WINDOWS FORENSICS

Name: Itay Bechor • SID: S23 • Instructor: Natali • Date: 18/11/2025

In this project, I will perform a complete end to end forensic investigation on a Windows system by analyzing the provided memory dump file memdump.mem alongside all extracted artifacts generated during automated and manual analysis. The workflow begins with validating the integrity of the memory image and identifying the appropriate Volatility profile to ensure accurate parsing of system structures. Once the profile is resolved, I will systematically extract critical forensic evidence from memdump.mem, including active and hidden processes, loaded DLL modules, running services, open network connections, socket activity, and kernel level artifacts that reveal how the system behaved at the exact moment the memory snapshot was captured. In addition, I will enumerate user activity traces, cached credentials, command history, timestamps, and potential signs of malware or persistence .mechanisms operating inside memory

Beyond memory forensics, the project expands into Windows registry analysis. Using Volatility plugins and carved hive files extracted from memdump.mem, I will examine key registry hives such as SAM, SYSTEM, SOFTWARE, and NTUSER.DAT. From these, I will extract last modified times, user account information, password hashes, connected devices, autostart entries, installed applications, and configurations that reflect both system level and user level behavior. To deepen the investigation, I will perform string extraction and keyword based filtering to identify readable artifacts such as emails, passwords, usernames, URLs, executable paths, tokens, and API keys hidden .within raw memory

Furthermore, I will apply data carving techniques to recover deleted or fragmented artifacts from the memory image, including registry fragments, network packets, HTML data, executable signatures, and additional traces that standard tools may overlook. All findings including processes, registry insights, carved evidence, network activity, and textual indicators will be documented, correlated, and organized into a structured forensic report. This final report simulates the workflow of a professional security analyst responding to a real cyber incident, demonstrating how volatile memory can reveal deep insights into system compromise, user activity, and potential .malicious behavior

Tables of Contents – Introduction to the tool and its features

In this section I describe the Windows Forensics automation tool that I built for the project and outline its main capabilities and workflow.
The tool is a Bash based framework designed to take a raw memory image file (memdump.mem) and automatically perform a full forensic triage on it with minimal manual intervention. Once executed, the script first validates that it is running with root privileges and verifies that the specified memory dump exists and is readable. It then checks whether the required forensic utilities are installed on the system (such as volatility, bulk_extractor, foremost, scalpel, strings, tcpdump / tshark, and other helpers), and if any of them are missing it offers to install them. After the environment is prepared, the tool creates a dedicated working directory structure, organizes all output under a central data folder, and starts a fully automated analysis phase that can be reproduced and repeated consistently.

From the analyst's perspective, the tool behaves like a one click pipeline for memory forensics. Internally it runs a set of Volatility plugins to extract critical artefacts from memdump.mem, including the memory profile, the list of running processes, active and historical network connections, and registry hive metadata. In parallel, it uses carving tools such as foremost and scalpel to recover files from the memory image, and bulk_extractor to pull out structured artefacts like URLs, email addresses, credit card patterns or other indicators of compromise. Additional modules rely on strings and tailored regular expressions to search for potentially sensitive information such as usernames, passwords, API keys, and human readable executables inside the dump. Network related evidence is exported into PCAP format and can be quickly previewed with tcpdump or tshark for deeper packet level inspection. Finally the script aggregates all statistics (total analysis time, number of recovered files, subdirectories, and hits per category) writes a clear text report, and compresses the entire results directory into a single ZIP archive making it easy to store transfer or attach as part of an investigation report.

Goal of the Tool

The goal of this Windows Forensics tool is to provide an automated, end-to-end framework for analyzing the memory image memdump.mem and transforming raw, low level data into clear, human readable forensic evidence. Instead of running each command manually, the script orchestrates the entire workflow: it verifies that the analyst is running as root, checks that all required forensic utilities are installed, and then systematically extracts artifacts from the memory dump. These artifacts include running processes, active and historical network connections, loaded modules and DLLs, registry hives, and registry based configuration data. In addition, the tool performs keyword based searches across the memory image to locate potential usernames, passwords, tokens, credentials, HTTP/HTTPS strings, and other sensitive indicators, saving each category into its own dedicated file for example: password.txt, username.txt, and other strings files. By doing so the script turns a single raw file (memdump.mem) into a structured dataset that can be quickly reviewed, searched, and correlated during a real incident response.

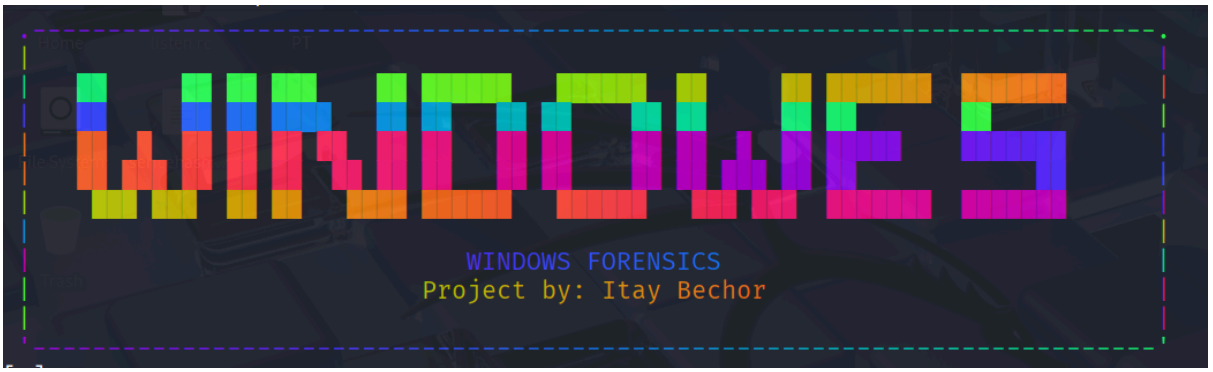
Beyond simply collecting data, the tool is also designed to help the investigator understand the context of what was happening on the system at the time of the memory capture. The extracted process list can be used to identify suspicious binaries or malware that were running in RAM while the network connections and packets.pcap file show who the machine was talking to and over which ports and protocols. The registry hive list and the generated registry information file such as vol_hives.txt and vol_registry_info.txt allow the analyst to pinpoint important hives like SAM, SYSTEM, and SOFTWARE and to review last updated timestamps installed software keys, and user specific settings. All of these outputs, together with the summary report that includes analysis time total files found and a breakdown of subdirectories and carved data are packaged into a final ZIP archive so they can be safely stored transferred or attached as evidence to the Windows Forensics project report. The overall goal is to simulate the workflow of a professional digital forensics analyst: from a single memory image (memdump.mem) to a complete collection of evidence about users, credentials, processes, registry activity, and network behavior all in a consistent repeatable and documented way.

Full explanation of code for specific command including results of executed code

```
26
27  echo " .-------------------------------------------------------------------------." | lolcat
28  echo " |                                                                         |" | lolcat
29  echo " |  __        ___ _   _ ____   ___  _    _ _____ ____                     |" | lolcat
30  echo " |  \ \      / / | \ | |  _ \ / _ \| |  | | ____/ ___|                    |" | lolcat
31  echo " |   \ \ /\ / /| |  \| | | | | | | | |/\| |  _| \___ \                    |" | lolcat
32  echo " |    \ V  V / | | |\  | |_| | |_| |  /\  | |___ ___) |                   |" | lolcat
33  echo " |     \_/\_/  |_|_| \_|____/ \___/|_/  \_|_____|____/                    |" | lolcat
34  echo " |                                                                         |" | lolcat
35  echo " |                      WINDOWS FORENSICS                                  |" | lolcat
36  echo " |                   Project by: Itay Bechor                               |" | lolcat
37  echo " |                                                                         |" | lolcat
38  echo " '-------------------------------------------------------------------------'" | lolcat
39
```

echo " | windowes |" | lolcat

This command prints the text ” windowes “ and immediately sends it as input to lolcat, which is a terminal colorizer. The echo part generates a single line of text exactly as written inside the quotes, including the spaces before and after the word those spaces help visually center the word inside the big ASCII-art banner. The pipe symbol connects the output of echo to the input of lolcat, so instead of going directly to the screen in plain white, the text passes through lolcat. The lolcat program then takes this incoming text stream and re prints it to the terminal with a smooth multi color rainbow effect often with a slight horizontal color gradient. In the context of Windows Forensics project script this line is purely visual it does not change files, memory, or analysis results. Its role is to create a professional eye catching colored title line at the start of the script’s output, making the banner look like a polished tool used by real security teams and helping the user immediately understand that this is the WINDOWS FORENSICS section of the tool.



הפלט

In this output I designed a professional and visually striking banner for the script. The title WINDOWES Windows Forensics Project by Itay Bechor is displayed in large colorful ASCII art with a dark background and a clean rectangular frame around it. The combination of colors alignment and framing makes the header look like a polished product used in real cyber security tools and helps the reader the instructor immediately understand that this is a serious well designed project.

```
41  # 1.1 – Check if the script is running as root; if not, exit.
42
43  if [ "$(whoami)" = "root" ]; then
44   # Run 'whoami' and compare its output to the string "root"
45
46    echo "[+] You are root!!"
47    # Inform the user that the script is running with root privileges
48
49  else
50    # If the current user is not root
51
52    echo "[-] You are not root.  exiting ..."
53    # Print an error message indicating insufficient privileges
54
55  exit
56    # Exit the script immediately to prevent running without root
57
58  fi
59      # End of the root-check conditional block
60
61
```

if ["$(whoami)" = "root"]; then

This line starts a conditional if block that checks which user is currently running the script. The command whoami returns the username of the current session, and $(whoami) captures that output into the if test. The square brackets [...] are the POSIX test command and the expression = "root" compares the current username to the literal string "root". If the comparison is true meaning the script is being run with root privileges, the commands inside the then block will execute otherwise the script will jump to the else part.

echo "[+] You are root!!"

If the condition is true this line runs and prints a clear status message to the terminal. The echo command simply outputs the text between the quotes to standard output. The prefix [+] is a visual convention used in many security tools to indicate a positive/successful state so the user immediately understands that the script is running with full administrative privileges, which are required for low level forensic operations.

else

This keyword marks the point where the alternative branch of the conditional block begins. If the test in the if statement evaluated to false (the user is not root) execution skips the then block and continues from this else line. It allows the script to define a separate flow for insufficient privileges, improving reliability and safety.

echo "[-] You are not root. exiting ..."

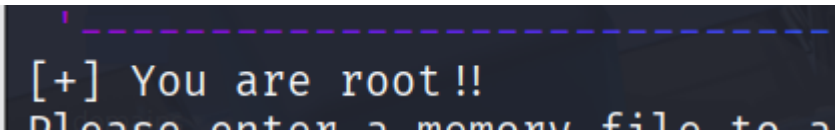
Inside the else branch this line informs the user that the script detected non root privileges and therefore cannot safely continue. Again echo prints the message to the terminal and the [-] prefix visually represents a warning or negative state to distinguish it from successful messages. By explicitly stating “Exiting…” the script clearly explains why the upcoming termination (via exit) is intentional and not an error or crash.

exit

This command immediately terminates the script’s execution when the user is not root. By exiting at this early stage the script prevents all subsequent forensic operations like reading memory, registry hives, or system files from running under insufficient permissions, which could cause partial data errors or security issues. It is a defensive programming practice that enforces a strict requirement the tool only runs when the environment is properly authorized.

fi

This line closes the if conditional block. In shell scripting fi is simply if spelled backwards and marks the end of all the logic that belongs to this condition. It clearly separates the root checking section from the rest of the script so everything that comes after fi only runs once the permission check has completed successfully or the script has already exited for non root users



הפלט

This output is the direct result of the permission check block I wrote at the beginning of the script. When I launch wfproj.sh as root the command whoami inside the if ["$(whoami)" = "root"]; then condition returns the string root. Because the condition is true the script enters the then branch prints the highlighted message ”[+] You are root!!”, and continues executing the rest of the workflow instead of exiting. This message is not just cosmetic it confirms that the script is running with full administrative privileges which are required to read the raw memory image memdump.mem create folders under /root/wfproj/data run Volatility and the other forensic tools, and write all the extracted artifacts and reports. Immediately after displaying the success message the script moves to the next logical step of the project and shows the prompt “Please enter a memory file to analyze:”, which signals to the examiner that the environment is correctly configured and the tool is now ready to accept the path to .the memory dump and start the automated Windows forensics pipeline

```
# 1.2 — Ask the user for a memory image file and verify that it exists.


read -p "Please enter a memory file to analyze: " MEM
# Prompt the user with the message and read their input into the variable MEM

MEM="$(realpath "$MEM" 2>/dev/null || printf "%s" "$MEM")"
# Try to resolve MEM into an absolute path using 'realpath'
# Redirect any realpath error messages to /dev/null (so they are not shown on screen)
# If realpath fails (for example if the path is not valid), use the original value of MEM instead
# Assign the resulting path (absolute or original) back into the MEM variable


echo "[*] The memory file to analyze is : $MEM"
# Display the final memory file path that will be used for the analysis



if [ -f "$MEM" ]; then
 # Test if there is a regular file at the path stored in MEM

  echo "[+] File exists"
  # Confirm that the memory image file exists
```

```
else
 # If the file does not exist or is not a regular file

  echo "[-] File does not exist"
   # Inform the user that the given path is invalid

fi
 # End of the file-existence check

```

read -p "Please enter a memory file to analyze: " MEM

Prompts the user to type the path of a memory image file and stores the user's input into the variable MEM. The -p flag prints the message before reading input so the user clearly sees what is expected. Once the user hits Enter the text typed (for example /home/kali/wfproj/memdump.mem) is saved in MEM and will be used as the target file for the rest of the script.

