CHAPTER 9

Memory Analysis

Chapter Overview

01

Baselines & Memory Sources

What good looks like, and where memory data comes from

02

Volatility Framework

Profile selection, plugin architecture, Vol2 vs Vol3

03

Examining Processes

pslist, pstree, dlllist, malfind and detection techniques

04

Network, Services & Anomalies

Active connections, services, and detecting injected code

Baselines & Memory Sources

Know normal before hunting abnormal

Baselines & Memory Data Sources

Importance of Baselines

Memory analysis requires knowing what is NORMAL

Collect baseline memory images from clean, known-good systems

Compare running processes, loaded DLLs, and network sockets to baseline

Environmental baselines beat generic lists — every org is different

Sources of Memory Data

Full memory dump: WinPmem, DumpIt, Magnet RAM Capture

Hibernation file: hiberfil.sys (compressed RAM image on disk)

Page file: pagefile.sys (swapped-out memory pages)

Crash dump: MEMORY.DMP — may be left by attackers

VM snapshot: includes memory state of running VM

Volatility Framework

The open-source memory forensics standard

Volatility: Setup and Profile Selection

Volatility: Python-based memory analysis framework — open source

Volatility 2: vol.py — mature, wide plugin support

Volatility 3: redesigned architecture, no profile required

Profile selection (Vol2): match OS version and architecture to image

imageinfo plugin: identify correct profile for unknown image

Command syntax: vol.py -f memory.raw --profile=Win10x64 [plugin]

Plugin output: tabular data suitable for grep, sort, and scripted analysis

Volatility 2 vs. Volatility 3

Volatility 2 (vol.py)

Requires explicit profile (--profile=Win10x64)

imageinfo: identifies correct profile for unknown image

Mature ecosystem: hundreds of third-party plugins

Python 2 based — no longer maintained but widely used

Syntax: python vol.py -f mem.raw --profile=Win7SP1x64 pslist

Volatility 3 (vol3)

