Memory analysis project

TMagen773637.S11.NX212

Ron Nisinboym

June 24, 2025

Overview

Project Description

The project aims to deliver a simple bash script that uses Kali Linux tools such as Volatility and Carving to automate the process of analyzing memory files by extracting the relevant data to a folder, writing down a report file with the outcome and finally compressing it all into a ZIP file for further investigation. This process simplifies the workflow for security analysts and forensic investigators by streamlining the steps required to perform memory forensics.

Technologies Used

- **Bash Scripting**: Used for automating the execution of commands, managing file paths, and automating the report generation.
- **Volatility**: A memory forensics tool used to extract valuable information from memory dumps, such as running processes, network activity, and other relevant data. This project uses *Volatility 2.5* version.
- **Carving**: Used for extracting embedded files and data from the memory dump that may lead to further investigation. Carving tools used in this project are – Bulk_Extractor, Binwalk, Foremost and Strings.
- **ZIP Compression**: Employed for packaging the analysis results into a single compressed file for easy handling.

Detailed Workflow

1. **Input**: The script receives the memory dump file as input.
2. **Tools installation**: The script checks if all necessary are installed on the system prior to execution stage and installs any missing ones.
3. **Execution**:
    a. The script lets the user choose to run **Volatility** and **Carving** tools on the memory dump to analyze and extract data.
    b. Relevant data (such as process lists, open network connections, and any recovered files) is extracted and saved into an organized folder structure.
    c. Prioritized data is also displayed in the terminal.
4. **Report Generation**: in the exiting stage, a report is automatically generated, summarizing:
    a. Date and time in which the script was completed.
    b. A list of all extracted files and the number of files found.
5. **Compression**: After gathering the necessary data and generating the report, the script compresses the folder into a ZIP file for easy handling and sharing.

Design Principles:

1. **Simplicity Over Decoration:** By reducing the use of bold colors, fancy fonts, and overly complex layouts, the script avoids visual clutter, providing an **"easy on the eyes" experience**.
2. **Clear, Readable Output**:
    a. The font choice is kept **simple and legible**, ensuring that the user can quickly read and interpret the analysis results, even over long periods of use.
    b. Colors are used sparingly, with only essential information (e.g., warnings, errors and successes) highlighted using subtle color changes. This reduces eye strain while still providing the necessary emphasis on important results.

Credits:

1. **Erel Regev (lecturer) –** for kickstarting the bash script, providing the initial framework and guidance by supplying the foundational script structure and key concepts which was later modified and expanded to my own vision of this project.
2. **ChatGPT** – The use of AI was made exclusively as a helping tool. Used to fix some of the errors and bugs occurring while writing the script, consulting in regards of the script's logic when more difficult loops were used(e.g. time of analysis for each tool) and asking for general information(e.g. colors, fonts, clearing outputs, timestamps, etc.), and things I struggled to remember(e.g. certain flags). The choice of ChatGPT over Google, forums or other sources was made mostly to save time and get straight forward explanations for each piece of advice to keep learning from it.

Detailed Walkthrough of the Script

This section provides a breakdown of the script's functionality, including detailed explanations of key code snippets, along with screenshots of the code and its practical execution.

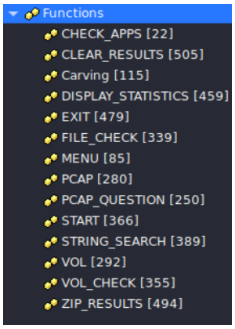
Global Variables -

In this screenshot we can see all the global variables that are being used by the script.
These variables **are not** nested in any function, to be available for any function that needs to call them.

```
#Global Variables.
#Used for folder:
HOME=$(pwd)

#Used for colors and fonts:
RED='\033[0;31m'
CYAN='\033[36m'
GREEN='\033[38;5;34m'
BOLD='\033[1m'
NC='\033[0m'
```

Functions in general -

In this screenshot we can see a list of all functions that are being used by the script.
The idea was to encapsulate repetitive chunks of code or any code that has a unique behavior, to achieve ease of access and further modifying of specific functionality.
Therefore, all pieces of code in this script, **except** global variables, are nested inside functions.