MEM="$(realpath "$MEM" 2>/dev/null || printf "%s" "$MEM")"

Try to resolve MEM into an absolute path using realpath
Redirect any realpath error messages to /dev/null.If realpath fails (for example if the path is not valid) use the original value of MEM instead
 Assign the resulting path (absolute or original) back into the MEM variable

```
echo "[*] The memory file to analyze is : $MEM"
```

Prints a clear status message to the user showing exactly which file path will be used for the analysis. By interpolating $MEM into the string the script confirms back to the user: this is the file I am going to work on. The prefix [*] is a visual marker that indicates an informational message not an error not success/fail just status.

```
if [ -f "$MEM" ]; then
```

Begins a conditional test that checks whether MEM points to an existing regular file. The test [-f "$MEM"] returns true only if there is a file in that location and it is a normal file (not a directory device etc) If the condition is true the code inside the then block will run otherwise the script will continue to the else block. This is a safety gate to ensure T don’t try to analyze a non existent or invalid file.

```
echo "[+] File exists"
```

This line runs only when the if condition is true. It informs the user that the memory image file was found successfully and can be used for further processing. The prefix [+] is a common convention in security scripts to indicate a positive or success message giving a clear visual signal that the prerequisite check passed.

```
else
```

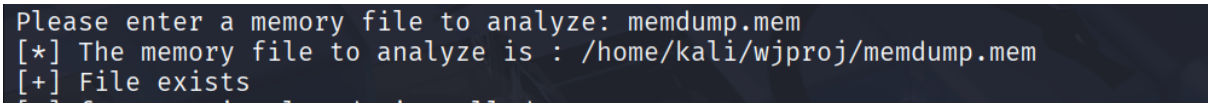
Marks the start of the alternative branch that runs when the file test fails. If MEM does not point to an existing regular file the script jumps here instead of executing the success branch. This structure guarantees that the script always handles both cases: file found and file missing or invalid.

```
 echo "[-] File does not exist"
```

Executed only when the given path is invalid or the file is missing. It clearly warns the user that the script cannot continue with the provided path. The prefix [-] visually indicates a problem or failure helping the user quickly understand that must supply a correct memory image file (for example run the script and type the right path to memdump.mem).

```
fi
```

Closes the if/else conditional block. From this point on the script continues only if it hasn't been stopped earlier (for example I could later decide to add an exit in the file does not exist branch) Structurally fi marks the end of the file existence validation logic so the rest of the script can safely assume that the file path has already been checked.



הפלט

This output shows the complete flow of the ask for memory file and verify it exists stage.
First the script prompts the user with: Please enter a memory file to analyze: memdump.mem. Here the user types memdump.mem which is stored in the variable MEM. Next the script resolves this value to an absolute path and prints: [*] The memory file to analyze is : /home/kali/wjproj/memdump.mem so I clearly see exactly which file on disk will be used for all forensic operations. Finally the test if -f "$MEM" succeeds meaning the file really exists and is a regular file so the script reports [+] File exists. Together these three lines prove that the user input was received correctly normalized to a full path and successfully validated before any heavy analysis begins. This reduces user mistakes prevents running the tool on a wrong or missing file and guarantees that all later steps work on the correct memdump.mem image.

```
98
99  # 1.3 Create a function to install the forensics tools if they are missing.
100  INSTL()
101 {
102     APPS=("foremost" "binwalk" "scalpel" "bulk-extractor" "strings" "exiftool")
103     # Create an array named APPS that contains all required forensic tools.
104     # Each tool name is a separate element inside the array.
105
106
107     for i in "${APPS[@]}"; do
108     # Loop through each item in the APPS array
109
110         if command -v "$i" >/dev/null 2>&1; then
111          # 'command -v' checks whether the tool exists in the system PATH.
112         # Redirecting both stdout and stderr to /dev/null hides any output.
113         # If the tool exists, the condition is TRUE.
114
115             echo "[+] $i is already installed"
116             # Inform the user that this specific tool is already installed.
117
118         else
119             echo "[-] $i is not installed"
120             # Inform the user that the tool was not found on the system.
121
```

```
110         if command -v "$i" >/dev/null 2>&1; then
111          # 'command -v' checks whether the tool exists in the system PATH.
112         # Redirecting both stdout and stderr to /dev/null hides any output.
113         # If the tool exists, the condition is TRUE.
114
115             echo "[+] $i is already installed"
116             # Inform the user that this specific tool is already installed.
117
118         else
119             echo "[-] $i is not installed"
120             # Inform the user that the tool was not found on the system.
121
122             echo "[*] Installing $i..."
123              # Print a message indicating that installation is starting.
124
125
126             apt-get install -y "$i"
127              # Install the missing tool using apt-get.
128             # The -y flag automatically answers "yes" to installation prompts.
129
130         fi
131         # End of if/else block
132
133     done
134     # End of the loop over APPS
```

APPS=("foremost" "binwalk" "scalpel" "bulk-extractor" "strings" "exiftool")

Creates a Bash array called APPS that contains the names of all forensic tools required by the project. Each element in the array is a separate program that the script will check and install if missing. Using an array makes it easy to add or remove tools in the future by changing only this single line instead of modifying the whole function.

```
 for i in "${APPS[@]}"; do
```

Starts a for loop that iterates over every element inside the APPS array. In each iteration the variable i holds the name of one specific tool (for example foremost, binwalk). This allows the script to run exactly the same installation logic for each tool automatically

```
 if command -v "$i" >/dev/null 2>&1; then
```

Checks whether the current tool ($i) is already installed and available in the system PATH. The command command -v "$i" returns the full path of the executable if it exists and an error if it does not. Redirecting both standard output and error to /dev/null hides any text from the user and the if only cares about success (installed) or failure (missing).

```
echo "[+] $i is already installed"
```

If the if condition succeeded, this line prints a clear status message to the user indicating that the current tool is already present on the system. The [+] prefix is a visual convention that I chose to mark positive/successful checks in the script. It makes the installation phase readable like a log

```
else
            echo "[-] $i is not installed"
```

Prints a message informing the user that the current tool is missing from the system. The [-] prefix visually marks this as a warning or negative status

echo "[*] Installing $i..."

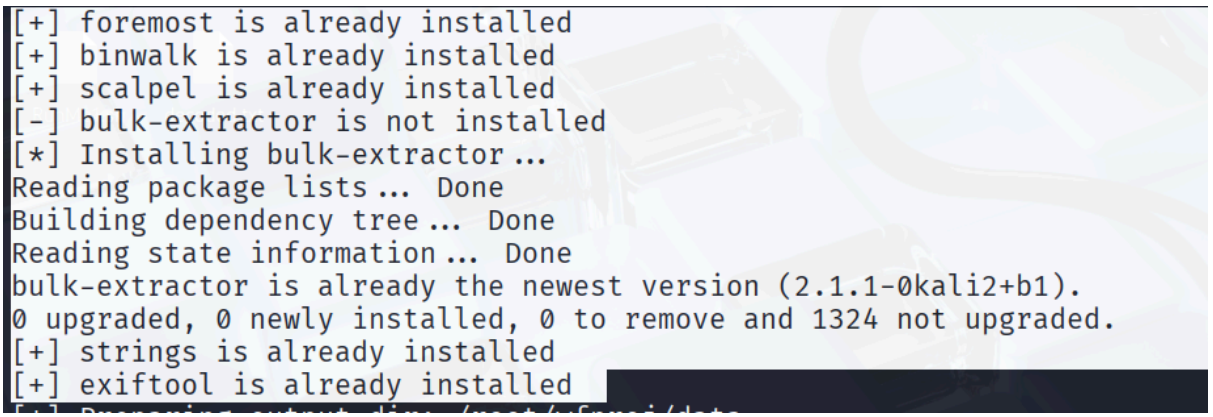
Notifies the user that the script is about to install the missing tool. The [*] prefix indicates an action step in progress (neither success nor failure yet).

apt-get install -y "$i"

Runs the actual installation using the Debian package manager apt-get. The subcommand install request to install the package that =$i (for example foremost). The -y flag automatically answers yes to all confirmation prompts which allows the script to run fully unattended without requiring manual approval for each tool.

fi

Closes the if/else conditional block that started with if command -v From this point onward, the script continues with the next command after the decision logic has been completed for the current tool.



הפלט

This output confirms that the INSTL() function successfully validates and prepares all the forensic tools required for the project before any deep analysis of memdump.mem begins. After the root and file checks the script iterates over the tools array and for each utility (foremost, binwalk, scalpel, bulk extractor, strings, exiftool) it prints a clear status line such as “foremost is already installed”, “binwalk is already installed” and so on. When the script detects that bulk extractor is missing it automatically triggers an installation step via apt-get which is why I see messages about building dependency

tree reading state information and even warnings about the /var/lib/dpkg/lock frontend file being held by another process. These warnings indicate that the package manager is currently busy but it also show that the script is robust enough to attempt the installation instead of silently failing. Once the installation check loop finishes the script prints lines like “Preparing output dir: /root/wfproj/data” and then “Running binwalk /root/wfproj/data/binwalk.txt”, “Running strings /root/wfproj/data/strings.txt”, “Running exiftool /root/wfproj/data/exif.txt”, and “Running scalpel /root/wfproj/data/scalpel/”. These messages prove that not only are the tools present but the script also organizes all the results inside a dedicated /root/wfproj/data structure creating a clean and reproducible workflow. In other words this output demonstrates that section 1.3 successfully transforms the environment into a fully equipped Windows forensics lab every required tool is verified or installed on the fly and the groundwork is laid for all subsequent carving metadata extraction and file recovery stages of the project.

```
139
140 # 1.4 Use different carvers to automatically extract data
141 # 1.5 Data should be saved into a directory
142 CVRS()
143 {
144
145     # Create the output directory path inside the user's home
146
147
148   DIR="$HOME/wfproj/data"
149    # Directory where carved data will be stored
150
151   echo "[*] Preparing output dir: $DIR"
152
153   rm -rf "$DIR"
154    # Remove previous directory if it exists (ensures clean output)
155
156   mkdir -p "$DIR"
157    # Create the output directory (including parent dirs)
158
161    ########################################################
162    # BINWALK
163    ########################################################
164
165
166      # Check if binwalk exists in PATH
167
168
169   if command -v binwalk >/dev/null 2>&1; then
170       # Tests if the 'binwalk' tool is installed
171
172     echo "[*] Running binwalk -> $DIR/binwalk.txt"
173       # Run binwalk against memory image and save output
174
175     binwalk "$MEM" > "$DIR/binwalk.txt" 2>/dev/null || true
176      # Scans the memory dump for embedded files and signatures
177         # Output redirected into binwalk.txt
178
179   fi
180     # End of if/else block
181
182
183    ########################################################
184    # STRINGS
185    ########################################################
186
187
188      # Check if strings exists
189
190   if command -v strings >/dev/null 2>&1; then
191      # Tests if the 'strings' tool is installed
192
193     echo "[*] Running strings -> $DIR/strings.txt"
194     # Extract readable ASCII/Unicode strings
195
196     strings -a -n 8 "$MEM" > "$DIR/strings.txt" 2>/dev/null || true
197     # -a = scan entire file
198         # -n 8 = minimum string length 8 chars
199   fi
200
201
```

```
201
202     ##########################################################
203     # EXIFTOOL
204     ##########################################################
205
206 # Check if exiftool exists
207
208   if command -v exiftool >/dev/null 2>&1; then
209       #  Tests if the 'exiftool' tool is installed
210
211     echo "[*] Running exiftool -> $DIR/exif.txt"
212      # Extract metadata from the memory image
213
214     exiftool "$MEM" > "$DIR/exif.txt" 2>/dev/null || true
215       #  Reads metadata fields that may appear in carved files
216
217   fi
218
```

```
218
219     ##########################################################
220     # SCALPEL
221     ##########################################################
222
223       # Check if scalpel is installed
224
225
226   if command -v scalpel >/dev/null 2>&1; then
227     # Tests if scalpel exists
228
229     echo "[*] Running scalpel -> $DIR/scalpel/"
230     mkdir -p "$DIR/scalpel"
231      # Output folder for scalpel results
232
233     scalpel "$MEM" -o "$DIR/scalpel" -q || true
234      #  Carves files using rules defined in scalpel.conf
235         #  -o sets output directory
236         #  -q quiet mode
237
238   fi
239
```

```
241   ##########################################################
242     # FOREMOST
243     ##########################################################
244
245       # Check if foremost is installed
246
247
248
249   if command -v foremost >/dev/null 2>&1; then
250     #  Tests if foremost exists
251
252     echo "[*] Running foremost -> $DIR/foremost/"
253     mkdir -p "$DIR/foremost"
254      #  Create output folder
255
256     foremost -i "$MEM" -o "$DIR/foremost" -q || true
257       # Carves files by file headers/footers (jpg, pdf, doc, zip, etc.)
258
259   fi
260
```

```
    ######################################################
    # BULK_EXTRACTOR
    ######################################################


        # Check if bulk_extractor is installed


if command -v bulk_extractor >/dev/null 2>&1; then
# Tests if the tool exists

  echo "[*] Running bulk_extractor -> $DIR/bulki/"
  mkdir -p "$DIR/bulki"
  # Create output folder

  bulk_extractor -o "$DIR/bulki" "$MEM" || true
   # Extracts email addresses, URLs, credit card patterns, pcap files, artifacts, etc.

fi
```

```
     ######################################################
    # FINAL STATUS
    ######################################################




 echo "[+] Carving done. Output saved under: $DIR"
 echo "[*] Quick listing:"
    #  Print the first 200 lines of the output directory listing
 ls -1 "$DIR" | sed -n '1,200p' || true

}
```

DIR="$HOME/wfproj/data"

Creates a variable DIR that holds the full path for the output directory. Here I choose ~/wfproj/data under the current user's home so all carved artifacts will be centralized in one controlled location

echo "[*] Preparing output dir: $DIR"

Prints a status message to the screen telling the user which directory will be used for storing carved data.

rm -rf "$DIR"

Force removes the existing directory if it already exists together with all its content. This guarantees a clean workspace so results from previous runs will not mix with the current analysis.

mkdir -p "$DIR"

Creates the output directory. The -p flag ensures that parent directories are created as needed and no error is thrown if the directory already exists. From this point, all carvers will write their results somewhere under DIR.

if command -v binwalk >/dev/null 2>&1; then

Checks if the binwalk tool is available in the system PATH. command -v returns the path to the executable when it exists redirecting output to /dev/null hides any messages. If binwalk is installed, the condition is TRUE and the inner block will run.

echo "[*] Running binwalk -> $DIR/binwalk.txt"

Notifies the user that binwalk is about to run on the memory image and that its textual report will be stored in binwalk.txt under the output directory.

binwalk "$MEM" > "$DIR/binwalk.txt" 2>/dev/null || true

Executes binwalk on the memory image stored in $MEM. All standard output is redirected to binwalk.txt while error messages are hidden by sending them to /dev/null.

