Essentials of Cybersecurity & Cyber Warfare practical

Practical A — Windows Memory Forensics: Finding a Stealthy Malware Process (Volatility 3)

Objective: Acquire a memory image from a Windows 10/11 VM and use Volatility 3 to identify a hidden/malicious process, dump its executable, and extract suspicious network connections and loaded DLLs.

Prerequisites: - Windows 10/11 lab VM (snapshot before the exercise) with an intentionally-infected sample in a controlled environment (instructor-provided) or an instructor-supplied memory image. - A separate analysis workstation with Linux/Windows that has Volatility 3 installed (Python 3.8+), and rekall or strings available. - Administrator access to the target VM for acquisition.

Tools & files: winpmem or FTK Imager for memory acquisition, volatility3 (pip install volatility3), python3, scalpel/strings, network capture (optional).

tasks (step-by-step):

1. Prepare the target VM snapshot and ensure no external network access. Note the time. 2. Acquire physical memory using winpmem (recommendation) to \lab\dumps\vm_mem.raw: - winpmem.exe -o vm_mem.raw (run as Administrator) — or use FTK Imager's physical memory capture wizard.

3. Transfer vm_mem.raw to the analysis workstation (read-only copy). Compute sha256 checksum for integrity: sha256sum vm_mem.raw. 4. Identify the Windows profile and use Volatility

4 to list processes: - vol.py -f vm_mem.raw windows.pslist.PsList (or volatility3 -f vm_mem.raw windows.pslist.PsList).

5. Look for suspicious process names, processes without parent, or command lines that are odd. Use psscan to find hidden processes: - vol.py -f vm_mem.raw windows.psscan.PsScan

6. Once you identify a PID of interest (e.g., 0x3f8), dump the process memory/executable: - vol.py -f vm_mem.raw windows.dumpfiles.DumpFiles --pid 0x3f8 --dump-dir=./dumps

7. Extract network connectors and sockets from memory: - vol.py -f vm_mem.raw windows.netscan.NetScan — note remote IPs/ports.

8. Extract loaded DLLs for that PID: - vol.py -f vm_mem.raw windows.dlllist.DllList --pid 0x3f8

9. Run strings on the dumped binary to find IoCs (domains, URLs): strings dumps/<exe> | egrep -i "http|\.com|\.exe|base64".

10. Create a short report: PID, process name, parent PID, suspicious indicators, dumped filename, sha256 of dumped exe, remote IPs/ports.

Solution / Expected outputs (worked example): - pslist shows normal processes; psscan reveals a hidden PID 1016 (0x3f8) not shown in pslist. - netscan lists remote connection 198.51.100.24:443 associated with PID 1016. - dlllist for PID 1016 shows unusual DLL xyzhook.dll loaded from C:\Windows\Temp. - Dumped file malicious_1016.exe saved to ./dumps; sha256: e3b0c44298fc1c149afbf4c8996fb924... (example truncated). - strings reveals a hardcoded domain badexample[.]com/api/post and a base64 block — possible config.

Practical B — Linux Kernel Rootkit Detection & Analysis (Advanced Rootkit Artifacts)

Objective: Detect a simple LKM (Loadable Kernel Module) rootkit or userland hook that hides processes and network sockets. Identify the malicious module, list hidden PIDs, and restore visibility (non-destructive proving).

Prerequisites: - Isolated Linux lab VM (prefer Debian/Ubuntu). Instructor must provide either a VM with a benign simulated rootkit or a pcap and artifacts allowing analysis. - Student has sudo access on the VM (or analyzes a disk image snapshot offline).

Tools & files: lsmod, dmesg, chkrootkit, rkhunter, strace, readelf, strings, modinfo, insmod, rmmod (only on instructor permission), auditd logs.

tasks (step-by-step):

1. Baseline: run `uname -a`, `lsmod`, `ps aux | wc -l`, `ss -tulpn` and capture outputs into baseline.txt.

2. Run chkrootkit and rkhunter to see flagged anomalies (may produce false positives). 3. Use dmesg and `journalctl -k` to look for unusual module load entries: `dmesg | egrep -i "module|lkm|insmod|rmmod|rootkit"`.