Starting the script -

The function starts the script once the user tries to run the ".sh" file.
The choice to give full permissions to the "memory_analysis" folder came to solve different problems with nesting a new folder in it further down the script, that occured because it gets created by "root" user.
Once conditions are met, the script continues to the "FILE_CHECK" function.

The use of the –p flag with the "mkdir" command, used in different parts of the script to avoid errors if a folder already exists.

The use of the –e flag with the "echo" command, used in many parts of the script to be able to use different color\font variables, and new-line logic as intended.

```
# This function starts the script. checks if the user is root or not, creates a main directory.
function START ()
{
    USER=$(whoami)
    if [ "$USER" != "root" ]
    then
        echo "You are not root. Exiting.."
        exit
    else
        figlet -f small "Forensics Investigation"

        #creating memory_analysis folder and giving permission read,write and execute permissions to avoid script errors.
        sudo mkdir -p ./memory_analysis
        sudo chmod 777 ./memory_analysis

        echo -e "${BOLD}${CYAN}\n***Pay attention! memory_analysis folder and report.txt will be cleared once you proceed with this step. consider a backup!***${NC}"

        FILE_CHECK
    fi

}
```

Checking if user's path input to memory file exist -

This function waits for an input with the file's path from the user.
It also uses an "if" statement along with the "-s" flag to check if the file exists.
Finally, It calls the "CHECK_APPS" function in case input is valid.

```
#this function checks whether the given file exist on the system. if not, shows a message that informs the user, and asks to try again until the correct file is provided.
function FILE_CHECK()
{
    echo "Please insert a full path to the image file:"
    read path
    echo "Checking if file exists..."
    if [ -s "$path" ]
    then
        echo -e "${GREEN}File exists!${NC}\n"
        CHECK_APPS
    else
        echo -e "${RED}File does not exist. Please try again!\n${NC}"
        FILE_CHECK
    fi
}
```

Checking tools existence and installing -

This function uses a local variable that nests different tools, this variable is later used in a “for loop” to check availability of each tool, making the code cleaner.
In case it is missing, it uses “if” and “elif” statements to check which tool is missing and install it.

Volatility 2.5 is not part of the $APPS variable, because it gets downloaded in a different method in this script.
In this case - my own github repository with the use of a raw link.

The use of “sleep 5” is there to give the user time to understand that something is going to be installed and give them a chance to use “CTRL-C”, to break out.

Finally, it calls the “CLEAR_RESULTS” and “MENU” functions.

```
#Function to check if carving and volatility tools exist on system and download\install if not.
function CHECK_APPS()
{
    APPS="bulk_extractor binwalk foremost strings" #defining APPS
    echo "Checking if the following tools exist on the system:"
    echo ""
#list volatility along with carving tools that are about to be checked.
#volatility downloaded from different repository, therefore, is not part of APPS.
    echo volatility
    for app in $APPS
    do
    echo "$app"
    done


    if [ -f $HOME/vol ]; then #checks for volatility_tool inside the folder. Downloading from a raw github link in case it's missing.
        echo -e "${GREEN}Volatility Tool is already installed. skipping...${NC}"
    else
        echo -e "\n${CYAN}Volatility Tool does not exist. Downlading...${NC}"
        sleep 5 #using sleep so the user have time to read the last message.
        wget https://github.com/CookieBotXL/justtakeit/raw/refs/heads/main/vol
        echo -e "${GREEN}Volatility Tool has been downloaded susccesfully!\n${NC}"
fi

#checks each app if installed, and installs it if missing. Uses dev/null to remove the "which" output.
    for app in $APPS
do
    if which "$app" > /dev/null
    then
        echo -e "${GREEN}$app is already installed. skipping...${NC}"
    else
        echo -e "${CYAN}$app does not exist. Installation is starting!${NC}"
        sleep 5
        if [ "$app" == "bulk_extractor" ]
        then
            apt install bulk-extractor
            echo -e "${GREEN}Bulk_Extractor has been installed susccesfully!\n${NC}"

        elif [ "$app" == "binwalk" ]; then
            apt install binwalk
            echo -e "${GREEN}Binwalk has been installed susccesfully!\n${NC}"

        elif [ "$app" == "foremost" ];  then
            apt install foremost
            echo -e "${GREEN}Foremost has been installed susccesfully!\n${NC}"

        elif [ "$app" == "strings" ]; then
            sudo apt install binutils
            echo -e "${GREEN}Strings has been installed susccesfully!\n${NC}"

        fi

    fi
done


    #only after done checking or installing tools - call CLEAR_RESULTS and MENU functions to proceed with operation.
    CLEAR_RESULTS
    MENU
}
```