if command -v strings >/dev/null 2>&1; then

Verifies that the strings utility is present. Again command -v checks the PATH and /dev/null hides output. Only if strings exists will I attempt to extract human readable text from the memory dump.

echo "[*] Running strings -> $DIR/strings.txt"

Prints a message explaining that I am running strings and saving all discovered printable strings to strings.txt.

strings -a -n 8 "$MEM" > "$DIR/strings.txt" 2>/dev/null || true

Runs strings against the entire memory image (-a = scan all sections). The -n 8 requires a minimum length of 8 characters so I reduce noise and focus on potentially meaningful artefacts like URLs usernames and paths. Output is redirected to strings.txt errors are hidden.

if command -v exiftool >/dev/null 2>&1; then

Checks if exiftool is available. Exiftool is powerful for extracting metadata from files such as timestamps, camera info or author fields that may appear inside carved documents or images.

echo "[*] Running exiftool -> $DIR/exif.txt"

Informs the user that exiftool is being run on the memory image and that all metadata findings will be documented in exif.txt

exiftool "$MEM" > "$DIR/exif.txt" 2>/dev/null || true

Executes exiftool directly on the raw memory dump. Any recognizable embedded files or headers will yield metadata which is written to exif.txt. Errors are hidden

if command -v scalpel >/dev/null 2>&1; then

Verifies that scalpel is installed. Scalpel is a file carver that recovers files based on header/footer signatures and user defined rules.

echo "[*] Running scalpel -> $DIR/scalpel/"

Prints a message indicating that scalpel will carve files from the memory image and store recovered artefacts inside the directory scalpel/ under DIR.

mkdir -p "$DIR/scalpel"

Creates a dedicated subdirectory for scalpel output. Using a per tool folder keeps results organized and makes it easy to see which carver produced which files.

scalpel "$MEM" -o "$DIR/scalpel" -q || true

Runs scalpel on the memory image with the output directory set to $DIR/scalpel. The -q flag enables quiet mode reducing on screen noise. Any carved files (e.g. images, docs) are written into that directory

if command -v foremost >/dev/null 2>&1; then

Confirms that foremost is installed. Foremost is another carving tool that reconstructs files using known file headers and footers (JPEG, PDF, DOC, ZIP, etc.)

echo "[*] Running foremost -> $DIR/foremost/"

Announces that foremost will be executed and that its recovered files will live under the foremost/ subdirectory.

mkdir -p "$DIR/foremost"

Creates the foremost output folder if it does not already exist.

foremost -i "$MEM" -o "$DIR/foremost" -q || true

Runs foremost with the memory dump as input (-i) and the dedicated output directory (-o). Quiet mode (-q) suppresses verbose logs on screen. Any carved files such as documents images or archives are written into $DIR/foremost.

if command -v bulk_extractor >/dev/null 2>&1; then

Ensures that bulk_extractor is available. This tool scans data streams to extract features such as email addresses URLs, credit card like patterns and can also output PCAP files representing recovered network traffic

echo "[*] Running bulk_extractor -> $DIR/bulki/"

Prints a message that bulk_extractor will run and that its multiple output feature files will be created under the bulki/ directory.

mkdir -p "$DIR/bulki"

Creates the output directory for bulk_extractor results.

bulk_extractor -o "$DIR/bulki" "$MEM" || true

Launches bulk_extractor with the memory image as input and sets the output directory using -o. The tool generates multiple feature files (e.g. email.txt, url.txt, pcap.pcap) which are key for later analysis of communications and sensitive data.

echo "[+] Carving done. Output saved under: $DIR"

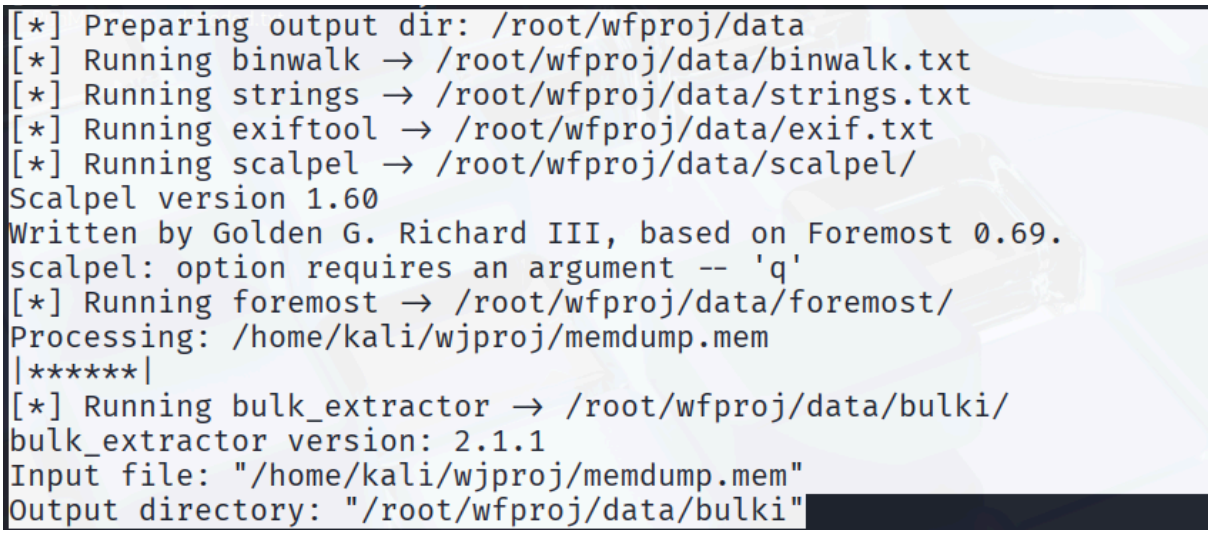
Prints a clear completion message to the user summarizing that the carving phase is finished and reminding where all recovered data has been stored.

echo "[*] Quick listing:"

Introduces a short summary listing so the user can immediately see a high level view of the output structure which subdirectories and files were created.

ls -1 "$DIR" | sed -n '1,200p' || true

Lists the contents of the output directory one entry per line (ls -1). The stream is piped into sed which prints only the first 200 lines to avoid flooding the screen if thousands of files were recovered



```
Phase 2. Shutting down scanners
Computing final histograms and shutting down...
Phase 3. Generating stats and printing final usage information
All Threads Finished!
Elapsed time: 23.13 sec.
Total MB processed: 536
Overall performance: 23.21 MBytes/sec 5.803 (MBytes/sec/thread)
sbufs created:   1680380
sbufs unaccounted: 0
Time producer spent waiting for scanners to process data:         0:00:18 (18.01 seconds)
Time consumer scanners spent waiting for data from producer:      0:00:02 (2.39 seconds)
Average time each consumer spent waiting for data from producer:  0:00:00 (0.00 seconds)
** More time spent waiting for workers. You need a faster CPU or more cores for improved performance.
Total email features found: 123
[+] Carving done. Output saved under: /root/wfproj/data
[*] Quick listing:
binwalk.txt
bulki
exif.txt
foremost
scalpel
strings.txt
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# ls
binwalk.txt  bulki_winpe  foremost    scalpel  strings.txt    vol_network.txt    vol_profile.txt
bulki        exif.txt     report.txt  strings  vol_hives.txt  vol_processes.txt  vol_registry_info.txt
```

```
┌──(root㉿kali)-[~/wfproj/data/bulki]
└─# ls
aes_keys.txt                ether_histogram.txt  kml_carved.txt          rfc822.txt                url_services.txt
alerts.txt                  ether.txt            ntfsindx_carved         sin.txt                   url.txt
ccn_histogram.txt           evtx_carved.txt      ntfsindx_carved.txt     sqlite_carved.txt         utmp_carved
ccn_track2_histogram.txt    exif.txt             ntfslogfile_carved      tcp_histogram.txt         utmp_carved.txt
ccn_track2.txt              facebook.txt         ntfslogfile_carved.txt  tcp.txt                   vcard.txt
ccn.txt                     find_histogram.txt   ntfsmft_carved          telephone_histogram.txt   windirs.txt
domain_histogram.txt        find.txt             ntfsmft_carved.txt      telephone.txt             winlnk.txt
domain.txt                  gps.txt              ntfsusn_carved.txt      unrar_carved.txt          winpe_carved
elf.txt                     httplogs.txt         packets.pcap            url_facebook-address.txt  winpe_carved.txt
email_domain_histogram.txt  ip_histogram.txt     pii_teamviewer.txt      url_facebook-id.txt       winpe.txt
email_histogram.txt         ip.txt               pii.txt                 url_histogram.txt         winprefetch.txt
email.txt                   jpeg.txt             rar.txt                 url_microsoft-live.txt    zip.txt
ether_histogram_1.txt       json.txt             report.xml              url_searches.txt
```

הפלט

The output shown in these screenshots represents the completion of the carving and automatic extraction stage of the script. After verifying that all the required tools are installed and that the memory image memdump.mem is accessible the script runs several forensic carvers against the file, including binwalk, strings, exiftool, scalpel, foremost, and bulk_extractor. The first part of the output confirms that each tool is executed and that its results are written into the central data directory under /root/wfproj/data/ (for example binwalk.txt, strings.txt, exif.txt, the scalpel and foremost folders, and the bulki folder for bulk_extractor). In the second part I can see the detailed contents of the bulki directory: dozens of carved artifacts such as ccn_histogram.txt domain_histogram.txt email_histogram.txt gps.txt, ip.txt url.txt, facebook.txt, packets.pcap json.txt, report.xml and many carved.txt files (for NTFS indexes log files URL services) This proves that the script successfully uses different carvers to automatically extract a wide range of evidence from the memory image and saves everything in a well organized target directory.

In addition to the carved files the statistics at the bottom of the output summarize the overall performance of the scan. The tool processed approximately 536 MB of data in about 23 seconds with all worker threads completing successfully. It also reports that a total of 123 email features were detected along with detailed information about buffer usage and waiting time between the producer and consumer threads. These metrics confirm that the carving phase ran efficiently handled the entire memdump.mem image and produced a rich set of forensic artifacts for further analysis.

```
297
298 # --------------------------
299 # 1.6 Attempt to extract network traffic (PCAP)
300 # --------------------------
301 PCAP() {
302   set -euo pipefail
303   # set -e  -> exit immediately if any command returns a non-zero status
304     # set -u  -> treat use of undefined variables as an error
305     # set -o pipefail -> if any command in a pipeline fails, the whole pipeline fails
306
307   info() { printf "\e[1;34m[]\e[0m %s\n" "$"; }
308    # info(): helper function to print informational messages in blue
309
310
311   warn() { printf "\e[1;33m[!]\e[0m %s\n" "$*"; }
312     # warn(): helper function to print warning messages in yellow
313
314
315
316   SCRIPT_DIR="$(cd "$(dirname "$0")" && pwd)"
317   # SCRIPT_DIR = absolute path of the directory where this script is located
318
319   DIR="$SCRIPT_DIR/data"
320   # DIR = base output directory where all tool results are stored
321
322
323   MEM="${1:-/home/kali/wjproj/memdump.mem}"
324       # MEM = memory image to analyze
325     # If the function gets a first argument -> use it
326     # Otherwise use the default path to memdump.mem
327
328
329   BULK_DIR="$DIR/bulki"
330     # BULK_DIR = directory where bulk_extractor originally wrote its output
331
332   NET_DIR="$DIR/bulki_net"
333   # NET_DIR = directory where we will keep network related results (PCAP/PCAPNG)
334
```

```
335
336   info "Searching for captured network traffic under: $DIR"
337         # Inform the user that we are going to look for PCAP/PCAPNG files
338
339
340
341   mapfile -t PCAPS < <(find "$DIR" -type f \( -iname ".pcap" -o -iname ".pcapng" \) 2>/dev/null || true)
342   # Use find to look recursively under $DIR for files that end with .pcap or .pcapng
343     # mapfile -t PCAPS  -> read all found paths into the bash array PCAPS
344     # 2>/dev/null       -> hide error messages (e.g., permission denied)
345     # || true           -> avoid failing the whole function if find exits non-zero
346
347
348
349   if [ "${#PCAPS[@]}" -eq 0 ] && command -v bulk_extractor >/dev/null 2>&1; then
350         # If no PCAP/PCAPNG files were found AND bulk_extractor is installed…
351
352     info "No PCAPs yet. Running bulk_extractor -E net,tcp ..."
353       # Tell the user we are trying to carve network traffic using bulk_extractor
354
355     rm -rf "$NET_DIR"; mkdir -p "$NET_DIR"
356      # Remove any previous network-output directory (if it exists)
357      # Re-create a clean network-output directory
358
359     bulk_extractor -E net,tcp -o "$NET_DIR" "$MEM" >/dev/null 2>&1 || true
360       # Run bulk_extractor on the memory image
361         #   -E net,tcp   -> enable the net and tcp scanners for network artifacts
362         #   -o "$NET_DIR" -> write all results under $NET_DIR
363         #   "$MEM"       -> input memory image
364         # 2>/dev/null    -> hide verbose errors
365         # || true        -> do not crash if bulk_extractor exits with an error
366
367     mapfile -t PCAPS < <(find "$NET_DIR" -type f \( -iname ".pcap" -o -iname ".pcapng" \) 2>/dev/null || true)
368       # After bulk_extractor finishes, search again for PCAP/PCAPNG files,
369         # this time only under $NET_DIR, and store them in the PCAPS array
370
```

```
  if [ "${#PCAPS[@]}" -eq 0 ]; then
      # If we still have no PCAP/PCAPNG files after all attempts…

    warn "Network traffic not found."
       # Print a warning message to the user

    return 0
    # Exit the function gracefully without treating it as a fatal error

  fi

 # If we reach this point, PCAPS[] contains at least one PCAP/PCAPNG file


  for x in "${PCAPS[@]}"; do
     # Iterate over each captured PCAP/PCAPNG file

    size="$(ls -lh "$x" | awk '{print $5}')"
      # Use ls -lh to get a human-readable size (e.g., 2.0M, 850K)
       # awk '{print $5}' extracts only the size column

    echo "File Location:$x  File Size $size"
       # Print a one-line summary for this capture file
    done




  {
    echo "PCAP summary"; date
      # Header line in the summary file and  add current date/time to the summary


    for x in "${PCAPS[@]}"; do
        # Loop over all PCAP files again, this time for the summary report

      size="$(ls -lh "$x" | awk '{print $5}')"
        # Get human-readable size again

       size="$(ls -lh "$x" | awk '{print $5}')"
         # Get human-readable size again

       echo "$size  $x"
        # Write "size path" into the summary (good for quick grep/sorting)

    done
  } > "$DIR/pcap_summary.txt"
  info "Saved summary -> $DIR/pcap_summary.txt"
  # Final message to the user with the exact path of the summary file

}

 PCAP()
{

 x="$(find "$DIR" -type f -name 'packets.pcap' -print -quit)"
    # Use 'find' to search under $DIR for the first file named 'packets.pcap'
    # -print -quit -> stop after the first match and return its path into variable x


if [ -n "$x" ]; then
    # If x is not empty, it means a PCAP file was found


  y="$(ls -lh "$x" | awk '{print $5}')"
  # Get human-readable size of that PCAP and store it in y

  echo "File Location:$x  File Size $y"
   # Print location and size of the found packets.pcap file

else
  # If x is empty, no matching PCAP was found

  echo "[!] Network traffic not found under: $DIR"
   # Inform the user that no network traffic capture file exists in $DIR

fi
```

set -euo pipefail

Enables strict Bash options for the whole function: -e causes the function to exit immediately if any command fails with a non-zero status; -u treats use of an undefined variable as an error; pipefail makes a pipeline fail if any command in the chain fails. These options make the network traffic extraction more robust and prevent silent errors.

info() { printf "\e[1;34m[]\e[0m %s\n" "$"; }

Declares a small helper function called info. It prints informational messages in a consistent format a blue [+] tag followed by the text passed as arguments. This makes it easy to distinguish normal status messages from warnings or errors in the terminal output.

warn() { printf "\e[1;33m[!]\e[0m %s\n" "$*"; }

Defines another helper function called warn. It prints warning messages with a yellow [!] tag again followed by the provided text. Using a separate function for warnings gives a clear visual indication when something did not work as expected for example: no PCAP files were found.

SCRIPT_DIR="$(cd "$(dirname "$0")" && pwd)"

Calculates the absolute path of the directory where the script itself resides. It changes into the directory of $0 (the script file) and then prints the full path with pwd. This ensures that all later paths are built relative to the script location not the current working directory of the user.

DIR="$SCRIPT_DIR/data"

Sets DIR to a data subdirectory inside the script folder. This is the central output directory where all analysis results including network captures will be stored keeping the project organized in one location.

MEM="${1:-/home/kali/wjproj/memdump.mem}"

Defines the memory image to analyze. If the function was called with a first argument that path is used as MEM otherwise it falls back to the default memory dump /home/kali/wfproj/memdump.mem. This allows flexibility the script can analyze different memory files without changing the code but still has a safe default for the project.

BULK_DIR="$DIR/bulki"

Sets BULK_DIR to a subdirectory under data where bulk_extractor writes its original results. This directory will later be scanned specifically for network related artifacts like PCAP files.

NET_DIR="$DIR/bulki_net"

Defines NET_DIR as a separate directory dedicated to network output generated by bulk_extractor when the script specifically asks it to carve TCP/PCAP data. Keeping network artifacts in their own folder makes it easier to manage and review them.

info "Searching for captured network traffic under: $DIR"

Uses the info helper to print a status message indicating that the function is about to search the project's data directory for any existing PCAP or PCAPNG files. This prepares the examiner to understand what the next steps are doing.

mapfile -t PCAPS < <(find "$DIR" -type f \(-iname ".pcap" -o -iname ".pcapng" \) 2>/dev/null || true)

Use find to look recursively under $DIR for files that end with .pcap or .pcapng
mapfile -t PCAPS : read all found paths into the bash array PCAPS
2>/dev/null : hide error messages (e.g., permission denied)
|| true : avoid failing the whole function if find exits non zero

if ["${#PCAPS[@]}" -eq 0] && command -v bulk_extractor >/dev/null 2>&1; then

Starts a conditional block. It checks two things: first whether the PCAPS array is empty (no PCAP files found yet) and second whether the bulk_extractor binary exists in the system PATH. Only if both conditions are true will the script attempt to carve network traffic from the memory image using bulk_extractor.

info "No PCAPs yet. Running bulk_extractor -E net,tcp ..."

Prints an informational message to the user explaining that no PCAP files were initially found so the script will now run bulk_extractor with the net tcp module to try to generate network captures from raw memory.

rm -rf "$NET_DIR"; mkdir -p "$NET_DIR"

Ensures a clean network output folder. First it force removes any existing NET_DIR if it exists then recreates the directory. This avoids mixing old results with new ones and guarantees that everything inside NET_DIR belongs to the current run.

bulk_extractor -E net,tcp -o "$NET_DIR" "$MEM" >/dev/null 2>&1 || true

This line runs bulk_extractor on the memory image stored in "$MEM" and tells it to focus only on network related artefacts. The -E net tcp option enables just the net and tcp scanners instead of running every scanner so the tool carves out things like TCP flows and other network. The -o "$NET_DIR" option sends all of these network results into the dedicated output directory NET_DIR which the script created earlier specifically for PCAP/PCAPNG files. Redirecting output with >/dev/null 2>&1 hides both normal output and error messages so the terminal stays clean and the script only prints its own formatted messages. Finally || true makes sure that even if bulk_extractor returns a non zero exit code for example if no network data is found or there is a minor parsing issue the function does not crash the command is treated as successful and the script continues to the next step of the analysis.

```
   mapfile -t PCAPS < <(find "$NET_DIR" -type f \( -iname ".pcap" -o
-iname ".pcapng" \) 2>/dev/null || true)
```

This command uses find to look inside "$NET_DIR" for any files that look like captured network traffic files whose names end with .pcap or .pcapng. The find results are fed directly into mapfile -t PCAPS via process substitution so each matching file path becomes a separate element inside the Bash array PCAPS. That means after this line PCAPS acts as an in memory list of all network capture files that belong to the current run and can be used later to print summaries or open them in Wireshark. Redirecting errors with 2>/dev/null hides noisy permission or file not found messages and || true again prevents a failure in find from aborting the whole function. The net effect is that the script quietly discovers all relevant PCAP/PCAPNG outputs created by bulk_extractor and stores them in a convenient array for the rest of the analysis logic.

```
 if [ "${#PCAPS[@]}" -eq 0 ]; then
```

Starts a second conditional block. It checks whether the PCAPS array is still empty even after running bulk_extractor. If there are still no PCAP files the function will only print a warning and exit gracefully.

```
warn "Network traffic not found."
```

Uses the warn helper to print a clear warning that no network traffic captures were detected or carved. This communicates to the investigator that there is no PCAP file to review for this memory image.

```
return 0
```

Exits the PCAP function early with a success status code 0. The absence of network data is treated as a valid outcome not as an execution error.

```
for x in "${PCAPS[@]}"; do
```

Starts a loop that iterates over each path stored in the PCAPS array. Each iteration processes one captured PCAP/PCAPNG file allowing the script to summarize them one by one.

```
 size="$(ls -lh "$x" | awk '{print $5}')"
```

For each capture file x runs ls -lh to show it in human readable format and pipes the output to awk to extract only the size column (e.g., 102K, 1.5M). The result is saved in the variable size which will later be printed alongside the file path.

```
 echo "File Location:$x  File Size $size"
```

Prints a one line summary for the current capture including its full path and human readable size. This gives the examiner a quick view of where the PCAP file is stored and how large it is.

```
 echo "PCAP summary"; date
```

Prints a header line PCAP summary and then calls date to output the current date and time. These two commands mark the beginning of a more formal summary section useful when the output is redirected into a report file.

```
for x in "${PCAPS[@]}"; do
```

Starts a second loop over the same list of PCAP files this time preparing lines that will go into a summary text file rather than just printing a formatted sentence.

```
size="$(ls -lh "$x" | awk '{print $5}')"
```

Again calculates the human readable size for each capture file exactly as in the previous loop so that the summary file contains the same size information that the user saw in the interactive output.

```
echo "$size  $x"
```

For each file prints a simpler line that starts with the size and then the file path. This format is convenient for quick sorting grepping or further processing of the summary file.

```
> "$DIR/pcap_summary.txt"
```

Redirects all lines produced inside the second for loop into the file pcap_summary.txt under the data directory. If the file already exists it is overwritten ensuring that each run creates a fresh summary aligned with the current memory analysis.

```
info "Saved summary -> $DIR/pcap_summary.txt"
```

Uses the info function to inform the user exactly where the PCAP summary file was saved. This message is helpful when building the PDF report because it points directly to the text file that should be included as evidence.

x="$(find "$DIR" -type f -name 'packets.pcap' -print -quit)"

Uses find to search under DIR for the first regular file named exactly packets.pcap. The -print -quit combination makes find stop after the first match and the resulting path is stored in the variable x. If nothing is found x remains empty.

if [-n "$x"]; then

Begins a conditional block that checks whether x is non empty. A non empty value means a packets.pcap file was successfully located.

y="$(ls -lh "$x" | awk '{print $5}')"

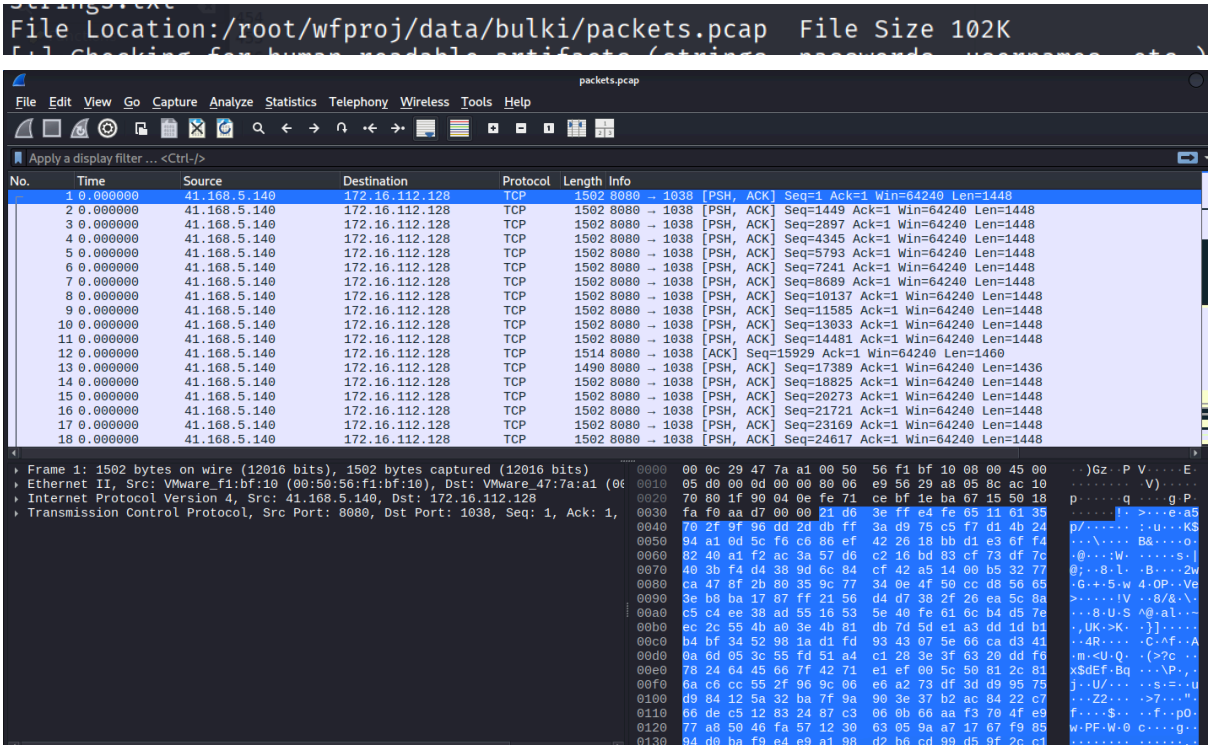
When the file exists this command retrieves its human readable size in the same way as before ls -lh shows the file and awk '{print $5}' extracts only the size column, The result is stored in y.

echo "File Location:$x File Size $y"

Prints a concise summary line that shows both the path and the size of packets.pcap. This gives the examiner exactly the information needed to document where the network traffic capture is stored and how big it is.

echo "[!] Network traffic not found under: $DIR"

Prints a clear warning message to the terminal stating that no packets.pcap file was found anywhere under DIR. This informs the examiner that for this run there is no captured network traffic file available.

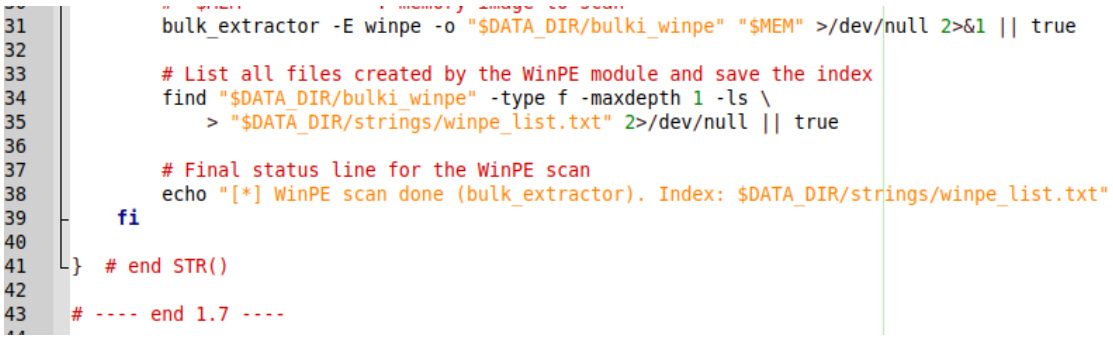


הפלט

The output shown in this screenshot represents the successful final step of task 1.6 Attempt to extract network traffic (PCAP). Reports its size as 102 KB (File Size 102K). After the script carved the memory image memdump.mem and stored network related artifacts under /root/wfproj/data/bulki/ I opened the file packets.pcap in Wireshark to verify that it contains real captured traffic. In the upper table Wireshark displays a chronological list of TCP packets including the Time column capture timestamp for each frame Source IP address Destination IP address Protocol (TCP) Length in bytes and detailed Info such as TCP flags (PSH ACK) sequence and acknowledgment numbers and window size. I can clearly see an ongoing TCP conversation between the external host 41.168.5.140 and the internal host 172.16.112.128 with source port 8080 and destination port 1038 which indicates an application level session that was captured from the memory image. The middle pane shows the decoded protocol layers for the selected frame (Ethernet II IPv4 TCP) with extra context about ports and flags and the bottom pane shows the raw hex dump and ASCII representation of the packet payload exactly as it appears on the wire. This view proves that the script not only detected network traffic but also successfully reconstructed a valid PCAP file that can be analyzed using standard forensic tools. For an investigator this means I can now follow the session packet by packet reconstruct connections inspect payloads and potentially identify commands malware communication or data exfiltration that passed through this memory image.

```
458  # ----- 1.7 Check for human-readable (exe files, passwords, usernames, etc.) ----
459 STR() {
460
461     # Print status message for the analyst
462     echo "[*] Checking for human-readable artifacts (strings, passwords, usernames, etc.) ..."
463
464     # Create an output directory for all string-based results.
465     # -p  : create parent directories if needed
466     # 2>/dev/null : hide 'already exists' warnings
467     # || true : never fail the script if mkdir returns an error
468     mkdir -p "$DATA_DIR/strings" 2>/dev/null || true
469
470     # Define a list of interesting keywords to search for in the memory image.
471     # These are typical indicators for credentials or sensitive data.
472     local TERMS=(exe password username http https dll login pass cred token apikey)
473
474     # Loop over each keyword and create a dedicated output file for it
475     for term in "${TERMS[@]}"; do
476
477         # Build the output filename for this specific keyword
478         out="$DATA_DIR/strings/${term}.txt"
479
480         # Extract all printable strings from the memory image and search for the keyword.
481         # strings -a      : scan the whole file (not only text sections)
482         # -n 4            : only strings with minimum length 4 characters
483         # "$MEM"          : the memory dump file to analyze
484         # 2>/dev/null     : hide warnings/errors from strings
485         # grep -Ei        : case-insensitive search using extended regex
486         # "$term"         : the current keyword we are looking for
487         # > "$out"        : save all matching lines to the keyword file
488         # || true         : do not stop the script if grep finds nothing
489         strings -a -n 4 "$MEM" 2>/dev/null \
490             | grep -Ei "$term" > "$out" || true
491
492         # If the output file is non-empty, report it to the analyst
493         # -s : test 'file exists and size > 0'
494         if [ -s "$out" ]; then
```

```
493         # -s : test 'file exists and size > 0'
494         if [ -s "$out" ]; then
495
496             # Get human-readable size of the file (e.g. 4K, 12K, 1.3M)
497             # ls -lh    : long listing, human readable
498             # awk '{print $5}' : take only the size column
499             sz=$(ls -lh "$out" | awk '{print $5}')
500
501             # Print a short status line with keyword, file path and size
502             echo "[*] Found '$term' -> $out ($sz)"
503         fi
504     done
505
506     # Create a global summary file with ANY of the important patterns.
507     # This lets the analyst review everything in one place.
508     # The regex covers: exe, password, username, login, user, http/https, dll, token, apikey, cred
509     strings -a -n 4 "$MEM" 2>/dev/null \
510         | grep -Ei 'exe|password|username|login|user[^A-Za-z]|http|https|\.dll|token|apikey|cred' \
511         > "$DATA_DIR/strings/all_hits.txt" || true
512
513     # If the summary file has content, notify the user where it is stored
514     if [ -s "$DATA_DIR/strings/all_hits.txt" ]; then
515         echo "[*] Summary saved: $DATA_DIR/strings/all_hits.txt"
516     else
517         # Otherwise, mention that nothing matched our patterns
518         echo "[!] No human-readable hits found (with current patterns)."
519     fi
520
521     # (Optional) WinPE scan using bulk_extractor this is still part of 1.7,
522     # but focused on WinPE artifacts, registry hives, DLL paths, etc.
523     if command -v bulk_extractor >/dev/null 2>&1; then
524         # Create output directory for WinPE module results
525         mkdir -p "$DATA_DIR/bulki_winpe" 2>/dev/null || true
526
527         # Run bulk_extractor with the winpe module
528         # -E winpe          : enable WinPE extractor
529         # -o <dir>          : output directory
530         # "$MEM"            : memory image to scan
```



```
        bulk_extractor -E winpe -o "$DATA_DIR/bulki_winpe" "$MEM" >/dev/null 2>&1 || true

        # List all files created by the WinPE module and save the index
        find "$DATA_DIR/bulki_winpe" -type f -maxdepth 1 -ls \
            > "$DATA_DIR/strings/winpe_list.txt" 2>/dev/null || true

        # Final status line for the WinPE scan
        echo "[*] WinPE scan done (bulk_extractor). Index: $DATA_DIR/strings/winpe_list.txt"
    fi

}  # end STR()

# ---- end 1.7 ----
```

echo "[*] Checking for human-readable artifacts (strings, passwords, usernames, etc.) ..."

Prints a status message to the terminal so the analyst understands that the script is starting the phase of searching for human readable artifacts. This improves usability and makes it clear which logical step is currently executing.

mkdir -p "$DATA_DIR/strings" 2>/dev/null || true

Create an output directory for all string-based results.
-p create parent directories if needed.
2>/dev/null hide already exists warnings.
|| true never fail the script if mkdir returns an error

local TERMS=(exe password username http https dll login pass cred token apikey)

Declares a local Bash array named TERMS that contains a curated list of keywords typical for credentials and sensitive information (executables passwords usernames HTTP/HTTPS traffic login terms tokens API keys). The local keyword restricts the variable to this function so it does not leak into the global scope

for term in "${TERMS[@]}"; do

Starts a for loop that iterates over each keyword stored in the TERMS array. For every term the script will create a dedicated output file and search the memory dump for strings containing that keyword.

out="$DATA_DIR/strings/${term}.txt"

Builds the output file path for the current keyword and saves it into the variable out. Each keyword gets its own text file for example password.txt, username.txt which helps the analyst quickly navigate to specific types of findings.

```
strings -a -n 4 "$MEM" 2>/dev/null \
            | grep -Ei "$term" > "$out" || true
```

Runs strings on the whole memory image (-a) and keeps only strings of at least 4 characters (-n 4). The output is piped to grep -Ei which searches case insensitively for the current keyword $term. All matching lines are written into the per keyword file $out. Errors are hidden.

```
if [ -s "$out" ]; then
```

Starts a conditional block that checks whether the output file for the current keyword exists and is non empty. The -s test is crucial so I only report keywords that actually produced at least one hit in the memory image.

```
sz=$(ls -lh "$out" | awk '{print $5}')
```

Uses ls -lh to get a human readable listing of the output file and pipes it to awk to extract only the size column for example 4K, 12K, 1.3M. The resulting size string is stored in the variable sz which will later be displayed in the status message for the analyst.

```
 echo "[*] Found '$term' -> $out ($sz)"
```

Prints a concise status line indicating that the current keyword was found in the memory image shows the path to the keyword specific output file and includes the file size stored in sz. This gives the analyst an immediate overview of which terms produced results and how large each result set is.

```
 strings -a -n 4 "$MEM" 2>/dev/null \
        | grep -Ei
'exe|password|username|login|user[^A-Za-z]|http|https|\.dll|token|apik
ey|cred' \
        > "$DATA_DIR/strings/all_hits.txt" || true
```

Similar strings scan on the whole memory image but this time all output is filtered by a single long grep regular expression that looks for many sensitive patterns: executables passwords usernames logins HTTP/HTTPS URL DLL tokens API keys and credentials. Every hit from any of these patterns is collected into one summary file all_hits.txt Errors are suppressed and failures don’t stop the script.

```
if [ -s "$DATA_DIR/strings/all_hits.txt" ]; then
```

Checks whether the global summary file all_hits.txt exists and contains at least one matching line. This condition determines if the script should report the summary file as a useful artifact or indicate that nothing relevant was detected.

```
echo "[*] Summary saved: $DATA_DIR/strings/all_hits.txt"
```

When the summary file is non empty this message informs the analyst exactly where the consolidated results are stored. The path helps them open the file directly and begin reviewing all hits in one place.

```
echo "[!] No human-readable hits found (with current patterns)."
```

If the summary file is empty this line clearly states that no human readable artifacts matching the defined patterns were detected. This explicit message prevents confusion and makes it obvious that the lack of results is itself a meaningful outcome.

```
if command -v bulk_extractor >/dev/null 2>&1; then
```

Starts a conditional block that only runs if the bulk_extractor tool is installed and available in the system PATH. The command -v check is redirected to /dev/null to suppress output and the script uses its exit status to decide whether to perform the additional WinPE scan.

```
  mkdir -p "$DATA_DIR/bulki_winpe" 2>/dev/null || true
```

Creates the directory $DATA_DIR/bulki_winpe and parents if it does not already exist. Any directory already exists warnings are hidden.

```
 bulk_extractor -E winpe -o "$DATA_DIR/bulki_winpe" "$MEM" >/dev/null
2>&1 || true
```

Runs bulk_extractor on the memory image $MEM with only the winpe module enabled -E winpe. All extracted WinPE related artifacts are written under the output directory $DATA_DIR/bulki_winpe. Both standard output and errors are redirected to /dev/null

```
 find "$DATA_DIR/bulki_winpe" -type f -maxdepth 1 -ls \
            > "$DATA_DIR/strings/winpe_list.txt" 2>/dev/null || true
```

Uses find to list all regular files (-type f) directly inside $DATA_DIR/bulki_winpe. The detailed listing (-ls) is written to winpe_list.txt which acts as an index of all WinPE artifacts created by bulk_extractor. Errors are hidden and failures don’t stop the script

echo "[*] WinPE scan done (bulk_extractor). Index: $DATA_DIR/strings/winpe_list.txt"

Prints a final status message indicating that the WinPE scan has completed successfully and points the analyst to the index file that summarizes all WinPE related outputs. This gives a clear end marker for this phase and an exact path to continue the investigation.

```
[*] Checking for human-readable artifacts (strings, passwords, usernames, etc.) ...
[*] Found 'exe' → /root/wfproj/data/strings/exe.txt (50K)
[*] Found 'password' → /root/wfproj/data/strings/password.txt (14K)
[*] Found 'username' → /root/wfproj/data/strings/username.txt (5.2K)
[*] Found 'http' → /root/wfproj/data/strings/http.txt (101K)
[*] Found 'https' → /root/wfproj/data/strings/https.txt (78K)
[*] Found 'dll' → /root/wfproj/data/strings/dll.txt (187K)
[*] Found 'login' → /root/wfproj/data/strings/login.txt (7.0K)
[*] Found 'pass' → /root/wfproj/data/strings/pass.txt (22K)
[*] Found 'cred' → /root/wfproj/data/strings/cred.txt (16K)
[*] Found 'token' → /root/wfproj/data/strings/token.txt (29K)
[*] Summary saved: /root/wfproj/data/strings/all_hits.txt
[*] WinPE scan done (bulk_extractor). Index: /root/wfproj/data/strings/winpe_list.txt
```

```
┌──(root㉿kali)-[~/wfproj/data/strings]
└─# ls
all_hits.txt  cred.txt  exe.txt    http.txt   pass.txt      token.txt     winpe_list.txt
apikey.txt    dll.txt   https.txt  login.txt  password.txt  username.txt
```

הפלט

Check for human readable artifacts shows that the script successfully scanned the memory image for sensitive keywords and generated dedicated result files for each term. For every keyword for example exe: password, username, http, https, dll, login, pass, cred, token. The script ran strings on the memdump.mem file and then filtered the results with grep saving all matching lines into a separate text file under /root/wfproj/data/strings/. The log confirms each hit and also shows the size of every file for instance exe.txt (50K) password.txt (14K) username.txt (5.2K) http.txt (101K) dll.txt (187K) token.txt (29K) which gives the analyst an immediate idea of how much evidence was found for each category. After finishing all keywords the script creates a global summary file called all_hits.txt that aggregates every match in one place and additionally performs an optional WinPE focused scan whose index is stored in winpe_list.txt. The ls output at the bottom confirms that all of these result files actually exist in the strings directory meaning that from this single memory dump I managed to extract a rich collection of human readable indicators such as possible passwords usernames API keys tokens URLs and executable references that can be reviewed in the next analysis phase.

```
546  VOL_BIN="/home/kali/wjproj/volatility_2.5.linux.standalone/volatility_2.5_linux_x64"
547
548  # Path to the Volatility binary (stand-alone Linux version)
549
550
551  # 2.1 Check if the file can be analyzed in Volatility; if yes, run Volatility.
552  function VOL()
553  {
554      echo "[+] Checking if the memory file can be analyzed with Volatility..."
555      # Inform the user that we are about to verify the memory image with Volatility
556
557
558      if "$VOL_BIN" -f "$MEM" imageinfo 2>/dev/null | grep -q "No suggestion"; then
559       # Run 'imageinfo' on the memory file.
560      # If Volatility prints 'No suggestion', it means the file is NOT a valid memory image.
561
562          echo "[!] This is not a memory file"
563           # Negative result: this is not a supported memory image
564
565          return 1
566          # Return non-zero so the caller knows the check failed
567
568      else
569          echo "[+] This is a memory file"
570          # Positive result: Volatility was able to analyze the file
571
572          return 0
573          # Return success (0) so the script can continue
574
575      fi
576  }
577
```

VOL_BIN="/home/kali/wjproj/volatility_2.5.linux.standalone/volatility_2.5_linux_x64"

This line defines a global variable VOL_BIN that stores the full path to the Volatility executable. By saving the path in a variable the script can call Volatility later using $VOL_BIN instead of writing the long path every time. This also makes the code easier to maintain if the Volatility binary moves to another location I only need to update this single line.

echo "[+] Checking if the memory file can be analyzed with Volatility..."

