


Optimizar la seguridad PowerShell

1. La seguridad PowerShell por defecto

Como podrá entender, la seguridad es algo muy importante, sobre todo en el ámbito del scripting. Es por ello que los creadores de PowerShell incluyeron dos normas de seguridades por defecto.

Los archivos **ps1** asociados al bloc de notas


La extensión «.ps1» de los scripts PowerShell, está por defecto asociada al editor de texto bloc de notas (o Notepad). Este procedimiento permite evitar lanzar scripts potencialmente peligrosos por una mala manipulación. El bloc de notas es ciertamente un editor un poco clásico, pero tiene la doble ventaja de ser inofensivo y no bloquear la ejecución de un script cuando éste está abierto con el editor.

 Este tipo de seguridad no está establecido con los scripts VBS cuya apertura está directamente relacionada al Windows Script Host. ¡Aquéllos que nunca hayan hecho doble clic en un script VBS al querer editarlo que tiren la primera piedra!

Una política de ejecución limitada

La segunda barrera de seguridad es la aplicación de políticas de ejecución «restricted» por defecto (véase las políticas de ejecución).

Esta estrategia es la más restrictiva. Es decir que bloquea sistemáticamente la ejecución de cualquier script. Solamente se ejecutaran los comandos escritos en el shell. Para remediar este bloqueo del script, PowerShell requiere que el usuario cambie el modo de ejecución con el comando `Set-ExecutionPolicy <modo de ejecución>`.

 Tal vez comprende ahora porqué el despliegue de PowerShell en sus máquinas no constituye un aumento de los riesgos, en la medida en que se respeten determinadas normas.

2. Las políticas de ejecución

PowerShell integra un concepto de seguridad que se ha denomina políticas de ejecución (execution policies) para evitar que un script no autorizado pueda ejecutarse sin el conocimiento del usuario. Existen cuatro configuraciones posibles: **Restricted**, **RemoteSigned**, **AllSigned** y **unrestricted**. Cada una de ellas corresponde a un nivel de autorización de ejecución de un script en particular y que podríamos tener la necesidad de cambiar en función de la estrategia que desea aplicar.

a. Las diferentes políticas de ejecución

Restricted: ésta es la política más restrictiva, y es también la política por defecto. No permite la ejecución de script, sólo permite las instrucciones de línea de comandos, es decir únicamente en el shell. Esta política puede considerarse como la más radical, dado que protege la ejecución involuntaria de archivos «.ps1».

Durante un intento de ejecución de script con esta política, un mensaje de este tipo se muestra en la consola:

```
No se puede cargar el archivo C:\script.ps1,
porque en el sistema está deshabilitada la ejecución de scripts.
```

Como esta política se define por defecto durante la instalación de PowerShell, habrá que cambiarla para la ejecución de su primer script.

AllSigned: es la política que permite la ejecución de script más «segura». Autoriza únicamente la ejecución de scripts firmados. Un script firmado es un script que contiene una firma digital como la que se muestra en la figura siguiente.

Ejemplo de script firmado

Con la política **AllSigned**, la ejecución de scripts firmados necesita que se esté en posesión de los certificados correspondientes (véase el apartado acerca de la firma de los scripts).

RemoteSigned: esta política es similar a la **AllSigned** con la diferencia de que sólo los scripts con un origen distinto al local requieren de una firma. Por consiguiente, esto significa que todos los scripts creados localmente podrán ejecutarse sin ser firmados.

Si intenta ejecutar un script procedente de Internet sin que éste esté firmado, se mostrará en consola el mensaje siguiente.

```
No se puede cargar el archivo C:\script.ps1.
El archivo C:\script.ps1 no está firmado digitalmente.
El script no se ejecutará en el sistema.
```

➤ Se preguntará seguramente cómo hace PowerShell para saber que nuestro script procede de Internet. Respuesta: gracias a las «Alternate Data Strings» que están implementadas en forma de flujos ocultos desde aplicaciones de comunicación como Microsoft Outlook, Internet Explorer, Outlook Express y Windows Messenger (véase el apartado relativo a los Alternate Data Streams).

Unrestricted: es la política menos vinculante, y por lo tanto la menos segura. Con ella, todo script, poco importa su origen, puede ejecutarse sin solicitud de firma. Es pues la estrategia donde el riesgo de ejecutar scripts potencialmente peligrosos es más elevado.

Esta estrategia muestra un aviso cuando se intenta ejecutar un script descargado de Internet.

```
PS > .\script.ps1

Advertencia de seguridad
Ejecute sólo los scripts de confianza. Los archivos procedentes de Internet
pueden ser útiles, pero algunos archivos podrían dañar su equipo.
¿Desea ejecutar C:\MiScript.ps1?
```

[N] No ejecutar	[Z] Ejecutar una vez
[U] Suspender	[?] Ayuda (el valor predeterminado es "N"):

PowerShell v2 aporta dos políticas de ejecución suplementarias:

Bypass: no bloquea nada y no se muestra ningún mensaje de advertencia.

Undefined: no existe una política de ejecución definida en el ámbito actual. Si todas las políticas de ejecución de todos los ámbitos son de tipo Undefined entonces la política efectiva aplicada será la política **Restricted**.

b. Los ámbitos de las políticas de ejecución

Esta parte sólo se aplica a PowerShell v2.

PowerShell v2 aporta un nivel de granularidad que PowerShell v1 no tenía en la gestión de las políticas de ejecución. PowerShell v1 sólo controlaba la política de ejecución del ordenador, dicho de otra forma, PowerShell v1 únicamente estaba dotado del ámbito LocalMachine.

Además del ámbito LocalMachine, PowerShell v2 aporta los dos nuevos ámbitos siguientes: Process, y CurrentUser. Si se desea, es posible asignar una política de ejecución a cada ámbito.

La prioridad en la aplicación de las políticas es la siguiente:

- **Ámbito Process:** la política de ejecución únicamente afecta a la sesión en curso (procesos Windows PowerShell). El valor asociado al ámbito Process se almacena únicamente en memoria; por lo que no se conserva al cerrar la sesión PowerShell.
- **Ámbito CurrentUser:** la política de ejecución aplicada al ámbito CurrentUser sólo afecta al usuario actual. El tipo de política se almacena de forma permanente en la parte del registro HKEY_CURRENT_USER.
- **Ámbito LocalMachine:** la política de ejecución aplicada al ámbito LocalMachine afecta a todos los usuarios de la máquina. El tipo de política se almacena de forma permanente en la parte del registro HKEY_LOCAL_MACHINE.

La política con una prioridad 1 es más prioritaria que la de una prioridad 3. Por consiguiente, si el ámbito LocalMachine es más restrictivo que el ámbito Process, la política que se aplicará será al menos la política del ámbito Process. A menos que ésta sea de tipo Undefined en cuyo caso PowerShell aplicará la política del ámbito CurrentUser, y después intentará aplicar la política LocalMachine.

Remarcar que el ámbito LocalMachine es el predeterminado cuando se aplica una política de ejecución sin precisar un ámbito concreto.

c. Identificar la política de ejecución actual

La política de ejecución en curso se obtiene con el commandlet **Get-ExecutionPolicy**.

Ejemplo:

```
PS > Get-ExecutionPolicy
Restricted
```

Con PowerShell v1, no se plantea la cuestión de saber cuál es el ámbito asociado al modo devuelto por este comando. En efecto, PowerShell v1 únicamente gestiona el ámbito LocalMachine.

Con PowerShell v2, nos beneficiamos del switch **-List**. Gracias a él vamos a saber qué políticas se aplican a nuestros ámbitos.

Por ejemplo:

```
PS > Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
MachinePolicy	Undefined
UserPolicy	Undefined
Process	Undefined
CurrentUser	AllSigned
LocalMachine	Restricted

En este ejemplo vemos que en el ámbito `CurrentUser` hemos aplicado la política `AllSigned`, mientras que el ámbito `LocalMachine` está asignado a la política `Restricted` (valor por defecto). Si ha seguido el tema hasta aquí, ¿cuál es, según su opinión, la política que se aplica a nuestra sesión PowerShell en curso?

Para saberlo llamaremos a **Get-ExecutionPolicy**:

```
PS > Get-ExecutionPolicy
AllSigned
```

Pues sí, se trata de la política `AllSigned` ya que el ámbito `CurrentUser` tiene la prioridad sobre el ámbito `LocalMachine`.

Observe que tenemos en la lista de los ámbitos devueltos los ámbitos `MachinePolicy` y `UserPolicy`. Estos ámbitos corresponden respectivamente a los ámbitos `LocalMachine` y `CurrentUser` mientras que las directivas de grupos (GPO) se utilizan para parametrizar el comportamiento de PowerShell en los puestos de los usuarios de un dominio.

El orden de aplicación de las políticas de ejecución nos lo devuelve el comando **Get-ExecutionPolicy -List**. Es necesario saber que las políticas de ejecución definidas por GPO son prioritarias con respecto a las otras.

d. Aplicar una política de ejecución

La política **Restricted** es la política aplicada por defecto en el entorno PowerShell. Ésta no es la más adecuada, ya que no permite la ejecución de los scripts. Le aconsejamos por tanto que opte por una política más flexible, con el fin de poder aprovechar las numerosas ventajas que ofrece el scripting PowerShell.

El cambio de la política de ejecución se realiza con el comando `Set-ExecutionPolicy` seguido del modo escogido.

Por ejemplo: `Set-ExecutionPolicy RemoteSigned` aplicará la política de ejecución `RemoteSigned` al ámbito `LocalMachine`.

Con PowerShell v2, para aplicar una política a un ámbito diferente al `LocalMachine`, es necesario utilizar el parámetro `-Scope` seguido del nombre del ámbito.

Ejemplo: aplicación de la política `RemoteSigned` al ámbito `Process`

```
PS > Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process
```

Cambio de directiva de ejecución

La directiva de ejecución le ayuda a protegerse de scripts en los que no confía. Si cambia dicha directiva podría exponerse a los riesgos de seguridad descritos en el tema de la Ayuda `about_Execution_Policies`.

¿Desea cambiar la directiva de ejecución?

[S] Sí [N] No [U] Suspendir [?] Ayuda (el valor predeterminado es "S"):

El cambio de política de ejecución va acompañada sistemáticamente de un mensaje de advertencia pidiéndole que confirme la acción.

Verificamos ahora el resultado:

```
PS > Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
MachinePolicy	Undefined
UserPolicy	Undefined
Process	RemoteSigned
CurrentUser	AllSigned
LocalMachine	Restricted

```
PS > Get-ExecutionPolicy
RemoteSigned
```



Únicamente con permisos de administrador es posible cambiar la política de ejecución aplicada al ámbito LocalMachine. Por lo tanto, seleccione la opción como administrador al arrancar PowerShell (clic con el botón secundario del ratón **ejecutar como**).



La clave de registro correspondiente al ámbito LocalMachine es la siguiente:
HKEY_Local_Machine\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.Powershell\ExecutionPolicy. Otra forma de configurar la política de ejecución de las máquinas, consiste en utilizar las GPO (*Group Policy Objects*). Para ello, Microsoft facilita un archivo .adm (ver más adelante en este capítulo).

3. Los scripts procedentes de Internet

Si usted ha leído el apartado anterior sabrá que los scripts creados localmente no están sujetos a las mismas obligaciones que los procedentes de Internet.

En primer lugar, con la política `RemoteSigned`, no le será posible ejecutar scripts descargados de Internet si no están firmados o desbloqueados. Sin embargo, incluso firmados, para ejecutarse sin problemas, es necesario que los scripts procedan de una entidad autorizada.

¡Intentemos clarificar un poco este tema! Por ejemplo, cuando descargue un script desde Internet, pasando por las herramientas de comunicación Microsoft (Outlook, Internet Explorer, Outlook Express y Windows Live Messenger), éstas asocian a su script un flujo de datos adicionales llamados Alternate Data Stream (ver apartado acerca de Alternate Data Streams) que permiten a PowerShell identificar la procedencia de un script. Y si el script no tiene una procedencia local, entonces pueden darse dos situaciones distintas:

Caso de un script firmado

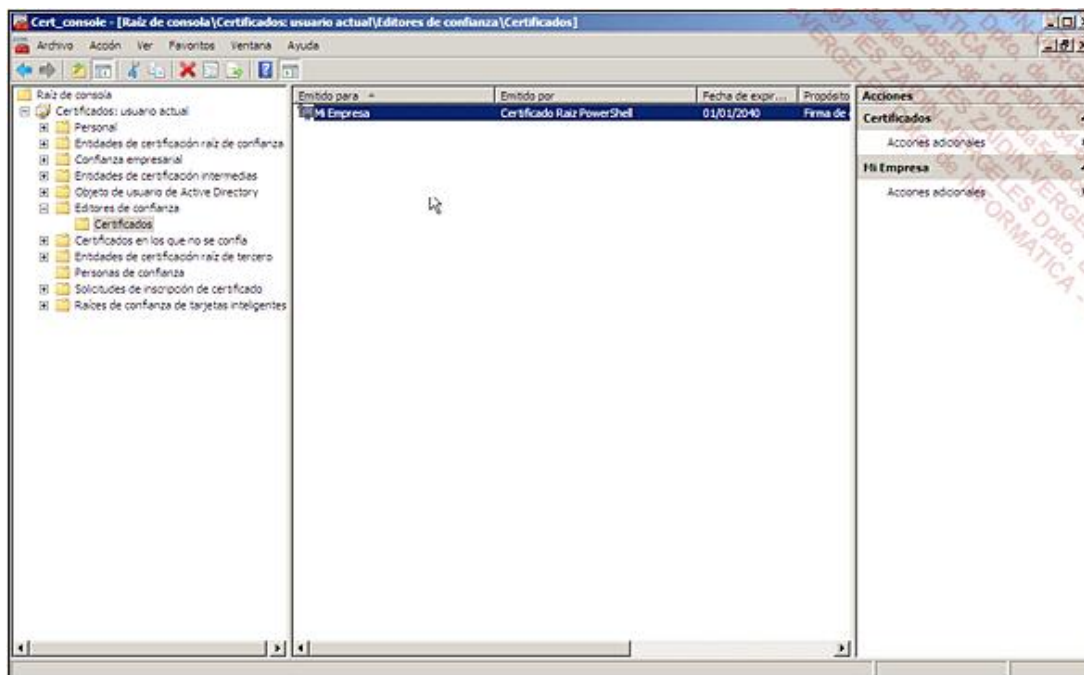
Si el script está firmado digitalmente, es decir, si contiene una firma que permita a la vez identificar el editor y garantizar la integridad del script, entonces PowerShell le preguntará si desea aprobar este editor.

La elección de aprobar o no un editor puede realizarse en el momento de la ejecución del script. En este preciso momento, PowerShell le preguntará si quiere ejecutar el software de un editor no aprobado, y tendrá las siguientes opciones de respuesta:

```
[O] No ejecutar nunca
[N] No ejecutar
[U] Ejecutar una vez
[E] Ejecutar siempre
[?] Ayuda
```

Observe que si elige la opción **No ejecutar nunca [O]** o **Ejecutar siempre [E]**, esta acción sobre el editor no se le volverá a solicitar nunca más.

Al escoger **Ejecutar siempre**, se crea una relación de confianza con el editor, y de ahí, como puede verse en la consola de gestión (véase figura inferior), el certificado correspondiente al editor se importa al almacén de certificados «editores de confianza».



Editores de confianza mostrados en la consola de administración

Observe que para ejecutar un script firmado, se necesita obligatoriamente estar en posesión de un certificado de una Entidad de certificación raíz de confianza correspondiente. Esto es lo que vamos a ver posteriormente en este capítulo (véase el apartado sobre los certificados).

➤ El primer reflejo que se suele tener cuando se recupera un script de Internet u otro lugar, es verificar su contenido. Un ojo crítico identifica rápidamente los posibles riesgos, si el código no es demasiado largo. Y si finalmente, tras inspeccionar el código llega a la conclusión de que no presenta ningún peligro para su sistema, entonces puede ejecutarlo.

Caso de un script no firmado

Si intenta ejecutar un script no firmado procedente de una máquina remota y se encuentra en modo **RemoteSigned**, veamos lo que PowerShell va a responderle.

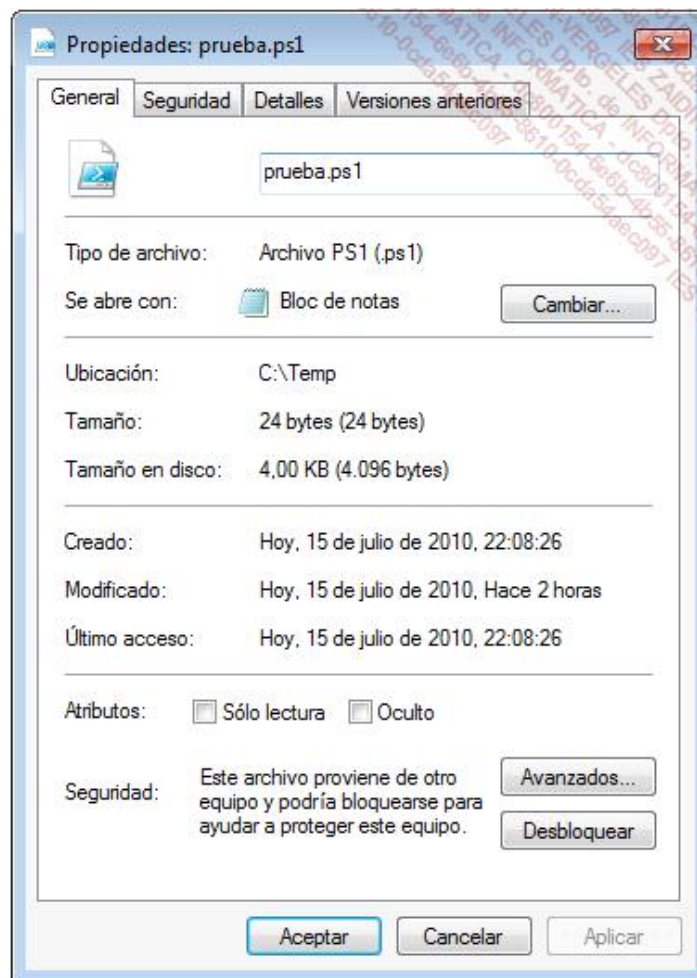
```
No se puede cargar el archivo C:\Temp\prueba.ps1. El archivo
C:\Temp\prueba.ps1 no está firmado digitalmente. El script no se ejecutará
en el sistema. Vea "get-help about_signing" para obtener más información.
```

Sin embargo, es posible ejecutar scripts no firmados. Para ello es necesario realizar lo que llamamos «desbloquear» el script. Desbloquear un script corresponde a suprimir el Alternate Data Stream que contiene la información de su procedencia. Una vez desbloqueado el script, podrá ejecutarlo como si se tratase de un script local.