Clearing previous results from output folder -

This function is being called by the “CHECK_APPS” function, after notifying the user.

```
#Function to clear all results folders, report file,
function CLEAR_RESULTS () {
    rm -rf ./memory_analysis/* #deletes all folders inside memory_analysis
    echo -e "${CYAN}\nmemory_analysis folder has been cleared!"
    > report.txt #clears report.txt
    echo -e "report.txt has been cleared!${NC}"
}
```

Main-Menu -

This function uses a “while true” loop so the user will always return to that stage after any tool finishes its work, until the user decides to manually exit and shut down the script.

Using the “case in” method to provide the user with the option to navigate through the menu.

The use of - echo “” - is done for aesthetic reasons to maintain spacings.
The reason it wasn’t used outside of “case” is to avoid that space if the choice is invalid. Again, purely for aesthetic reasons.

Based on the input, it calls the “CARVING”, “VOL” or “EXIT” functions.

```
#Function to open the Main Menu for the user.
function MENU()
{
    while true; do
    echo ""
    echo -e "What would you like to do with the file? [CHOOSE A NUMBER]\n [+] 1 - Carvers Menu\n [+] 2 - Run Volatility\n\n [+] 0 - Save Report And Exit"
    read -p "Enter your choice: " OPTIONS

    case $OPTIONS in

    #using echo "" on each step to make a new line ONLY when the option is valid, avoiding that space when the option is invalid.
    1)  echo ""
        Carving #use carving tools
    ;;

    2)  echo ""
        VOL #use volatility tool
    ;;

    0)  echo ""
        EXIT #use EXIT function to save a report and ZIP results.

    ;;

    *) echo -e "${RED}Invalid choice. Please try again.${NC}" #blocking user from choosing invalid option.
    esac
    done
}
```

Using Carving -

This function uses a case for different carving options with the use of the $path variable which stores the memory file. It also gives the option to run all carving tools or return to the Menu.

Each case uses “start_time=$(date +%s)” to update the “start_time” global variable that is being used later in the “DISPLAY_STATISTICS” function that is being called, to show how much time it took for the tool to finish, number of files, and output folder path.

This function calls other functions when needed, such as - “PCAP_QUESTION”, “STRING_SEARCH” and “MENU”.

Searching for a specific string -

This function is being called by the “Strings” tool which is used in the “CARVING” function.
It uses three “while true” loops:

1. **First loop** - asks the user if they want to search for a specific string output and will continue asking until the input is valid.
2. **Second loop –** In case the user chooses to search for a string in the first loop, this one will use “grep -i“ to perform the search and move to the third loop.
3. **Third loop** – This loop handles the case in which the user is being asked if he wants to perform another search after already doing one previously. This loop is also looking for a valid input from the user.
   In case the input is “y”, we move back to the 2nd loop and so on until the input for the 3rd loop is “n”,at which point the “return” command is being used to exit the function.