No profile required — auto-detects OS and builds symbols

Python 3 based — actively maintained and faster

ISF (Intermediate Symbol Files) auto-downloaded on demand

Syntax: python vol3.py -f mem.raw windows.pslist.PsList

Plugin namespaces: windows.*, linux.*, mac.*

Key Volatility Plugin Reference

windows.pslist: process list from EPROCESS doubly-linked list

windows.psscan: pool tag scanning — finds hidden/unlinked processes

windows.pstree: parent-child process hierarchy

windows.dlllist: DLLs loaded by each process

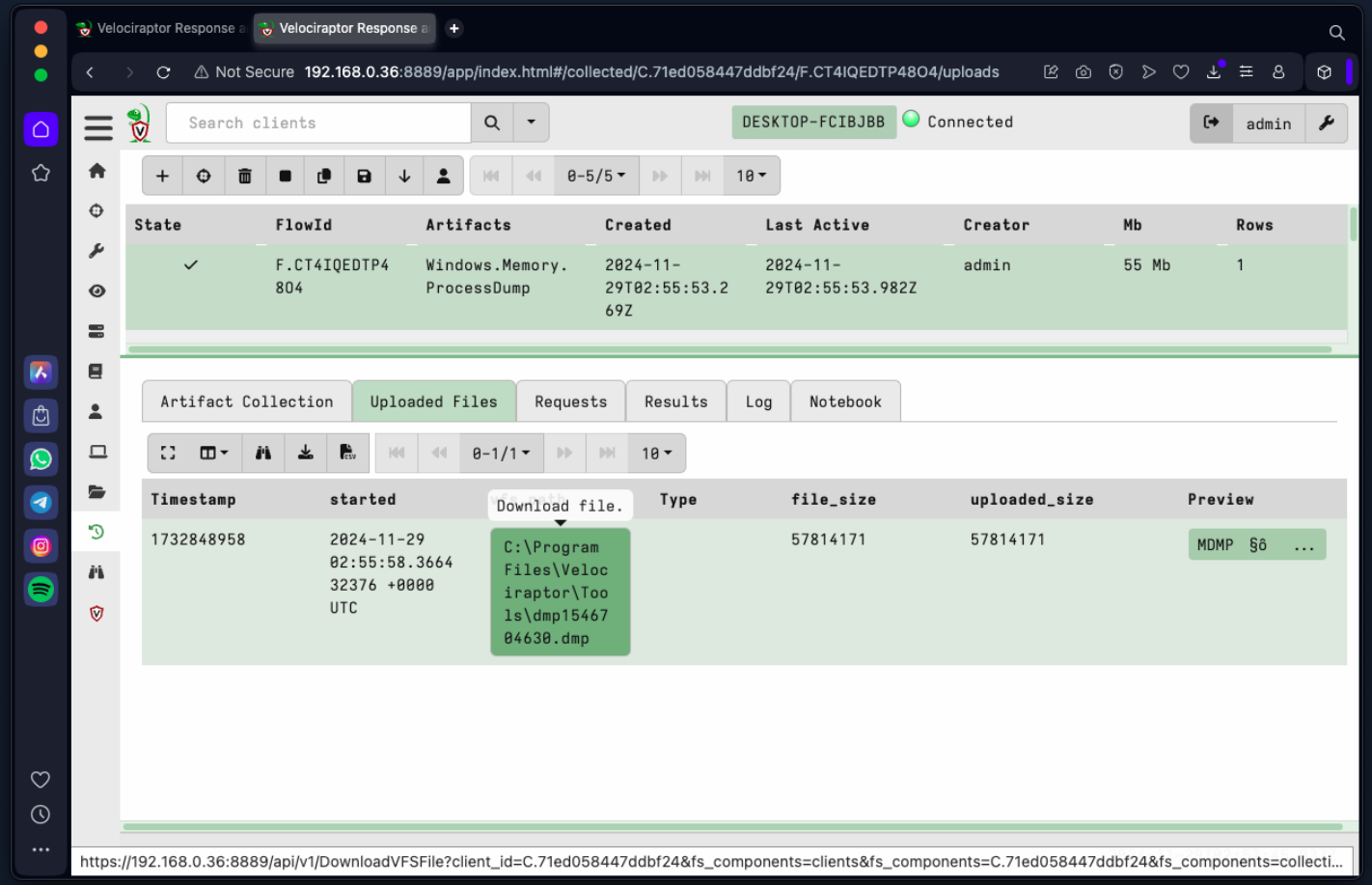
windows.netscan: active and recently-closed network connections

windows.malfind: executable, non-image-backed, writable memory regions

windows.cmdline: command-line arguments for each process

windows.hashdump: extract SAM database hashes (offline cracking)

windows.filescan: cached file objects in memory



Process memory capture with Velociraptor

Examining Processes

Finding malicious processes in memory

Process Analysis Plugins

Listing & Tree View

pslist: standard process list from EPROCESS linked list

pstree: parent-child relationships — spot unusual parents

psscan: find processes using pool tag scanning (finds hidden)

psxview: cross-view comparison — differences = hidden processes

dlllist -p PID: all DLLs loaded by a process

Advanced Detection

malfind: find injected code (executable, not image-backed, RWX)

handles -p PID: open handles — reveals attacker's files/keys

cmdline: command-line arguments — reveals attacker tools run

hollowshell: detect process hollowing (module vs. VAD mismatch)

ldrmodules: detect unlinked/hidden DLLs

Malfind Deep Dive — Detecting Injected Code

malfind identifies: MZ header present + VAD tag = mapped image, but no backing file on disk

Suspicious flags: PAGE_EXECUTE_READWRITE (RWX) permissions on a non-image region

Output: VAD address, page permissions, first 64 bytes as hex+disasm

False positives: .NET JIT-compiled code, some graphics drivers — investigate, don't blindly trust

Extracting injected code: -D /output/dir — dumps each suspicious region to a file

