Windows Memory Acquisition (winpmem)

CREATING AN AFF4 (Open cmd.exe as Administrator)
```
C:\> winpmem_<version>.exe -o output.aff4
```
*INCLUDE PAGE FILE
```
C:\> winpmem_<version>.exe -p c:\pagefile.sys -o output.aff4
```

EXTRACTING TO RAW MEMORY IMAGE FROM AFF4
```
C:\> winpmem<version>.exe output.aff4 --export
PhysicalMemory -o memory.img
```

EXTRACTING TO RAW USING REKALL
```
$ rekal -f win7.aff4 imagecopy --output-image=
"/cases/win7.img
```

Live Windows Memory Analysis

(Open cmd.exe as Administrator)
CREATING LIVE REKALL SESSION VIA MEMORY
```
C:\Program Files\Rekall> Rekal --live
```

CREATING LIVE REKALL SESSION VIA API ANALYSIS
```
C:\Program Files\Rekall> Rekal --live API
```
**LIVE WMI COMMANDS
```
[] Live (API) 16:52:10> wmi "select SID,Disabled from
Win32_UserAccount"
```
**LIVE GLOB SEARCH
```
 [] Live (API) 16:52:10> select * from glob("c:\windows\*.exe")
```

MacOS Memory Live Analysis & Acquisition

MAC OSXPMEM (Run commands with Root privileges)
Extract osxpmem.zip and ensure file/dir permissions are root:wheel

CREATING AN AFF4
```
$ sudo kextload MacPmem.kext
$ sudo ./osxpmem --output test.aff4
$ sudo kextunload MacPmem.kext/
<clean up by removing driver>
```

LIVE OSX MEMORY ANALYSIS
```
$ sudo kextload MacPmem.kext/
$ rekal -f /dev/pmem
<begin interactive session>
$ sudo kextunload MacPmem.kext/
<clean up by removing driver>
```

Registry Analysis Plugins

ENUMERATE AND EXTRACT REGISTRY HIVES

HIVES- Find and list available registry hives
```
$ rekal -f be.aff4 hives
```

REGDUMP- Extracts target hive

--hive_regex	Regex Pattern Matching
- D "<dir>"	Dump directory

```
$ rekal -f be.aff4 regdump --hive_regex="SAM" -D "/cases"
```

PRINTKEY- Output a registry key, subkeys, and values
-K "Registry key path"
```
[1] be.aff4 11:14:35> printkey –K
  "Software\Microsoft\Windows\CurrentVersion\Run"
```

USERASSIST- Find and parse userassist key values

Additional Functionality

ANALYZE_STRUCT Interprets and identifies windows memory structures when given a virtual offset
```
[1] be.aff4 11:15:35> analyze_struct 0x8180e6f0
```

DT Displays Specific Kernel Data Structures
```
[1] be.aff4 11:14:35> dt "_EPROCESS",offset=<virtual offset>
```

PTOV Determine owning process with physical to virtual address translation (decimal offset shown below)
```
$ rekal -f test.img ptov 21732272
```
VMSCAN Allows for the identification of virtual machines
CERTSCAN Dumps RSA private and public keys
dump_dir= Dumps output to a specified directory
MIMIKATZ Extracts and decrypts credentials from lsass

Linux Memory Acquisition

LINUX PMEM (TO CREATE PROFILE)
```
# tar vxzf linux_pmem_1.0RC1.tgz
# cd linux
# make
```

LINPMEM (TO CREATE IMAGE VIA /proc/kcore)
```
# gzip -d linpmem_2.0.1.gz
# chmod 755 linpmem_2.0.1
# ./linpmem_2.0.1 -o linux.aff4
# cd linux
# rekal convert_profile 3.11.0-26-generic.zip  Ubuntu.zip
# rekal --profile=Ubuntu.zip -f ../linux.aff4
```

SANS DFIR

DIGITAL FORENSICS & INCIDENT RESPONSE

Rekall Memory Forensic Framework

Cheat Sheet v3.1

POCKET REFERENCE GUIDE

Purpose

The Rekall Memory Forensic Framework is a collection of memory acquisition and analysis tools implemented in Python under the GNU General Public License. This cheatsheet provides a quick reference for memory analysis operations in Rekall, covering acquisition, live memory analysis and parsing plugins used in the 6-Step Investigative Process. For more information on this tool, visit **rekall-forensic.com**.