Important notice - Without the 2nd loop, the user would only be able to search once.

```
#function letting the user search for specific strings while using the strings tool.
function STRING_SEARCH() {
    while true; do
    # Ask if the user wants to search for a word in the strings output
    echo -n "Do you want to search for a specific string in the strings output? (y/n): "
    read search_choice

    case "$search_choice" in
    [Yy]*)
        #If user chooses 'y', ask for the string to search for and filter results
        while true; do
        echo -n "Enter the string to search for: "
        read search_string
        echo -e "Filtered results for '$search_string':\n"
        grep -i "$search_string" ./memory_analysis/strings_results/strings.txt

        #Ask if the user wants to search again
        while true; do
        echo -en "\nDo you want to search again? (y/n): "
        read search_again

        #Check for valid input in additional searches (y/n)
        case "$search_again" in

        [Yy]*)
            # Continue searching if 'y' is chosen
            break  # Break out of this inner loop, continue searching

        ;;

        [Nn]*)
            # Break out and return to menu if 'n' is chosen
            echo ""
            echo -e "Returning to Menu..."
            MENU
            return

        ;;

        *)
            # If invalid input, prompt again without moving forward
            echo -e "${RED}Invalid input. Please enter 'y' or 'n'.${NC}\n"

        ;;

        esac
        done
        done

        ;;

        [Nn]*)
                #If user chooses 'n' in the starting question, skip the search and return to the menu
                echo ""
                echo -e "Returning to Menu..."
                MENU
                return

        ;;

        *)
            #If invalid input for the starting question, ask again
            echo -e "${RED}Invalid input. Please enter 'y' or 'n'.${NC}\n"

        ;;

    esac
    done
}
```

Checking if memory file can be analyzed by Volatility -

This function uses an “if” statement that checks if the memory file doesn’t have one of the following extensions – .mem, .raw, .dmp, to define if it is a valid memory file prior to continuing or moving back to Main-Menu in case it isn’t.
In this case it uses “=~” which is the regular expression matching operator that allows to compare a string against a regular expression pattern.
To simplify - The expression `[[ "$path" =~ \.(mem|raw|dmp)$ ]]` checks if the string in `$path` ends with one of the file extensions: `.mem`, `.raw`, or `.dmp`.

```
#This function checks if the file can be analyzed by Volatility tool.
function VOL_CHECK(){
    echo "Analyzing $path..."
    if [[ ! "$path" =~ \.(mem|raw|dmp)$ ]]; then #If it's not a .mem, .raw, or .dmp file, print error and go back to MENU.
        echo -e "${RED}The file '$path' is not a valid memory image file. Only .mem, .raw, or .dmp files can be analyzed by Volatility.${NC}"
        MENU
    fi
    # If it's a valid memory image file, continue with the script
    echo -e "${GREEN}The file '$path' is a valid memory image file. Continuing...${NC}"
}
```

Running Volatility tool -

This function calls the “VOL_CHECK” function.
It uses different local variables such as $PROFILE, to find, display and later use the memory profile of the current memory file by running “imageinfo” while using text manipulation to get only the relevant info and save it.

Later, it uses an “if” statement to target “pslist” and “connscan” plugins and display them in the terminal along with saving them into output text files just like the rest of the plugins.
It uses the “tee” command to simultaneously execute that logic of displaying the output on one hand and saving it to a text file, on the other.

```
#Function that runs Volatility Tool plugins.
function VOL()
{
    VOL_CHECK
    sudo chmod +x ./vol #making sure volatility has permission to run(doesn't run wihtout it if been downloaded from github)
    start_time=$(date +%s)
    mkdir -p ./memory_analysis/volatility_results
    PROFILE=$(./vol -f "$path" imageinfo | grep Suggested | awk -F',' '{print $1}' | awk -F':' '{print $2}' | sed 's/ //g')
    echo -e "${BOLD}${CYAN}OS: $PROFILE${NC}"
    PLUGINS="hivelist cmdline driverscan mftparser svcscan hashdump userassist dlllist shutdowntime psscan pslist connscan  "
    KEYS="Run RunOnce"
    REG_KEY_PATH="Software\\Microsoft\\Windows\\CurrentVersion"


    #running dumpregistry to extract all registry data from memory.
    echo ""
        sudo mkdir -p ./memory_analysis/volatility_results/regdump_results
        echo "Using dumpregistry against the file..."
        sudo ./vol -f "$path" --profile=$PROFILE dumpregistry -D ./memory_analysis/volatility_results/regdump_results > /dev/null 2>&1


    #using printkey with a variable to extract specific registry keys into seperate text files.
    for key in $KEYS
    do
        echo "Using printkey $key against the file..."
        ./vol -f "$path" --profile=$PROFILE printkey -K "$REG_KEY_PATH\\$key" > /dev/null 2>&1 > "memory_analysis/volatility_results/$FILE_NAME/results_printkey_$key.txt"
    done

    for plugin in $PLUGINS
    do
        echo "Using $plugin against the file..."
        if [[ "$plugin" == "pslist" || "$plugin" == "connscan" ]]; then
        echo -e "${BOLD}${CYAN}Top priority plugin has been detected. Output will be displayed here as well:${NC}"
        sleep 2
        #Display output in terminal and save to file with the use of tee
        ./vol -f "$path" --profile=$PROFILE $plugin | tee "memory_analysis/volatility_results/$FILE_NAME/results_$plugin.txt"
        else
        #For other plugins, save output to file silently
        ./vol -f "$path" --profile=$PROFILE $plugin > /dev/null 2>&1 > "memory_analysis/volatility_results/$FILE_NAME/results_$plugin.txt"
        fi
    done

        DISPLAY_STATISTICS "memory_analysis/volatility_results" "Volatility"
}
```