Analyze dumps with: strings, file command, and YARA rules for shellcode patterns

Common injected patterns: shellcode starts with NOP sled + call instruction

Correlate: injected code in process X + network connection from process X = C2 payload

DLL Analysis — dlllist, ldrmodules, dllscan

dlllist -p PID: DLLs in process's PEB (Process Environment Block) loader list

ldrmodules: cross-reference three DLL lists — discrepancies = hidden/reflectively loaded DLL

Hidden DLL: present in VAD but not in any PEB loader list — reflective loading

dlldump -p PID -D /output: extract all DLLs from a process to disk for analysis

Suspicious DLL paths: anything outside C:\Windows, C:\Program Files, or C:\Windows\SysWOW64

DLL search order hijacking: legitimate DLL overridden in application directory

Verify: compare DLL hashes against known-good hashes from clean system images

VirusTotal DLL lookup: hash of suspicious DLL → check reputation before deep analysis

Network, Services & Anomalies

Connecting processes to network activity

Network Connections & Windows Services

netscan: active and recently-closed TCP/UDP connections with owning PID

Correlate PID from netscan → pslist: which process owns each connection?

svcscan: Windows services extracted from memory — find services not in registry

getservicesids: map service names to SIDs

Anomalies to investigate: svchost.exe without ServiceDLL, rundll32 with network conn.

notepad.exe, calc.exe, or explorer.exe with outbound connections

Processes injected into: lsass.exe, explorer.exe, svchost.exe are common targets

Registry Artifacts in Memory

hivelist: find all loaded registry hives and their virtual addresses in memory

printkey: dump a specific registry key from memory

printkey -K 'SOFTWARE\Microsoft\Windows\CurrentVersion\Run': persistence keys

printkey -K 'SAM\Domains\Account\Users': local account information

userassist: decode UserAssist keys (ROT-13 encoded GUI execution history)

shimcache / amcache: application compatibility cache — evidence of execution

Registry analysis advantage: in-memory hives include unsaved changes not yet on disk

dumpregistry -D output/: export all hives for offline analysis with RegRipper

Memory Timeline & Correlation

Timeline: combine process creation times, network connections, and file access into one chronology

mftscan + pstree: correlate files modified during attack with processes active at the time

Timestomp detection: $STANDARD_INFORMATION timestamp earlier than $FILE_NAME timestamp = stomped

Correlate event IDs: memory artifact timestamps + Event 4688 (process create) = ground truth

plaso/log2timeline: ingest memory artifacts, event logs, and filesystem timestamps together

Volatility timeliner plugin: generates unified timeline from memory artifacts alone

Visual tools: Timeline Explorer (Eric Zimmerman), Plaso + Kibana, Timesketch

Goal: reconstruct minute-by-minute attacker activity to answer 'what happened and when'

Memory Analysis Workflow — End to End

Step 1 — Identify image: vol3 -f mem.raw windows.info to confirm OS and architecture

Step 2 — Process list: pslist + psscan + pstree — any process only visible in one list?

Step 3 — Network: netscan — note all ESTABLISHED/LISTEN connections with owning PID

Step 4 — Pivot to process: correlate suspicious PID across pslist, cmdline, dlllist, malfind

Step 5 — Code analysis: malfind — dump suspicious regions, scan with YARA or AV

Step 6 — Credentials: hashdump, lsadump — extract any credentials present in memory

Step 7 — Registry: hivelist + printkey — check persistence and recently written keys

Step 8 — Timeline: timeliner or log2timeline — place all artifacts in chronological order

Conclusion

Memory contains active secrets: credentials, encryption keys, injected code

Baselines are essential — memory analysis without context is guesswork

Volatility's cross-view plugins (psxview, malfind) find what attackers hide

Correlate processes with network connections to identify C2 communications

Modern attackers avoid disk — memory is often the only place malware lives

A structured workflow (list → network → pivot → extract → timeline) produces reliable results

Knowledge Check