4. Compare /proc/modules and lsmod to see any discrepancies.

5. If processes are hidden, iterate over /proc and search numeric directories missing from ps output: - `for p in /proc/[0-9]*; do echo $(basename $p) $(cat $p/comm 2>/dev/null); done | sort -n > proc_list.txt` - Compare with `ps -eo pid,comm | sort -n`.

6. Inspect kernel module files listed in /lib/modules/$(uname -r)/ and recently modified files: - `find /lib/modules -type f -mtime -7 -ls` (find modules modified in last week). 7. Identify suspicious module name (e.g., kshellx) and view its modinfo: `sudo modinfo kshellx`.

8. Without unloading the module in production, create a copy for offline static analysis: `sudo cp /lib/modules/$(uname -r)/extra/kshellx.ko /tmp/` and run `readelf -a /tmp/kshellx.ko` and `strings /tmp/kshellx.ko | egrep -i "hide|hook|proc|netfilter|ss"`.

9. To *prove* the effect and then restore, on an instructor-approved ephemeral lab only: unload the module (`sudo rmmod kshellx`) and show ps and ss now list previously-hidden items. If rmmod fails, show how to use `cat /sys/module/kshellx/holders` to diagnose dependencies.

Solution / Expected outputs (example): - chkrootkit flagged suspicious kernel module. - proc_list.txt shows PID 4321 present in /proc but missing from ps output. - dmesg contains insmod: module kshellx.ko by root at 12:01:22. - strings kshellx.ko contains hide_process, hook_sys_getdents, and netfilter_register — indicating malicious hooks. - After rmmod kshellx (ephemeral lab), ps now reports PID 4321 and ss -tulpn lists previously hidden socket 0.0.0.0:6667.

Practical C — Advanced Network Forensics: Reconstructing HTTP Transactions & Extracting Files from a Large PCAP

Objective: From a provided large pcap (instructor-supplied), reconstruct full HTTP sessions, extract transferred files (images, executables, archives), identify exfiltration via HTTP POST, and produce IOCs.

Prerequisites: - Analysis workstation with tshark/Wireshark, tcpflow, brotli/zlib installed, and enough disk space. - Instructor-provided pcap lab_http_big.pcap (contains mixed normal traffic and malicious exfiltration)

Tools & files: tshark, tcpflow, foremost/scalpel, strings, python3 (for simple parsing), binwalk (optional).

tasks (step-by-step):

1. Get pcap summary: tshark -r lab_http_big.pcap -q -z conv,ip to find top talkers and initial time ranges.

2. Filter HTTP traffic and extract HTTP objects using tshark's export feature: - tshark -r lab_http_big.pcap -Y http -T fields -e http.request.method -e http.request.full_uri -e ip.src -e ip.dst -E separator=, > http_requests.csv 3. Identify large POST requests (possible exfil): - tshark -r lab_http_big.pcap -Y 'http.request.method == "POST"' -T fields -e frame.number -e frame.len -e ip.src -e ip.dst -e http.content_length then sort by content_length.

4. Use tcpflow to reconstruct TCP streams to disk: tcpflow -r lab_http_big.pcap -o ./flows/.

5. For each flow file corresponding to HTTP (port 80/443 if decrypted), run file and strings to detect embedded files: - file flows/* | egrep -i "(jpeg|png|zip|gzip|exe|pdf)" and copy matches to extracted/.

6. If traffic uses Content-Encoding: gzip or brotli, decompress payloads and run foremost or scalpel on the decompressed streams.

7. Correlate HTTP headers to identify user-agents, cookies, and suspicious endpoints. Note endpoints receiving large POST bodies.

8. Calculate total bytes exfiltrated per destination IP and create a top-10 list: - tshark -r lab_http_big.pcap -Y 'http.request.method == "POST"' -T fields -e ip.dst -e http.content_length | awk -F"\t" '{sum[$1]+= $2} END{for(i in sum) print sum[i], i}' | sort -nr | head

Solution / Expected outputs (example): - http_requests.csv shows multiple POSTs to https://upload.badexample.com/receive from host 10.0.2.15 between 2025-11-05 10:12–10:18. - tcpflow extraction yields flows/10.0.2.15-192.0.2.5-00000012 containing a gzipped payload. After decompress, foremost recovers confidential.zip (md5: example d41d8cd98f00b204e9800998ecf8427e). - Top exfil target 198.51.100.200 with 2.4 MB transferred.

Practical D — Cross-Host Timeline & Correlation Using Plaso (log2timeline) and Timesketch

Objective: Build a cross-host forensic timeline using Plaso log2timeline.py from multiple disk images/log bundles, import results into Timesketch, and create correlated timeline queries to identify the chain of compromise.

Prerequisites: - Analysis VM with Plaso (log2timeline) and Timesketch (or use a hosted Timesketch instance). Instructor provides two disk image files or exported logs: hostA.dd and hostB.dd. - Python3, pip-installed plaso, and Timesketch access.

Tools & files: log2timeline.py, psort.py, plaso-containers (optional), Timesketch web UI.

tasks (step-by-step): 1. Prepare work directory and compute checksums for images: sha256sum hostA.dd hostB.dd.

2. Run log2timeline to create Plaso storage files: - log2timeline.py plaso_hostA.plaso hostA.dd - log2timeline.py plaso_hostB.plaso hostB.dd

3. Use psort.py to filter interesting event types (e.g., authentication, process creation, network events): - psort.py -o L2tcsv -w hostA_events.csv plaso_hostA.plaso "--source=windows:evtx,syslog" (adjust as needed)

4. Alternatively, upload .plaso files to Timesketch (UI) and create a sketch for correlation across hosts.

5. In Timesketch create queries to find a pivoting sequence, for example: - index:hostA OR index:hostB event_type:process_creation "cmd.exe" - Search for same user account logging into both hosts within short time window: user:alice earliest:-1h latest:+1h. 6. Produce a timeline of the compromise: initial access event, lateral movement (PsExec/ssh), data staging, exfil.

Solution / Expected outputs (example): - psort CSV shows 2025-11-05T03:14:22Z Process rundll32.exe on hostA executing C:\Windows\Temp\mal.dll. - On hostB there is a wmi remote execution entry 5 minutes later with same alice user. - Timesketch visualization shows clustering of events indicating lateral movement and data staging into C:\Users\alice\AppData\Local\Temp\archive.zip.

Practical E — Detection Engineering: Writing & Testing Sigma Rules and ElasticSearch/OSQuery Queries (Hunting)

Objective: Create Sigma detection rules for suspicious PowerShell and abnormal process spawning, convert them to ElasticSearch queries (or Suricata/YARA where applicable), and test them against provided log samples.

Prerequisites: - should have Python3 and sigmac (Sigma converter) installed; access to a set of Windows Event logs or ELK-styled JSON logs .

Tools & files: sigma rules repo templates, sigmac (Sigma converter), sample logs winlogs_sample.json or sysmon_evtx.

tasks (step-by-step): 1. Review sample logs to identify hunting gaps: search for event IDs 1 (Process Create) in Sysmon or PowerShell ScriptBlock logging. 2. Write a Sigma rule targeting PowerShell encoded commands + unusual parent process (e.g., cmd.exe spawning powershell.exe with -EncodedCommand):

```
title: Suspicious PowerShell EncodedCommand
id: f1a2b3c4-d5e6-7890-abcd-ef0123456789
status: experimental
description: Detects PowerShell started with -EncodedCommand by non-administrative shells.
logsource:
  product: windows
  service: sysmon
detection:
  selection:
    EventID: 1
    Image|endswith: '\\powershell.exe'
    CommandLine|contains: '-EncodedCommand'
  condition: selection
level: high
```

3. Convert the Sigma rule to an ElasticSearch query using sigmac -t es-qs -c config/elk-windows.yml rule.yml and inspect the output.
4. Test the rule against sample logs using a simple grep or Python script that loads JSON and applies the rule conditions.
5. Tune the rule to reduce false positives by adding ParentImage filters and excluding known admin tools.
6. Document the test methodology and false-positive rate on the provided dataset.

Solution / Expected outputs (example): - Sigma rule matches 5 events in winlogs_sample.json where CommandLine contains -EncodedCommand and ParentImage is cmd.exe. - Converted Elastic query successfully produces a query DSL block; running against a test Elasticsearch index returns the 5 hits. False positives reduced by excluding C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe when launched by schtasks.exe (legitimate automation) — add an exclude clause.