Asking user to search for network traffic (PCAP) file -

This function uses the same “while true” with “case” logic as before.
It looks for a Y\y\N\n input from the user, instead of numbers to make the experience more intuitive.

This function is being called by the “Bulk_Extractor” when the “CARVING” function is running.

```
#Function used to ask user if they want to search for network traffic.
function PCAP_QUESTION () {
    while true;do
    echo -n "Do you want to check if a Network traffic file was extracted and get its details? (y/n): "
    read REPLY


    case $REPLY in

    [Yy])
        PCAP #Calls a seperate function to check if PCAP file exists and providing its location and size.
        break

    ;;

    [Nn])
        echo -e "Skipping Network traffic file check and cotinuing... \n"
        break

    ;;

    *)
        echo -e "${RED}Invalid choice. Please try again.${NC}\n"

    ;;
        esac
        done

}
```

Providing network traffic location and size -

This function can be called by the “PCAP_QUESTION”
It is using the “du” (disk-usage) command to check file size, along with the “-h” flag to make the output easier to understand in case the size is 1000kb or larger. (e.g. 1000kb will show as 1MB)
The function also uses some text manipulation to avoid repetition of the path to the PCAP file, as it’s being shown to the user by the previous line of code.

```
#Function checks for a PCAP file informaion if it exists after extracting, and providing that info.
function PCAP(){
    PCAP_FILE="$OUTPUT_DIR/packets.pcap"
        if [ -f "$PCAP_FILE" ]; then # checks if file exists.
        echo -e "${GREEN}\nNetwork traffic file found!${NC}"
        #showing location of the PCAP file. using -e flag to read backslashes in order to paint text in red for user better visibility and user convenience.
        echo -e "${BOLD}${CYAN}Location: $PCAP_FILE${NC}"

        #using disk usage with human readble flag to display the size and using awk to prevent repeating path output
        echo -e "${BOLD}${CYAN}The size of the file is: $(du -h "$PCAP_FILE" | awk '{print $1}')${NC}\n"
    else
        echo -e "${RED}No network traffic file (packets.pcap) found.\n${NC}"
        fi
}
```

Live output of each tool statistics -

This function is being called by all "CARVING" plugins and "VOL" function.

By accepting arguments(output directory and the name of running tool) and using "end_time" timestamp to subtract from "start_time" variables of each tool, it calculates how much time passed between two events to provide the user with the time it took to run each tool.

It also uses a "find" command along with a "-type f" flag to count only the number of files, preventing count of folders, to provide information about how many files were created by running each tool and show this data in the terminal.

```
#Function to display statistics(number of files and the time it took to run)
function DISPLAY_STATISTICS() {
#Accept output directory and tool name as an arguments when running the function.
  local tool_output_dir=$1
  local tool_name=$2
  #Calculate the time taken for the analysis
  end_time=$(date +%s) #using timestamp to compare with "start_time" variable.
  time_taken=$((end_time - start_time))

  #count the number of files in the memory_analysis directory.
  num_files_carv=$(find "$tool_output_dir" -type f | wc -l)

  #Display statistics to user.
  echo -e "${GREEN}--- $tool_name Analysis Completed ---${NC}"
  echo -e "${GREEN}Time of analysis: $time_taken seconds${NC}"
  echo -e "${GREEN}Number of files found: $num_files_carv${NC}"
  echo -e "${GREEN}files were saved to: $tool_output_dir/${NC}\n"

}
```

Compressing all results into a ZIP file -

This function defines a timestamp to be added to the ZIP file name for two reasons.