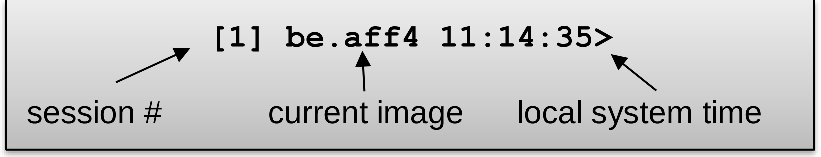
Rekall Memory Forensic Framework

Memory analysis is one of the most powerful investigation techniques available to forensic examiners. Rekall auto-detects the target system's profile, using a repository of more than 100 kernel versions available either online or stored locally.

When launching Rekall, you can run single commands or drop into an interactive session to take advantage of caching, preventing the need to obtain the same data with subsequent plugin runs. This cheatsheet shows command line examples using both techniques for Rekall version 1.5.3+

Getting Started with Rekall

Single Command Example
```
$ rekal -f be.aff4 pslist
```
Starting an Interactive Session
```
$ rekal -f be.aff4
```
Starting an Interactive Session (sends output to specified tool)
```
$ rekal -f be.aff4 --pager=gedit
```


```
[1] be.aff4 11:14:35>
```

Memory Analysis Basics

GETTING HELP

```
[1] be.aff4 11:14:35> plugins.<tab>
```
(lists plugins applicable for use for this image)

```
[1] be.aff4 11:14:35> pslist?
```
(lists options available for specific plugin)

COMMON OPTIONS IN INTERACTIVE SESSION
describe(<plugin>) Print the output fields of a plugin
verbosity=# Specify amount of output (1-10, default=1)
proc_regex=”process name” Regex to select process by name
<pid> Positional Argument: Filter by process PID
dump_dir=”path to directory” Path to output directory
output=”path to output dir\file” Required if outputting to file
quit Exit interactive session

IMAGE DETAILS (list OS version, physical layout, uptime)

```
[1] be.aff4 11:14:35> imageinfo
```

ARTIFACT COLLECTOR (Carving for defined artifacts)

```
[] Live (API) 16:52:10> artifact_list
[] Live (API) 16:52:10> artifact_collector
[“WMIProcessList”,”WMILoggedOnUsers”,”WMIDrivers”],output_path=”c:\\cases\\exercises”
```

Step 1. Enumerating Processes

PSLIST Enumerate Processes

```
[1] be.aff4 11:14:35> pslist
```
Customize pslist output with efilters

```
[1] be.aff4 11:14:35> describe(pslist)
[1] be.aff4 11:14:35> select EPROCESS,ppid,process_create_time from pslist() order by process_create_time
```

PSTREE (WITH VERBOSITY) – List Processes with path and command line

```
[1] be.aff4 11:14:35> describe(pstree)
[1] be.aff4 11:14:35> select _EPROCESS,ppid,cmd,path from pstree()
```

PEINFO Display detailed process & PE info

```
[1] be.aff4 11:14:35> procinfo <PID>
```

DESKTOPS Enumerate desktops and desktop threads

```
[1] be.aff4 11:14:35> desktops verbosity=<#>
```
SESSIONS Enumerate sessions and associated processes

```
[1] be.aff4 11:14:35> sessions
```

Step 2. Analyze Process DLLs and Handles

DLLLIST List of loaded dlls by process.
Filter on specific process(es) by including the process identifier <PID> as a positional argument

```
[1] image.img 11:14:35> dlllist [1580,204]
```