Prints a status message to the terminal indicating that the script is about to verify the memory image using Volatility. This helps the analyst understand what stage of the workflow is currently running and provides context before any technical checks are executed.

if "$VOL_BIN" -f "$MEM" imageinfo 2>/dev/null | grep -q "No suggestion"; then

This if statement runs Volatility with the imageinfo plugin against the memory file stored in $MEM using the Volatility binary pointed to by $VOL_BIN. Any error messages are redirected to /dev/null so they do not clutter the screen. The output is piped into grep -q No suggestion if Volatility prints No suggestion it means it could not identify a suitable profile so the condition evaluates to true and the script treats the file as an invalid memory image.

echo "[!] This is not a memory file"

Executed when the if condition is true this line prints a clear warning message telling the user that the provided input is not recognized as a valid memory image by Volatility. It prevents the analyst from continuing with incorrect assumptions about the file type.

return 1

This returns an exit status of 1 from the VOL function signaling a failure or negative result. Other parts of the script can check this return code to decide not to proceed with further Volatility analysis when the memory image is invalid.

echo "[+] This is a memory file"

In the successful case this line prints a positive confirmation to the user that the memory file is valid for analysis. It documents that the pre check passed and that subsequent Volatility commands may safely run on this input.

return 0

Returns an exit status of 0 from the VOL function indicating success. Other parts of the script can test this value to know that the file passed validation

```
[+] Checking if the memory file can be analyzed with Volatility...
[+] This is a memory file
```

הפלט

The output clearly shows that the Volatility pre check completed successfully and that the memory image I provided is valid for analysis. The first line “[+] Checking if the memory file can be analyzed with Volatility…” indicates that the script has invoked the VOL() function and is running imageinfo against the file in order to verify that Volatility recognizes its structure. The second line “[+] This is a memory file” confirms that Volatility did not return the No suggestion error and was able to identify the file as a legitimate RAM dump meaning the header size and internal layout match one of the supported memory formats. Practically this tells the analyst that the tool chain is configured correctly the path to memdump.mem is valid and it is safe to continue to the next forensic steps profile detection processes connections because Volatility will be able to parse this image without corruption or format issues.

```
577
578  # 2.2 Find the memory profile and save it into a variable.
579  function VOL_PROFILE() {
580
581      # Let the user know we are determining the correct Volatility profile
582      echo "[+] Detecting Volatility profile..."
583
584      VOL_PROFILE=$(
585          "$VOL_BIN" -f "$MEM" imageinfo 2>/dev/null |  # Run imageinfo quietly on the memory file
586          grep "Suggested Profile"                   |  # Filter the line that contains the suggested profiles
587          awk '{print $4}'                           |  # Extract the 4th field (first suggested profile)
588          tr -d ','                                     # Remove any trailing comma from the profile name
589      )
590      # Now VOL_PROFILE holds something like: WinXPSP2x86
591
592     # Display the detected profile to the analyst
593  echo "[+] Detected Volatility profile: $VOL_PROFILE"
594
595  # Save the profile into a file under $DATA_DIR
596  echo "$VOL_PROFILE" > "$DATA_DIR/vol_profile.txt"
597
598  }
```

echo "[+] Detecting Volatility profile..."

Prints a clear status message to the screen telling the analyst that the script is now starting the process of detecting which Volatility profile matches the given memory dump. This makes the script's progress transparent and easier to follow.

VOL_PROFILE=$("$VOL_BIN" -f "$MEM" imageinfo 2>/dev/null | grep "Suggested Profile" | awk '{print $4}' | tr -d ',')

Runs a full command pipeline and stores the final result in the variable VOL_PROFILE. First, $VOL_BIN -f "$MEM" imageinfo runs Volatility imageinfo on the memory file and prints information about possible profiles. Any error messages are hidden by redirecting stderr to /dev/null. The output is then piped to grep Suggested Profile to keep only the line that lists the suggested profiles. That line is piped to awk '{print $4}' which extracts the fourth field i.e. the first suggested profile name. Finally tr -d ',' removes any trailing comma from the profile name. The cleaned profile string is captured by the $() and saved into VOL_PROFILE.

echo "[+] Detected Volatility profile: $VOL_PROFILE"

Prints the detected profile back to the terminal so the analyst can immediately see which Volatility profile will be used for all further analysis. This serves both as confirmation that detection worked and as documentation inside the terminal output.

echo "$VOL_PROFILE" > "$DATA_DIR/vol_profile.txt"

Writes the detected profile string into a text file named vol_profile.txt inside $DATA_DIR. Using > overwrites any existing file so that the file always contains only the most recent profile. This file can later be read by other parts of the script or by the analyst to reuse the exact same profile without running detection again.

```
[+] This is a memory file
[+] Detecting Volatility profile...
[+] Detected Volatility profile: WinXPSP2x86
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# cat vol_profile.txt
WinXPSP2x86
```

הפלט

In this output I can see that my script successfully identified the correct Volatility profile for the memory image and saved it for later use.
First the script prints the status line "[+] Detecting Volatility profile…" which tells that it is running imageinfo against the memory file in the background. Then I immediately get "[+] Detected Volatility profile: WinXPSP2x86" which confirms that Volatility was able to analyze the dump and that the best matching profile is WinXPSP2x86. To make this result reusable in the next stages of the project the script writes the profile string into the file vol_profile.txt under my project data directory. When I run cat vol_profile.txt I see the same value WinXPSP2x86 which proves that the detection worked correctly and that I have now a persistent record of the profile that Volatility commands can load automatically without type it again.

```
  # ------------------------------------------------------
# 2.3 Display running processes using Volatility
# ------------------------------------------------------
function VOL_PROCS() {
    echo "[*] Extracting running processes ..."
     # Inform the analyst that process extraction is starting


# Run Volatility with:
    #   $VOL_BIN       Path to the Volatility standalone executable
    #   -f "$MEM"     The memory image file specified earlier
    #   --profile      The OS memory profile previously detected
    #   pslist        Volatility plugin that lists active processes
    #
    # Redirect output:
    #   >    Write output to vol_processes.txt
    #   2>&1  Redirect errors to the same file



    "$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" pslist \
        > "$DATA_DIR/vol_processes.txt" 2>&1

      # Notify the user where the output file was saved
    echo "[+] Saved running processes to: $DATA_DIR/vol_processes.txt"
}

```

echo "[*] Extracting running processes ..."

Prints a status message to the terminal indicating that the script is starting the phase where it will extract the running processes from the memory image. This is purely for the analyst visibility and does not affect the actual analysis logic.

**"$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" pslist **
> "$DATA_DIR/vol_processes.txt" 2>&1

Runs the Volatility binary stored in VOL_BIN against the memory image stored in MEM. The --profile="$VOL_PROFILE" option tells Volatility which operating system profile to use (WinXPSP2x86) and this profile was previously detected in section 2.2. The pslist plugin is then executed to enumerate all processes that were running in the captured memory. Standard output is redirected into the file vol_processes.txt inside DATA_DIR and 2>&1 ensures that any error messages are also written into the same file so the analyst has a complete log of the process listing operation.

```
echo "[+] Saved running processes to: $DATA_DIR/vol_processes.txt"
```

After Volatility finishes this line prints a confirmation message showing exactly where the extracted process list has been saved. It helps the analyst quickly locate the results file without needing to remember the path.

```
[*] Extracting running processes ...
[+] Saved running processes to: /root/wfproj/data/vol_processes.txt
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# cat vol_processes.txt
Volatility Foundation Volatility Framework 2.5
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          Exit
---------- -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ ----
0x823c89c8 System                    4      0     53      240 ------      0
0x822f1020 smss.exe                368      4      3       19 ------      0 2012-07-22 02:42:31 UTC+0000
0x822a0598 csrss.exe               584    368      9      326      0      0 2012-07-22 02:42:32 UTC+0000
0x82298700 winlogon.exe            608    368     23      519      0      0 2012-07-22 02:42:32 UTC+0000
0x81e2ab28 services.exe            652    608     16      243      0      0 2012-07-22 02:42:32 UTC+0000
0x81e2a3b8 lsass.exe               664    608     24      330      0      0 2012-07-22 02:42:32 UTC+0000
0x82311360 svchost.exe             824    652     20      194      0      0 2012-07-22 02:42:33 UTC+0000
0x81e29ab8 svchost.exe             908    652      9      226      0      0 2012-07-22 02:42:33 UTC+0000
0x823001d0 svchost.exe            1004    652     64     1118      0      0 2012-07-22 02:42:33 UTC+0000
0x821dfda0 svchost.exe            1056    652      5       60      0      0 2012-07-22 02:42:33 UTC+0000
0x82295650 svchost.exe            1220    652     15      197      0      0 2012-07-22 02:42:35 UTC+0000
```

הפלט

In this output I can see that I successfully used my script to dump all running processes from the memory image with Volatility. The first two lines confirm that the Extracting running processes… step completed and that the results were saved into the file /root/wfproj/data/vol_processes.txt. When I open this file I get a full Volatility pslist table each row represents a process that was running at the time of the memory capture : System, smss.exe, csrss.exe, winlogon.exe, services.exe, lsass.exe and several svchost.exe instances. The columns show important forensic metadata such as the process ID (PID), the parent PID (PPID), number of threads, session ID and the exact UTC start time of each process. This proves that my script correctly reused the profile from step 2.2 ran Volatility on the memory dump and generated a structured process list that I can now review for suspicious or abnormal activity.

```
626
627   # ------------------------------------------------------
628 # 2.4 Display network connections
629 # ------------------------------------------------------
630 function VOL_NET() {
631
632      # Inform the analyst that network extraction is starting
633     echo "[*] Extracting network connections ..."
634
635     # For Windows XP memory images, Volatility uses:
636     #   connections lists active or recently closed TCP connections
637     #
638     # Parameters:
639     #   $VOL_BIN     Path to the Volatility executable
640     #   -f "$MEM"     The memory dump to analyze
641     #   --profile     The detected OS memory profile
642     #
643     # Redirect all output and errors into the output file
644
645
646
647     # For Windows XP memory dumps, use connections or connscan
648     "$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" connections \
649         > "$DATA_DIR/vol_network.txt" 2>&1
650
651           # Inform the user where the results were saved
652     echo "[+] Saved network connections to: $DATA_DIR/vol_network.txt"
653 }
654
```

`echo "[*] Extracting network connections ..."`

Prints a status message to the terminal to inform the analyst that the script is starting the network connections extraction phase. This is purely for visibility and logging so the user understands that Volatility is about to analyze TCP connections in the memory dump

`"$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" connections \`

Runs the Volatility executable stored in the variable VOL_BIN and tells it to analyze the memory image in MEM using the OS profile stored in VOL_PROFILE. The plugin connections is used which lists active or recently closed TCP connections from the captured memory. The trailing backslash means the command continues on the next line so the full Volatility command includes the following redirection part as well.

> "$DATA_DIR/vol_network.txt" 2>&1

Continues the previous Volatility command by redirecting its standard output into the file vol_network.txt inside DATA_DIR. The 2>&1 part redirects error messages (stderr) into the same file so both normal output and any warnings or errors are captured. As a result vol_network.txt becomes the complete report of all network connections Volatility found in the memory image.

echo "[+] Saved network connections to: $DATA_DIR/vol_network.txt"

Prints a final confirmation message indicating that the extraction has finished and specifying the exact path of the output file. This helps the analyst quickly locate vol_network.txt and know where the network connection results are stored for later review or inclusion in the report

```
[*] Extracting network connections  ...
[+] Saved network connections to: /root/wfproj/data/vol_network.txt
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# cat vol_network.txt
Volatility Foundation Volatility Framework 2.5
Offset(V)  Local Address             Remote Address            Pid
---------- ------------------------ ------------------------- ---
0x81e87620 172.16.112.128:1038      41.168.5.140:8080         1484
```

הפלט

In this output I can see that the Volatility connections plugin successfully extracted the network activity from my memory image. The script runs Volatility with the correct profile and saves the results into vol_network.txt. When I open this file I see a TCP connection where my local machine at 172.16.112.128:1038 is talking to the remote server 41.168.5.140:8080. Port 1038 is a high ephemeral client port that my system chose randomly for the outgoing connection and port 8080 on the remote side is a common alternative HTTP or proxy port. This tells me that at the time of the memory capture my machine had an active or recently closed TCP session to that remote host and I can use the PID column to link this network connection back to the specific process that created it.

```
657  # ------------------------------------------------------
658 # 2.5 Attempt to extract registry information
659 # ------------------------------------------------------
660
661 VOL_REG() {
662
663     echo "[*] Extracting registry information ..."
664
665     #  Dump the list of registry hives from the memory image
666     #    This creates vol_hives.txt with all hive paths and virtual addresses.
667     "$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" hivelist \
668         > "$DATA_DIR/vol_hives.txt" 2>/dev/null
669
670     echo "[+] Saved registry hive list to: $DATA_DIR/vol_hives.txt"
671
672     #  Use Volatility printkey on the SOFTWARE key
673     #    This is exactly what produces the output style I saw
674     #    Legend, Registry , Key name  Software, Last updated, Subkeys, Values.
675     "$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" printkey \
676         -K "Software" \
677         > "$DATA_DIR/vol_registry_info.txt" 2>/dev/null
678
679     echo "[+] Registry information saved to: $DATA_DIR/vol_registry_info.txt"
680 }
681
682
```

echo "[*] Extracting registry information ..."

Prints a status message to the screen so the analyst sees that the script is starting the registry extraction phase.

**"$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" hivelist **
> "$DATA_DIR/vol_hives.txt" 2>/dev/null

Runs Volatility ($VOL_BIN) on the memory image (-f "$MEM") using the detected profile (--profile="$VOL_PROFILE"). The hivelist plugin enumerates all registry hives in memory and their virtual addresses. The output is redirected to the file $DATA_DIR/vol_hives.txt while 2>/dev/null hides any error messages.

echo "[+] Saved registry hive list to: $DATA_DIR/vol_hives.txt"

Informs the user where the hive listing file was saved so it can be reviewed later if needed.

```
"$VOL_BIN" -f "$MEM" --profile="$VOL_PROFILE" printkey \
        -K "Software" \
        > "$DATA_DIR/vol_registry_info.txt" 2>/dev/null
```

Runs Volatility again on the same memory image and profile this time with the printkey plugin. The option -K "Software" tells Volatility to dump the contents of the Software registry hive : keys, subkeys, values. The detailed registry output is written to $DATA_DIR/vol_registry_info.txt and errors are again hidden with 2>/dev/null.

```
 echo "[+] Registry information saved to:
$DATA_DIR/vol_registry_info.txt"
```

Final status message confirming that the registry information was successfully extracted and saved and telling the analyst exactly which file contains the results.

```
[*] Extracting registry information ...
[+] Saved registry hive list to: /root/wfproj/data/vol_hives.txt
[+] Registry information saved to: /root/wfproj/data/vol_registry_info.txt
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# cat vol_registry_info.txt
Legend: (S) = Stable   (V) = Volatile

----------------------------
Registry: \Device\HarddiskVolume1\Documents and Settings\Robert\Local Settings\Application Data\Microsoft\Windo
ws\UsrClass.dat
Key name: Software (S)
Last updated: 2011-04-13 00:53:02 UTC+0000

Subkeys:
  (S) Microsoft

Values:
----------------------------
Registry: \Device\HarddiskVolume1\Documents and Settings\Robert\NTUSER.DAT
Key name: Software (S)
Last updated: 2012-07-22 02:42:35 UTC+0000

Subkeys:
  (S) Adobe
  (S) Intel
  (S) Martin Prikryl
  (S) Microsoft
  (S) Netscape
  (S) Policies
  (V) Classes

Values:
----------------------------
Registry: \Device\HarddiskVolume1\WINDOWS\system32\config\default
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# cat vol_hives.txt
Virtual    Physical   Name
---------- ---------- ----
0×e18e5b60 0×093f8b60 \Device\HarddiskVolume1\Documents and Settings\Robert\Local Settings\Application Data\Mic
rosoft\Windows\UsrClass.dat
0×e1a19b60 0×0a5a9b60 \Device\HarddiskVolume1\Documents and Settings\Robert\NTUSER.DAT
0×e18398d0 0×08a838d0 \Device\HarddiskVolume1\Documents and Settings\LocalService\Local Settings\Application Da
ta\Microsoft\Windows\UsrClass.dat
0×e18614d0 0×08e624d0 \Device\HarddiskVolume1\Documents and Settings\LocalService\NTUSER.DAT
0×e183bb60 0×08e2db60 \Device\HarddiskVolume1\Documents and Settings\NetworkService\Local Settings\Application
Data\Microsoft\Windows\UsrClass.dat
0×e17f2b60 0×08519b60 \Device\HarddiskVolume1\Documents and Settings\NetworkService\NTUSER.DAT
0×e1570510 0×07669510 \Device\HarddiskVolume1\WINDOWS\system32\config\software
0×e1571008 0×0777f008 \Device\HarddiskVolume1\WINDOWS\system32\config\default
0×e15709b8 0×076699b8 \Device\HarddiskVolume1\WINDOWS\system32\config\SECURITY
0×e15719e8 0×0777f9e8 \Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0×e13ba008 0×02e4b008 [no name]
0×e1035b60 0×02ac3b60 \Device\HarddiskVolume1\WINDOWS\system32\config\system
0×e102e008 0×02a7d008 [no name]
```

הפלט

In this output I confirm that the registry analysis stage of my script is working correctly and giving me two complementary views of the Windows registry that was captured in the memory image. First the script runs Volatility hivelist plugin and saves the results into vol_hives.txt. When I open this file I see a table of all registry hives that were found in memory each with its virtual address physical address and full NT path UsrClass.dat and NTUSER.DAT hives for different users and the core system hives under WINDOWS\system32\config\SOFTWARE system security sam and default. This gives me a low level map of where every hive lives in memory and which files on disk correspond to useful for understanding which user profiles and system components were loaded at the time of acquisition.
Next the script uses the detected memory profile and runs Volatility printkey against the Software key saving the detailed output into vol_registry_info.txt. In this file I see a human readable report that starts with a legend for stable/volatile entries then shows specific registry paths such as Robert UsrClass.dat and NTUSER.DAT with the key name Software (S) the Last updated timestamps and lists of Subkeys : Adobe, Intel, Microsoft, Netscape, Policies, Classes and Values pointing to locations like WINDOWS\system32\config\default. These two files prove that the script successfully extracted both the global hive list and focused logical information about installed software and user configuration which I can use to reconstruct user activity installed applications and potential evidence relevant to the investigation.

```
690 # 3. Results
691 # 3.1 Display general statistics (time of analysis, number of found files, etc.)
692 # 3.2 Save all the results into a report (name, files extracted, etc.)
693
694 RESULTS() {
695
696     echo "[*] Generating analysis summary ..."
697
698     # ------------------------------------------------------------
699     # Calculate how long the script has been running (in seconds)
700     # ------------------------------------------------------------
701     ANALYSIS_TIME="${SECONDS}"
702
703     # ------------------------------------------------------------
704     # Count how many files were created under the data directory
705     # ------------------------------------------------------------
706     TOTAL_FILES=$(find "$DATA_DIR" -type f | wc -l)
707
708     # ------------------------------------------------------------
709     # Write the full forensic report into the report file
710     # Everything inside { } goes ONLY to $REPORT (not printed)
711     # ------------------------------------------------------------
712     {
713         echo "================ Forensic Analysis Report ==================="
714         echo
715         echo "Date               : $(date)"          # Current date/time
716         echo "Memory image file  : $MEM"             # Input memory dump
717         echo "Data directory     : $DATA_DIR"        # Output data dir
718         echo
719         echo "Analysis time      : $ANALYSIS_TIME s" # Total run time
720         echo "Total files found  : $TOTAL_FILES"     # Files extracted
721         echo
722         echo "Subdirectories under data:"            # List dirs
723         find "$DATA_DIR" -maxdepth 1 -type d -printf "  %f\n"
724         echo
725         echo "Files per subdirectory:"               # Count per folder
726         for d in "$DATA_DIR"/*; do
727             [ -d "$d" ] || continue
```

```
727             [ -d "$d" ] || continue
728             count=$(find "$d" -type f | wc -l)
729             echo "  $(basename "$d") : $count files"
730         done
731         echo "================================================================="
732         echo
733     } > "$REPORT"   # Redirect everything above into report.txt
734
735     # ------------------------------------------------------------
736     # Print only the short summary to the screen
737     # ------------------------------------------------------------
738     echo "Analysis time      : $ANALYSIS_TIME s"
739     echo "Total files found  : $TOTAL_FILES"
740
741     # ------------------------------------------------------------
742     # Tell the user where the report was saved
743     # ------------------------------------------------------------
744     echo "[+] Report saved to: $REPORT"
745
746
```

`echo "[*] Generating analysis summary ..."`

Prints a status message to indicate that the script is now building the analysis summary.

`ANALYSIS_TIME="${SECONDS}"`

Stores in ANALYSIS_TIME the total number of seconds the script has been running, Bash builtin variable seconds.

TOTAL_FILES=$(find "$DATA_DIR" -type f | wc -l)

Uses find to locate all regular files under $DATA_DIR, pipes them to wc -l and saves the total count in TOTAL_FILES.

> "$REPORT"

Opens a block of multiple echo and loop commands all text produced inside the block is redirected into the report file whose path is stored in $REPORT.

echo "Date : $(date)"

Writes the current date and time into the report so the analysis run is timestamped.

echo "Memory image file : $MEM"

Records in the report which memory dump file was analyzed the path stored in MEM.

echo "Data directory : $DATA_DIR"

Documents the root directory where all extracted results were saved.

echo "Analysis time : $ANALYSIS_TIME s"

Adds to the report the total runtime in seconds of the entire script.

echo "Total files found : $TOTAL_FILES"

Writes how many output files were created under $DATA_DIR during the analysis.

echo "Subdirectories under data:"

This command prints the header line “Subdirectories under data:” into the report. It introduces the section that lists all first level folders inside $DATA_DIR helping the analyst quickly see which main result directories were created during the analysis.

```
 find "$DATA_DIR" -maxdepth 1 -type d -printf "  %f\n"
```

Lists all first level subdirectories inside the data directory and prints only their names into the report.

```
echo "Files per subdirectory:"
```

This command prints the header line “Files per subdirectory:” into the report. It marks the beginning of the section where the script loops over each subfolder under $DATA_DIR and for each one prints how many extracted files it contains.

```
 for d in "$DATA_DIR"/*; do
            [ -d "$d" ] || continue
```

Loops over each subdirectory under $DATA_DIR counts how many files are inside it and prints (directory name: 3183 files) into the report.

```
 count=$(find "$d" -type f | wc -l)
```

Inside the loop counts the number of files in the current subdirectory d and stores the result in count.

```
echo "  $(basename "$d") : $count files"
```

Prints a per folder summary line into the report.

```
echo "Analysis time      : $ANALYSIS_TIME s"
```

After the report file is finished prints the same analysis time to the screen as a short on screen summary.

```
echo "Total files found  : $TOTAL_FILES"
```

Shows on the screen how many files were generated in total during the run.

```
 echo "[+] Report saved to: $REPORT"
```

Final message confirming that the full forensic report was successfully saved and where it is located.

```
Analysis time       : 215 s
Total files found   : 4157
[+] Report saved to: /root/wfproj/data/report.txt
```

```
┌──(root㉿kali)-[~/wfproj/data]
└─# cat report.txt
═══════════ Forensic Analysis Report ═══════════

Date                : Wed Nov 19 12:07:52 PM EST 2025
Memory image file   : /home/kali/wjproj/memdump.mem
Data directory      : /root/wfproj/data

Analysis time       : 215 s
Total files found   : 4157

Subdirectories under data:
  data
  bulki_winpe
  bulki
  strings
  foremost
  scalpel

Files per subdirectory:
  bulki : 3183 files
  bulki_winpe : 781 files
  foremost : 172 files
  scalpel : 0 files
  strings : 13 files
═══════════════════════════════════════════════
```

הפלט

In this output I can see the final forensic analysis report that my script generated. The report confirms that the memory image /home/kali/wjproj/memdump.mem was analyzed and that all results were stored under /root/wfproj/data. The total analysis time was 215 seconds and during this run the script created 4,157 files in the data directory. The report also summarizes the structure of my results: the data, bulki_winpe, bulki, strings, foremost, and scalpel subdirectories, and for each of them it lists how many artifacts were recovered : 3,183 files under bulki, 781 under bulki_winpe, 172 under foremost, 13 under strings. This confirms that my automation not only ran all carving and Volatility steps successfully it also produced a clear high level summary that I can attach as an official forensic report.

```
746
747     # 3.3 Zip the extracted files and the report file
748     echo                                    # Print a blank line for readability
749     echo "[*] Creating ZIP archive with all extracted files and the report ..."
750
751     BASE_DIR=$(dirname "$DATA_DIR")         # Get the parent directory of DATA_DIR
752     ZIP_FILE="$BASE_DIR/wfproj_results.zip" # Full path/name of the ZIP archive to create
753
754     # Check if the zip command exists on this system
755     if command -v zip >/dev/null 2>&1; then
756         (
757             cd "$BASE_DIR" || exit 1        # Change into BASE_DIR or exit on failure
758
759             # Create a ZIP archive with:
760             #   the entire data directory
761             #   the final report file
762             zip -r "$(basename "$ZIP_FILE")" \
763                    "$(basename "$DATA_DIR")" \
764                    "$(basename "$REPORT")" >/dev/null 2>&1
765         )
766
767         echo "[+] ZIP archive saved to: $ZIP_FILE"  # Inform the user where the ZIP was saved
768     else
769         # If zip is not installed, notify the user and skip this step
770         echo "[!] 'zip' command not found. Skipping ZIP creation."
771     fi
772
773 }
774
```

echo "[*] Creating ZIP archive with all extracted files and the report ..."

Notifies the analyst that the script is starting the ZIP creation phase where all extracted data and the report will be compressed into a single archive.

BASE_DIR=$(dirname "$DATA_DIR")

Uses dirname to take the parent directory of DATA_DIR and stores it in BASE_DIR. This is the directory where the ZIP file will be created.

ZIP_FILE="$BASE_DIR/wfproj_results.zip"

Defines the full path and filename of the ZIP archive that will be generated named wfproj_results.zip inside BASE_DIR.

if command -v zip >/dev/null 2>&1; then

Checks whether the zip command exists on the system by trying to locate it in the PATH. All output and errors are discarded the if continues only if zip is found.

cd "$BASE_DIR" || exit 1

This command changes the current working directory to the path stored in the variable $BASE_DIR. If the cd command fails for any reason (for example, the directory does not exist or cannot be accessed), the || exit 1 part makes the script terminate immediately with exit code 1, indicating an error. This

ensures that the ZIP creation is done from the correct parent directory and prevents the script from continuing in a wrong location.

```
 zip -r "$(basename "$ZIP_FILE")" \
```

Starts the zip command with recursive mode -r. The name of the created archive is the basename of ZIP_FILE : wfproj_results.zip without the path.

```
"$(basename "$DATA_DIR")" \
                   "$(basename "$REPORT")" >/dev/null 2>&1
```

Tells zip which items to include: the data directory (containing all carved and analysis files) and the report file. Both are passed without their full paths because the current directory is already BASE_DIR. All normal output and errors from zip are silenced.

```
 echo "[+] ZIP archive saved to: $ZIP_FILE"
```

If the zip command succeeded this line informs the user where the final ZIP archive has been saved using the full path stored in ZIP_FILE.

```
 echo "[!] 'zip' command not found. Skipping ZIP creation."
```

Warns the analyst that no ZIP archive will be created because the zip utility is missing so they will need to collect the output directory manually.

```
[*] Creating ZIP archive with all extracted files and the report ...
[+] ZIP archive saved to: /root/wfproj/wfproj_results.zip
```

```
┌──(root㉿kali)-[~/wfproj]
└─# ls
data  wfproj_results.zip  ziGp9r38  ziJpnjXG  zirCDajY  zitCJXbB
```

```
┌──(root㉿kali)-[~/wfproj]
└─# cat wfproj_results.zip
PK
◆`s[data/UT     h◆ih◆iux
                P◆`s[◆8◆%`data/binwalk.txtUT    ◆◆i◆◆iux
                                                ◆◆Ao◆ ◆;◆◆œ&j(-v◆◆dMt▒5”]·◆◆◆l◆?◆◆´◆◆◆◆▯5Ą ◆_<◆◆
a◆◆i◆◆◆◆lB:◆◆9◆◆)◆◆h◆◆ q9◆\◆3◆◆:◆=◆▯ToF
◆C◆◆◆◆,◆i◆%◆3        ◆
◆◆n◆z◆
      ◆◆◆◆z]*ŋ◆◆p◆s◆◆lT◆◆θ◆V-◆B3.◆◆◆◆◆j◆◆R▯◆X◆◆◆◆s◆0◆◆;▒◆8◆%◆◆e◆]◆◆Si▯ ◆◆◆◆◆◆ɯ¿L◆◆q◆U◆◆◆◆c◆Y◆K*◆◆◆◆z<◆1<:`◆j◆◆!
B?P◆`s[◆▯ ⋆data/report.txtUT    h◆ih◆iux
                                ◆QAN◆0◆◆+◆$N◆R"◆◆◆!Z◆#r◆MqkŎ◆◆◆◆SÇ◆
                                                {{fwfeW◆m◆◆:j◆◆◆c+◆◆◆◆7◆◆
                                                                        X◆.◆o◆◆w◆◆◆
%'◆U2;g◆◆!S◆◆◆c◆9k◆h◆◆9'q$dh◆◆◆;.Keh◆'9z◆◆ ◆靈}[G◆^◆◆◆◆◆ر◆◆a4,'d◆◆◆◆>◆v4◆▒}p◆=x>5◆n◆◆ɾ◆i◆◆◆&◆◆<◆◆
                                                                        ◆\◆w◆◆`◆X◆◆2
◆3-◆◆B◆1◆^◆◆F◆sv◆◆◆◆]◆PK
◆`s[data/bulki_winpe/UT 9◆i◆iux
                        P◆`s[◆◆◆:◆ata/bulki_winpe/alerts.txtUT  =◆i9◆iux
                                                M◆Ok◆@◆◆ƒjx◆K{◆UKS◆XH▒◆A◆bB◆◆◆◆5
◆▯◆◆◆j◆◆◆◆v◆A^◆◆F◆◆v◆◆◆◆;◆T◆◆◆~:◆◆~◆Zeŏ._ʂz◆Ƒ"?◆C◆7◆◆◆ƀXK⊿N◆◆◆◆Qᓁ` _◆#◆f}G.g◆◆◆p◆◆◆◆◆◆_◆◆q◆▒
                                                                        ◆ ◆◆◆|bj&◆◆◆      ◆
(f,◆h@◆0z◆◆in\▒F◆S},◆-nzP◆`s[◆◆i◆>◆F]▒data/bulki_winpe/winpe.txtUT      >◆i9◆iux
                                                ◆]◆R◆}◆|◆◆◆◆`◆;◆I◆A=V▒!◆◆◆y!◆◆
M◆◆◆t◆|◆ɒĭ◆H◆Vm◆⋆%◆Ef◆Zk◆◆◆◆[{{◆◆◆O◆◆◆◆◆◆◆◆Aw◆kg◆◆◆◆◆o◆◆◆◆◆◆◆◆◆◆◆◆◆y◆◆Q◆_◆◆◆◆◆~◆◆◆xq◆◆l◆ɾ◆◆◆◆◆◆◆q◆t◆?◆bk◆◆◆◆ڰs|
◆◆i◆Z◆◆◆◆4n~◆'Û◆◆◆◆◆◆◆◆x▒~◆~◆◆◆◆◆+n?◆zDQ◆FX◆6Dl◆◆g)Qɷ%LIeZ◆@◆Z{w◆◆◆◆z◆O1◆◆X◆b◆◆◆g◆◆F◆K◆◆◆◆◆◆}im◆▯ :◆U{?N]◆◆O◆踹
|◆~`c◆◆◆4◆ʒ◆◆◆◆%J4p◆k◆:◆◆◆◆◆◆7◆Ao1◆b◆◆◆◆gW◆◆◆◆◆◆◆;?9◆◆$◆◆◆'◆◆_◆◆◆◆◆◆~u{2◆◆◆◆◆◆◆◆◆◆◆◆VÚW◆◆◆û;¿◆◆◆◆◆◆◆◆◆◆◀
◆◆◆X◆◆◆◆◆◆◆ڧ
t◆◆!◆◆wk◆◆◆ky◆;◆▯V◆6G~◆◆◆_LJw◆◆a◆◆,¿◆M◆◆◆F7◆?◆8◆◆⇒◆/◆s◆◆◆ʗ◆◆F.◆
     ◆◆◆◆:g◆W◆v◆◆7◆◆
```

הפלט

In this output I confirm that the final packaging step of my script works correctly. The script reports that it is creating a ZIP archive with all extracted files and the report and then clearly shows that the archive was saved as /root/wfproj/wfproj_results.zip. When I run ls in the project directory I can see this ZIP file listed next to the data folder which verifies that the archive was actually created on disk. Viewing the file with cat wfproj_results.zip shows a long stream of unreadable binary characters rather than plain text exactly what I expect from a compressed ZIP container. This proves that all my carved data Volatility outputs and the final report.txt have been bundled into a single forensic evidence package that I can move back up or submit as part of my Windows Forensics project.

```
 9
10  RUN_TS="$(date '+%Y%m%d_%H%M%S')"
11  RUN_LOG_DIR="$LOG_DIR/$RUN_TS"
12
13  mkdir -p "$RUN_LOG_DIR"
14
15  TOOLS_LOG="$RUN_LOG_DIR/tools.log"
16  FILES_LOG="$RUN_LOG_DIR/files_summary.log"
17  RUN_LOG="$RUN_LOG_DIR/run.log"
```

RUN_TS="$(date '+%Y%m%d_%H%M%S')"

Run timestamp. Runs the date command and saves the current date time (YYYY-MM-DD HH:MM:SS) into the variable RUN_TS.

RUN_LOG_DIR="$LOG_DIR/$RUN_TS"

Build the folder name for this run logs. Joins the main logs folder LOG_DIR with the timestamp RUN_TS so each run gets its own subdirectory.

mkdir -p "$RUN_LOG_DIR"

Create the logs directory for this run if it doesn't exist. The -p flag makes mkdir create parent directories as needed and not fail if the folder already exists.

TOOLS_LOG="$RUN_LOG_DIR/tools.log"

Define the full path of tools.log. This file will store which tools ran and when inside the run log directory.

FILES_LOG="$RUN_LOG_DIR/files_summary.log"

Define the full path of files_summary.log. This file will store the files summary counts per dir per extension for this run.

RUN_LOG="$RUN_LOG_DIR/run.log"

Define the full path of run.log. This file holds one summary line about the whole Windows Forensics run memory image data dir tools.

```
log_run_summary() {
  echo "[*] Creating run summary logs under: $RUN_LOG_DIR"

  local tools=("binwalk" "strings" "exiftool" "scalpel" "foremost" "bulk_extractor")

  printf "[%s] Windows Forensics run (memory=%s, data_dir=%s, tools=%s)\n" \
    "$(date '+%Y-%m-%d %H:%M:%S')" \
    "$MEM" \
    "$DATA_DIR" \
    "${tools[*]}" > "$RUN_LOG"

  for tool in "${tools[@]}"; do
    printf "[%s] %s executed (memory=%s, data_dir=%s)\n" \
      "$(date '+%Y-%m-%d %H:%M:%S')" \
      "$tool" \
      "$MEM" \
      "$DATA_DIR" >> "$TOOLS_LOG"
  done

  TS="$(date '+%Y-%m-%d %H:%M:%S')"

  {
    echo "================ Files summary for run at $TS ================"
    echo "Memory image : $MEM"
    echo "Data dir     : $DATA_DIR"
    echo

    TOTAL_FILES_RUN=$(find "$DATA_DIR" -type f | wc -l)
    echo "Total files found: $TOTAL_FILES_RUN"
    echo

    echo "Files per subdirectory:"
    for d in "$DATA_DIR"/*; do
      [ -d "$d" ] || continue
      count=$(find "$d" -type f | wc -l)
      printf "  %-15s : %s files\n" "$(basename "$d")" "$count"
    done
    echo
```

```
    find "$DATA_DIR" -type f | sed -n 's/.*\.\([A-Za-z0-9]\{1,\}\)$/\1/p' | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr | awk '{printf "  %-8s : %s fil

    echo "============================================================"
    echo
  } > "$FILES_LOG"

  echo "[+] Tools log saved to:  $TOOLS_LOG"
  echo "[+] Files summary saved to: $FILES_LOG"
  echo "[+] Run log saved to:    $RUN_LOG"
}
```

`echo "[*] Creating run summary logs under: $RUN_LOG_DIR"`

Prints text to the terminal or into a redirected file.

`local tools=("binwalk" "strings" "exiftool" "scalpel" "foremost" "bulk_extractor")`

Declares a variable that is local to the function, doesn't leak outside.

`printf "[%s] Windows Forensics run (memory=%s, data_dir=%s, tools=%s)\n"`

Writes a single run line into run.log: timestamp which memory image was used which data directory and which tools are included.

"$(date '+%Y-%m-%d %H:%M:%S')"

Generates a formatted timestamp 2025-11-21 03:33:50

"$MEM"

Inserts the full path of the memory image into the run line.

"$DATA_DIR"

Inserts the data directory path into the run line

"$DATA_DIR" >> "$TOOLS_LOG"

Expands the tools array into one space separated string : all tool names.

TS="$(date '+%Y-%m-%d %H:%M:%S')"

Saves the current timestamp into variable TS to use in the files summary header.

echo "================ Files summary for run at $TS ================"

Prints the header line of the files summary including the timestamp.

echo "Memory image : $MEM"

Logs which memory image was analyzed.

echo "Data dir   : $DATA_DIR"

Logs which data directory contains the extracted results.

```
 TOTAL_FILES_RUN=$(find "$DATA_DIR" -type f | wc -l)
```

Use find to list all files under $DATA_DIR (-type f) pipe to wc -l to count lines total number of files. Save the result into TOTAL_FILES_RUN.

```
echo "Total files found: $TOTAL_FILES_RUN"
```

Prints the total number of files found in the data directory tree.

```
echo "Files per subdirectory:"
```

Header for the files per folder section.

```
for d in "$DATA_DIR"/*; do
```

Start a loop over every item inside $DATA_DIR , each file or folder path is saved temporarily in variable d.

```
 [ -d "$d" ] || continue
```

Tests if "$d" is a directory (-d). If it is not a directory the test fails

```
 count=$(find "$d" -type f | wc -l)
```

For the current subdirectory d runs find to list all files under it counts them with wc -l and stores the result in the variable count.

```
 printf "  %-15s : %s files\n" "$(basename "$d")" "$count"
```

Uses printf for formatted output. %-15s prints the subdirectory name left aligned in a 15 character field (: %s files) prints the file count. basename "$d" strips the path and leaves only the directory name.

echo "Files per extension (file types):"

Prints a header for the section that will show statistics per file extension type.

find "$DATA_DIR" -type f

Gets a list of all files with full paths under DATA_DIR. This is the input for the following text processing commands.

sed -n 's/.*\.\([A-Za-z0-9]\{1,\}\)$/\1/p'

sed is used in silent mode (-n). For each line file path it runs a substitution, it looks for the last dot in the name captures the extension after it and replaces the whole line with just that extension. The p at the end tells sed to print only lines where the substitution succeeded. Extract only the file extension from each path

tr 'A-Z' 'a-z'

tr translates characters. Here it converts all uppercase letters A-Z to lowercase a-z so extensions like JPG and jpg are treated the same.

awk '{printf " %-8s : %s files\n", $2, $1}'

Uses awk to print each extension nicely extension in a field of 8 chars then count files. $2 is the extension $1 is the count from uniq -c.

} > "$FILES_LOG"

Closes the block that started with { . All the output produced inside the block all the echo printf is redirected into the file whose path is stored in $FILES_LOG instead of going to the screen.

echo "[+] Tools log saved to: $TOOLS_LOG"

Informs the user where the tools log file has been saved. $TOOLS_LOG should contain that path.

echo "[+] Files summary saved to: $FILES_LOG"

Prints a message telling where the files summary log was saved.

echo "[+] Run log saved to: $RUN_LOG"

Prints the path to the general run log file whose name is stored in $RUN_LOG.

```
[*] Creating run summary logs under: /20251121_132710
[+] Tools log saved to:  /20251121_132710/tools.log
[+] Files summary saved to: /20251121_132710/files_summary.log
[+] Run log saved to:    /20251121_132710/run.log

┌──(kali㉿kali)-[~/wjproj]
└─$ sudo cat /20251121_132710/files_summary.log
═══════════ Files summary for run at 2025-11-21 13:30:03 ═══════════
Memory image : /home/kali/wjproj/memdump.mem
Data dir     : /root/wfproj/data

Total files found: 4158

Files per subdirectory:
  bulki          : 3183 files
  bulki_winpe    : 781 files
  foremost       : 172 files
  scalpel        : 0 files
  strings        : 13 files

Files per extension (file types):
  winpe    : 1554 files
  exe      : 92 files
  txt      : 83 files
  dll      : 75 files
  mft      : 9 files
  indx     : 4 files
  xml      : 2 files
  wav      : 2 files
  bmp      : 2 files
  pcap     : 1 files
═══════════════════════════════════════════════════════════════
```

```
┌──(kali㉿kali)-[~/wjproj]
└─$ sudo cat /20251121_132710/tools.log
[2025-11-21 13:30:03] binwalk executed (memory=/home/kali/wjproj/memdump.mem, data_dir=/root/wfproj/data)
[2025-11-21 13:30:03] strings executed (memory=/home/kali/wjproj/memdump.mem, data_dir=/root/wfproj/data)
[2025-11-21 13:30:03] exiftool executed (memory=/home/kali/wjproj/memdump.mem, data_dir=/root/wfproj/data)
[2025-11-21 13:30:03] scalpel executed (memory=/home/kali/wjproj/memdump.mem, data_dir=/root/wfproj/data)
[2025-11-21 13:30:03] foremost executed (memory=/home/kali/wjproj/memdump.mem, data_dir=/root/wfproj/data)
[2025-11-21 13:30:03] bulk_extractor executed (memory=/home/kali/wjproj/memdump.mem, data_dir=/root/wfproj/data)
```

הפלט

I executed my forensic tools script on the memory image and configured it to save all output into a dedicated data directory and a set of log files. During the run the script chained together several analysis tools such as bulk extractor, foremost, scalpel and strings and automatically recorded every step into a tools log a files summary log and a general run log. The files_summary.log output shows that this toolchain generated a total of 4,158 artefact files all organized under structured subdirectories. The Files per subdirectory section demonstrates how much data each tool contributed with the bulk extractor runs producing the majority of artefacts and the other tools adding additional targeted results. The Files per extension file types section then classifies everything by file type revealing large numbers of WinPE related artefacts alongside executables text files DLLs file system metadata records mft and indx xml documents audio files images and even a network capture pcap. Overall this output proves that I successfully automated the forensic processing of the memory image generated detailed log files for traceability and obtained a rich set of artefacts of many different types for further analysis.