1. Prevent from overwriting previous ZIP files that might exist in the same folder by giving it a unique name.
2. Making it easier for the analyst to find the desired ZIP by date and time of creation.

It is being called by the "EXIT" function"

```
#Function to zip all result folder + report.txt.
function ZIP_RESULTS (){
        #creating a small loading so the user will have time to read the following message before zipping starts -
        echo -e "\n${BOLD}${CYAN}WAIT! Before you go, we will pack the results nicely in a ZIP-file for you!"
        echo -e "ZIPPING your files, please wait!${NC}"
        #zipping with -q flag to silence default output and using a timestamp to avoid overwriting the prvious zip and give creation date and time indication while maintaining the same file name.
        timestamp=$(date +%Y%m%d_%H%M%S)
        zip -rq memory_analysis_"$timestamp".zip memory_analysis report.txt
        echo -e "${GREEN}\nWe're Done! Everything is packed nicely into ./memory_analysis_{$timestamp}.zip${NC}"
}
```

Exiting the script -

This function creates the "report.txt" file by using the "find" with a "type -f" flag twice.
First time it does it to count the total number of files in the output folder, while the second time it is used without the "wc -l" command, to write down the paths for all existing files in that folder into "report.txt".

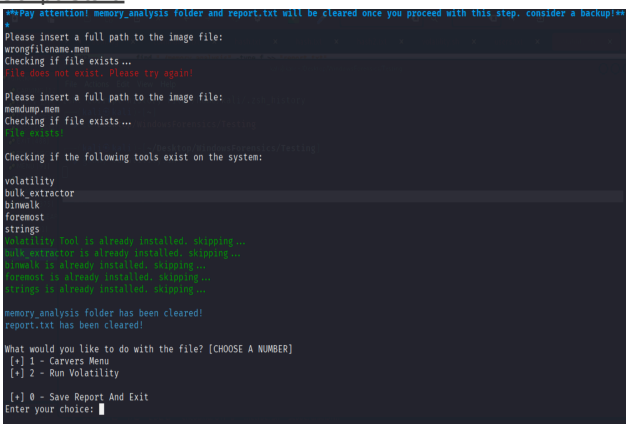
Finally it calls the ZIP_RESULTS function prior to exiting.

```
#Exit function including creating a report file and creating a ZIP file with all the results.
function EXIT () {
        total_results_count=$(find "./memory_analysis" -type f | wc -l)
        echo -e "Report generated on: $(date)\n" > "report.txt"
        echo -e "Total number of files found: $total_results_count\nHere are everything that you have found:\n" >> "report.txt"
        find "./memory_analysis" -type f >> "report.txt"

        echo -e "${GREEN}Your report was saved to ./report.txt${NC}"

        ZIP_RESULTS #Call ZIP_RESULTS function before exiting script.

        echo -e "\n${CYAN}See you later, aligator!!! ^_^${NC}"
        exit 0 #exiting script
}
```

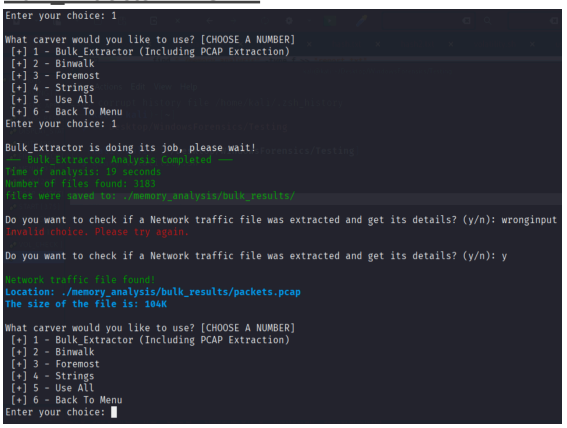
Demonstrating Script Performance -

This part will show **some** of the script functionality in work.

