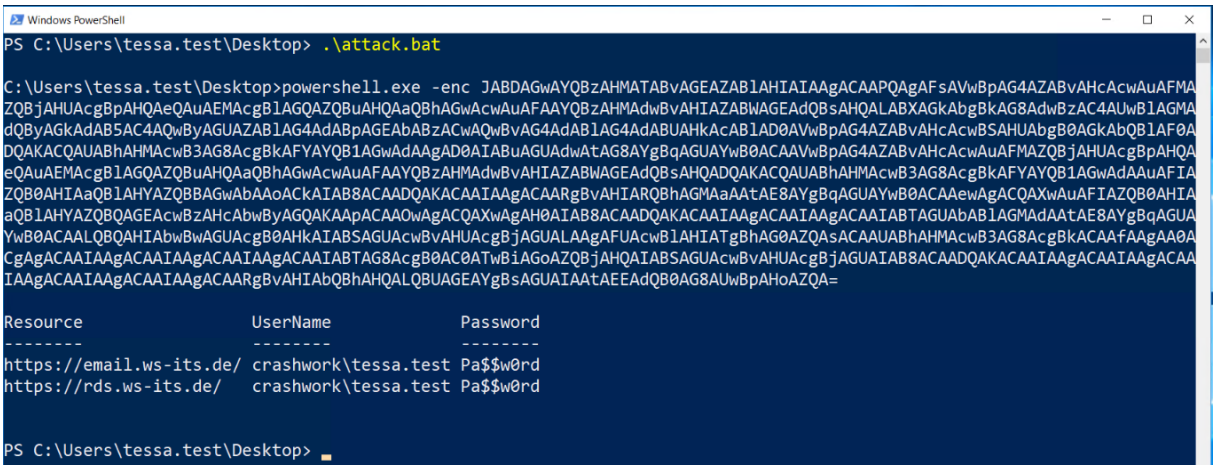
Auch hier ist die Konfiguration über eine GPO möglich:



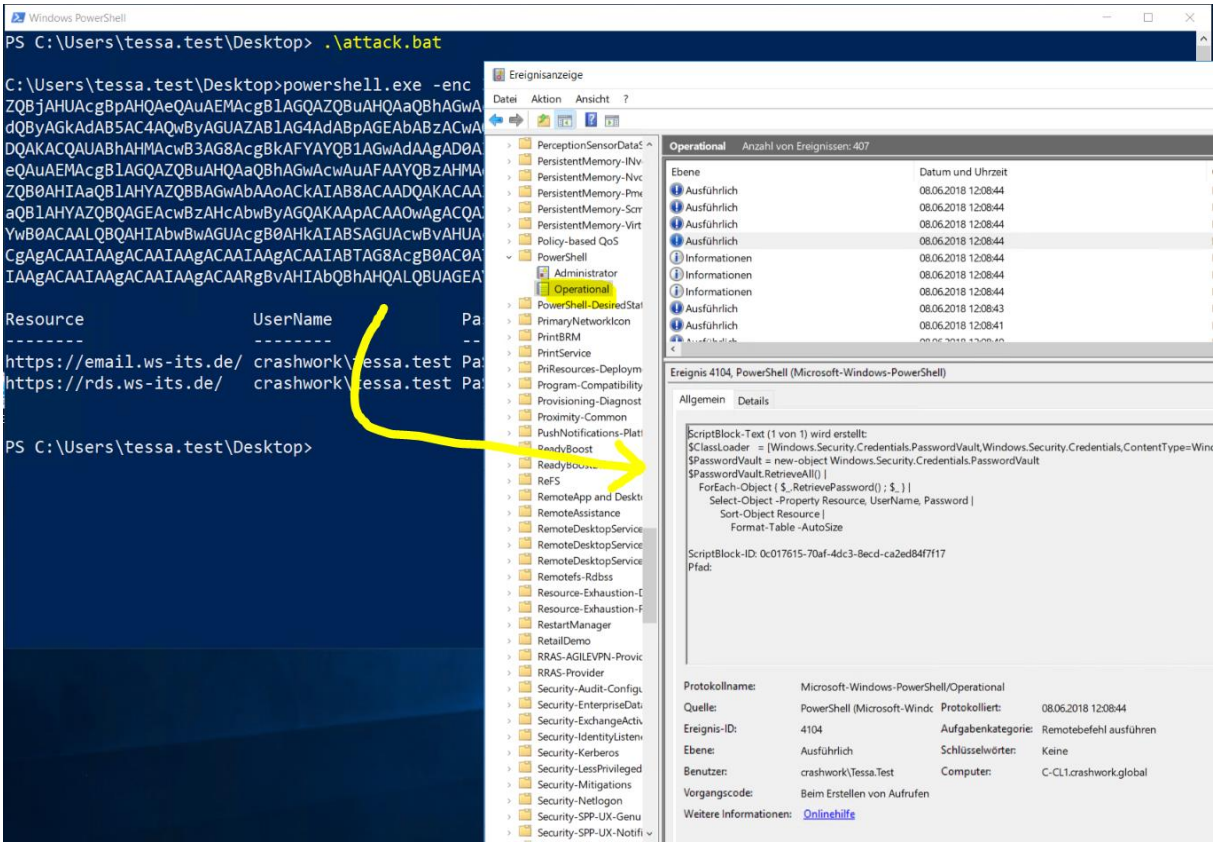


Angriff eines geschützten Systems

Durch eine reine Protokollierung wird natürlich kein Angriff verhindert:



Aber er kann nachvollzogen werden. Im Eventlog findet man nun eben auch den Code im Klartext:



Mit einem Monitoring auf übliche Schlüsselworte könnte im Hintergrund ein Alarm ausgelöst werden…

Nebenwirkungen und nützliche Hinweise

Nachteilig ist hierbei aber die lokale Protokollierung. Bei vielen Clients ist eine zentrale Auswertung nahezu unmöglich. Und natürlich kann ein Angreifer mit ausreichenden Rechten die Eventlogs nach seiner Aktion auch löschen… ☹

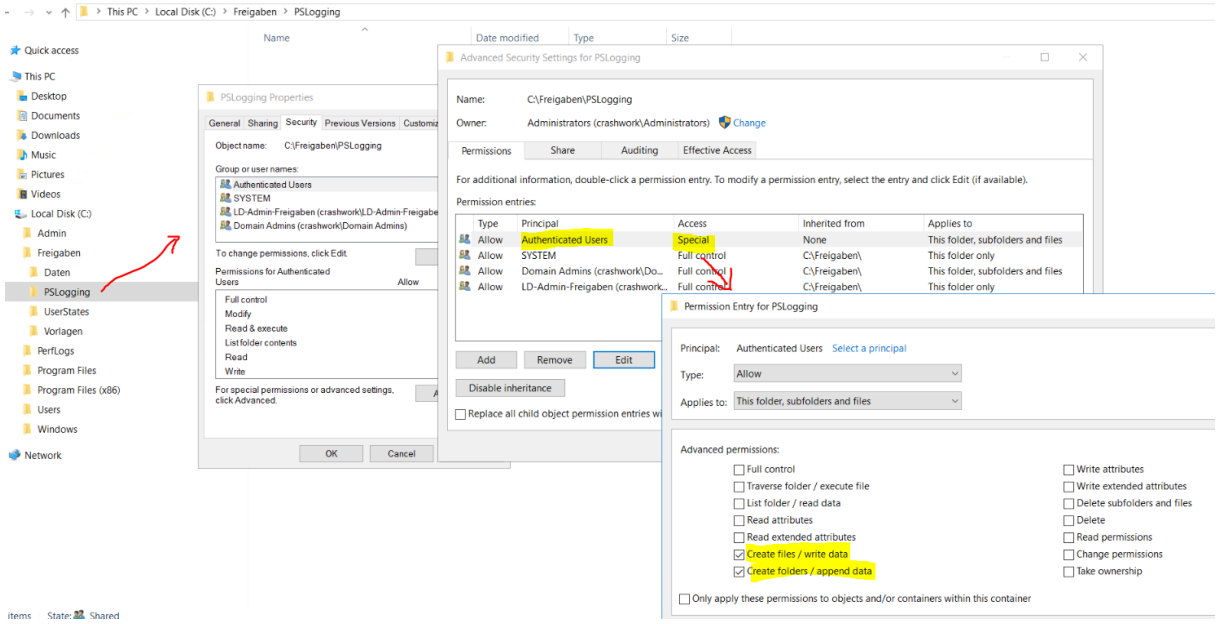
Szenario 3 - PowerShell Transcription

Angriff ohne Schutz

Besser als die Protokollierung in Eventlogs wäre gleich eine zentrale Sammlung von Protokollen auf einem Laufwerk. Das lässt sich mit Transcripts erzwingen. Ohne ist die Ausgangslage die gleiche wie im Punkt vorher: der Angriff kann nachträglich nicht analysiert werden.

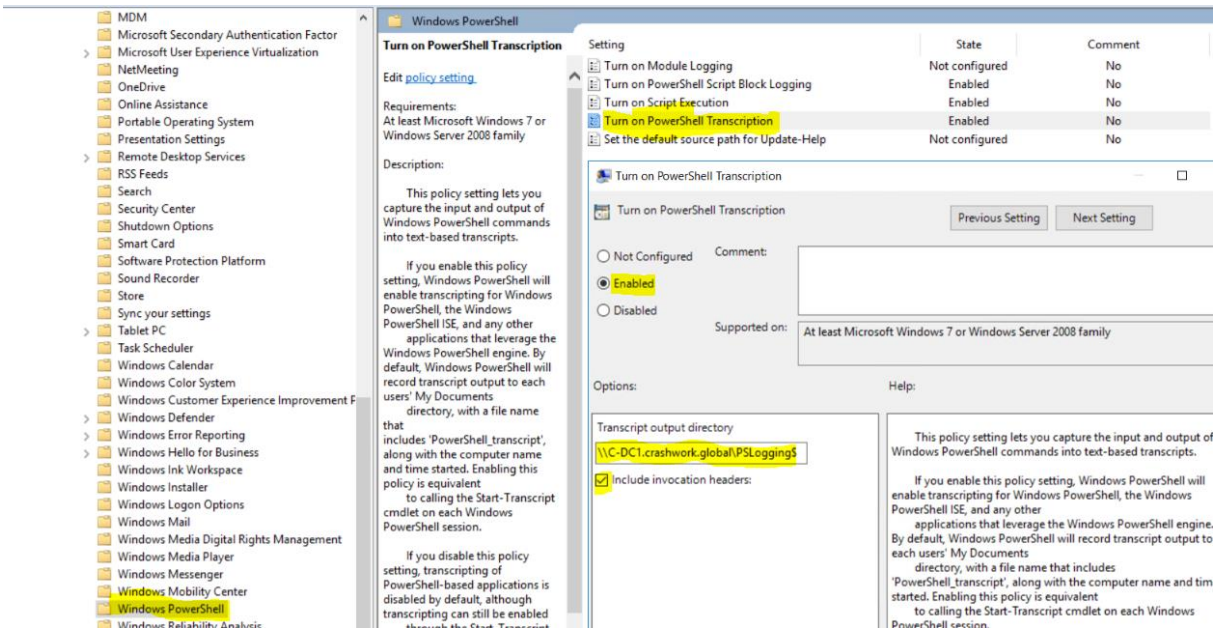
Konfiguration der Schutzkomponente

Transcripts werden ebenfalls in einer GPO definiert. Zusätzlich benötigen wir aber auch ein zentrales Laufwerk. Dieses habe ich auf meinem LAB-Server erstellt:



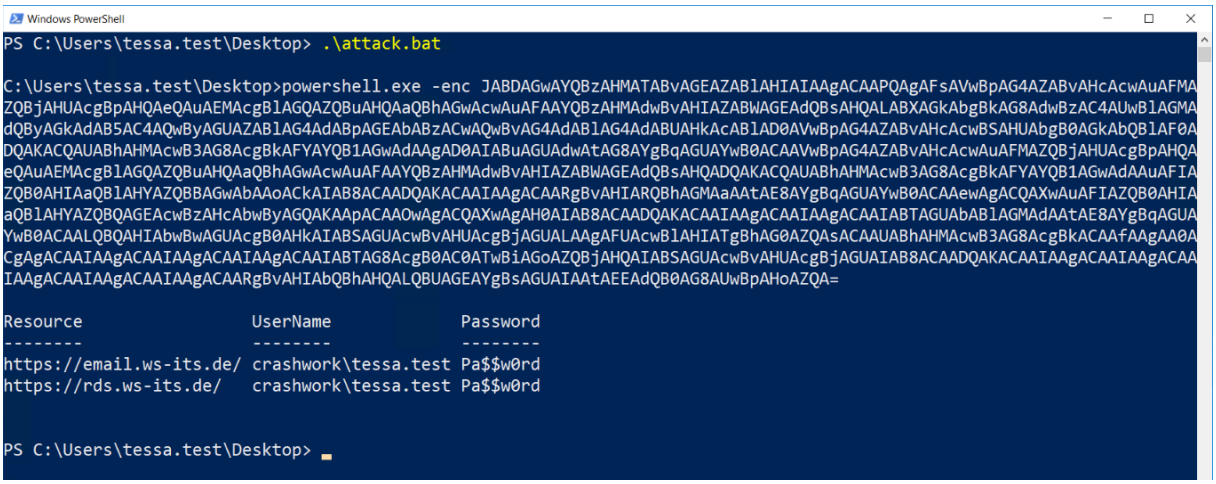
Man erkennt (hoffentlich), dass ich den authentifizierten Benutzer nur erlaube, Daten zu schreiben oder anzuhängen. Auf diese Weise verhindere ich, dass ein Angreifer die Transcript-Dateien nachträglich löscht.

In der GPO fehlt nicht mehr viel:

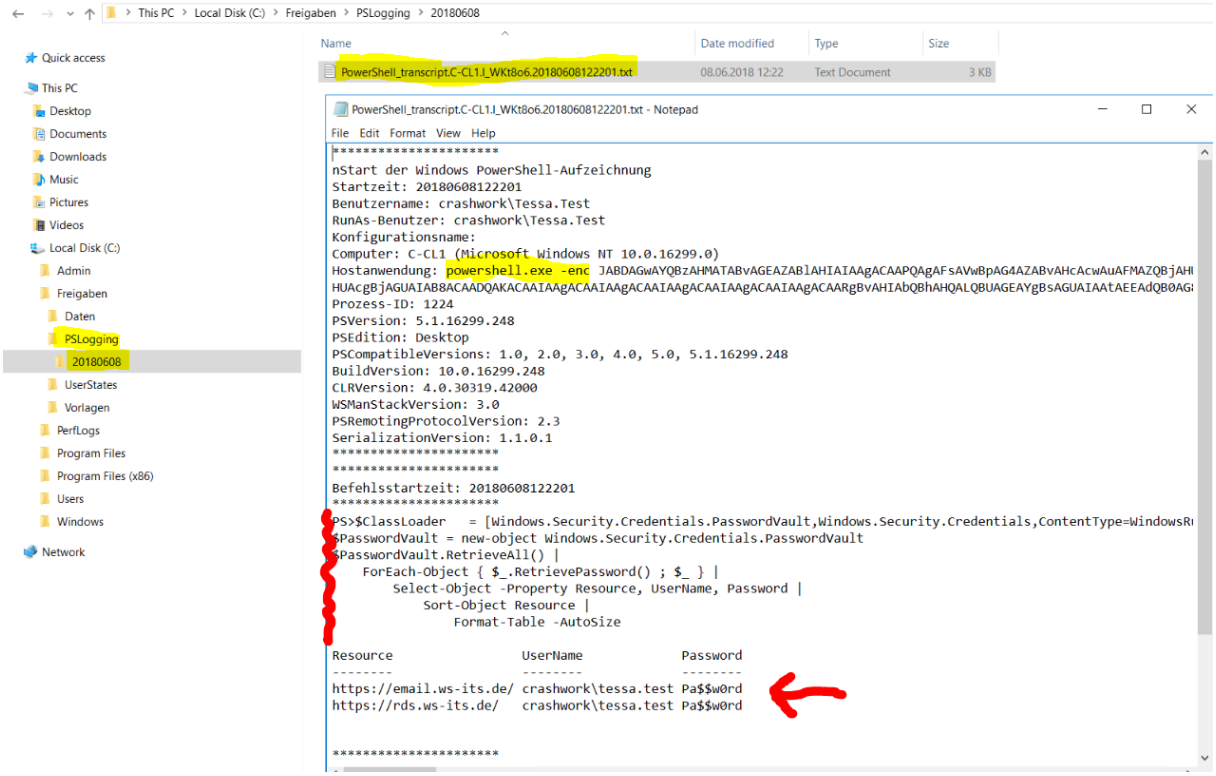


Angriff eines geschützten Systems

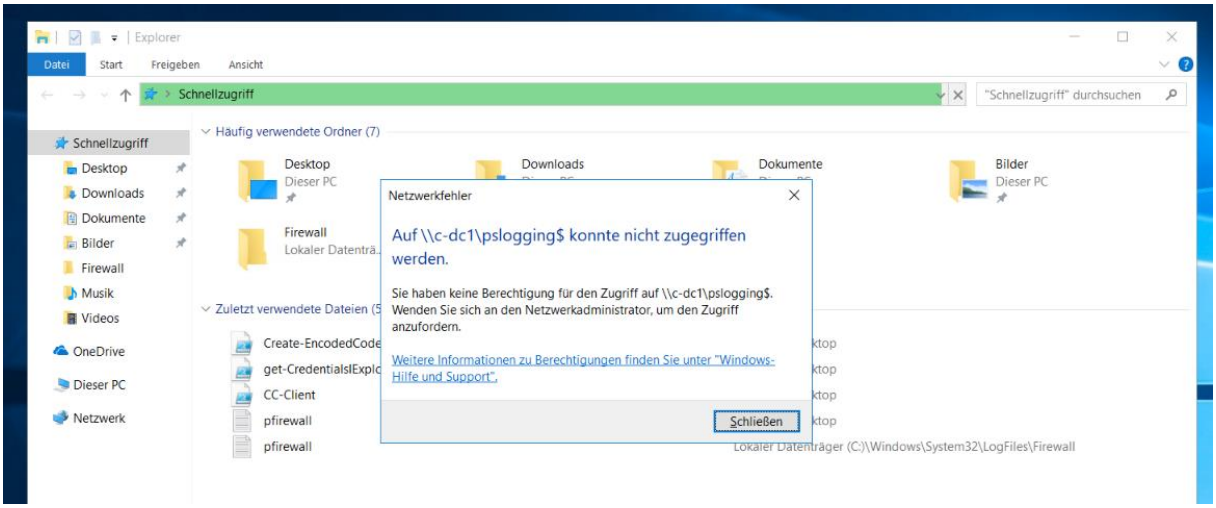
Nachdem die Richtlinie aktiviert wurde, wird die PowerShell jede Ausführung zentral protokollieren. Der User bekommt davon nichts mit:



Auf dem Server wird je Session eine eigene Datei erstellt. In dieser finden wir neben dem EncodedCommand auch die Klartext-Codes – und auch die Ergebnisse:



Da so natürlich auch sensible Informationen abgegriffen werden können muss ein Schutz des Verzeichnisses eine hohe Priorität haben. Ggf. könnten die Files nach deren automatischer Analyse gezippt und verschlüsselt werden – oder man löscht sie einfach nach deren Analyse. Bis dahin gibt es bei mir dank NTFS-Permission nichts zu sehen:

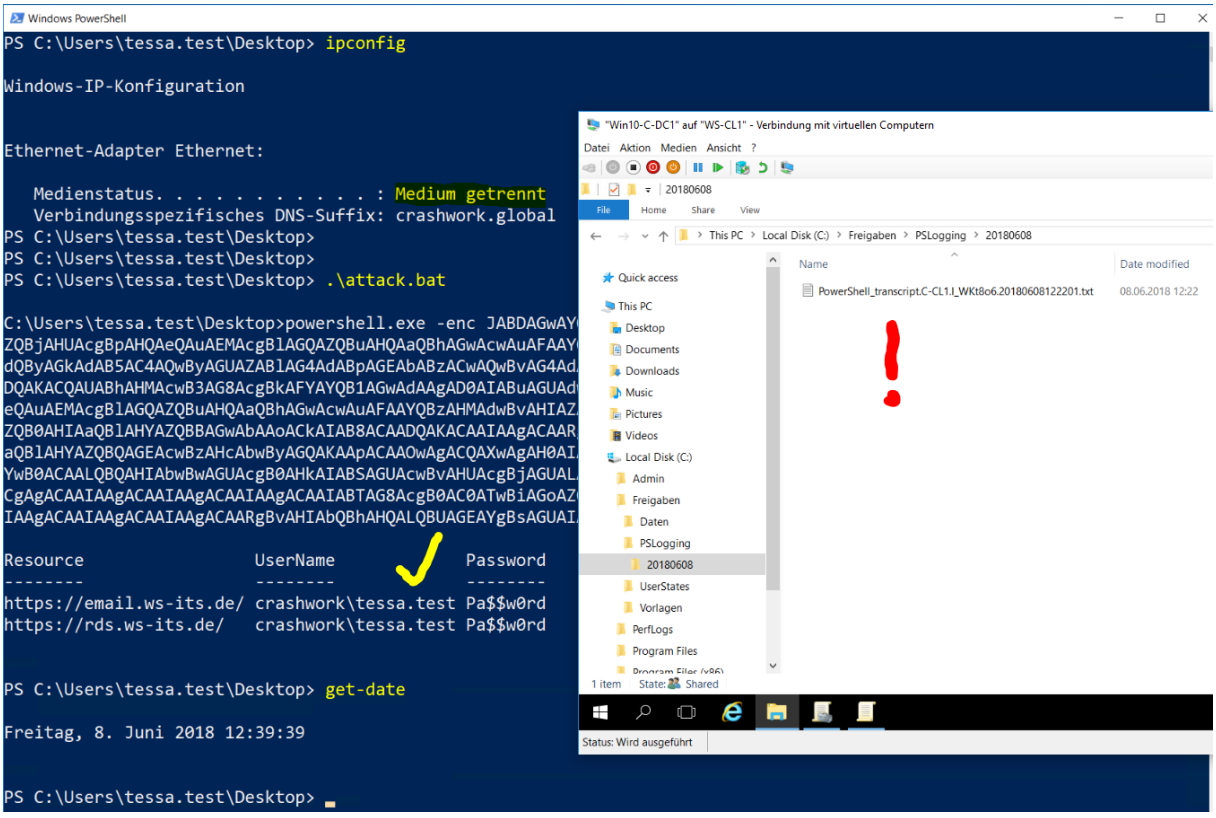


Nebenwirkungen und nützliche Hinweise

Der Schutz ist hier ebenfalls nur die Möglichkeit einer forensischen Analyse. Ggf. findet eine Automatik in den Textdateien Schlüsselworte, die auf einen Angriff hinweisen.

Aus Angreifersicht könnte man nun aber einfach den SMB-Datenstrom vom Client zum Server mitschneiden und so an sensible Informationen herankommen. Die PowerShell wird ja doch eher von Admins verwendet. Und die führen damit natürlich ganz interessante Tätigkeiten aus…

Ebenso nachteilig sind mobile Arbeitsstationen. Die PowerShell versucht natürlich, den Zielserver zum Schreiben des Transcripts zu erreichen, aber wenn das fehlschlägt, wird eben OHNE Protokollierung ausgeführt:

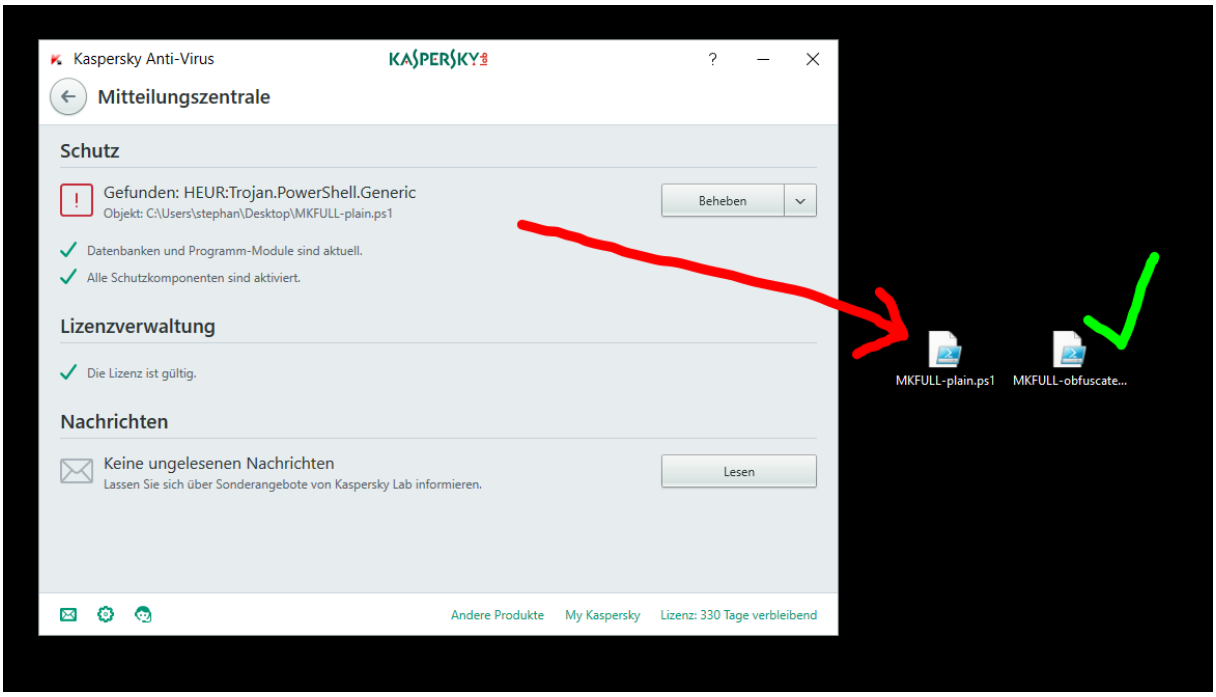


Ein Angreifer vor Ort könnte also einfach das Netzkabel herausziehen bevor er loslegt…

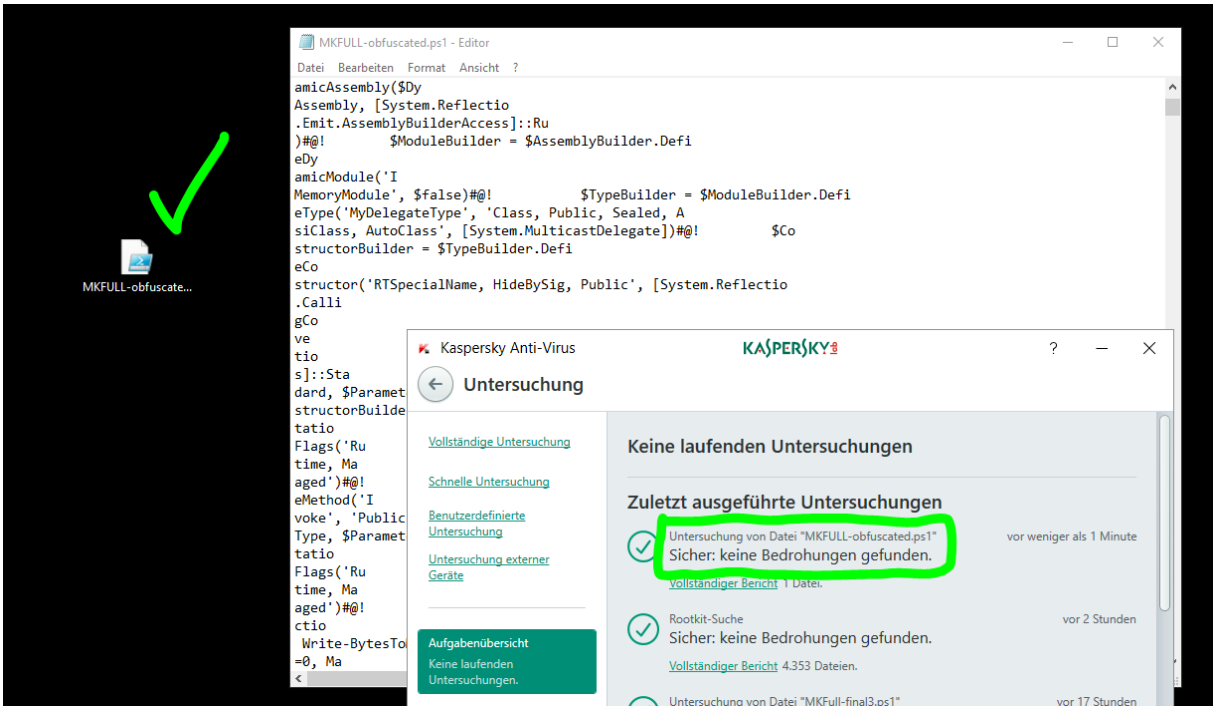
Szenario 4 – AMSI (Anti Malware Scan Interface)

Angriff ohne Schutz

Hilft ein Virenscanner gegen Schadcode-Dateien? Nun ja, Scripte sind zunächst einmal Textdateien. Weisen diese ein bestimmtes Muster auf, dass der Virenscanner erkennt, dann kann er reagieren. Auf meinem Desktop liegen 2 Dateien, die den gleichen Code beinhalten. Aber eine davon habe ich modifiziert:

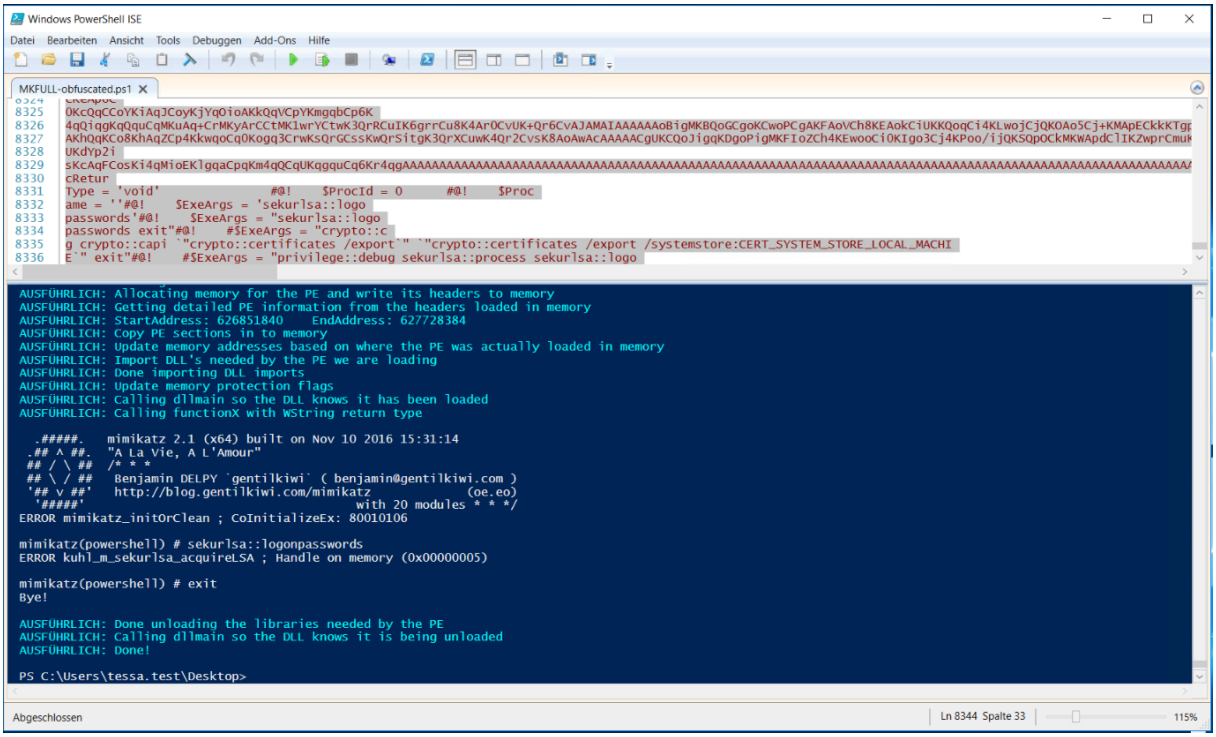


Mein Virenscanner ist also in der Lage, den Schadcode im Klartext zu identifizieren. Aber modifiziert ist alles OK:



Klassischerweise hilft ein Virenscanner bei maskiertem Code (obfuscaded code) nicht mehr, da er mehr auf die Datei und deren Aufbau fixiert ist. Seit Windows 10 hat Microsoft eine Schnittstelle für AV-Lösungen bereitgestelt, mit der Schadcode vor der Ausführung im RAM überprüft werden kann – AMSI, oder Anti Malware Scan Interface. Damit KANN ein Virenscanner in den Prozess eingreifen, bevor der Code ausgeführt wird.

Hier sieht man den maskierten Code OHNE AMSI:

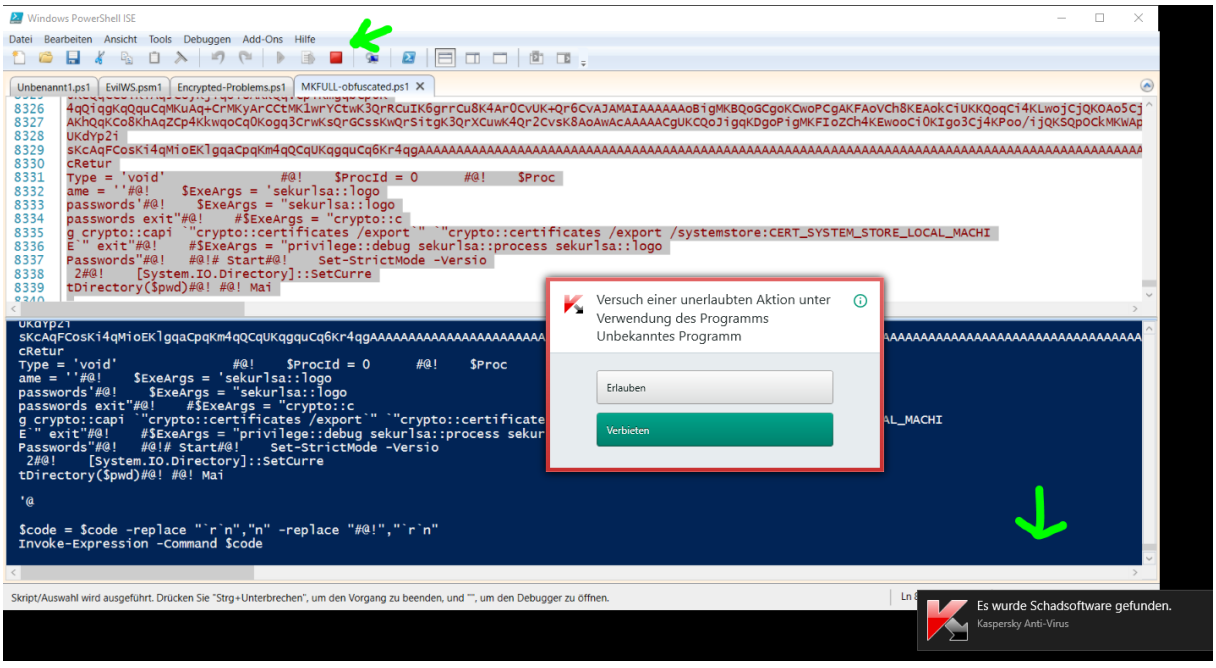


Konfiguration der Schutzkomponente

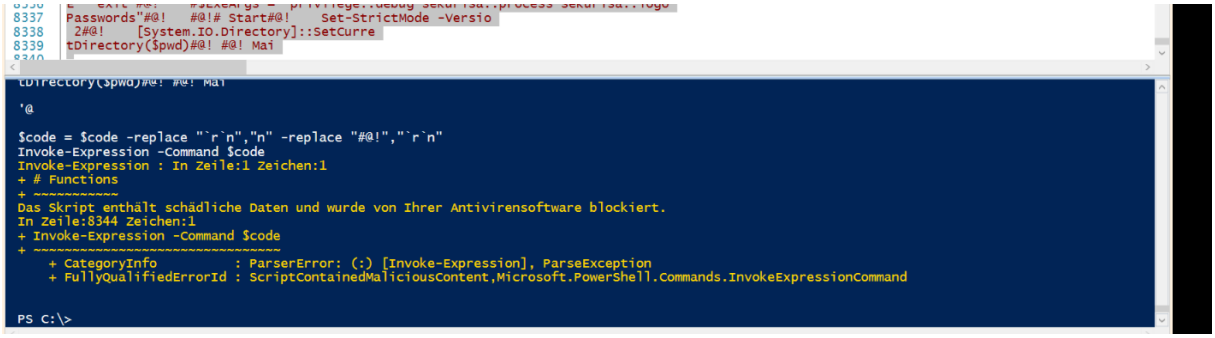
Für AMSI benötigt man Windows 10 und einen kompatiblen Virenscanner.

Angriff eines geschützten Systems

Mit allen Voraussetzungen wird der Code einfach aufgehalten:



Danach hält der Code einfach an:



Nebenwirkungen und nützliche Hinweise

Wie bei jedem Virenscanner ist hier nur eine Wahrscheinlichkeit der Erkennung vorhanden. Wenn ich den Code oben VOR dem Maskieren umbaue, dann erkennt mein Kaspersky auch nichts mehr…

Aber es ist besser als nichts!

Szenario 5 - PowerShell Constraint Mode

Angriff ohne Schutz

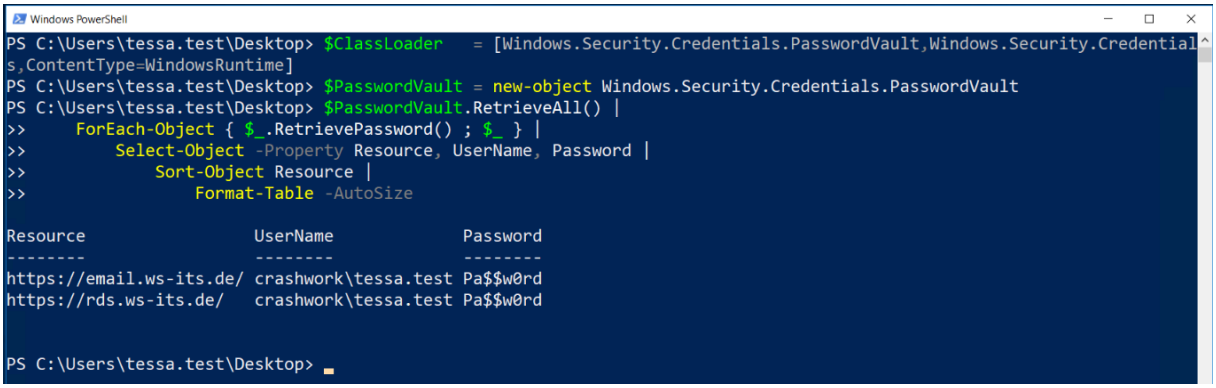
Die PowerShell kann nahezu auf alle Ressourcen und Werkzeuge des Betriebssystems zugreifen. Selbst wenn es keine passenden cmdlets gibt, kann mit .net-Unterstützung oder über die WMI fast jede Hürde überwunden werden.

Für alltägliche Aufgaben genügen meist die eingebauten cmdlets. Warum soll man daher die Erweiterungen zulassen? Genau das lässt sich mit dem LanguageMode definieren. Aktuell werden 2 Modi unterschieden

- im FullLanguageMode ist alles erlaubt
- im ContrainedMode sind nur Basisbefehle möglich

Abfragen lässt sich dies recht einfach mit dieser Zeile:

Im Standard-Modus FullLanguage gibt es keine Einschränkung. Ein Angreifer kann daher .net-Konstrukte verwenden, um an seine Beute zu gelangen:

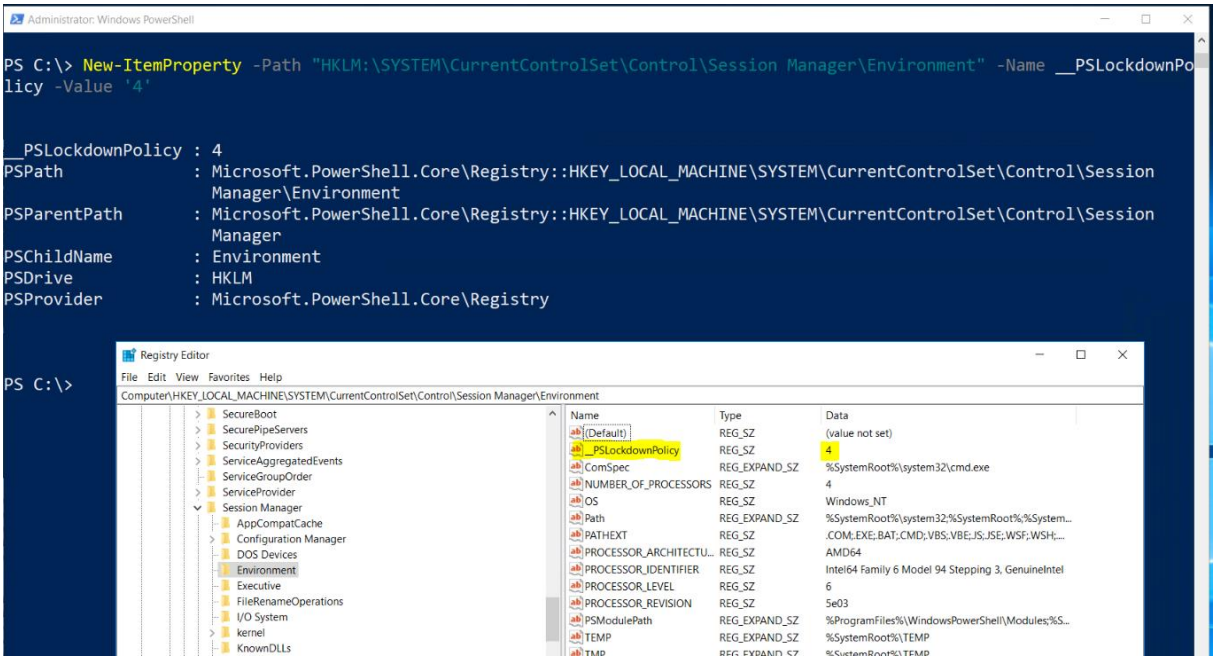


Konfiguration der Schutzkomponente

Die Aktivierung kann über eine Systemvariable erfolgen. Diese muss __PSLockdownPolicy heißen und den Wert 4 für den ConstrainedMode aufweisen. Der Wert 1 oder das Fehlen der Variable schaltet die PowerShell wieder in den FullLanguageMode.

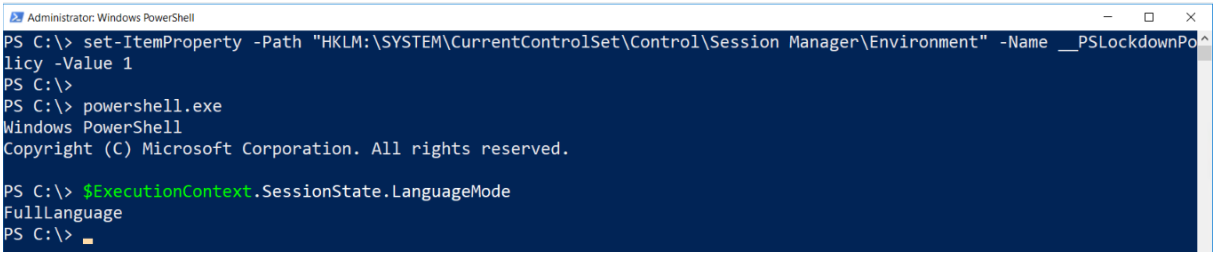
Das Erstellen der Variable geht natürlich auch mit der PowerShell:

```
New-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" -Name __PSLockdownPolicy -Value '4'
```

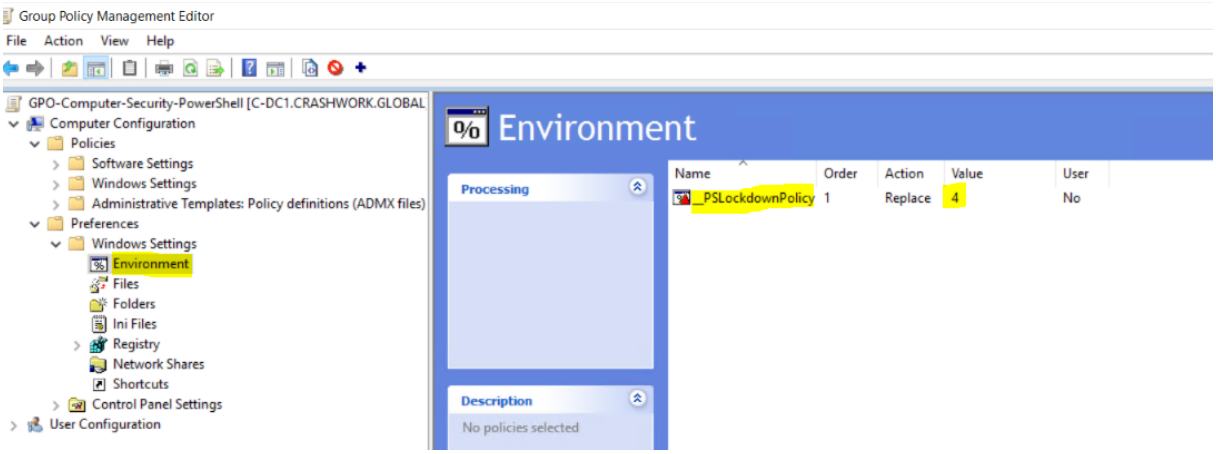


Danach ist jede neue Powershell limitiert:

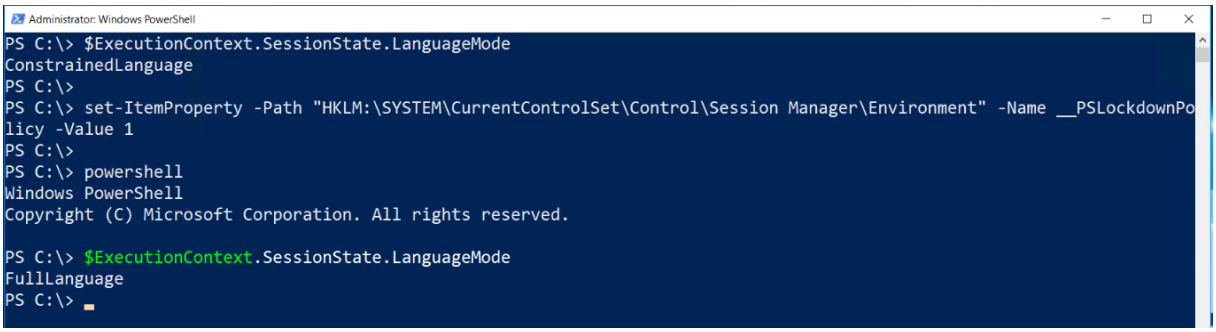
Das ist aber wie ein set-executionpolicy zu bewerten: wer die erforderlichen Rechte hat kann sich einfach darüber hinwegsetzen:



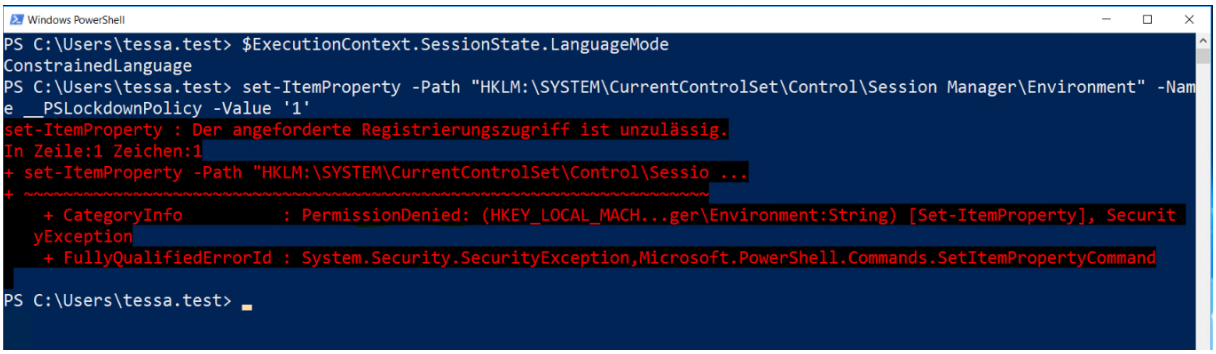
Daher – und auch wegen der zentralen Konfiguration – ist der Einsatz einer GPO sinnvoller. Hier kann man ebenfalls Systemvariablen erstellen:



Nur leider ist der Teil mit den Environment-Variablen in den Preferences zuhause: auch diese kann ein Administrator (wenn auch mit meinem Replace nur bis zum nächsten gpupdate) überschreiben:

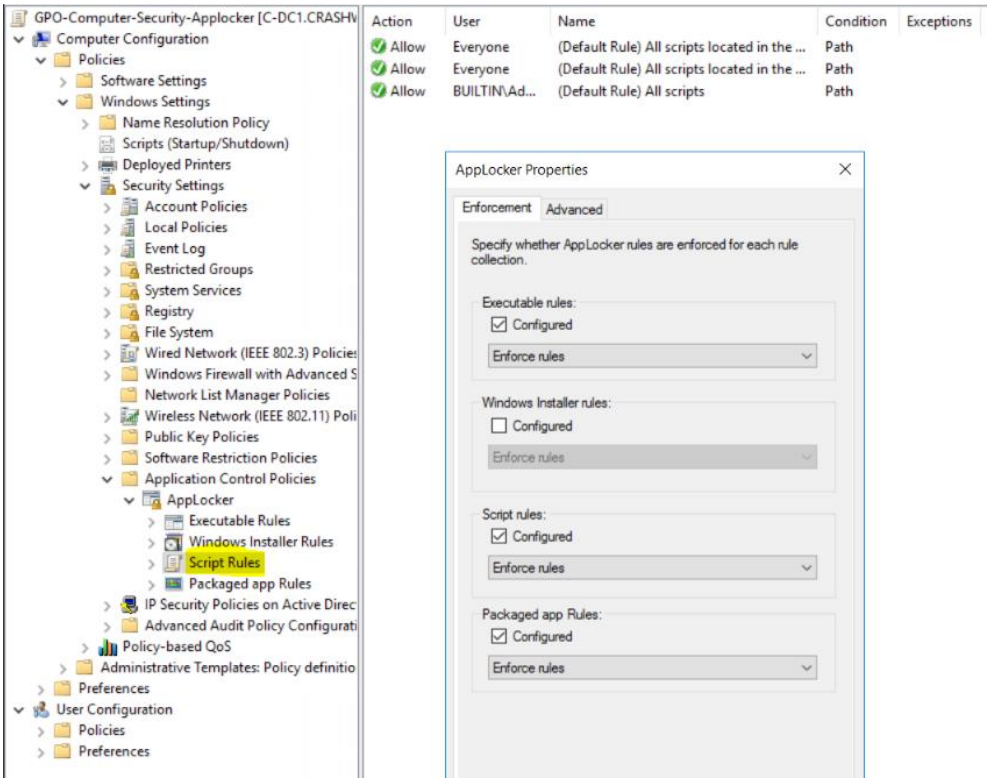


Aber dafür benötigt man eben auch administrative Rechte: ein Standardbenutzer kann sich da nicht mehr rauswinden:

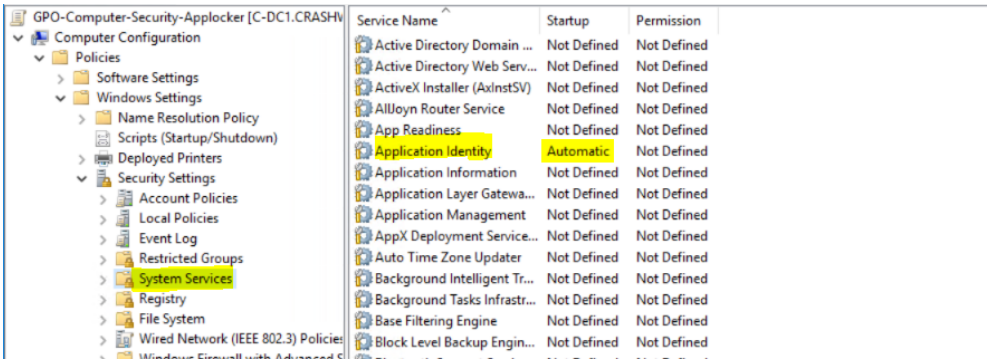


Ich weiß nicht, wie es euch dabei geht, aber mir missfällt diese Lösung. Abgesehen davon wäre dann auch eine unterschiedliche Mode-Konfiguration wünschenswert: für normale Benutzer der ConstrainedMode und für Admins der FullLanguage-Mode.

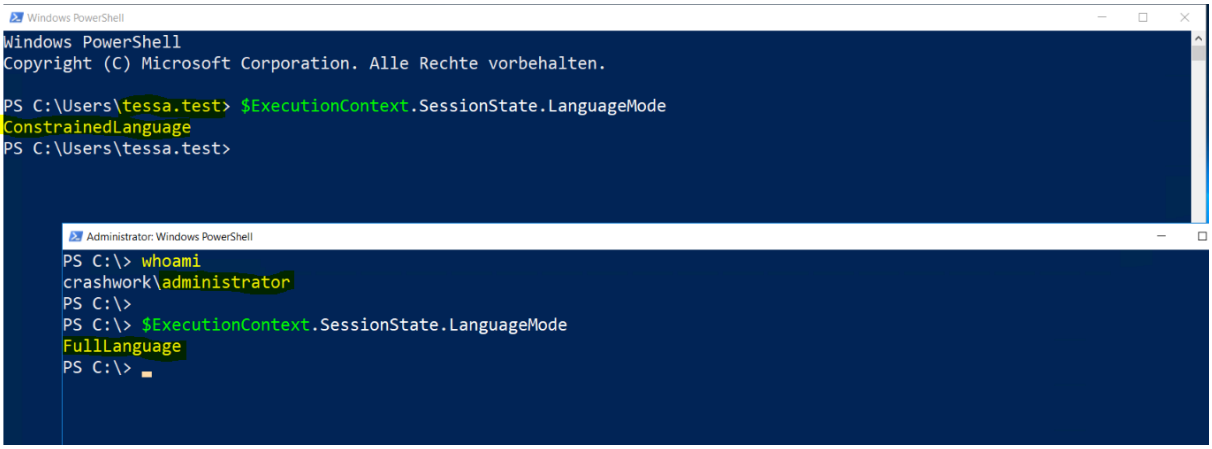
Genau das liefert der ConstrainedMode mit Applocker! Mit aktiviertem und konfigurierten Applocker wird die Anwendung des ConstainedModes für Standardbenutzer erzwungen. Und das kann mit einer einfachen GPO konfiguriert werden:



Bitte vergesst aber nicht, dass der Applocker-Dienst mitgestartet werden muss:

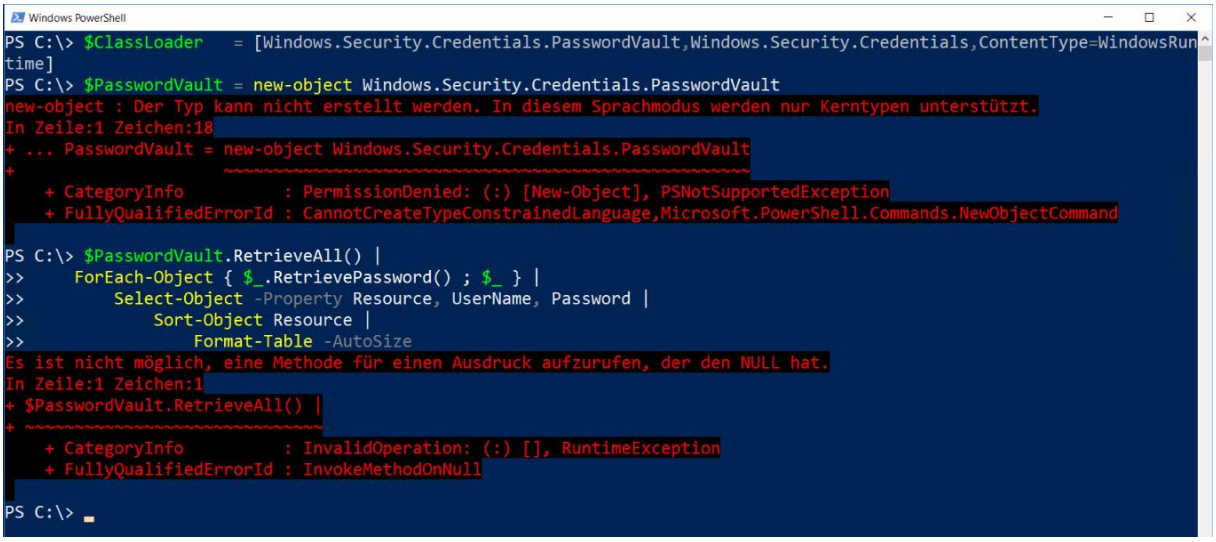


Ebenso wird für den Applocker ein Windows Client in der Enterprise-Edition benötigt. Nach einem gpupdate und ein/zwei Neustarts greift das Regelwerk. Fragt nun ein Standardbenutzer den Language-Mode ab, sieht er den eingeschränkten Modus. Ein Admin hingegen ist im FullLanguage-Mode unterwegs! Und das auch ohne die Systemvariable!



Angriff eines geschützten Systems

Aber wie schützt der Constrained Mode nun eigentlich? Eigentlich recht simpel: es werden entliche für Angreifer interessante Technologien abgeschaltet. Statt den Passwörtern bekommt dieser Angreifer nur Fehlermeldungen. Die erste kennzeichnet die Einschränkung des Sprachmodus. Der zweite Fehler ist daher nur ein Folgefehler.



Egal: ein Ergebnis gibt es nicht mehr! 😊

Nebenwirkungen und nützliche Hinweise

Auch hier ist es mal wieder sinnvoll, nicht immer als Admin unterwegs zu sein. Aber

Man muss die Einschränkungen des Constrained Modes genau kennen. Das wären diese hier:

- Alle cmdlets aus allen Windows Modulen funktionieren wie gewohnt – Ausnahmen können aber nicht ausgeschlossen sein (siehe weiter unten)
- Alle Scriptelemente (Schleifen, Bedingungen, …) sind erlaubt
- Das cmdlet Add-Type kann nur noch digital signierte Assemblies laden
- new-Object kann nur noch „erlaubte“ Typen referenzieren
- nur „erlaubte“ Typen können verwendet werden
- Typkonvertierungen sind innerhalb der „erlaubten“ Typen möglich
- cmdlets mit einer integrierten Typkonvertierung müssen „erlaubten“ Typen verwenden
- Aus .net ist nur noch die ToString() Methode erlaubt
- Es können alle möglichen Typwerte abgefragt werden. Geändert werden können nur noch Objekteigenschaften mit „erlaubten“ Typen

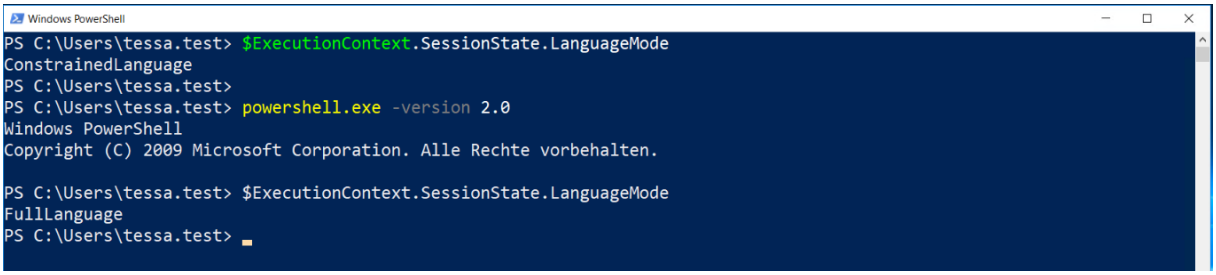
Welche Datentypen sind denn nun noch „erlaubt“?

AliasAttribute	Hashtable	PSTypeNameAttribute
AllowEmptyCollectionAttribute	int	Regex
AllowEmptyStringAttribute	Int16	SByte
AllowNullAttribute	long	string
Array	ManagementClass	SupportsWildcardsAttribute
Bool	ManagementObject	SwitchParameter
byte	ManagementObjectSearcher	System.Globalization.CultureInfo
char	NullString	System.Net.IPAddress
CmdletBindingAttribute	OutputTypeAttribute	System.Net.Mail.MailAddress
DateTime	ParameterAttribute	System.Numerics.BigInteger
decimal	PSCredential	System.Security.SecureString
DirectoryEntry	PSDefaultValueAttribute	TimeSpan
DirectorySearcher	PSListModifier	UInt16
double	PSObject	UInt32
float	PSPrimitiveDictionary	UInt64
Guid	PSReference	

Vielleicht habt ihr bisher auch .net oder andere Datentypen als kleine Hilfsmittel verwendet. Das wird dann problematisch.

Achtung!

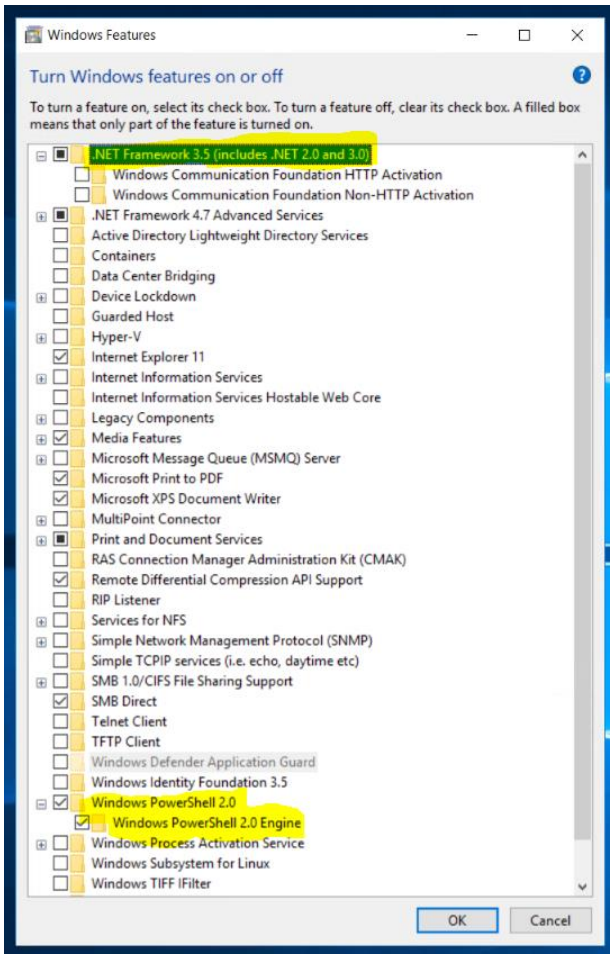
Der Constrained-Mode setzt den Einsatz der PowerShell-Version V5 voraus! Diese gibt es ab Windows 10 per Default an Bord des Betriebssystemes. Wenn aber die PowerShell V2 auf dem gleichen System zuhause ist, kann ein Angreifer OHNE administrative Rechte einfach den Constrained Mode umgehen:



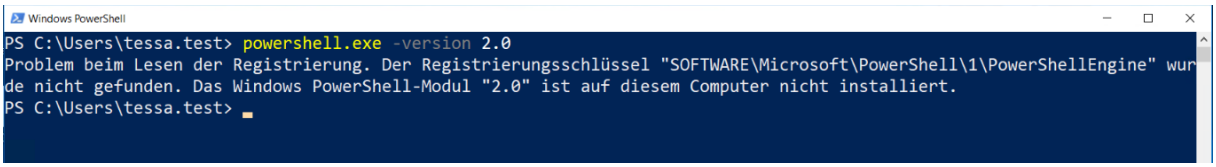
Natürlich ist die „alte“ Powershell nicht ganz so mächtig, wie man in meinem Beispiel sehen kann:

```
Windows PowerShell
PS C:\Users\tessa.test> $ClassLoader   = [Windows.Security.Credentials.PasswordVault,Windows.Security.Credentials,Conten
tType=WindowsRuntime]
Windows.Security.Credentials.PasswordVault,Windows.Security.Credentials,ContentType=WindowsRuntime : Der Typ [Windows.S
ecurity.Credentials.PasswordVault,Windows.Security.Credentials,ContentType=WindowsRuntime] kann nicht gefunden werden:
Stellen Sie sicher, dass die Assembly, die diesen Typ enthält, geladen wird.
Bei Zeile:1 Zeichen:118
+ $ClassLoader   = [Windows.Security.Credentials.PasswordVault,Windows.Security.Credentials,ContentType=WindowsRuntime]
 <<<<
    + CategoryInfo          : InvalidOperation: (Windows.Securit...=WindowsRuntime:String) [], RuntimeException
    + FullyQualifiedErrorId : TypeNotFound

PS C:\Users\tessa.test> $PasswordVault = new-object Windows.Security.Credentials.PasswordVault
new-object : Der Typ [Windows.Security.Credentials.PasswordVault] kann nicht gefunden werden. Stellen Sie sicher, dass
die Assembly, die diesen Typ enthält, geladen wird.
Bei Zeile:1 Zeichen:28
+ $PasswordVault = new-object <<<<  Windows.Security.Credentials.PasswordVault
    + CategoryInfo          : InvalidType: (:) [New-Object], PSArgumentException
    + FullyQualifiedErrorId : TypeNotFound,Microsoft.PowerShell.Commands.NewObjectCommand

PS C:\Users\tessa.test> $PasswordVault.RetrieveAll() |
>>     ForEach-Object { $_.RetrievePassword() ; $_ } |
>>         Select-Object -Property Resource, UserName, Password |
>>             Sort-Object Resource | Format-Table -AutoSize
```

Aber es gibt genug Schadcode der für Windows 7 und damit für die PowerShell V2 geschrieben wurde. Daher würde ich empfehlen, dass die PowerShell V2 aus modernen Systemen entfernt wird. In Windows 10 wird sie zwar per Default erst dann nutzbar, wenn das DotNet-Framework 3.5 installiert ist, aber zur Sicherheit sollte die PS2 als optionales Windows-Feature deaktiviert werden:



Es genügt, die beiden Haken bei der Windows PowerShell zu entfernen. Danach kann ein Angreifer nicht mehr auf die alte Plattform wechseln:



Fazit

Aus meiner Sicht hat Microsoft noch einiges an Arbeit vor sich, um das nützliche Werkzeug PowerShell vernünftig abzusichern. Erste Ansätze sind erkennbar. Ich bleibe auf jeden Fall dran. Ihr auch?

Stay tuned!