Evidentemente, se necesita más que nunca inspeccionar detalladamente el script. Recuerde que un script descargado de Internet de un lugar en el que no confíe es potencialmente peligroso.

He aquí los pasos a seguir para desbloquear un script no firmado descargado de Internet.

- Pulse el botón secundario del ratón sobre el script correspondiente y escoja **Propiedades**. Después, en la pestaña **General**, en la parte inferior de la ventana, pulse **Desbloquear**.



Ventana Propiedades que permite desbloquear un script

El script está ahora desbloqueado, por lo que ya no será posible conocer su procedencia.

4. Los Alternate Data Streams (ADS)

a. Los orígenes

Desconocidos por muchos especialistas en informática, los Alternate Data Streams (en castellano: Flujos Alternativos de Datos) no datan de ayer. En efecto, los ADS vieron la luz con el sistema de archivos NTFS (*New Technology File System*) utilizado por la familia Windows desde principios de los años 90 con la aparición de Windows NT 3.1.

El principio de los ADS que no ha evolucionado desde su creación, consiste en insertar los flujos de datos adicionales en un archivo. Hasta aquí nada sorprendente, me podría decir. Sí, pero lo sorprendente es que el contenido, así como el tamaño de los flujos son invisibles. Es decir que puede «ocultar» un ejecutable de varios megabytes en un archivo de texto de algunos bytes sin que el tamaño del archivo, visible por el usuario desde la pestaña **Propiedades**, indique la presencia de los bytes ocupados por el ejecutable. Poco documentados y poco explicados, comprenderá porqué los ADS son a día de hoy un instrumento de numerosos virus. La cuestión es saber por qué estos flujos de datos se han vuelto invisibles.

Lo que hay que saber es que la utilización de los ADS hoy en día ha derivado de su función principal. En su origen los ADS se integraron en los sistemas operativos Windows para permitir una compatibilidad con el sistema de ficheros de Macintosh: el Hierarchical File System (HFS). Porque puede que no sepa que los archivos de Macintosh (sobre los OS anteriores a 'OS X') son el resultado de la asociación de dos componentes: el *Data Fork* y el *Resource Fork*. Como su nombre indica, el Resource Fork contiene los recursos utilizados por una aplicación. Encontrará por ejemplo, elementos

del interfaz gráfico (menús, ventanas, mensajes, etc.) y de otros elementos relacionados a la traducción de la aplicación en diversos idiomas. Y por oposición, el Data Fork contiene el código binario de la aplicación el cual es a priori inmutable. Sin embargo, en el caso de aplicaciones compatibles a la vez en PowerPC y Motorola, el Data Fork contiene las dos versiones del código.

Por una cuestión de interoperabilidad entre sistemas, NTFS ha integrado los data streams. Estos últimos desempeñan el rol del «Resource Fork» versión Windows. Sin embargo, en la práctica, esta voluntad de interoperabilidad no ha llegado a ninguna aplicación concreta. Tanto que a día de hoy, los ADS sirven principalmente en el sistema NTFS para insertar información en los ficheros (tipos de metadatos). Por ejemplo, cuando descargue un script PowerShell desde Internet con Internet Explorer, este último va a crear un Alternate Data Stream en su script para especificar su procedencia. Y es así como PowerShell va a poder determinar si el script que le desea ejecutar fue creado en local o no.



Observe que sólo el tamaño del flujo principal de los ficheros es tomado en cuenta por los gestores de cuotas, el de los ADS se ignora.

b. Crear y leer los ADS

Paradójicamente, PowerShell que utiliza él mismo los ADS para conocer la procedencia de un script, no los gestiona nativamente. Y es por la sencilla razón que el Framework .NET no los gestiona tampoco. Por ello vamos a emplear excepcionalmente el CMD.exe (pero prometo, será la única vez) para leer y crear los ADS.

He aquí las instrucciones sobre la creación de un ADS conteniendo otro archivo de texto.

- Creamos un archivo de texto vacío gracias al comando siguiente:

```
PS > New-Item -name MiArchivo.txt -type file

Directorio: C:\temp

Mode                LastWriteTime         Length Name
----                -
-a---            15/07/2010    23:05             0 MiArchivo.txt
```

El resultado del comando nos permite comprobar que el tamaño de nuestro archivo es cero.

- Creemos después un segundo archivo de texto que esta vez no estará vacío sino que incluirá la ayuda del comando `Get-Process`.

```
PS > Set-Content -path AyudaComando.txt -value $(help get-process)
```

- Verifiquemos ahora el tamaño de nuestros dos archivos con un simple `Get-ChildItem`.

```
PS > Get-ChildItem

Directorio: C:\temp

Mode                LastWriteTime         Length Name
----                -
-a---            15/07/2010    23:07        1763 AyudaComando.txt
-a---            15/07/2010    23:05             0 MiArchivo.txt
```

Puesto que el conjunto de operaciones se ejecutan con CMD.exe, teclee simplemente `cmd` en la consola PowerShell.


```
PS > cmd
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.
```

- Inserte ahora el contenido del texto AyudaComando.txt como ADS del archivo vacío MiArchivo.txt con el comando siguiente:

```
C:\temp> type AyudaComando.txt > MiArchivo.txt:MiPrimerAds.txt
```

Como puede constatar, los ADS de un archivo son accesibles a través del carácter «:» (<nombre_archivo>:<nombre_flujo>) que es por consiguiente un carácter prohibido en la atribución del nombre del archivo.

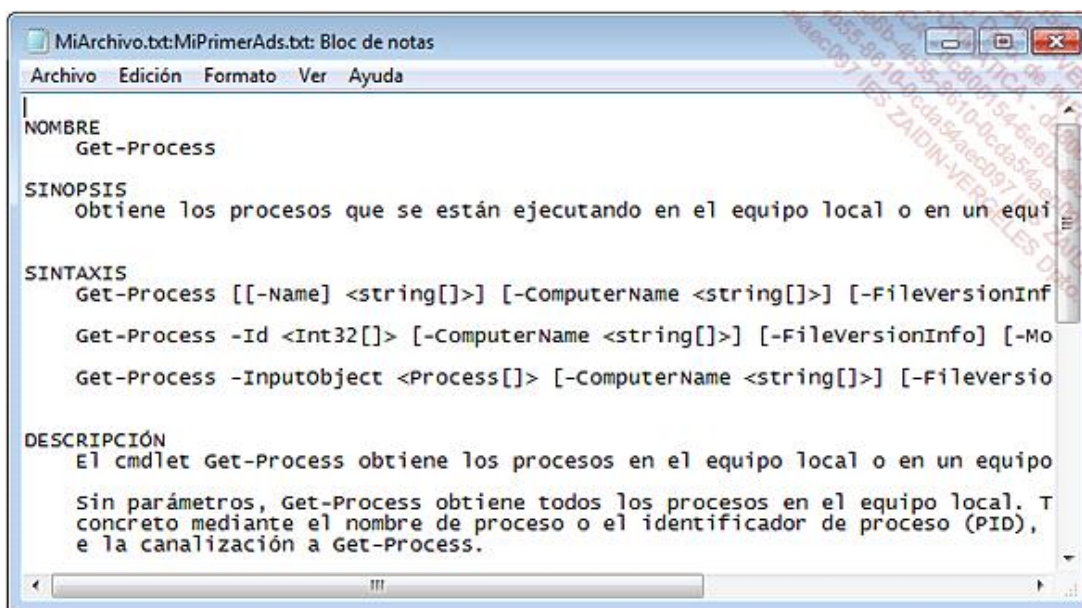
Si intenta leer el archivo MiArchivo.txt utilizando el comando `more` (siempre con `cmd.exe`), como es lógico, no se visualizará nada en la consola ya que el archivo está vacío.

```
C:\temp> more MiArchivo.txt
```

```
C:\temp>
```

Pero si usted prueba de leer el ADS de nombre MiPrimerAds.txt asociado al archivo MiArchivo.txt con el bloc de notas, use el comando siguiente:

```
C:\> notepad MiArchivo.txt:MiPrimerAds.txt
```



Extracto del contenido del ADS

Obtendrá la ayuda en línea del comando `Get-Process`. Sin embargo, el archivo MiArchivo.txt mostrará siempre un tamaño cero. ¡Sorprendente!

```
C:\temp> dir MiArchivo.txt
```

```
Directorio de C:\temp
```

```
15/06/2010  23:12                0 MiArchivo.txt
               1 archivos                0 bytes
               0 dirs 16.484.016.128 bytes libres
```

Procederemos ahora a insertar no un texto sino un ejecutable como ADS de nuestro archivo. Para ello, escriba el

comando (siempre en cmd.exe):

```
C:\temp> type c:\WINDOWS\system32\calc.exe > MiArchivo.txt:calc.exe
```

Verifiquemos de nuevo el tamaño del nuestro archivo.

```
C:\temp> dir MiArchivo.txt

Directorio de C:\temp

15/06/2010  23:17                0 MiArchivo.txt
               1 archivos                0 bytes
               0 dirs 16.484.016.128 bytes libres
```

■ ¡Siempre cero! Finalmente, para ejecutar el ADS, teclee el comando:

```
C:\> start C:\temp\MiArchivo.txt:calc.exe
```

En resumen, es posible ocultar mucho las cosas con los Alternate Data Streams, ya se trate de imágenes, ejecutables, vídeos, etc. brevemente todo flujo de datos puede ser «albergado» por un archivo «padre» sin que el tamaño de este último cambie.