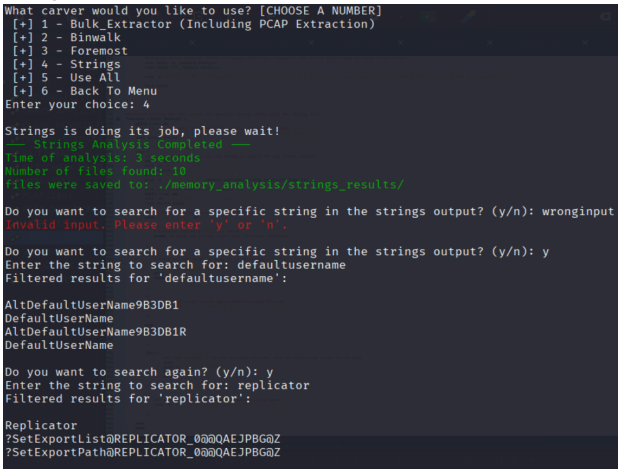
Script Start:



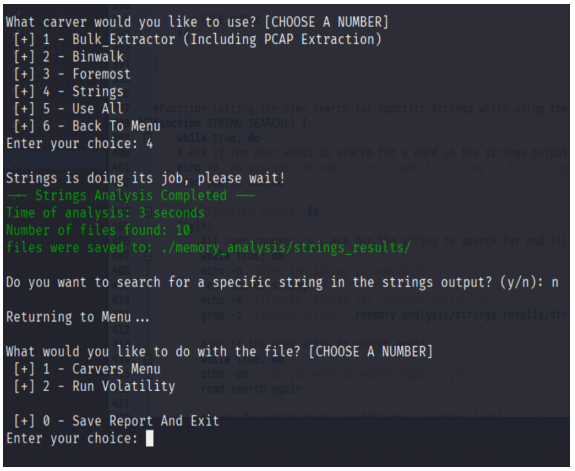
Bulk_Extractor + PCAP:



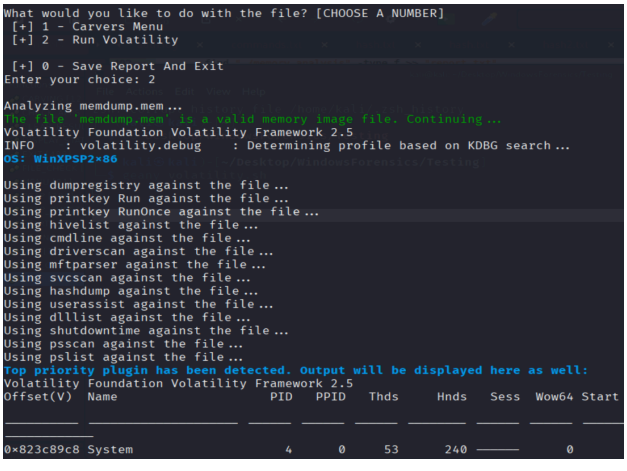
Strings+Search:



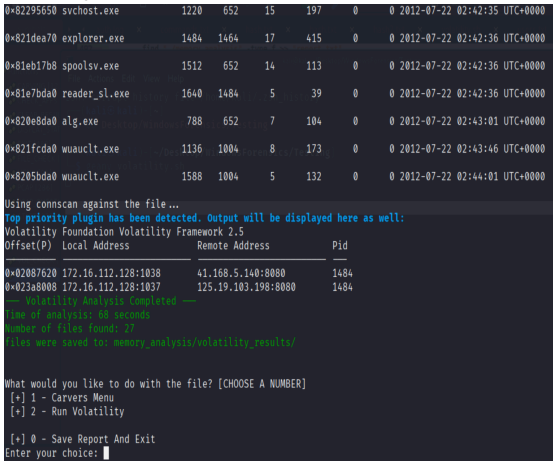
Strings without search:



Volatility:



Volatility continues:



Exit script + zip + report: