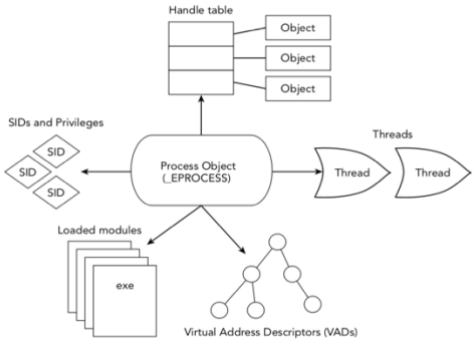

comando | findstr lsass → equivalente *grep*
comando | more → listar poco a poco
Ver perfil (*profile*): `python vol.py -f memory.vmem imageinfo`

Cambios de formato

Algunas operaciones dependen del formato. El comando imagecopy pemite cambiar de **.dmp** o **.sys** a **.raw**

```
$ python vol.py imagecopy -f hiberfil.sys -O hiber.raw
```

Listar procesos

```
$ python vol.py -f lab.mem --profile=WinXPSP3x86 pslist
$ python vol.py -f lab.mem --profile=WinXPSP3x86 pstree
```

Visualizar en forma de árbol (depende de GraphViz)

```
$ python vol.py psscan -f memory.bin [--profile=Win7SP1x64] --output=dot --output-file=processes.dot
$ dot -Tpng processes.dot -o psscan.png
```

Listar Hebras

```
$ python vol.py -f prolaco.vmem thrdscan
```

Enumerar procesos

Enumera procesos de siete manera distintas para permitir hacer correlación de datos: proceso activo enlazado (**pslist**), escaneos sobre el objeto del proceso (**psscan**), hebras (**thrdproc**), tabla PspCid (tabla de manejadores especial en el Kernel) (**pspcid**), manejadores de CSRSS (**csrss**), procesos de sesión (todos los procesos que pertenecen a una misma sesión de usuario) (**session**), hebras de escritorio (**deskthrd**).

- True – Si el proceso ha sido encontrado empleando el método de la columna
- False – Otro caso
- --apply-rules vemos Okey en las columnas cuando el proceso no ha sido encontrado pero satisface al menos una de las excepciones siguientes:
  - Procesos que comienzan antes que csrss.exe (incluyendo System, smss.exe y csrss.exe) y **no están en la tabla de manejadores CSRSS**.
  - Procesos que comienzan antes que smss.exe (incluyendo System y smss.exe) y **no están en el listado de procesos de la sesión** o la **lista de hebras de escritorio**.
  - Procesos que han hecho "exit" no serán encontrados por cualquiera de los métodos empleados salvo por el **escaneo del objeto del proceso** y el **escaneo de hebras** (si un _EPROCESS o _ETHREAD aún se encuentran residentes en la memoria).
- Recordatorio: Los procesos que han hecho un "exit" real deben tener 0 hebras y una tabla de manejadores no válida.

```
$ python vol.py -f prolaco.vmem psxview --apply-rules
```

Consultar claves de registro

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 printkey -K "Microsoft\Windows\CurrentVersion\Run"
```

Obtener información sobre un usuario a partir de su SID

```
$ python vol.py -f memory.img --profile=Win7SP0x86 printkey -K "Microsoft\Windows
NT\CurrentVersion\ProfileList\S-1-5-21-4010035002-774237572-2085959976-1000"
```

Listar DLLs en Memoria

LoadCount debe interpretarse como un entero (short integer). 0xffff representa -1, que significa que la DLL se cargó porque así se especificó en la IAT. Otros valores (p.ej. 0x3, 0x27, 0x4 y 0x1) significan que las DLLs fueron cargadas empleando LoadLibrary. Para procesos de 32bits en máquina de 64bits usar **ldrmodules**.

- Base: Es la dirección base donde la DLL está cargada en el proceso. Es esa dirección la que se usa en el **dlldump**.

```
$ python vol.py -f mem.dmp --profile=Win7SP0x86 dlllist -p 3108
```

Detectar Librerías ocultas (no enlazadas) – Inconsistencias VAD y PEB

También se usa para listar las DLLs para proceso WOW64 ***(Proceso de 32bits corriendo en entorno de 64bits)***

Coteja la información de los ficheros mapeados en memoria en el VAD y la información en el PEB, permitiendo encontrar discrepancias. Enumera todos los nodos VAD que contienen imágenes ejecutables mapeadas. La opción -v (--verbose) imprime el path completo. Técnica: ***VAD cross-referencing***.
Tres formas en las que las DLLs quedan listadas en una estructura PEB:

- inLoadOrder (**InLoad**)
- inInitOrder (**InInit**)
- inMemmoryOrder (**InMem**)

Si una dll muestra valor Falso o 0 para las tres columnas (no se encuentra en las listas del PEB), y sin embargo tiene path asociado (el del VAD), podría ser indicativo de una inyección de código empleando por ejemplo VirtualAlloc(Ex) y WriteProcess Memory.

```
$ python vol.py -f mem.dmp [--profile=Win7SP1x64] ldrmodules -p 616
```

Para checkear por **Process Hollowing**, comparar la carga de proceso legítimo (p.ej. 121) con las cargas de ilegítimo(s) (p.ej. 212), viendo si se cargó la imagen base (ejecutable) y la dirección (en columna base, p.ej. 0x01000000) -p 121,212, o cualquier otra diferencia relevante en base al análisis.

Obtener información sobre secciones VAD

Emplear ***vadwalk*** para ver la información en forma de tabla. La opción -p es para indicar un proceso específico.

```
$ python vol.py -f mem.raw [--profile=Win7SP1x64] vadinfo -p 296
$ python vol.py -f mem.raw [--profile=Win7SP1x64] vadwalk -p 296
```

Visualizar en forma de árbol la estructura VAD

La opción –output=**dot** --output-file=**graph.dot** permite guardar el archivo .dot para abrirlo.

- Rojo: Heaps, Gris: DLLs, Verde: Stacks, Amarillo: Ficheros Mapeados
- Tipos de estructuras: VadS (_MMVAD_SHORT), Vad (_MMVAD), Vadl (_MMVAD_LONG)

```
$ python vol.py -f mem.raw [--profile=Win7SP1x64] --output=dot --output-file=mivad.dot vadtree -p 296
$ dot -Tpng mivad.dot -o mivad.png
```

Volcar contenido de zona de memoria

Extraer toda las páginas de memoria de un proceso en un fichero individual. "/dump" es un directorio que creamos antes del ejemplo

```
$ python vol.py -f stuxnet.vmem --profile=WinXPSP3x86 memdump -p 1928 -D dump/
```

Volcar contenido de VADs

Las páginas para cada nodo VAD se guardan en ficheros separados. "vads" es un directorio que creamos antes del ejemplo.

```
$ python vol.py -f stuxnet.vmem --profile=WinXPSP3x86 vaddump -D vads
```

Detectar Hooks

```
$ python vol.py -f stuxnet.vmem --profile=WinXPSP3x86 apihooks
```

Mediante messagehooks se puede saber el tipo de mensaje filtrado (p.ej. WH_KEYBOARD). Si entre los flags vemos "HF_GLOBAL" el Hook es global, por lo que son resultado de usar **SetWindowsHookEx**. El valor "Procedure" nos da el offset que debemos sumar sobre el offset de la DLL para desensamblar el código con volshell (ver adelante): **1.** Identificar la DD y su offset (dlllist), **2.** En volshell seleccionar el contexto del proceso afectado (cc(pid=-..)), **3.**dis(offset total).

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 messagehooks --output=block
```

Tanto con la inyección DLL como con los hooks globales, el path completo en disco de la DLL puede verse en la tabla **atom**:

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 atoms
```

Inline Hooking afectando a funciones de la **ssdt**, incluir la función --verbose para que indique información sobre el tipo de hook:

```
$ python vol.py -f skynet.bin --profile=WinXPSP3x86 ssdt --verbose
```

Detectar Inyección de Código

Los resultados se pueden guardar con --dump-dir, siendo en el ejemplo output_dir el directorio donde se guardarán los ficheros separados.

```
$ python vol.py -f memory.mem --profile=WinXPSP3x86 malfind
$ python vol.py -f memory.mem --profile=WinXPSP3x86 malfind -p 1648 --dump-dir ./output_dir
```

Extraer Ejecutables .EXE de memoria

Formas de extracción (incluir la opción **-p** para especificar un PID de proceso, mirar **impcan** para la posible reconstrucción de la **IAT**):

Extraer todos los ejecutables en la lista de procesos activos

```
$ python vol.py -f mem.dmp --profile=Win7SP1x64 procdump --dump-dir=OUTDIR/
```

Usando el desplazamiento físico (offset, -o) de su estructura _EPROCESS que podemos obtener con **psscan** o **psxview**. **--unsafe** evita check PE.

```
$ python vol.py -f mem.dmp --profile=Win7SP1x64 procdump --offset=0x000000003e1e6b30 --dump-dir=OUTDIR/
```

Obtener información perdida de la IAT

```
$ python vol.py -f mem.dmp --profile=Win7SP1x64 impscan -p 1180
```

- Para generar fichero IDC que usar en IDA Pro:

```
$ python vol.py -f mem.dmp --profile=Win7SP1x64 impscan -p 1180 --output=idc --output-file=imps.idc
```

Extraer DLLs de memoria

Empleando expresiones regulares (--regex) sólo para DLLs en la lista PEB (no vale para DLLs ocultas o inyectadas). *Ejemplo: Extraer todas las DLLs del proceso con PID 1408 que tengan la cadena "crypt" en su nombre o path:*

```
$ python vol.py -f mem.dmp --profile=Win7SP1x64 dlldump -p 1408 --regex=crypt --ignore-case --dump-dir=OUTDIR/
```

Empleando la dirección base donde la cabecera DOS existe.

```
$ python vol.py -f mem.dmp --profile=Win7SP1x64 dlldump -p 1408 --base=0x000007fef7310000 --dump-dir= OUTDIR/ --memory
```

Buscar procesos que contienen *Strings* específicas

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 strings -s ./fichero_con_strings.txt
```

Mostrar en el historial de comandos (cmd) y consola

Busca por buffers COMMAND_HISTORY.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 cmdscan
```

Busca por buffers CONSOLE_INFORMATION. Por ejemplo, podremos ver commandos ejecutados con ***powershell***.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 consoles
```

Ver manejadores para un proceso

Por ejemplo, la opción **-t file** indica que queremos que muestre sólo manejadores de ficheros, y **-t Mutant** para mutex

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 handles -p 1648
```

Buscar un patrón empleando Yara

Por ejemplo, buscar la cadena (**-Y**) "Thunderbird" en el volcado para el proceso 1648 (**--yara-file=ruta** para fichero de reglas yara específico):

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 yarascan -Y "Thunderbird" -p 1648
```

Correspondencia entre dirección de memoria virtual, dirección física y dirección de volcado

La primera columna representa la dirección virtual, la segunda la física y la última la dirección en el volcado realizado con memdump del proceso.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 memmap -p 1648
```

Extracción de Ficheros de Memoria

Se realizan búsquedas en la estructura de VADs para identificar el path del fichero en memoria. **Output** es un directorio creado para volcar la salida. summary.txt es el nombre del fichero donde guardar el resumen de la operación. Con la opción -r se pueden aplicar expresiones regulares (--regex). En el ejemplo se guardarán los ficheros de eventos (.evt). También puede guardarse el **summary** como **.json**.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 dumpfiles -r evt$ -D output/ -S summary.txt
```

Shell de Volatility (VolShell)

Abrir Shell de contexto para hacer consultas detalladas sobre las estructuras y contexto de los procesos. Se abrirá una Shell de la que salimos con **quit()** – o no...

```
$ python vol.py -f mem.bin --profile=Win7SP1x64 volshell
```

hh()	Ayuda (muestra todos los comandos).	`hh()`
ps()	Listar procesos activos en una tabla.	`ps()`
dt(object, address=None, space=None, recursive=False, depth=0)	Describe un objeto o muestra información del tipo.	`dt("_EPROCESS", )`
dd(address, length=128, space=None)	Imprime como DWORDS	`dd("", )`
cc(offset=None, pid=None, name=None, physical=False)	Establecer un contexto específico.	`cc()`
dis(address, length=128, space=None, mode=None)	Desensamblado de instrucciones en memoria.	`dis()`
db(address, length=128, width=16, space=None)	Imprime bytes como canonical hexdump.	`db`

Chuleta para ver DKOM con Volatility

1. Hacer ***psscan*** → Esto nos permite ver un listado de procesos.
2. Hacer ***psxview*** → Para ver qué procesos tienen valores a **False** para los listados.
    a. Observad si hay procesos que tienen pslist (listado de procesos) a **False** sin que tengan fecha de "exit".
        i. **Anotarlo/s como sospechosos.**
        ii. **Obtener el offset para el EPROCESS del proceso sospechoso. Se hace así:**
        iii. **VALOR1 es el offset indicado en la primera columna para el proceso sospechoso.**
        iv. **Empleando procdump hacemos que se calcule el offset del EPROCESS:**

```
python vol.py prolaco.vmem --profile=WinXPSP2x86 procdump -o VALOR1 -D dump/
```

        v. **VALOR2 es el offset indicado en la primera columna.**
3. Emplear **volshell**
4. Con **ps()** se hace listado de procesos
    a. Nos permite comprobar las direcciones offset para los procesos (última columna)
    b. Observad si está listado o no el proceso sospechoso.
        i. **Si no lo está, observad que VALOR2 tampoco está en la columan de Offset.**
    c. Como curiosidad, observad que el FLINK del último proceso (al estar listados por orden) debería apuntar a System (el primer proceso). *Esto lo hemos probado antes de meternos en este lío de comprobar el DKOM del proceso oculto.*
5. dt("_EPROCESS", **VALOR2**) – Con esto vemos la estructura _EPROCESS del proceso oculto (o sospechoso).
    a. Si no sale una estructura _EPROCESS, entonces algo de lo que hemos hecho antes no está bien...
    b. Observad el campo ActiveProcessLinks. Ese campo contiene Flink y Blink.
    c. VALOR3 es la tercera columna, que es la dirección donde encontramos la estructura ActiveProcessLinks.
    d. VALOR4 es el desplazamiento de esa estructura (primera columna).
6. dt("_LIST_ENTRY", **VALOR3**)
    a. Vemos Flink y Blink. Nos interesan ambos. Primero miraremos Flink.
    b. **VALOR5 es el número que aparece junto a Flink. Ese valor lo convertimos a dirección con dd.**
7. dd(**VALOR5**)
    a. La primera columna son direcciones.
    b. **VALOR6 --** El primer valor es la dirección al Flink del siguiente EPROCESS. Si no es nulo vamos al paso 8
8. dt("_EPROCESS", **VALOR6-VALOR4**)
    a. Ahora estamos en la siguiente estructura a la que apunta el EPROCESS del proceso oculto.
    b. **VALOR7 = VALOR6-VALOR4, que lo podemos observar arriba del todo de la estructura donde estamos.**
    c. Ese VALOR7 es el offset de uno de los procesos que vemos si hacemos ps()
9. ps()
    a. Si VALOR7 == VALOR2 entonces el enlace apunta al propio proceso oculto, y lo siguiente es comprobar Blink.
    b. Anotar el nombre del proceso cuyo offset es VALOR7 – Llamémoslo **ProcesoPosterior.exe**
    c. ** SI NO HUBIESE OCULTACIÓN, EL PUNTERO BLINK DEL EPROCESS DE DICHO PROCESO SERÍA NUESTRO PROCESO SOSPECHOSO **
10. POR COMPROBAR:
    a. Repetimos la operación para el proceso sospechoso hasta el punto 6, en el cual tomamos el valor de Blink y seguimos a partir de ahí. Eso nos dará el nombre del proceso "anterior" al proceso oculto – Llamémoslo **ProcesoAnterior.exe**
        i. **** SI NO HUBIESE OCULTACIÓN, EL PUNTERO FLINK DE DICHO PROCESO SERÍA NUESTRO PROCESO SOSPECHOSO ****
    b. Ahora, conforme a la teoría:
        i. Repetimos toda la operación para ProcesoPosterior.exe, para ver cuál es el EPROCESS al que apunta su Blink. Tiene que coincidir con el EPROCESS de **ProcesoAnterior.exe porque el malware lo ha cambiado.**
        ii. Repetimos toda la operación para ProcesoAnterior.exe, para ver cuál es el EPROCESS al que apunta su Flink. Tiene que coincidir con el EPROCESS de **ProcesoPosterior.exe porque el malware lo ha cambiado.**
11. **POSIBLES SORPRESAS**:
    a. ProcesoAnterior.exe y ProcesoPosterior.exe no se apuntan el uno al otro.
        i. Aún se puede comprobar que los enlaces han sido cambiados, encontrando que la estructura EPROCESS a la que apunta el ProcesoOculto también es apuntada por otro proceso.

Ejemplo de recorrido volshell pslist

Este ejemplo mostrará algunos datos sobre Flink pero ***sólo de los procesos listados a través de pslist*.** Campos del **_EPROCESS** en negrita

```
>>for proc in win32.tasks.pslist(addrspace()):
        process_space = proc.get_process_address_space()
        pidp = proc.UniqueProcessId
        flink = proc.ActiveProcessLinks.Flink
        print "Flink (value5) for Pid {0}: {1}, value6 first of:".format(pidp, flink)
        value6 = dd(flink)
```

Identificar Drivers – comprobaciones para identificar Rootkits

El comando "modules" permite ver los módulos cargados en el sistema (drivers del kernel). Si el momento de la infección es reciente, podríamos fijarnos por ejemplo en los últimos módulos cargados en el sistema.

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 modules
```

El comando "modscan" busca estructuras LDR_DATA_TABLE_ENTRY. Esto, además de cargar los drivers anteriores también puede mostrar los **drivers que han sido ocultados por los Rootkits**. A diferencia de "modules", el orden de resultados no tiene relación con el orden en el que fueron cargados. Adicionalmente se puede usar el comando "**driverscan**" para una comprobación general, ya que este último busca estructuras _DRIVER_OBJECT.

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 modscan
```

También se pueden analizar los hooks para intentar ver el driver responsable, o bien emplear comandos: driverirp, ssdt, idt, etc.

Identificar Hebras (huérfanas) sobre drivers – otra comprobación para identificar Rootkits

Otra opción si sospechamos de rootkit pero no vemos nada con los módulos anteriores es buscar hebras ocultas. El comando **threads** con la opción -F OrphanThread permite identificar hebras huérfanas. En StartAddress se indica la posición del código y junto a dicha dirección **UNKNOWN**. Es un check que devuelve aquellas entradas para las que **la dirección de comienzo no mapea a drivers cargados / conocidos**. Esa dirección apuntará a una función dentro del fichero PE.

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 threads -F OrphanThread
```

Extraer Drivers

Si se suprime la opción -b del ejemplo se extraerán todos los drivers del Kernel. El valor -b se obtiene de la columna "base" (tercera columna) tras la operación "modules" anterior. El valor -p permite indicar un patrón.

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 moddump -b 0xf8c5200 -D outdir
```

Los drivers extraídos de esta forma sufrirán el mismo problema de la IAT incompleta que los ejecutables, sobre todo si el driver estaba incialmente empaquetado. Se usará **impscan** también en este caso para obtener los datos de la IAT para el análisis en IDA Pro, indicando -b para la dirección base.

```
$ python vol.py -f laqma.vmem --profile=WinXPSP3x86 impscan -b 0xf8c5200 --output-file="iatinfo.txt"
```

***Si el driver tiene dirección de inicio y tamaño a 0 (anti-dumping), probar a buscar el fichero del driver con filescan y aplicar dumpfiles.**

Conexiones de red y sockets

***NOTA: CONEXIONES DE RED OCULTAS – Si en la columna del PID del proceso (tras hacer connscan o netscan) aparece -1 u otro valor, realizar una búsqueda con yarascan empleando como cadena (-Y) la IP buscada. También comprobar con apihooks si existen hooks sobre procesos de red.**

Connections permite ver conexiones TCP **activas** en el momento de la adquisición de memoria.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 connections
```

Connscan permite ver conexiones que han estado activas en algún momento.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 connscan
```

Netscan encuentra puntos finales TCP y UDP, así como puertos a la escucha. Distingue entre IPv4 e IPv6. También muestra la IP y puerto remoto, así como datos sobre el estado de la conexión (TCP). Puede fallar en capturas de versions de windows anteriores a Windows Vista. Para versiones anteriores usar connscan.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 netscan
```

Sockscan busca estructuras _ADDRESS_OBJECT, y puede volcar información residual de puertos a la escucha. Sockets muestra puertos a la escucha activos en el momento de la captura.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 sockscan
```

Identificadores de seguridad y privilegios

Comprobar usuarios logueados, nombres y si existen evidencias de escalado de privilegios.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 getsids
```

Comprobar procesos con privilegios para cargar drivers del kernel (si no se dispone de grep, buscar las entradas Enabled).

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 privs -r driver | grep Enabled
```

Listado de servicios instalados

Se listan por orden de instalación, siendo los últimos los más recientes.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 svcscan
```

Extraer fichero

Plan B por si algunas de las extracciones directas no se pueden realizar… intentar buscar el fichero origen y exportarlo.

```
$ python vol.py -f mem.raw --profile=Win7SP1x64 dumpfiles -Q 0x0000000000777540 -D . --name
```