Los ADS son propios de un único ordenador, es decir que los ADS no estarán asociados al archivo si usted lo transfiere (correo, llave USB, etc.).



El lanzamiento de archivos ejecutables en un ADS ya no funciona en Windows 7 y Windows 2008 R2.

c. Observar y comprender los ADS de sus archivos .ps1

Si ha estado atento, sabrá que el modo de ejecución **RemoteSigned** reconoce la procedencia de los scripts gracias a los ADS. Y vamos a ver exactamente qué ADS se crean y qué contienen. Pero la vida de informático no es siempre fácil, es por ello, que no es posible listar los ADS nativamente en Windows. Por tanto obtendremos un ejecutable (streams.exe descargable desde el site de Microsoft) que nos permita de efectuar un listado de los ADS asociados a un archivo.

Utilización del ejecutable Streams

■ Para ejecutar streams.exe, bastará con iniciar una primera vez el archivo setup.exe, después de aceptar la licencia.



Muestra los términos de la licencia referente a la instalación de Streams.exe

Una vez aceptada la licencia, sólo necesita escribir en la consola el nombre completo del ejecutable streams.exe seguido del nombre del archivo o directorio.

El ejecutable streams.exe dispone de dos opciones:

- -s: lista repetidamente todos los ADS asociados al archivo(s) de un directorio.
- -d: suprime los ADS asociados al archivo(s).

Evidentemente, entrever los ADS asociados a un script supone que éste último procede de un instrumento de comunicación Microsoft como Internet Explorer, Windows Live Messenger, Outlook, etc.

Tomemos el ejemplo del script *list-group.ps1* descargado del site www.powershell-scripting.com. Listamos los ADS con el ejecutable Streams.exe utilizando el comando siguiente:

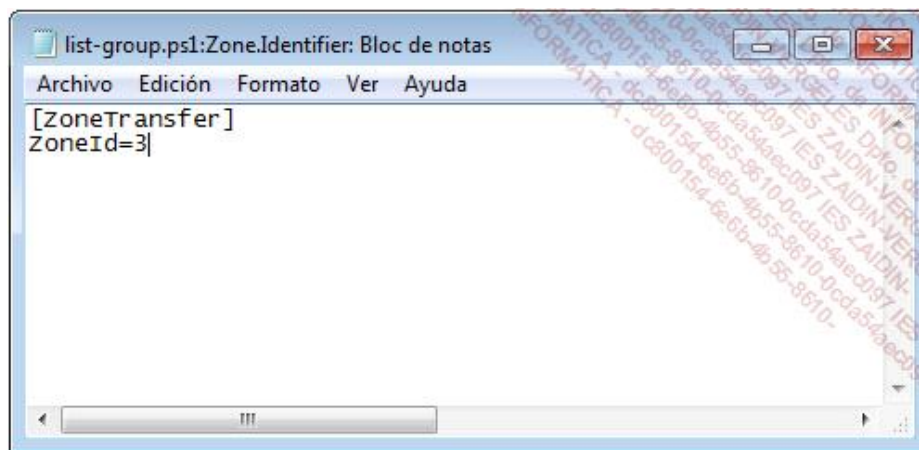
```
PS > ./streams.exe list-group.ps1

Streams v1.56 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\Scripts\list-group.ps1:
:Zone.Identifier:$DATA      26
```

Se observa claramente que un ADS denominado Zone.Identifier ha sido detectado.

Y si se quiere ver lo que contiene, he aquí lo que se va encontrar:



Vista del contenido del ADS Zone.Identifier

De hecho, cuando descargue un script desde un instrumento de comunicación Microsoft, este último va a crear un Data Stream para incluir información sobre su procedencia. Esta información se traduce por un identificador de zona (ZoneID) que puede adoptar diferentes valores según la procedencia del script, y según la seguridad elegida de Internet Explorer. En efecto, el concepto de Zona de Internet es muy utilizado por el navegador. Las modificaciones introducidas en las opciones de seguridad con la adición de sitios/servidores de confianza tienen un impacto directo en ZoneID y por tanto en lo que PowerShell considera como local o remoto.

Zona Internet	Valor	Considerado como local
NoZone	-1	Si
MyComputer	0	Si
Intranet	1	Si
Trusted	2	Si
Internet	3	No
Untrusted	4	No

d. Modificar el ZoneId o cómo transformar un script remoto en script local

Como ha podido observar, el identificador de zona es similar a un trazador que va a seguir a su script para recordar a PowerShell que usted no ha sido el autor.

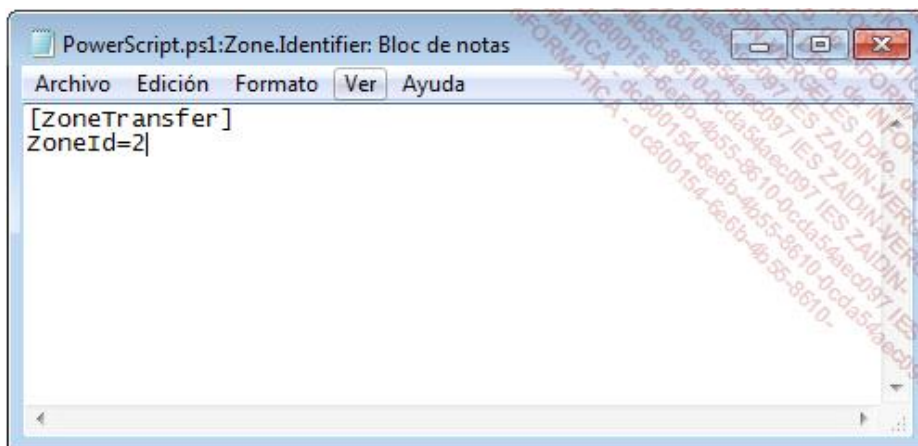
Sólo ahora que se ha familiarizado con los ADS y, en particular con los creados por los instrumentos de comunicación Microsoft, veremos cómo hacer creer a PowerShell que un script es un script local. Para ello, existen dos técnicas:

- La primera consiste, como enunciamos anteriormente en este capítulo, en hacer un clic en el botón secundario del ratón en el script en cuestión y elegir **Propiedades**. Después, en la pestaña **General**. En la parte inferior de la ventana, pulsar **Desbloquear**.
- La segunda es un poco más larga, pero todo y así eficaz, se basa en el cambio del ZoneID. Para modificar el identificador de zona, después del shell, empezamos por abrir el contenido del ADS con notepad:

```
PS > cmd
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.
```

```
C:\> notepad PowerShell.ps1:Zone.Identifier
```

- Después cambiamos el valor del ZoneId (inicialmente a 3 si el script procede de Internet) para poner el valor 2 (Trusted).



Modificación gráfica del ADS Zone.Identifier

- Finalmente, guardaremos el archivo ADS pulsando el botón **Guardar**. Reiniciamos PowerShell para probar de ejecutar nuevamente el script. Y aquí lo tenemos, «Eureka», el script se ejecuta como si se tratase de un script local.

5. Las cadenas securizadas

Saber enmascarar los datos sensibles contenidos en los scripts, debería formar parte de sus tareas habituales. Decimos "debería" ya que todavía hoy en día, son muchos los scripts donde los datos confidenciales están sin cifrar. Existen muchas técnicas para encubrir las cadenas de caracteres en un script, pero la más eficaz es la securización de la cadena aportada por el Framework .NET.

Con PowerShell, hay que distinguir una cadena cifrada de una cadena segura. Se habla de cadena cifrada cuando su contenido se vuelve incomprensible para toda persona que no disponga de una clave de descifrado (véase el apartado sobre el cifrado de cadenas), y se habla de cadenas seguras cuando tienen:

- **Un contenido cifrado:** el contenido de las cadenas seguras se cifra carácter por carácter y luego se registra en la memoria. El cifrado del contenido no requiere ninguna clave, el Framework .NET cifra él mismo los datos.
- **Un acceso controlado de escritura:** una vez creada, a una cadena segura se le puede agregar texto sólo carácter por carácter. Sin olvidar que una metodología propia al tipo `SecureString` permite acceso de sólo lectura, lo que impide cualquier modificación posterior en la cadena.
- **Una no duplicación del contenido en memoria:** PowerShell forma parte de los lenguajes de objetos comúnmente llamados lenguajes de alto nivel. Este tipo de lenguaje tiene la particularidad de no entorpecer al creador del script con algunos detalles de programación, como la asignación o la liberación de la memoria, operaciones confiadas al Garbage Collector (GC) o recolector de basura en castellano. Aunque extremadamente práctico, puede ocurrir que el Garbage Collector efectúe numerosas copias en memoria para optimizar la asignación dinámica de las variables. Esto es lo que se denomina «Mark and Compact». Así, para paliar este problema de seguridad, una cadena `SecureString` se almacena en un espacio de memoria no administrado por el GC, y no se duplica nunca en la memoria. Y una vez que la variable se suprima, el espacio asignado es borrado inmediatamente de la memoria y no deja traza alguna.

a. Securizar una cadena

Para securizar una cadena vía PowerShell, existen dos métodos.

El primer método consiste en utilizar el comando `ConvertTo-SecureString` asociado a los parámetros:

- **-asplaintext**, indica que usted utilizará este commandlet para convertir una cadena estándar en una cadena segura.
- **-force**, indica el hecho de que usted confirma el uso de este commandlet.

Parámetro	Descripción
String	El parámetro string permite determinar la cadena que debe ser descifrada.
SecureKey	El parámetro secureKey permite utilizar una cadena segura como valor clave. En realidad, el valor de la cadena segura se convierte a una matriz de bytes y puede ser utilizada como clave.
Key	El parámetro key determina la clave a utilizar. Como recordará, la clave debe tener una longitud de 128, 192 o 256 bits. Es decir que si está usando una matriz de enteros como clave, reconociendo que un entero está codificado en 8 bits, puede utilizar las matrices de 16, 24 o 32 enteros.
AsPlainText	Este parámetro no se utiliza en el marco del descifrado, sirve únicamente si el comando se usa para transcribir una cadena en una cadena segura.
Force	El parámetro force se utiliza como complemento al parámetro asPlainText indicando que se desea realmente securizar una cadena utilizando asPlainText .

Por ejemplo, veamos la securización de un texto: «Hola»

```
PS > $cadena = ConvertTo-SecureString 'Hola' -asplaintext -force
PS > $cadena
System.Security.SecureString
```

Como puede observar cuando estamos tratando de leer este valor en el Shell, el resultado no aparece, sino que únicamente se muestra el tipo.

El segundo método consiste en introducir un texto en la consola mediante el commandlet **Read-Host** y convertir este texto en cadena segura gracias al parámetro **-AsSecureString**. Ejemplo:

```
PS > $cadena = Read-Host -assecurestring
*****
PS > $cadena
System.Security.SecureString
```

En los dos casos, el objeto devuelto es de tipo **SecureString**, y no puede leerse directamente.

Para tener una visión de lo que es posible hacer con la cadena segura que acabamos de crear, eche un vistazo rápido al cuadro siguiente, donde se listan los diferentes métodos asociados a el objeto **SecureString**.

Método	Descripción
AppendChar	Anexa un carácter al final de la cadena segura actual.
Clear	Elimina el valor de la cadena segura actual.
Copy	Crea una copia de la cadena segura actual.
Dispose	Libera todos los recursos utilizados por el objeto SecureString actual.
GetHashCode	Recupera en formato entero de 32 bits el código del hash.

GetType	Identifica el tipo: <code>SystemString</code> .
get_Length	Devuelve en formato entero de 32 bits la longitud de la cadena.
InsertAt	Inserta un carácter en la cadena segura en la posición de índice especificada.
IsReadOnly	Devuelve el valor booleano <code>True</code> si la cadena está marcada como sólo lectura y <code>False</code> si no lo está.
MakeReadOnly	Hace el contenido de la cadena inalterable. Esta operación es irreversible.
RemoveAt	Quita de la cadena segura el carácter que se encuentra en la posición de índice especificada.
SetAt	Reemplaza con otro carácter el carácter existente en la posición de índice especificada.

Al listar los métodos de un objeto `SecureString` con el comando que ha utilizado desde el comienzo del libro (recuerde `Get-Member`), como buen observador no habrá pasado por alto la falta de dos métodos omnipresentes en los objetos vistos hasta ahora: `Equals` y `ToString`.

No es un olvido por nuestra parte sino más bien una voluntad por parte de Microsoft de no permitir estos métodos con un objeto de tipo `SecureString`, lo que constituiría evidentemente un problema de seguridad. El método `Equals` permite probar si dos objetos son idénticos: si la igualdad es respetada, entonces se devuelve el booleano `true` en caso contrario se devuelve el valor `false`. Solamente este método aplicado a un objeto de tipo `SecureString` devuelve el valor `false` incluso si las dos cadenas seguras son idénticas, ejemplo:

```
PS > $cadena1 = Read-Host -assecurestring
****
PS > $cadena2 = $cadena1.Copy()
PS > $cadena1.Equals($cadena2)
False
```

De este modo, esta seguridad permite evitar el descubrimiento de cadenas mediante procedimientos automatizados de pruebas sucesivas, denominados de «fuerza bruta».

En cuanto al método `ToString`, que permite transformar el objeto cadena de caracteres, devuelve únicamente el tipo del objeto `System.Security.SecureString`.

Veamos de forma más detallada qué ocurre cuando utiliza alguno de los métodos definidos en el cuadro anterior.

- En primer lugar, creamos una cadena segura con el comando `Read-Host` y le insertamos una palabra de cuatro letras:

```
PS > $cadena = Read-Host -assecurestring
****
```

Verificamos seguidamente su longitud con el comando siguiente:

```
PS > $cadena.Get_Length()
4
```

La consola nos muestra la cifra 4.

- Probamos ahora de añadirle un carácter:

```
PS > $cadena.AppendChar('P')
PS > $cadena.Get_Length()
5
```

La longitud de la cadena ha aumentado un carácter.

Sin embargo si intenta insertar varios caracteres, no podrá hacerlo directamente:

```
PS > $cadena.AppendChar('Hola')
No se puede convertir el argumento "0", con el valor: "Hola", para
"AppendChar", al tipo "System.Char": "No se puede convertir el valor "Hola"
al tipo "System.Char". Error: "La cadena debe contener exactamente un
carácter." "
```

Pero como nada es imposible, veamos como sortear el problema:

```
PS > $insert= 'Hola'
PS > For($i=0;$i -lt $insert.length;$i++)
{$cadena.AppendChar($insert[$i])}
```

■ Verificamos si la cadena se ha incrementado correctamente:

```
PS > $cadena.Get_Length()
8
```

■ Hagamos ahora que esta cadena sea de sólo lectura:

```
PS > $cadena.MakeReadOnly()
```

■ Intentemos añadirle un carácter:

```
PS > $cadena.AppendChar('P')
Excepción al llamar a "AppendChar" con los argumentos "1":
"Instancia de sólo lectura."
```

Lógicamente, PowerShell genera un mensaje de error ya que el objeto SecureString ya no es accesible en escritura.

b. Leer una cadena segura

No le sorprenderá si le decimos que no existe un método propiamente dicho para convertir una cadena segura en una cadena clásica. Esto es fácilmente comprensible por el hecho de que eso tendría como consecuencia anular las precauciones adoptadas para cumplir algunos puntos de seguridad enunciados anteriormente. En efecto, si el contenido de una cadena segura se copia en una cadena estándar, entonces será nuevamente copiada en la memoria por el Garbage Collector, y por tanto, dejará de ser confidencial.

Pero como de costumbre, tenemos una vez más una solución a proponerle, ya que existen con el Framework .NET, una clase llamada `Runtime.InteropServices.Marshal` que propone dos métodos:

- `SecureStringToBSTR` que va a permitirle asignar la memoria no gestionada por el Garbage Collector para volver a copiar el contenido de la cadena segura,
- `PtrToStringUni` quien duplicará el contenido de la cadena almacenada en una partición de la memoria no gestionada hacia un objeto de tipo `String` en Unicode, el cual es gestionado por el Garbage Collector.

Ejemplo:

Leer una cadena segura.

En primer lugar, creamos una cadena segura con el comando `Read-Host` y le insertamos una palabra de cuatro letras:

```
PS > $cadenaSec = Read-Host -assecurestring
****
```

■ Procedemos ahora a la lectura de este `SecureString` utilizando los métodos estáticos de la clase `Marshal`:

```
PS > $ptr = [System.Runtime.InteropServices.Marshal]
::SecureStringToBSTR($cadenaSec)
PS > $cadenaenClaro = [System.Runtime.InteropServices.Marshal]::
PtrToStringUni($ptr)
```

Observará que la variable `$cadenaenClaro` es de tipo `String` y contiene el valor de la cadena segura.

Sin embargo, una vez que usted ha recuperado su cadena en claro, es mejor borrarla de la zona de memoria para limitar los riesgos. Esto podría parecer extremo, pero muchos son los instrumentos de piratería que se basan en una lectura de las zonas memoria. Para ello, llevaremos a cabo en primer lugar la liberación del puntero de la cadena no gestionada por el Garbage Collector. Luego sustituimos el contenido de la variable `$cadenaenClaro` por un valor cualquiera, y finalmente forzamos al Garbage Collector a ejecutarse. Traducido a PowerShell esto quedará de la forma siguiente:

```
# Liberar el puntero de la cadena
PS > [System.Runtime.InteropServices.Marshal]::
ZeroFreeCoTaskMemUnicode($ptr)

# Modificación del contenido de la variable $cadenaenClaro por 40 asteriscos
PS > $cadenaenClaro = '*' * 40

# Llamada al Garbage Collector
PS > [System.GC]::Collect()
```

6. El cifrado

El cifrado es una técnica con más de treinta siglos de antigüedad que consiste en transformar una información no codificada (comprensible) en una información que no puede ser comprendida por una persona no autorizada. Esta transformación se realizaba generalmente mediante el intercambio de letras del mensaje (transposición), o por la sustitución de una o varias letras por otras (sustitución).

El esquema siguiente pone en escena a los personajes Alicia y Javier que tratan de comunicarse a través de un canal de transmisión público, tal como Internet.



Envío de un mensaje confidencial

En el esquema superior, las operaciones de cifrado y descifrado se simbolizan por la llave y los envíos/recepciones de mensajes por las flechas.

En esta puesta en escena, Alicia transforma su mensaje editado en claro en mensajes cifrados. Luego lo transmitirá a Javier, que hará la transformación inversa, es decir, descifrar el mensaje. De este modo Alicia y Javier hacen que su mensaje sea incomprensible para Óscar que habría podido interceptarlo durante el intercambio. En efecto, Óscar no tiene la clave: él no sabe cómo Alicia ha cifrado el mensaje ni como Javier va a descifrarlo.

Los primeros sistemas de cifrado estaban esencialmente basados en el alfabeto, como el famoso código llamado «César», donde cada letra del texto a cifrar se sustituía por otra letra situada en la enésima posición más lejos en el alfabeto. Así, si el desfase es de 1 la A vale B y la B valdrá C. Por ejemplo, si tomamos un desfase de 3.

alfabeto original:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
alfabeto codificado:	D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Lo que dará:

Mensaje original = PowerShell es fácil

Mensaje cifrado = SrzhuVkhoo hv idflo

➤ La ventaja de utilizar una cadena cifrada es que se puede grabar en un archivo para reutilizarla posteriormente, lo que no es posible, con una cadena segura.

Noción de clave de cifrado

Un cifrado se compone generalmente de un algoritmo fijo al que se asocia una clave variable. En el caso del código César, el algoritmo es el desfase de letras en el alfabeto y la clave es el número de desfase a aplicar. Por ejemplo, si le damos el mensaje siguiente: «Hqkrudexhd hv xvwhg xq fudfn» y le decimos que está cifrado con el código César no podrá descifrarlo. Es necesario que le indiquemos el valor de la clave que corresponde al número de desfases a aplicar. En este caso, si le decimos que la clave es 3, podrá entonces descifrar el mensaje.

Ejemplo:

Script que cifra un mensaje con el código César.

Como puede constatar, el script necesita los parámetros -texto y -clave que contienen respectivamente el texto a cifrar y la clave a utilizar.

```
# Script Cesar.ps1
```

```
# Cifrado de un mensaje gracias al código César

Param ([string]$texto, [int]$clave)

$mensaje_original = $texto
$alfabeto_MAY='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
$alfabeto_MIN='abcdefghijklmnopqrstuvwxyz'

for($i=0;$i -lt $mensaje_original.length;$i++)
{
    $encontrado = 0
    for($j=0;$j -lt $alfabeto_MAY.length;$j++)
    {
        $tmp = $clave
        While(($j+$tmp) -ge $alfabeto_MAY.length)
        {
            $tmp -= 26
        }
        If($mensaje_original[$i] -ceq $alfabeto_MAY[$j])
        {
            $mensaje_modif += $alfabeto_MAY[$j+$tmp]
            $encontrado = 1
        }
        ElseIf($mensaje_original[$i] -ceq $alfabeto_MIN[$j])
        {
            $mensaje_modif += $alfabeto_MIN[$j+$tmp]
            $encontrado = 1
        }
    }
    if(!$encontrado) {$mensaje_modif += $mensaje_original[$i]}
}

Write-host "`nMensaje Original: $mensaje_original "
Write-host "`n`nMensaje Codificado: $mensaje_modif `n"
```

Resultado en la consola PowerShell:

```
PS > ./cesar.ps1 -texto "PowerShell es fácil" -clave 14

Mensaje Original: PowerShell es fácil

Mensaje Codificado: DksfGvszz sg toqgwz
```

El cifrado con clave simétrico


También se denomina cifrado con una clave, éste es el sistema que utiliza PowerShell para cifrar un mensaje. Su principal característica es que el emisor y el receptor utilizan ambos la misma clave para cifrar y descifrar el mensaje. En el ejemplo del código César, si el emisor cifra su mensaje con una clave de 13, el receptor deberá utilizar el mismo valor para efectuar la rotación en sentido contrario y poder así descifrar el mensaje. Por ello es un sistema simétrico.

Evidentemente este principio exige que la clave siga siendo secreta durante toda la transacción.

El cifrado de cadenas con PowerShell está basado en el algoritmo de Rijndael en versión AES por *Advanced Encryption Standard* (estándar de Cifrado Avanzado).

Este sistema, creado a finales de los años 90 por dos investigadores belgas, utiliza el principio de cifrado con clave simétrico de longitud 128, 192 o 256 bits. Por tanto, en el cifrado de mensajes de correo, necesitará anotar

cuidadosamente su clave.


 El sistema de clave simétrico tiene sus límites cuando varias personas tratan de transmitir mensajes cifrados entre ellas. Si cinco personas constituyen una red de intercambio de mensajes secretos, cada persona debe conocer la clave secreta de las otras cuatro personas. Lo que constituye ya un buen número de claves.

a. Cifrar una cadena

El cifrado de cadenas con PowerShell se realiza a partir del commandlet siguiente: `ConvertFrom-SecureString`. El nombre explícito del comando («convertir desde una cadena segura») deja entrever que para encriptar una cadena, primero hay que asegurarse de que se trate realmente de una cadena segura de tipo `SecureString`. Tendrá por tanto que transformar una cadena de caracteres en una cadena segura y luego utilizar el comando `ConvertFrom-SecureString`.

Este comando dispone de 3 parámetros (aparte de los comunes), veamos el detalle:

Argumento	Descripción
<code>secureString</code>	El parámetro <code>secureString</code> permite determinar la cadena a cifrar. Compruebe que esta cadena sea de tipo <code>SecureString</code> .
<code>key</code>	El parámetro <code>key</code> determina la clave a utilizar. Como información, la clave debe tener una longitud de 128, 192 o 256 bits. Es decir, si está usando una matriz de enteros como clave y cada entero está codificado en 8 bits, usted puede usar las matrices de 16, 24 o 32 enteros.
<code>secureKey</code>	El parámetro <code>secureKey</code> permite utilizar una cadena segura como valor de clave. En realidad, el valor de la cadena segura se convierte en matriz de bytes y puede ser utilizada como clave.

 Si no se especifica ninguna clave, PowerShell utilizará la API Win32 DPAPI (Data Protection API) para cifrar y descifrar los datos.

Para entender mejor como cifrar un texto, veamos algunos ejemplos.

Ejemplo:

Cifrar una cadena sin clave.

En primer lugar, empezaremos por crear una cadena segura que va a contener nuestra información confidencial:

```
PS > $secure_string_pwd = ConvertTo-SecureString `
"Código entrada edificio: 101985" -asplaintext -force
```

Convertimos posteriormente la cadena segura en cadena numérica con el commandlet `ConvertFrom-SecureString` sin especificar la clave, y dirigiremos el resultado a un archivo de texto:

```
PS > ConvertFrom-SecureString $secure_string_pwd > c:\cadena_c1.txt
```

Al recuperar el contenido del archivo mediante el commandlet `Get-Content`, se observa que éste contiene gran cantidad de datos cifrados.

```
PS > Get-Content c:\cadena_c1.txt

01000000d08c9ddf0115d1118c7a00c04fc297eb01000000e9af423c8b1f5b46aa63acdd15e
faa800000000020000000000106600000001000020000000b578b7a5c290ad6df21d1a35e783
```



```
467f708a17eba662cb94aedac3066625f111000000000e8000000002000020000000038e4f1e
51e6bf45080af69ea51aefff38f7b381398b524e489c4db7ce7d1ebc5000000048914285c76c
e259bc0e68d4aefe82d24801fe7c2c346a97831dfebc6b7aae5e480bcec57ae0e716d6d3d6a7
e83da18f8cfbf5c2339d08e3eee7b977ca4f7c7e9dc4040c8b4e49a955349d736c1b58f64000
00004ed6ef86ee6d68db30405fb6ebafee44c4a99b786c6cbb4318eee927f74c596289164f72
5f0843cffa472ba5ald6148c63c29073d49748cce0166986aaf6380e
```

Ejemplo: cifrar una cadena con una clave de 256 bits.

Veamos ahora más detalladamente cómo cifrar una cadena utilizando una clave de 256 bits, es decir de 32 bytes (32 x 8 = 256). Empecemos de nuevo por crear la cadena segura:

```
PS > $secure_string_pwd = ConvertTo-SecureString "Código entrada
edificio: 101985" -asplaintext -force
```

Después, creamos nuestra clave de 32 bytes especificando los valores inferiores a 256 y la asignamos al comando **ConvertFrom-Securestring** mediante el parámetro **Key**, para finalmente enviar todo a un archivo de texto:

```
PS > $clave = (6,10,19,85,4,7,19,87,13,3,20,13,3,6,34,43,56,34,23,14,87,56,
34,23,12,65,89,8,5,9,15,17)

PS > ConvertFrom-SecureString -secureString $secure_string_pwd `
-key $clave > c:\cadena_c2.txt
```

El archivo contiene la información cifrada, pero esta vez, ha sido con la clave de 256 bits que nosotros mismos hemos especificado.

```
PS > Get-Content c:\cadena_c2.txt

76492d1116743f0423413b16050a5345MgB8AGQAaQBQAEsAZQBUBuAHYAUQBBSAFMATQBQAFQ
AagBlAHoAMgBLADgA0ABUAHcAPQA9AHwANAAyADUAMwA1AGYAYwA1ADYAYQA5ADcAMAAyAD
IANwA1AGMAZABjAGQA0QA3AGUAMgBkADUANAA4ADEAYwAyADgANgBkAGEAMwAyADEAZgAzA
DYAZABmAdcANwAwADAA0AA0AGIAZQBjADYAOAAwADUAMwBmADUAMAA5AGMAMAA3AGYAZQBi
ADYANAA1ADcAYwA5ADUANwBkADIAMABiAGYAOQA1AGIAZgAyADAAMQA4ADYAMgAyAGQAOQB
iADMAZAA2ADcAZAAwADUANwA0ADMANwBhAGIAMgA1ADQAMQA3ADgANwAyADcAMQAYADkAMw
A2ADEANQBkADgAZABjADIAZAA5ADUAZAA5ADMAZgAxAGYAMABmAGQAMwA0ADgAZgBiAGUAA
QA5ADgANgA1ADEAMABhADYAOAAwADgAOAA=
```

Ejemplo: cifrar un texto con una cadena segura.

Finalmente, para terminar, tratemos ahora de cifrar un texto con una clave de 128 bits utilizando una cadena de caracteres como clave, lo que es mucho más práctico de retener ya que serán 16 números.

Se sabe que una variable de tipo «char» se destina a representar cualquiera de los 65.536 caracteres Unicode en dos bytes (16 bits). Por consiguiente, es necesario utilizar 8 caracteres (128/16=8) para alcanzar 128 bits. Para que una cadena pueda ser utilizada como clave, ésta debe ser absolutamente segura.

Empecemos por crear nuestra cadena segura que va a servirnos de clave (con 8 caracteres exactamente):

```
PS > $clave = ConvertTo-SecureString 'tititata' -asplaintext -force
```

Después, una vez más, securizamos la cadena que contendrá nuestra información confidencial:

```
PS > $secure_string_pwd = ConvertTo-SecureString `
"Código entrada edificio: 101985" -asplaintext -force
```

Y para finalizar, ciframos la cadena con el commandlet **convertFrom-SecureString**, pero esta vez especificaremos el

parámetro -securekey asociado a la cadena cifrada que nos sirve de clave:

```
PS > ConvertFrom-SecureString -secureString  
$secure_string_pwd `  
-securekey $clave > c:\cadena_c3.txt
```

Y si miramos el contenido del archivo, observamos naturalmente una cadena cifrada, pero que tendrá por clave de cifrado, una cadena segura.

```
PS > Get-Content c:\cadena_c3.txt  
76492d1116743f0423413b16050a5345MgB8AFEAVgB6AHYAaAB5AFcARwB4AGEANgBGAFYAMg  
BkADMALwAlAGQAUwBzAFEPQA9AHwAZABLAGUANABjADcANgAxADYAOQBLAGIAOQBmAGIAMQAw  
AGYAZgBhAGMAZQA4ADgAZAA5AGYAYwAxAGUANwBjADYAOAA2AGYAZABmADQAZgBhAGYANABhAD  
EAZQB1ADcAOQBjADMAZQA2ADUAMQAZAGYAMAA5AGUAZgBjADYANQBhAGIANwAwADMAYQA1ADAA  
YwAzAGUAMQAxAGMAYwBjAGUAMgBiAGYAOQBhADgAZABkADcAZgA3ADIAMABhAGEAOQBhADIAZg  
BjAGQANAAzAGUAOQA2AGUAOQA1AGYANQBkADQANQA1ADIANQAZADAAOQA0ADkAYwAzADkAZAA1  
AGYAMAAyADAAZgAzADMAZgBhADgAYwBjADQAZQAwADgAZgA4AGUAZgA2AGIAMABkADMANAA2AD  
IAOAA3AGYAMQA=
```

b. Descifrar un texto

Retomemos el texto cifrado en la sección anterior mediante una clave de 256 bits. Para lograr la operación inversa, es decir descifrar, utilizaremos el comando `ConvertTo-SecureString` acompañado de la clave correspondiente:

```
PS > $cadena_cifrada = Get-Content c:\cadena_c2.txt  
PS > $cadena_original = ConvertTo-SecureString -key $clave -string  
$cadena_cifrada
```

A raíz de este comando, la variable `$cadena_original` no contiene el texto en claro, sino el texto en formato de cadena segura.

He aquí lo que PowerShell nos muestra en la consola al leer la variable `$cadena_original`:

```
PS > $cadena_original  
System.Security.SecureString
```

Faltará entonces utilizar un método del Framework .NET para ver lo que nos esconde esta cadena.

```
PS > $ptr = [System.Runtime.InteropServices.Marshal]  
::SecureStringToBSTR($cadena_original)  
PS > [System.Runtime.InteropServices.Marshal]  
::PtrToStringUni($ptr)
```

Y naturalmente, el resultado corresponde al texto introducido inicialmente.

```
Código entrada edificio: 101985
```

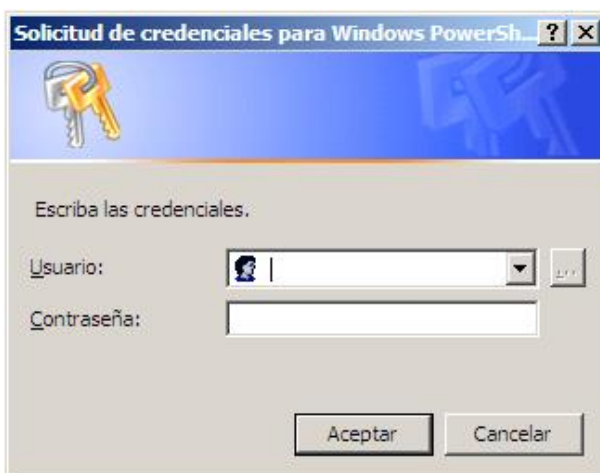
7. Las credenciales

En el complejo mundo de la seguridad informática, la primera forma de minimizar los riesgos es no trabajar normalmente con una cuenta administrador. Aunque como ya sabemos, esta regla está lejos de aplicarse de forma generalizada. Es evidente que efectuar varias veces la misma operación, por ejemplo: cerrar una sesión de usuario para abrir otra con una cuenta administrador para efectuar una acción concreta, y después cerrarla para finalmente reabrir la sesión con su cuenta de usuario limitada puede hacernos perder la paciencia...

Observe que podemos iniciar una consola PowerShell con la opción **Ejecutar como administrador**. En este modo, todos

los comandos escritos en la consola se ejecutarán con los permisos de administrador.

El comando `Get-Credential` de un género en particular permite obtener las «credentials», es decir la pareja usuario/contraseña de un usuario. Así, gracias a este mecanismo, un usuario puede autenticarse bajo otro nombre. Utilizado en la consola, `Get-Credential` muestra una interfaz gráfica, que propone introducir un nombre de usuario así como la contraseña asociada, y devuelve en forma de un objeto `PSCredential` el nombre de usuario en claro al igual que la contraseña en formato de cadena segura.



Interfaz gráfica del commandlet `Get-Credential`

Ejemplo:

Valor de retorno para el nombre de usuario «Administrador».

UserName	Password
-----	-----
\Administrador	System.Security.SecureString

Este comando dispone también del parámetro opcional: `-credential` que permite introducir el nombre de usuario directamente en la línea de comandos. De esta forma, sólo tendremos que informar la contraseña.

Supongamos otro ejemplo con la supresión de un archivo con el comando `Remove-Item`. Si está efectuando el comando con una cuenta limitada que no tiene ningún derecho sobre este archivo, he aquí lo que PowerShell muestra:

```
Remove-Item: Imposible suprimir el elemento C:\temp: El acceso
a la ruta 'C:\temp' está denegado.
```

Ahora, aplicamos a este mismo comando los derechos de un usuario autorizado a suprimir este tipo de archivo. Para ello, basta con añadir al parámetro `-credential` el valor devuelto por el comando `Get-Credential`.

Ejemplo:

```
PS > Remove-Item ArchivoASuprimir -credential $(get-credential)
```

Así, durante la ejecución, la ventana asociada al comando `Get-Credential` propone entrar la información de identificación de un usuario con los derechos apropiados, y transmite dicha información al commandlet que podrá entonces ejecutarse como el usuario especificado. Aquí tiene un conjunto de los comandos que cuentan con `-credential` entre sus parámetros.

Add-Content	Move-ItemProperty
Clear-Content	New-Item
Clear-Item	New-Item-Property

Clear-ItemProperty	New-PSDrive
Copy-Item	New-Service
Copy-ItemProperty	Remove-Item
Get-Content	Remove-ItemProperty
Test-Path	Rename-Item
Get-Item	Rename-ItemProperty
Get-ItemProperty	Resolve-Path
Get-WmiObject	Set-Content
Invoke-Item	Set-Item
Join-Path	Set-ItemProperty
Move-Item	Split-Path

Ejemplo:

Creación de archivos con utilización de credenciales.

Tomamos una situación habitual, la creación de un archivo. Para que se realice esta operación, es preciso que usted disponga de los permisos adecuados. Centrémonos en lo que ocurre durante un intento de crear un archivo sin disponer de los permisos suficientes para hacerlo.

```
PS > New-Item -name prueba.txt -type file

New-Item: El acceso a la ruta 'C:\Users\Oscar\Documents\Temp\
prueba.txt' está denegado.
En línea: 1 Carácter: 9
+ New-Item <<<< -name prueba.txt -type file
```

- Probemos ahora de utilizar el mismo comando, pero añadiéndole el parámetro `-credential`. Para ello, tecleamos este comando:

```
PS > New-Item -name prueba.txt -type file -credential `
$(get-credential -credential administrador)
```

El cuadro de diálogo que nos pedirá nuestra información de autenticación se abre, y al introducirle el usuario/contraseña con los permisos de acceso adecuados se ejecuta correctamente la acción.

Ejemplo:

Utilización de un objeto WMI utilizando credenciales.

Todo y que no hemos abordado todavía los objetos WMI (*Windows Management Instrumentation*), vamos a hacer un pequeño salto hacia adelante para mostrarle cómo aplicar las autenticaciones de usuario con peticiones WMI remotas. Imaginemos por un instante que por alguna razón, necesitamos preguntar a la máquina local para conocer el nombre de los discos lógicos presentes. La petición local es la siguiente:

```
PS > foreach($i in $(Get-WmiObject Win32_LogicalDisk)){ $i.name }
```

Hasta aquí no presenta una dificultad especial. Ahora, para efectuar esta petición en un ordenador remoto debemos utilizar el parámetro `-computer`. Por suerte para nosotros, `Get-WmiObject` soporta credenciales; podremos entonces pasarle las credenciales de administrador para acceder a esta máquina.

```
PS > foreach($i in $(Get-WmiObject Win32_LogicalDisk `
-credential $(get-credential) -computer 192.168.1.2)){ $i.name }
```

8. Enmascarar una contraseña

Seguramente conoce la expresión: «¡Para vivir felices, vivamos ocultos!» Pues bien, esta máxima se aplica también a las contraseñas. La contraseña es un elemento crítico de la cadena «seguridad informática» y es un error introducirla en claro en un archivo. Es como llevar escrito su número secreto en la tarjeta de crédito.

Antes de pensar en ocultar la introducción de contraseñas, valore en primer lugar si el uso de soluciones como las `credentials` (ver sección Las credenciales, de este capítulo) o también la ejecución de scripts con una cuenta de servicio cubrirían sus necesidades.

Y si usted realmente necesita hacer que el usuario introduzca una contraseña durante la ejecución del script, le sugerimos varios métodos para hacer esta operación lo más confidencial posible.

a. Utilización del comando `Read-Host`

Como hemos visto en el apartado de las cadenas seguras, el comando `Read-Host` asociado al parámetro `-AsSecureString` permite ofrecerle confidencialidad al introducir una contraseña. Tecleando el siguiente comando, cada carácter introducido en el Shell se traducirá mostrando por pantalla el carácter asterisco (*), y cuando ha finalizado de introducir la contraseña, la variable `$psw` recibirá un objeto de tipo `SecureString`.

```
PS > $psw = Read-Host -assecurestring  
****
```

La utilización del comando `Read-Host` es a la vez la forma más sencilla de enmascarar la introducción de una contraseña y la forma de manejo menos fácil.

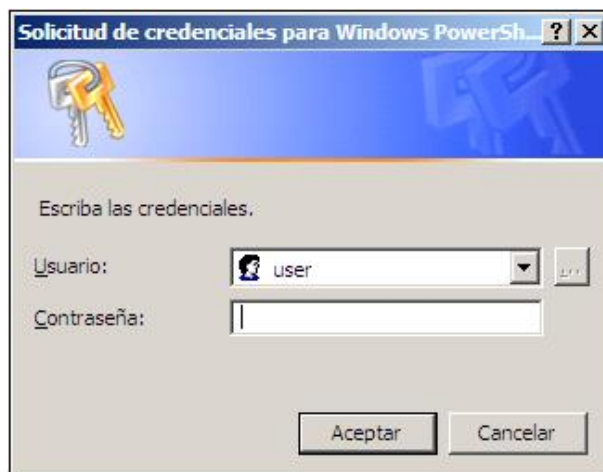
b. Utilización del commandlet `Get-Credential`

Aunque la finalidad de este comando es recuperar la información asociada a una cuenta diferente a la del usuario en curso, puede servir también para recuperar una contraseña haciendo un uso indebido del mismo.

Tecleando el comando siguiente, el interfaz gráfico de `Get-Credential`, nativo de PowerShell, le invita a introducir una contraseña que almacenaremos, en formato de cadena segura, en la variable `$password`. Observe que el campo «usuario» ya está informado, esto es gracias al parámetro `-credential`.

Ejemplo:

```
PS > $password = (get-credential -credential 'user').password
```



Interfaz gráfica Get-member preinformada



Si sólo le interesa la contraseña, el nombre de usuario no tiene ningún interés y tan sólo utilizará la propiedad `password` del objeto `PSCredential`. Sin embargo, si desea también recuperar el nombre de usuario, puede

c. Utilización de una interfaz gráfica personalizada

Una tercera opción consiste en desarrollar una pequeña interfaz gráfica que podrá utilizar posteriormente en diversas aplicaciones. La ventaja es que usted puede personalizarla para satisfacer mejor sus necesidades. En la que proponemos, la contraseña se devuelve mediante \$retorno en formato de cadena segura. Aunque no hemos comenzado a ver formularios gráficos de PowerShell (véase el capítulo .NET), esta función le ofrece una breve reseña de las posibilidades gráficas ofrecidas gracias al Framework .NET.

```
#Get-password.ps1
#Interfaz gráfica que permite recuperar una contraseña

# Carga del assembly para los formatos gráficos
[void][System.Reflection.Assembly]::LoadWithPartialName('System.windows.forms')

# creación del formato principal
$form = new-object Windows.Forms.form
# dimensionamiento del formato
$form.Size = new-object System.Drawing.Size(360,140)
$form.text = 'Get-Password'
# Creación botón validar
$boton_validar = new-object System.Windows.Forms.Button
$boton_validar.Text = 'Validar'
#Posicionamiento del botón
$boton_validar.Location = new-object System.Drawing.Size(135,60)
# Ajuste del tamaño
$boton_validar.size = new-object System.Drawing.Size(90,25)
# Creación de un cuadro de texto
$txtbox = new-object System.Windows.Forms.textbox
#Posicionamiento de la zona de texto
$txtbox.Location = new-object System.Drawing.Size(80,20)
# Ajuste del tamaño
$txtbox.size = new-object System.Drawing.Size(200,25)
# Utilización del método que permite ocultar el texto introducido
$txtbox.set_UseSystemPasswordChar($true )

# Acción del botón Validar
$boton_validar.add_Click(
{
    # Conversión de la cadena recibida en cadena securizada
    $form.hide()
    $result = ConvertTo-SecureString -string "$($txtbox.get_Text())" -asplaintext
-force
})

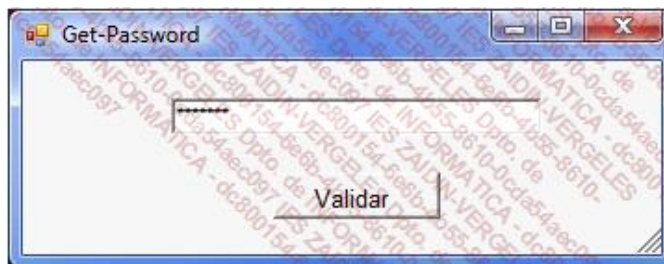
#Adición de componentes y visualización del formato
$form.Controls.Add($boton_validar)
$form.Controls.Add($txtbox)
$form.Add_Shown(
{
    $form.Activate()
})
```



```
[void]$form.ShowDialog()  
$result
```

Observe que utilizamos un método particular del objeto `TextBox`:

`Set_SystemPasswordChar`: permite remplazar el texto introducido en la zona de texto por asteriscos.



Interfaz gráfica del script `Get-Password.ps1`



Para subir un escalón más en la seguridad en este ejemplo, deberíamos, al igual que hemos hecho para otros ejemplos «limpiar correctamente» la variable que contiene la contraseña en claro, y después forzar la activación del garbage collector.