Windows Forensics Project

Student:Yaniv Juravliov
Project code:NX212
Student code:s26

Table Of Contents

Overview of the script

This Bash script functions as an **automated digital forensics** and **memory analysis framework** designed to streamline the examination of unknown files such as disk images, memory dumps, and raw binary artifacts. The script establishes a controlled and repeatable forensic environment by first validating execution privileges and ensuring all required analysis tools are present, automatically installing any missing dependencies to eliminate manual setup overhead. Once initialized, the script prompts the analyst for a target file, verifies its existence, and initiates a multi-layered extraction process that integrates several industry-standard forensic utilities, including Bulk Extractor, Binwalk, Foremost, and Strings, in order to maximize artifact recovery and evidence coverage.

During the extraction phase, the script systematically carves embedded files, extracts human-readable data, and identifies potential network artifacts such as packet capture files, enabling early detection of communication indicators commonly associated with malicious activity. Extracted strings are further analyzed to identify credentials, URLs, and IP addresses, providing immediate intelligence that supports threat hunting and incident response workflows. The script then performs file-type and signature analysis to determine whether the provided artifact is compatible with memory forensics, preventing unsupported or invalid analysis attempts that could lead to misleading results.

When a memory dump is detected, the script dynamically integrates the **Volatility 3** framework and automatically determines the operating system of the captured memory image, supporting **Windows**, **Linux**, and **macOS** environments. Based on the identified platform, the script safely executes a curated set of read-only Volatility plugins to extract critical runtime artifacts such as process listings, network connections, command execution history, and registry or kernel-level data. Each plugin execution is logged with explicit success or failure status, ensuring transparency, traceability, and reproducibility throughout the analysis process.

Following artifact extraction, the script compiles a structured **forensic report** that consolidates execution timestamps, file metadata, cryptographic hashes, extraction statistics, detected indicators, and memory analysis outcomes into a single authoritative document. To preserve evidentiary integrity and support chain-of-custody requirements, all extracted artifacts and reports are packaged into a compressed archive accompanied by a SHA256 hash. Overall, this script unifies environment preparation, automated forensic extraction, memory analysis, reporting, and evidence preservation into a single cohesive workflow, significantly reducing analyst workload while producing standardized, well-documented results suitable for security operations, digital forensics investigations, and professional reporting.

Breaking Down the script into sections

This script can be divided into 12 sections

1.Environment initialization & metadata

which includes the shebang declaration , ansi color variables , visual separators

2. Root user Validation

ensure the script is executed by a root account , which is convenient as a lot of tools require privileges and the script by itself will run smoothly

this includes the root user validation , safe termination if not run as root

3.Target File Acquisition & validation

Prompts the user to input the file , and the script will run a check if it exists and the path

4. Dependency Verification & Tool Installation

Guarantees all required tools and beyond are properly installed or if they already exist then notify about that , includes using an array of the tools , automatic installation if anything is missing .

5. Output Structure Initialization

prepares a clean organized workspace for all the extracted artifacts and logs

which are all going to be saved inside that directory

6. Multi-Tool Artifact Extraction (Carving Phase)

Extracting the file that the user prompted from the input by using various tools specifically the ones we are using are strings , binwalk , foremost , bulk extractor.

7. Network Artifact Discovery

Detect the existence of network traffic evidence if it was extracted by one of the tools,if it does exist it will output the path so the user seeing it can navigate if needed to the file

8. Human-Readable Artifact Analysis

identify any useful keyword that could be human readable to extract useful artifacts and evidence includes IP, URL , Credentials keywords such as user,pass,etc..

9. Volatility Installation

includes volatility installation and dependencies

10. Memory Dump Detection & Classification

determines if the file specified is suited for memory analysis using volatility3 to prevent unsupported or misleading analysis attempts

11. Memory Forensics Automation (Volatility)

Extracts artifacts from memory dumps in a safe , automated way . it includes the detection of the OS , execution of plugins.

12. Reporting, Packaging & Evidence Integrity

consolidate results and preserve forensic integrity its going to include statistics calculation , structured report text file , ZIP packaging of evidence and hashing of the final report zip archieve

Making of this Scripts

While developing this script, I intentionally chose to utilize AI assistance more than I normally would, not as a shortcut, but as a learning accelerator. The goal was to deepen my understanding of **Bash scripting, automation techniques, and defensive scripting practices**, while still maintaining full control over the logic, structure, and validation of the tool.

Throughout the development process, I focused on designing the script to be **automated, fault-tolerant, and reproducible**, rather than relying on manual execution or assumptions. This included learning how to declare and use associative arrays for dependency management, implementing conditional logic to prevent automation failures, and handling common edge cases such as missing files, unsupported artifacts, and non-critical command failures.

While AI assistance was used to propose ideas and initial structures, significant effort was spent **reviewing, correcting, and refining** the generated code. This process helped me better understand Bash behaviors, error handling patterns, and the importance of validating every command in an automated forensic workflow. Many small issues had to be identified and resolved to ensure the script behaved consistently and produced deterministic output.

Overall, this approach made the development process more efficient while significantly improving my understanding of Bash scripting and automation design. More importantly, it resulted in a reusable forensic framework that can be expanded in future projects and applied to real-world analysis scenarios, strengthening both my technical skills and my confidence in building reliable automated tools.

A few important notes when running the script

The most important one is the following:

1) The script uses **Volatility3**

Not all memory dumps are supported , Linux memory analysis requires a matching kernel and symbols , Some windows dumps may fail due to unsupported formats or corruption .

Vol3 mostly supports Win10,11,Server16,19,22 Versions

Partially supports win 8 and 8.1 , serv 2012 / r2

Win XP , Vista, Win 7 , win server 2008 / r2 are **NOT** supported this is

There are also some reliable plugins and some are less (mostly the registry related ones).

2) Input File location Restriction

The script searches only for the target file within the user's Desktop directory

`/home/$REAL_USER/Desktop` So make sure the file is located somewhere around it

Otherwise it will fail to find it and the script will break,

And also make sure no other file is named the same.

3) The Script must be executed as root

as simple as that otherwise it wont keep run and will display an error

Reason is , file carving tools , mem analysis and certain system utilities require elevated permissions. And it also makes the script easier to automate and prevent requirement to enter root password.

4) Operating System

The script is mostly for Debian based Distros such as (ubuntu , kali , parrot ,etc..)

Because it uses "apt" for package installation .

To sum it up

Make sure target file is located on the **Desktop** , run the script as **root** on a debian based system , and understand that **volatility 3** analysis is **subject** to mem dump **compatibility** and **OS support** limitations.

Breaking Down the scripts commands and sections

1.ANSI color section

Starting with section number 1

```
#============================================================================
# Ansi color variables because we love them for our eyes
#============================================================================
GREEN='\e[0;32m'
RED='\e[0;31m'
BLUE='\e[0;34m'
DARKGREY='\e[1;30m'
RESET='\e[0m'
LINE="============================================="

echo -e "${BLUE}${LINE}${RESET}"
```

Defining ANSI colors Variables
to make the output of the script clean to eye , wether its an error(red) or succesfull command execution (green) , a line that divides each section to understand exactly which part of the script is being executed (installation , memory and artifact extraction , and more)

To enable it , it is required to write echo with the -e flag in order to escape sequence properly otherwise it won't work
and we need to write ${THE COLOR} ..... sentence ..... ${RESET}
without the -e (escape) flag we would see something like this

```
#!/bin/bash
RED='\e[0;31m'
RESET='\e[0m'
echo "${RED}Hello world!${RESET}"
```

```
❯ bash newscript.sh
\e[0;31mHello world!\e[0m
```

But by applying -e , it will give us the following result:

```
❯ bash newscript.sh
Hello world!
```

The basic structure of an ANSI escape code is
\e[<style>;<color>m
\e – Escape character (ASCII ESC , 0x1B)
[– Start of ANSI sequence
<style> – Formatting (bold , normal, etc)
<color> – Foreground color
m – End of sequence

2.Root User Validation

```
REAL_USER="${SUDO_USER:-$USER}"
if [ $(whoami) != "root" ] ;
then
echo -e "${RED}Exiting ... run as root${RESET}"
    exit 1
    else
    echo -e "${GREEN}You're Root , Continuing...${RESET}"
fi
echo -e "${BLUE}${LINE}${RESET}"
```

This section ensures that the script is executed with as root to execute system-level operations such as installations of packages , accessing directories , running tools.. etc... , its good for the script to be automated as much as possible.
and it will **NOT** continue unless logged in as root.
Starting with `REAL_USER="${SUDO_USER:-$USER}"`
It determines the original non-root user who launched the script.
If the script is run using sudo the $SUDO_USER variable contains the username of the person who invoked sudo .
if the script run directly as root without sudo $SUDO_USER is empty , so it falls back to $USER.
Latter on the script creates a file on the user's desktop and changes ownership back to the real user. This prevents extracted files from being owned by root, which would be inconvenient . and also the files will be saved there and won't be visible to the user unless coping or accessing root in the CLI , another alternative could be promting the user executing the script to enter their username but I wanted to challenge myself and use another techniques and to make the script as less input as possible and more automated.
`if [ $(whoami) != "root" ] ;` a condition that if $(whoami) is **NOT** root
then it will exit and tell the user to run it as root.
with
```
echo -e "${RED}Exiting ... run as root${RESET}"
    exit 1
```
Otherwise it will continue and display succesful message in green that it is run as root.

```
else
echo -e "${GREEN}You're Root , Continuing...${RESET}"
```

So to sum up , a validation that the script is running with root privileges or as root user, safely identifies the original invoking user.
Stops executing and notifies if requirement isnt met.

3.Target File Acquisition & validation

```
read -p "Enter your file name to analyze:" FILENM
filepath=$(find /home/$REAL_USER/Desktop -type f -name "$FILENM" 2>/dev/null )
if [ -z "$filepath" ]; then
    echo -e "${RED}Your file doesn't exist,Please enter a valid file${RESET}"
    exit 1
else
    echo -e "${GREEN}Found required file at${RESET} ${RED}: "$filepath" ${RESET}"
    echo -e "${GREEN}Initiating analysis...${RESET}"
fi
echo -e "${BLUE}${LINE}${RESET}"
```

This section is responsible for collecting the target file from the user,verify that it exists,and ensures that the script won't continue without a valid input.

`read -p "Enter your file name to analyze:" FILENM`

Prompt the user to enter the file name , In the output it will look like the **following:**

`Enter your file name to analyze:m4.vmem` in my case I used m4.vmem for demonstration

Following by this

`filepath=$(find /home/$REAL_USER/Desktop -type f -name "$FILENM" 2>/dev/null )`

Which searches inside the **user**'s **desktop*** for a filename matching the input we gave it Stores the full path of the file incase its found inside the **filepath** for latter use.

find - Searches recursively

/home/$REAL_USER/Desktop – limits the search scope for safety and speed because depending on the environment if the system has a lot of user's it might take more time and even generate an error if there's another file matching the same name, so limiting it to the current user's Desktop in my opinion is the better choice.

-type f – ensures the type of search is specified for a file. F stands for file.

-name "$FILENM" matches the filename we provided the script with.

2>/dev/null – which is going to be seen a lot , suppresses error messages output.

`if [ -z "$filepath" ]; then` Checks wether the filepath variable is empty.

Which could potentially mean it doesn't exist and we will need to verify what we wrote.

If it doesn't exist , it will notify the user and close the script to make the user run it again correctly.

```
echo -e "${RED}Your file doesn't exist,Please enter a valid file${RESET}"
exit 1
```

If the file is found , will notify the user that it found the file includiing the file's **path** and proceeds further executing the script.

```
else
    echo -e "${GREEN}Found required file at${RESET} ${RED}: "$filepath" ${RESET}"
    echo -e "${GREEN}Initiating analysis...${RESET}"
```

The output example:

```
Found required file at : /home/kali/Desktop/m4.vmem
Initiating analysis...
```

4. Dependency Verification & Tool Installation

```
declare -A Packages=(
    [foremost]="foremost"
    [scalpel]="scalpel"
    [bulk_extractor]="bulk-extractor"
    [binwalk]="binwalk"
    [strings]="binutils"
    [file]="file"
    [exiftool]="libimage-exiftool-perl"
)
for tool in "${!Packages[@]}"; do
    if ! command -v "$tool" >/dev/null 2>&1; then
        echo -e "${RED}Installing missing tool: $tool ${RESET}"
        apt install -y "${Packages[$tool]}" >/dev/null 2>&1
    else
        echo -e "${GREEN}$tool already installed ${RESET}"
    fi
done
echo -e "${BLUE}${LINE}${RESET}"
```

Declare -A Packages = declares an **associative array** named Packages
Its used to allow mapping between the **Command name** and **Package name**
this is crucial because the command names and package names are **not always identical** such as exiftool and strings.

`for tool in "${!Packages[@]}"; do` the following for loop is important as it is iterates over all tool names in the associative array named Packages
the ! (exclamation mark) is telling Give me the indexes not the values.
so ${!Packages[@]} expands to **foremost scalpel bulk_extractor binwalk strings file exiftool.** Which is what the **command -v** needs
@ expands each element as a separate word .
so it becomes “foremost” “scalpel” “bulk_extractor” and so on.. each of them is assigned to the variable **tool**

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

if NOT command -v $tool (bulk,strings,foremost,scalpel,binwalk....)
which is saying if the tools do not exist then tell the user its missing and installs it.
with **apt install -y “${Packages[$tool]}”**
“${Packages[$tool]}” which retrieves the VALUE associated of that key
meaning .. that **strings** is seen as **binutils** and **exiftool** as **the package libimage-exiftool-perl**

```
❯ command -v exiftool
/usr/bin/exiftool
```

If the tools already exist the output is green and tells the user that the associated tool is installed , incase the tool is missing the script will install it for him and display that in red.

5. Output Structure Initialization

```
echo -e "${GREEN}Extracting Data using carvers, it will take a while... ${RESET}"
OUTDIR="/home/$REAL_USER/Desktop/Extracted_Data"
mkdir -p "$OUTDIR"
REPORT="$OUTDIR/REPORT.txt"
VOL_LOG="$OUTDIR/volatility_plugins.log"
: > "$VOL_LOG"
mkdir -p "$OUTDIR/bulk"
mkdir -p "$OUTDIR/binwalk"
mkdir -p "$OUTDIR/foremost"
```

This section prepares the directory structure and log files that will store all the extracted data and analysis results that will be latter reviewed by the user.
everything is going to be saved inside the directry called **“Extracted_Data”** which is saved inside the variable called **OURDIR** proceeding with **mkdir -p**
-p flag ensures that the command will not fail if the directory already exists.

```
-p, --parents     no error if existing, make parent directories as needed,
                  with their file modes unaffected by any -m option
```

REPORT=”$OUTDIR/REPORT.txt is going to save logged report such as timestamps,hash,plugins that were used,what was extracted and how much,and more.
VOL_LOG=”$OURDIR/volatility_plugins.log” volatility execution plugin log which is going to show what plugins were used and if they were successful.
: > "$VOL_LOG" this line create or clears the vol log file.
: is a no operation command , this ensures the log starts empty for each execution.
mkdir -p "$OUTDIR/bulk"
mkdir -p "$OUTDIR/binwalk"
mkdir -p "$OUTDIR/foremost"
These commands create a tool specific directory for bulk,binwalk,foremost outputs... instead of making them all in one , this will create a more organized enviornment to latter review the results so we can associate each result to the tool and its purpose.

6. Multi-Tool Artifact Extraction (Carving Phase)

```
echo -e "${GREEN}Running Bulk Extractor on the file [+]${RESET}"
bulk_extractor "$filepath" -o "$OUTDIR/bulk" >/dev/null 2>&1
echo -e "${GREEN}Running Binwalk on the file [+]${RESET}"
binwalk -e "$filepath" -C "$OUTDIR/binwalk" >/dev/null 2>&1
echo -e "${GREEN}Running foremost on the file [+]${RESET}"
foremost -i "$filepath" -o "$OUTDIR/foremost" >/dev/null 2>&1
echo -e "${GREEN}Running strings on the file [+}${RESET}"
strings "$filepath" > "$OUTDIR/strings.txt" 2>/dev/null
echo -e "${GREEN}[+] Extraction completed.${RESET}"
echo -e "${BLUE}${LINE}${RESET}"
```

Now we perform the actual data extraction using 4 tools **(binwalk, bulk, foremost, strings)**
Each one of them uses different technique which will be explained latter separately.
but its necessary to use different tool in order to extract useful artifacts.
each one of them uses the file which is specified with **$filepath** , the one user inputs at the start of the script and outputs it in the different directories we made for it $OUTDIR/(the tool associated directory)
Upon completion it will display that the extraction is complete for the designed tool and proceed to the next until extraction is complete.
Like the following example:

```
Extracting Data using carvers, it will take a while...
Running Bulk Extractor on the file [+]
Running Binwalk on the file [+]
Running foremost on the file [+]
Running strings on the file [+}
[+] Extraction completed.
```

bulk_extractor "$filepath" -o "$OUTDIR/bulk" >/dev/null 2>&1
-o stands for the output directory , notice that bulk doesn't require any flag for the specified filename because of how it works, it does not perform file carving or embedded file extraction it processes the input as a raw byte and extracts forensic artifacts such as URLs , Ips , Emails , etc.
binwalk -e "$filepath" -C "$OUTDIR/binwalk" >/dev/null 2>&1
-e flag option enables automatic extraction , while **-c** specifies the output directory
foremost -i "$filepath" -o "$OUTDIR/foremost" >/dev/null 2>&1
here **-i** specifies the input file and **-o** sets the output directory as specified
strings "$filepath" > "$OUTDIR/strings.txt" 2>/dev/null
strings by default doesn't create an output directory and does not manage files by itself.
it simply prints results to standard output
and thats exactly why **2>/dev/null** was used , The output needs to be saved eventually into a file and with the way it works to make it clean we only want to hide the error but keep the output
if for example it would be **>/dev/null 2>&1** like the other tools , no results will be saved.
2>&1 this statement actually says , send the **STERR (2)** to where **STOUT (1)** is going .

Quick note – each tool operates differently and has it's own manual so its very important to look inside of it with **–help** & **-h** & **man** , to look inside of it and see which flag corresponds with what we want to have as a result.

7. Network Artifact Discovery

```
echo -e "${GREEN}Checking for network traffic file...${RESET}"
PCAP=$(find "$OUTDIR" -type f -name "*.pcap" 2>/dev/null | head -n 1 )
if [ -z "$PCAP" ] ; then
echo -e "${RED}Network traffic file is not found [-]${RESET}"
else
echo -e "${RED}Found Network Traffic File at:${RESET}" "$PCAP"
fi
echo -e "${BLUE}${LINE}${RESET}"
```

This section focuses on finding a network traffic file if extracted by one of the tools, usually the tool that is able to extract PCAP files is bulk extractor (NOT always but most of the time it does)
network traffic files are very useful as they can contain evidence of network communication , data exfiltration and more which can be latter analyzed using wireshark

PCAP=$(find "$OUTDIR" -type f -name "*.pcap" 2>/dev/null | head -n 1)

the following line searches the output directory for files with a **.pcap** extension

find "$OUTDIR" searches recursively in the output directory which I named **Extrated_data**

-type f limits the search to regular files

-name "*.pcap" – matches network capture files (it can be 123.pcap or 341.pcap) ***** stands for global

head -n 1 returns only the first match , I never encountered multiple network file extraction from bulk it can be removed , but i still used it to limit the result to 1 file .

if [-z "$PCAP"] ; then

this condition checks whether a PCAP exists

-z checks if the variable is empty , returns true if the string has no characters

-n is the opposite

it's like using the **!** with the **command -v** previously.

the same logic could be written like this

```
if [ -n "$PCAP" ] ; then
echo -e "${RED}Found Network Traffic File at:${RESET}" "$PCAP"
else
echo -e "${RED}Network traffic file is not found [-]${RESET}"
```

so if nothing was store inside PCAP variable = no .pcap file found

```
echo -e "${RED}Found Network Traffic File at:${RESET}" "$PCAP"
```

The output result will look like this:

```
Found Network Traffic File at: /home/kali/Desktop/Extracted_Data/bulk/packets.pcap
```

So the user executing the script can know where the network file is located to perform extra analysis using network analyser tools like wireshark.

8. Human-Readable Artifact Analysis

```
echo -e "${GREEN}Extracting useful artifacts using strings...${RESET}"
echo -e "${BLUE}${LINE}${RESET}"
grep -iE "pass|user|key|token" "$OUTDIR/strings.txt" > "$OUTDIR/creds.txt"
grep -iE "http[s]?://" "$OUTDIR/strings.txt" > "$OUTDIR/url.txt"S
grep -E "([0-9]{1,3}\.){3}[0-9]{1,3}" "$OUTDIR/strings.txt" > "$OUTDIR/ips.txt"
echo -e "${GREEN}Finished Extraction, using strings!${RESET}"
echo -e "${BLUE}${LINE}${RESET}"
```

echo -e "${GREEN}Extracting useful artifacts using strings...${RESET}"
The first line used to notify the user that the script is beginning the artifact extraction phase
grep -iE "pass|user|key|token" "$OUTDIR/strings.txt" > "$OUTDIR/creds.txt"
-i makes the search case-insensitive
-E enables extended regular expressions which enables the use of **[s]?**
and keywords such as **pass , user , key , token** which are commonly associated with credentials
everything that matches that is saved into **creds.txt** for further analysis
grep -iE "http[s]?://" "$OUTDIR/strings.txt" > "$OUTDIR/url.txt"
this line to extract urls even though bulk extractor already does that maybe it missed something.
[s] in order to match both http and https , **?** question mark makes the letter **s** optional
allowing the pattern to match both http and https urls
grep -Eo '([0-9]{1,3}\.){3}[0-9]{1,3}' "$OUTDIR/strings.txt" > "$OUTDIR/ips.txt"
this command will extract IPv4 addresses
-E again to enable extended regular expressions
such as **{} ()**
-o Only matching output
[0-9]{1,3} -Any digit from 0 to 9 , repeated 1 to 3 times **{1,3}**
\. – means literal dot , the dot must be escaped because . normally means any character in regex
{3} – repeats the entire group 3 times
[0-9]{1,3} – again at the end which is the final octet of the IPv4.
because there are 3 dots and 4 octets, it was necessary to divide it.
'([0-9]{1,3}\.){3}[0-9]{1,3}'
is a very useful syntax especially when doing text manipulation that requires looking IPs in CTFs or auth.log labs can combine with **sort** and **uniq.**

9. Setting up volatility3 installation

```
echo -e "${GREEN}Installing dependencies and volatility to perform extraction...${RESET}"
VOL_DIR="/home/$REAL_USER/Desktop/volatility3"
VOL_PATH="$VOL_DIR/vol.py"
pip3 install capstone simplejson pycryptodome pillow openpyxl ujson yara-python --break-system-packages > /dev/null 2>&1
apt install python3 python3-pip -y   > /dev/null 2>&1
if [ ! -d "$VOL_DIR" ]; then
    git clone https://github.com/volatilityfoundation/volatility3.git >/dev/null 2>&1 "$VOL_DIR"
fi
chown -R "$REAL_USER:$REAL_USER" "$OUTDIR" "$VOL_DIR"
```

This section prepares the environment for **memory analysis using Volatility 3**.
It installs the required Python dependencies, ensures Volatility is available locally, and sets correct file ownership so the extracted data can be accessed by the original user.

echo -e "${GREEN}Installing dependencies and volatility to perform extraction...${RESET}"
to inform the user that the script is about to install the required tools and libraries needed

VOL_DIR="/home/$REAL_USER/Desktop/volatility3"
Defines the directory where volatility3 is going to live which is in Desktop under the directory volatilty3

VOL_PATH="$VOL_DIR/vol.py"
stores the path to the vol3 executable script each command will have to refer to it,
this avoids repeating long paths later in the script and improves readability.

pip3 install capstone simplejson pycryptodome pillow openpyxl ujson yara-python --break-system-packages > /dev/null 2>&1
Installs Python dependencies required by Volatility
These libraries support disassembly, JSON handling, cryptography, YARA scanning, and report parsing
--break-system-packages allows installation even on systems with restricted Python environments
Output is suppressed to keep execution clean.

apt install python3 python3-pip -y
Ensures that Python 3 and pip3 are installed on the system.
This guarantees that Volatility and its dependencies can run correctly.

if [! -d "$VOL_DIR"]; then
Basically checks if there's already a volatility directory that already exists
! = **NOT**, **-d** = directory
$VOL_DIR Its limited to this path which is the desktop, so if there's some other folder named like this somewhere else it will still install the vol3 to desktop as a new one additionally.

git clone https://github.com/volatilityfoundation/volatility3.git >/dev/null 2>&1 "$VOL_DIR"
Clones the Volatility 3 repository into the specified directory if it is not already present.
Output is suppressed to avoid cluttering the terminal.

chown -R "$REAL_USER:$REAL_USER" "$OUTDIR" "$VOL_DIR"
Recursively changes ownership of the extracted data directory and volatility directory.

10. Memory Dump Detection & Classification

```
echo -e "${GREEN}Checking if file is memory-dump compatible...${RESET}"
FILE_TYPE=$(file -b "$filepath")
echo -e "${BLUE}Detected file type:${RESET} $FILE_TYPE"
IS_MEMORY=false
FILE_EXT="${filepath##*.}"

if echo "$FILE_EXT" | grep -qiE 'mem|raw|dmp|vmem|lime'; then
    IS_MEMORY=true
elif echo "$FILE_TYPE" | grep -qiE 'memory|crash dump|vmcore|event trace'; then
    IS_MEMORY=true
fi
```

This section determines whether the provided file **looks like a memory dump** and whether it is potentially compatible with memory analysis tools such as Volatility.

Instead of relying on a single indicator, the script uses two detection methods:

1. File extension analysis
2. File signature / type analysis

FILE_TYPE=$(file -b "$filepath")

This command uses the file utility to detect the file's **content type** based on its binary signature.

-b (brief mode) removes the filename from the output

The result is stored in FILE_TYPE

This helps identify memory dumps even if the file extension is misleading.

echo -e "${BLUE}Detected file type:${RESET} $FILE_TYPE"

Prints the detected file type so the user can see how the system classified the file.

IS_MEMORY=false

Initializes a boolean flag indicating whether the file is considered a memory dump.

The default is set to false until proven otherwise.

FILE_EXT="${filepath##*.}"

Extracts the **file extension** from the file path.

Double hash operator - Remove the longest matching pattern from the beginning of the string

*. This is a glob pattern, not regular expression = /home/user/Desktop/memory_dump.

removes the longest such match Results in **vmem**

if echo "$FILE_EXT" | grep -qiE 'mem|raw|dmp|vmem|lime'; then

IS_MEMORY=true

This checks if the file extension matches **common memory dump formats**.

-q suppresses output , **-i** makes the match case-insensitive , **-E** allows multiple patterns

the extension included are **.mem,.raw,.dmp,.vmem,.lime**

elif echo **"$FILE_TYPE" | grep -qiE 'memory|crash dump|vmcore|event trace'; then**

IS_MEMORY=true

If the extension check fails, this block analyzes the file signature output instead.

In other words, If the filename doesn't look like memory, check what the file actually is

Using both methods improves detection accuracy and avoids relying on assumptions.

11. Memory Forensics Automation (Volatility)

```
CAN_ANALYZE=false

if [ "$IS_MEMORY" = true ]; then
    echo -e "${GREEN}Testing Volatility compatibility...${RESET}"

    if python3 "$VOL_PATH" -f "$filepath" windows.info > "$OUTDIR/vol_probe.txt" 2>&1; then
        CAN_ANALYZE=true
        OS_TYPE="windows"
    elif python3 "$VOL_PATH" -f "$filepath" linux.info > "$OUTDIR/vol_probe.txt" 2>&1; then
        CAN_ANALYZE=true
        OS_TYPE="linux"
    elif python3 "$VOL_PATH" -f "$filepath" mac.info > "$OUTDIR/vol_probe.txt" 2>&1; then
        CAN_ANALYZE=true
        OS_TYPE="mac"
    fi
fi
if [ "$CAN_ANALYZE" = true ]; then
    echo -e "${GREEN}Volatility confirmed this memory dump is analyzable.${RESET}"
else
    echo -e "${RED}Volatility could NOT analyze this memory dump.${RESET}"
fi
```

This section determines whether Volatility can analyze the memory dump and, if so, which operating system profile applies **(Windows, Linux, or macOS)**.

It acts as a capability probe, not a full analysis.

CAN_ANALYZE=false Initializes a control flag that tracks whether Volatility analysis is possible.

if ["$IS_MEMORY" = true]; then

Ensures Volatility probing only happens **if the file was already identified as memory-like.**

It avoids running volatility on disk images, documents , random binary files.

echo -e "${GREEN}Testing Volatility compatibility...${RESET}"

Informs the user that Volatility is starting

if python3 "$VOL_PATH" -f "$filepath" windows.info > "$OUTDIR/vol_probe.txt" 2>&1; then

CAN_ANALYZE=true

OS_TYPE="windows"

this is the windows probe , runs the windows.info volatility plugin and redirects output to vol_probe.txt

elif python3 "$VOL_PATH" -f "$filepath" linux.info > "$OUTDIR/vol_probe.txt" 2>&1; then

CAN_ANALYZE=true

OS_TYPE="linux"

linux probe which runs only if the Windows probe failed.

elif python3 "$VOL_PATH" -f "$filepath" mac.info > "$OUTDIR/vol_probe.txt" 2>&1; then

CAN_ANALYZE=true

OS_TYPE="mac"

and finally the mac OS probe , its the less common .

the **elif** logic is that It allows you to test **another condition only if the previous one failed**.

Overall, this section acts as a **validation gate** between raw memory detection and full forensic analysis, ensuring accuracy, efficiency, and safe execution of Volatility plugins.

```
run_vol_plugin() {
    local plugin="$1"
    local outfile="$2"

    if python3 "$VOL_PATH" -f "$filepath" "$plugin" > "$outfile" 2>&1; then
        echo -e "${GREEN}[+] $plugin completed${RESET}"
        echo "OK,$plugin,$outfile" >> "$VOL_LOG"
        return 0
    else
        echo -e "${DARKGREY}[-] $plugin not supported or failed${RESET}"
        echo "FAIL,$plugin,$outfile" >> "$VOL_LOG"
        return 1
    fi
}
```

This function **safely executes Volatility plugins**, captures their output, logs success or failure, and prevents the script from crashing if a plugin is unsupported or fails.

Starting with **run_vol_plugin() {** that defines a reusable bash function and inside it local variables

local plugin="$1" → plugin name

local outfile="$2" → output file path

local ensures variables exist only inside the function , no accidental overwriting of global variables.

if python3 "$VOL_PATH" -f "$filepath" "$plugin" > "$outfile" 2>&1; then

Executes Volatility with the selected plugin , All output is captured .

both stdout and stderr redirected to **volatility_plugins.log** file.

echo -e "${GREEN}[+] $plugin completed${RESET}"

echo "OK,$plugin,$outfile" >> "$VOL_LOG"

return 0

if the plugin ran successfully it will print a success message , appends a structured log entry and returns exit code 0 (success)

else

echo -e "${DARKGREY}[-] $plugin not supported or failed${RESET}"

echo "FAIL,$plugin,$outfile" >> "$VOL_LOG"

return 1

If the plugin fails or is unsupported:

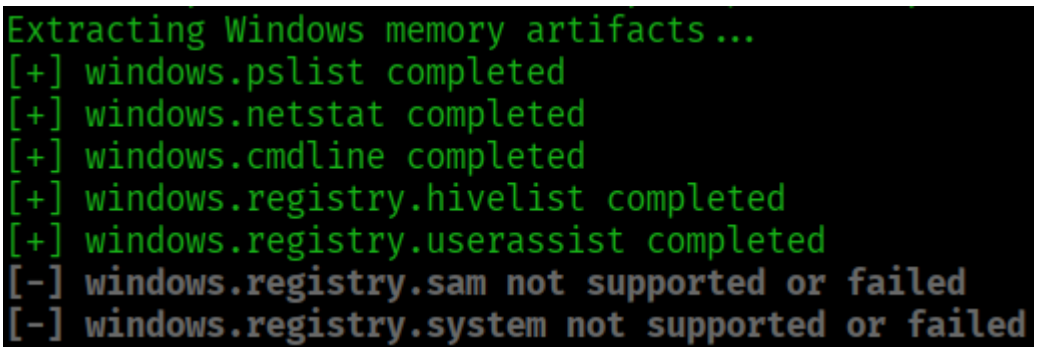
print a neutral failure message , logs the failure without stopping the script and returns exit code 1

this prevents script termination , false positive , crash.

In basic words

Run plugin → capture output → log result → keep going

Output will look like this in the terminal :



```
if [ "$CAN_ANALYZE" = true ] && [ "$OS_TYPE" = "windows" ]; then
    echo -e "${GREEN}Extracting Windows memory artifacts...${RESET}"

    run_vol_plugin windows.pslist "$OUTDIR/vol_windows_pslist.txt"
    run_vol_plugin windows.netstat "$OUTDIR/vol_windows_netstat.txt"
    run_vol_plugin windows.cmdline "$OUTDIR/vol_windows_cmdline.txt"
    run_vol_plugin windows.registry.hivelist "$OUTDIR/vol_windows_hivelist.txt"
    run_vol_plugin windows.registry.userassist "$OUTDIR/vol_windows_userassist.txt"
    run_vol_plugin windows.registry.sam "$OUTDIR/vol_windows_sam.txt"
    run_vol_plugin windows.registry.system "$OUTDIR/vol_windows_system.txt"
fi
```

The following section performs targeted windows memory analysis using volatility after it is validated that the file is a valid memory dump, volatility can analyze it and the operating system id is windows

if ["$CAN_ANALYZE" = true] && ["$OS_TYPE" = "windows"]; then

CAN_ANALYZE=true

Volatility successfully probed the memory dump

OS_TYPE=windows

The dump belongs to a Windows system

echo -e "${GREEN}Extracting Windows memory artifacts...${RESET}"

informs the user that the script has passed validation and is windows specific memory extraction.

Proceeds with **Plugin Execution**

run_vol_plugin what happens behind the scene is the following :

plugin=”windows.pslist”

outfile="/home/user/Desktop/Extracted_Data/vol_windows_pslist.txt"

python3 "$VOL_PATH" -f "$filepath" "$plugin" > "$outfile" 2>&1

This is the command that is being executed

python3 /home/<user>/Desktop/volatility3/vol.py | this is the $VOL_PATH

-f /home/<user>/Desktop/memdump.raw | this is the $filepath

windows.pslist | this is the local plugin = $1

> /home/<user>/Desktop/Extracted_Data/vol_windows_pslist.txt | this is the local outfile $2

2>&1

```
if [ "$CAN_ANALYZE" = true ] && [ "$OS_TYPE" = "linux" ]; then
    echo -e "${GREEN}Extracting Linux memory artifacts...${RESET}"

    run_vol_plugin linux.pslist "$OUTDIR/vol_linux_pslist.txt"
    run_vol_plugin linux.netstat "$OUTDIR/vol_linux_netstat.txt"
    run_vol_plugin linux.lsmod "$OUTDIR/vol_linux_modules.txt"
    run_vol_plugin linux.bash "$OUTDIR/vol_linux_bash.txt"
fi
```

Another section for the linux OS , same principles as explained above , and finally MacOS

```
if [ "$CAN_ANALYZE" = true ] && [ "$OS_TYPE" = "mac" ]; then
    echo -e "${GREEN}Extracting macOS memory artifacts...${RESET}"

    run_vol_plugin mac.pslist "$OUTDIR/vol_mac_pslist.txt"
    run_vol_plugin mac.netstat "$OUTDIR/vol_mac_netstat.txt"
fi
```

This makes the script OS “flexible” , even though MacOS and Linux have fewer plugins than windows, Their memory structures are less standardized , security protections are stronger , kernel data structures changes frequently , and windows is always the mainstream operating system.

12. Reporting, Packaging & Evidence Integrity

```
END_TIME=$(date +"%Y-%m-%d %H:%M:%S")
END_EPOCH=$(date +%s)
DURATION_SEC=$((END_EPOCH - START_EPOCH))

FILE_SIZE_BYTES=$(stat -c%s "$filepath" 2>/dev/null)
FILE_SHA256=$(sha256sum "$filepath" 2>/dev/null | awk '{print $1}')

TOTAL_EXTRACTED_FILES=$(find "$OUTDIR" -type f 2>/dev/null | wc -l)
CARVED_FILES_FOREMOST=$(find "$OUTDIR/foremost" -type f 2>/dev/null | wc -l)
CARVED_FILES_BINWALK=$(find "$OUTDIR/binwalk" -type f 2>/dev/null | wc -l)
BULK_FILES=$(find "$OUTDIR/bulk" -type f 2>/dev/null | wc -l)

PCAP_FOUND="no"
if [ -n "$PCAP" ]; then PCAP_FOUND="yes"; fi

CREDS_HITS=$(wc -l < "$OUTDIR/creds.txt" 2>/dev/null || echo 0)
URL_HITS=$(wc -l < "$OUTDIR/url.txt" 2>/dev/null || echo 0)
IP_HITS=$(wc -l < "$OUTDIR/ips.txt" 2>/dev/null || echo 0)

VOL_OK_COUNT=$(grep -c '^OK,' "$VOL_LOG" 2>/dev/null || echo 0)
VOL_FAIL_COUNT=$(grep -c '^FAIL,' "$VOL_LOG" 2>/dev/null || echo 0)
```

This section collects timestamps, file metadata, extraction statistics, analysis results in order to: Measure how long the analysis took, Identify the target file uniquely (hash & size), Count extracted artifacts, detect whether network traffic was found, quantify string-based hits, Summarize Volatility plugin success/failure.

END_TIME=$(date +"%Y-%m-%d %H:%M:%S") → Stores human-readable timestamp

END_EPOCH=$(date +%s) → Unix time **(seconds since 1970)**

Which is going to respond to this block at the start of the script at the very top.

```
START_TIME=$(date +"%Y-%m-%d %H:%M:%S")
START_EPOCH=$(date +%s)
```

DURATION_SEC=$((END_EPOCH - START_EPOCH)) → extracts duration of the script's execution

1.FILE_SIZE_BYTES=$(stat -c%s "$filepath" 2>/dev/null)

2.FILE_SHA256=$(sha256sum "$filepath" 2>/dev/null | awk '{print $1}')

1. Extracts the exact file size in bytes

2. Extracts the sha256 hash of the file (this provides integrity)

TOTAL_EXTRACTED_FILES=$(find "$OUTDIR" -type f 2>/dev/null | wc -l)

CARVED_FILES_FOREMOST=$(find "$OUTDIR/foremost" -type f 2>/dev/null | wc -l)

CARVED_FILES_BINWALK=$(find "$OUTDIR/binwalk" -type f 2>/dev/null | wc -l)

BULK_FILES=$(find "$OUTDIR/bulk" -type f 2>/dev/null | wc -l)

this section provides extraction statistics such total amount of extracted files via foremost, binwalk ,and bulk extractor tools to be latter displayed in the report file.

Which will be displayed like this :

```
---------------- CARVING OUTPUT --------------
Total files under OUTDIR:     52484
Foremost files:               1994
Binwalk extracted files:      0
Bulk Extractor files:         50475
```

PCAP_FOUND="no"

if [-n "$PCAP"]; then PCAP_FOUND="yes"; fi

Network file detection like seen previously in section 7 , will also be displayed in the report file along with the script execution itself in the terminal.

CREDS_HITS=$(wc -l < "$OUTDIR/creds.txt" 2>/dev/null || echo 0)

URL_HITS=$(wc -l < "$OUTDIR/url.txt" 2>/dev/null || echo 0)

IP_HITS=$(wc -l < "$OUTDIR/ips.txt" 2>/dev/null || echo 0)

Extraction of specific artifacts such as IP , URL , CREDS which contain words like (pass,user,token) always useful for a quick overview

Expected output in the report file as follows:

```
---------------- STRINGS HITS ----------------
Cred keywords lines:          104169
URL lines:                    47809
IP lines:                     9419
```

VOL_OK_COUNT=$(grep -c '^OK,' "$VOL_LOG" 2>/dev/null || echo 0)

VOL_FAIL_COUNT=$(grep -c '^FAIL,' "$VOL_LOG" 2>/dev/null || echo 0)

Volatility plugin success tracking

```
---------------- VOLATILITY PLUGINS ----------
Succeeded: 5
Failed:    2
```

wc-l < (somefile.txt)

the **<** redirects file content into STDIN , wc thinks input coming from a pipe not a file.

which results only in numbers.

|| echo 0

|| stands for logic ‘OR’

so if the first command fails it runs the second command

so if **wc** fails **echo 0** runs and variable **become 0**

This makes 2 possible outcome , File exists and displayed numbers of lines , File missing output 0

```
{
echo "================ FORENSICS REPORT ================"
echo "Start Time:   $START_TIME"
echo "End Time:     $END_TIME"
echo "Duration:     ${DURATION_SEC}s"
echo ""
echo "---------------- TARGET FILE ----------------"
echo "Path:         $filepath"
echo "Size (bytes): ${FILE_SIZE_BYTES:-unknown}"
echo "SHA256:       ${FILE_SHA256:-unknown}"
echo "file(1):      $FILE_TYPE"
echo ""
echo "---------------- DETECTION ------------------"
echo "Is memory-like:   $IS_MEMORY"
echo "Detected OS:      $OS_TYPE"
echo "Volatility OK:    $CAN_ANALYZE"
echo ""
echo "---------------- CARVING OUTPUT --------------"
echo "Total files under OUTDIR:     $TOTAL_EXTRACTED_FILES"
echo "Foremost files:               $CARVED_FILES_FOREMOST"
echo "Binwalk extracted files:      $CARVED_FILES_BINWALK"
echo "Bulk Extractor files:         $BULK_FILES"
echo "PCAP found:                   $PCAP_FOUND"
if [ -n "$PCAP" ]; then echo "PCAP path:                    $PCAP"; fi
echo ""
echo "---------------- STRINGS HITS ----------------"
echo "Cred keywords lines:          $CREDS_HITS"
echo "URL lines:                    $URL_HITS"
echo "IP lines:                     $IP_HITS"
echo ""
echo "---------------- VOLATILITY PLUGINS ----------"
echo "Succeeded: $VOL_OK_COUNT"
echo "Failed:    $VOL_FAIL_COUNT"
echo ""
if [ -s "$VOL_LOG" ]; then
    echo "Details (status,plugin,output_file):"
    cat "$VOL_LOG"
else
    echo "No volatility plugins were attempted."
fi
echo "================================================"
} > "$REPORT"
```

This whole block is what is going to build the REPORT.txt file log which audits everything was done from the previous block where the values are stored , and everything is now being used in order to build the report fext file.

the important syntrax here is **{** all the commands**; } > “$REPORT”**

which is useful for single redirection , cleaner audit trail .

```
TimeStamps=$(date +"%Y-%m-%d_%H-%M-%S")
ZipPath="/home/$REAL_USER/Desktop/forensics_results_${TimeStamps}.zip"
Hash="${ZipPath}.sha256"

echo -e "${GREEN}Creating ZIP archive...${RESET}"
zip -r "$ZipPath" "$OUTDIR" >/dev/null

echo -e "${GREEN}Calculating SHA256 hash...${RESET}"
sha256sum "$ZipPath" > "$Hash"

echo -e "${GREEN}Results packaged successfully:${RESET}"
echo " ZIP : $ZipPath"
echo " HASH: $Hash"
```

And finally the last block of this section, this block finalizes the forensic collection process by creating a uniquely timestamped ZIP archive of all collected artifacts and generating a SHA256 hash file to ensure data integrity. The results are saved to the real user's Desktop, enabling proper evidence handling and chain-of-custody validation.

Hash="${ZipPath}.sha256"

this creates a matching hash filename:

like this forensics_results_2025-12-17_10-15-15.zip.sha256

echo -e "${GREEN}Creating ZIP archive...${RESET}"

zip -r "$ZipPath" "$OUTDIR" >/dev/null

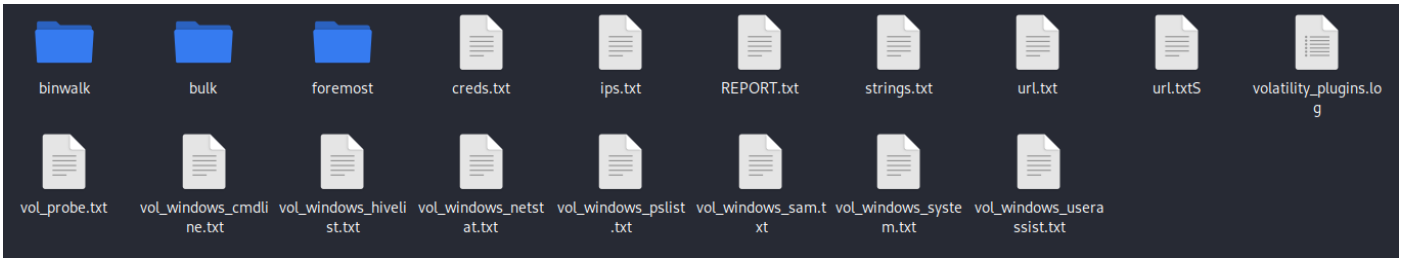
The creation of the ZIP file , recursively compresses all of the results in the Extracted_Data directory

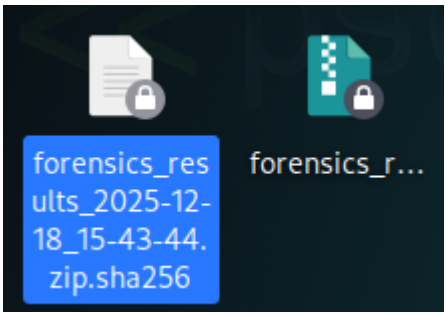
echo -e "${GREEN}Calculating SHA256 hash...${RESET}"

sha256sum "$ZipPath" > "$Hash"

will generate a cryptographic hash and store it separately in a text file with **.sha256** ending

Final Results.





The final output in the terminal looks like this:

```
❯ sudo bash TMagen773638_s26_NX212.sh
[sudo] password for kali:
═══════════════════════════════════════════════
You're Root , Continuing ...
═══════════════════════════════════════════════
Enter your file name to analyze:m4.vmem
Found required file at : /home/kali/Desktop/m4.vmem
Initiating analysis ...
═══════════════════════════════════════════════
file already installed
strings already installed
foremost already installed
exiftool already installed
bulk_extractor already installed
scalpel already installed
binwalk already installed
═══════════════════════════════════════════════
Extracting Data using carvers, it will take a while ...
Running Bulk Extractor on the file [+]
Running Binwalk on the file [+]
Running foremost on the file [+]
Running strings on the file [+}
[+] Extraction completed.
═══════════════════════════════════════════════
Checking for network traffic file ...
Found Network Traffic File at: /home/kali/Desktop/Extracted_Data/bulk/packets.pcap
═══════════════════════════════════════════════
Extracting useful artifacts using strings ...
═══════════════════════════════════════════════
Finished Extraction, using strings!
═══════════════════════════════════════════════
Installing dependencies and volatility to perform extraction ...
Checking if file is memory-dump compatible ...
Detected file type: data
Testing Volatility compatibility ...
Volatility confirmed this memory dump is analyzable.
Extracting Windows memory artifacts ...
[+] windows.pslist completed
[+] windows.netstat completed
[+] windows.cmdline completed
[+] windows.registry.hivelist completed
[+] windows.registry.userassist completed
[-] windows.registry.sam not supported or failed
[-] windows.registry.system not supported or failed
Report saved to: /home/kali/Desktop/Extracted_Data/REPORT.txt
Creating ZIP archive ...
Calculating SHA256 hash ...
Results packaged successfully:
 ZIP : /home/kali/Desktop/forensics_results_2025-12-18_15-43-44.zip
 HASH: /home/kali/Desktop/forensics_results_2025-12-18_15-43-44.zip.sha256
```

About Forensics tool

Tool	Purpose
Foremost	Signature-based file carving (images, docs, archives)
Binwalk	Extract embedded files and firmware components
Bulk Extractor	High-speed extraction of credentials, URLs, IPs, emails, PCAPs
strings	Extract human-readable strings from binary files
file (file(1))	Identify real file type regardless of extension
Volatility 3	Memory forensics (Windows, Linux, macOS artifacts)
sha256sum	Cryptographic hashing for evidence integrity
zip	Evidence packaging and delivery

Memory forensics is a crucial aspect of digital forensics that involves analyzing volatile data in a computer's memory (RAM) to uncover evidence of system use, malware infection, user actions, and more. Unlike traditional disk forensics, memory forensics allows investigators to see what was happening on a system at a specific point in time, including programs that were running, network connections that were active, and data that was in use but not necessarily saved to disk.

Volatility 3 represents a significant evolution in the field of memory forensics. Building on the success of its predecessors, it provides a robust framework for analyzing memory dumps across various operating systems, including **Windows, Linux, and macOS.** Volatility 3 offers numerous advantages over earlier versions, such as improved performance, a more modular architecture, support for the latest operating systems and file formats, and a more flexible command-line interface. These enhancements make Volatility 3 an indispensable tool for forensic analysts seeking to conduct detailed and efficient analyses of volatile memory