THREADS Enumerates process threads

```
[1] be.aff4 11:14:35> threads proc_regex= ”chrome”
```

HANDLES List of open handles for each process Include pid or array of pids separated by commas
object_types=”TYPE” – Limit to handles of a certain type {Process, Thread, Key, Event, File, Mutant, Token, Port}

```
[1] image.img 11:14:35> handles 868, object_types=”Key”
```

FILESCAN Scan memory for _FILE_OBJECT handles

```
[1] image.img 11:14:35> filescan output=”filescan.txt”
```

DUMPFILES Extract memory mapped files

```
[1] image.img 11:14:35> dumpfiles 1484,dump_dir="."
```

Step 3. Review Network Artifacts

NETSCAN -Scan for connections and sockets in Vista-Win7

```
[1] memory.aff4 11:14:35> netscan
```

NETSTAT -ID active TCP connections in Vista-Win7

```
[1] memory.aff4 11:14:35> netstat
```

DNS_CACHE- Dumps dns resolver cache

```
[1] memory.aff4 11:14:35> dns_cache
```

Step 4. Look for Evidence of Code Injection

MALFIND Find injected code and dump sections by VAD analysis
<pid> Positional Argument: Show information only for specific PIDs
phys_eprocess= Provide physical offset of process to scan
eprocess= Provide virtual offset for process to scan
dump_dir= Directory to save memory sections

```
[1] be.aff4 11:14:35> malfind eprocess=0x853cf460,
dump_dir=”/cases”
```

LDRMODULES Detect unlinked DLLs
verbosity= Verbose: show full paths from three DLL lists

```
[1] be.aff4 11:14:35> ldrmodules 1936
```

MESSAGEHOOKS Enumerates desktop and thread windows message hooks to aid in spotting SetWindowsHookEx code injection

Step 5. Check for Signs of a Rootkit

PSXVIEW Find hidden processes using cross-view
MODSCAN Scan memory for loaded, unloaded, and unlinked drivers
SERVICES Enumerates services from in-memory registry hive
SVCSCAN Scans for _SERVICE_RECORD objects

HOOKS_INLINE Detects API hooks
eprocess= Filters by virtual address EProcess
phys_eprocess= Filters by physical address of EProcess

HOOKS_EAT Detects Export Address Table hooks

```
[1] be.aff4 11:14:35> hooks_eat 6764
```

HOOKS_IAT Detects Import Address Table hooks
SSDT Hooks in System Service Descriptor Table

DRIVERIRP Identify I/O Request Packet (IRP) hooks
regex=”drivername”- Filter on REGEX name pattern
OBJECT_TREE Tracks named objects

```
[1] be.aff4 11:15:35> object_tree type_regex=”Driver”
```
CALLBACKS Enumerates registered system event callbacks

Step 6. Dump Suspicious Processes and Drivers

DUMP Hexdump data starting a specified offset

```
[1] be.aff4 11:14:35> dump <virtual offset>
```

COMMON OPTIONS FOR EXTRACTION
<pid> Positional Argument: Filter by process PID
proc_regex=”process name” Regex to select process by name
offset= Specify process by physical memory offset
dump_dir= Directory to save extracted files

DLLDUMP Extract DLLs from specific processes

```
[1] be.aff4 11:14:35> dlldump 1004,dump_dir=”.”
```

MODDUMP Extract kernel drivers

```
[1] be.aff4 11:14:35> moddump regex=”tcipip”,
dump_dir=”/tmp”
```

PROCDUMP Dump process to executable sample

```
 [1] be.aff4 11:14:35> procdump proc_regex=”csrss”,
dump_dir=”/tmp”
```

MEMDUMP Dump every memory section into a single file

```
[1] be.aff4 11:15:35> memdump 1004,dump_dir=”./output”
```