HOCHSCHULE
CAMPUS WIEN
UNIVERSITY OF
APPLIED SCIENCES

**Assignment 3**

Digital Forensics

# Memory Forensics

**Author:**
Philip Magnus

**Student identification number:**
c2410537022

**Date:**
19th of December 2025

# Contents

# 1. Introduction

## 1.1. Goal of the Assignment

The goal of this assignment is split into two parts:

1. Acquire a memory dump from a running system of your choice
   - include a very specific and unique artefact that makes your skills verifyable, such as a specific running process or a specific open network connection!
2. Analysing a RAM Dump we received from the instructor
   - Answeing questions regarding the analysis

## 1.2. Setup

- Ubuntu 24.04 LTS (32BG RAM)
- AVLM `v0.14.0` to aquire RAM (https://github.com/microsoft/avml)
- Volatility 3 `v2.27.0`
- dwarf2json `v0.9.0`

# 2. Assignment - Part 1

## 2.1. Setup and Acquisition of RAM Dump

First I created a RAM dump of my running Ubuntu 24.04 LTS system using `AVML`.

To comply with the exercise I added a unique artifact by opening a `gif` image, with the name `giphy-3348127193.gif` (see Figure 1), in the default image viewer application.

Figure 1: Unique artefact - opened gif image in default image viewer.

This was not the only process running on the system, I also had a Vivaldi browser window open with multiple tabs. For this I downloaded the latest pre compiled binary from the official `AVML` GitHub repository and executed the following command with `sudo` privileges to create a memory dump named `ram.dump`:

```
$ sudo ./avml ram.dump
```

Listing 1: Creating a RAM dump using AVML.

This resulted in a 32GB RAM dump file named `ram.dump` in the current working directory. I also generated the `sha256` checksum of the dump file to ensure integrity during analysis:

```
$ sha256sum ram.dump

992f44d0995022e472f3e23049b27879de01a78218651f894656ce58260391e1  ram.dump
```

Listing 2: Sha256 checksum of ram.dump.

## 2.2. Set up Volatility 3 for Analysis

For the analysis of the acquired RAM dump I used `Volatility 3`. First I installed `Volatility 3` using `pip`:

```
$ python -m venv .venv && source .venv/bin/activate

$ pip install git+https://github.com/Abyss-W4tcher/volatility3.git@issue_1761_module_sect_
attr_fix
```

Listing 3: Installing Volatility 3.

Note: Volatility3 needs to be installed from the branch corresponding with PR1773 due to issue 1883, which prevents analysing of dumps of current `Linux` kernel versions)

```
$ chmod -w+r ram.dump
```

Listing 4: Make dump read-only.

As shown in Listing 4, I made the dump file read-only to prevent accidental modification during analysis. Using the `vol -f ram.dump banners`, see Figure 2, command I checked for the `Linux` version of the acquired dump, this is necessary to create the correct symbols table for analysis.

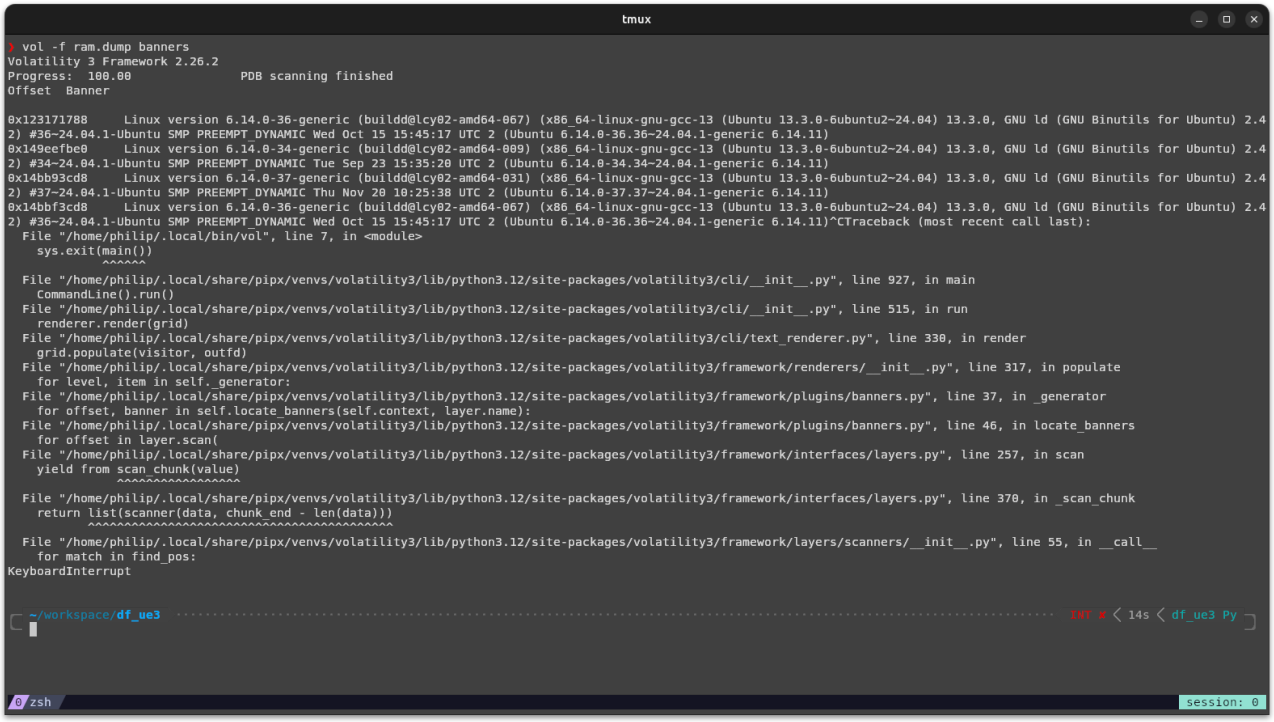


Figure 2: vol -f ram.dump banners output

The newest `Linux` kernel version found in the dump is `6.14.0-36-generic`. The other versions are previous kernels that are left over after system updates.

Next I created the correct symbols table which is crucial for the analysis of the dump. First I installed the kernel debug symbols using `apt`, see Listing 5.

```
$ sudo apt install linux-image-amd64-dbg -y
```

Listing 5: Downloading kernel debug symbols.

Next I downloaded and compiled `dwarf2json` as described in the git repository (dwarf2json). With the command shown in Listing 6 I generated the symbols table for `Volatility 3`.

```
$ ./dwarf2json linux --elf /usr/lib/debug/boot/vmlinux-6.14.0-36-generic | xz -c >
linux-6.14.json.xz
```

Listing 6: Creating symbols table for Volatility 3 using dwarf2json.

I then placed the generated `linux-6.14.json.xz` file in the `volatility3/sybols` directory to make it available for `Volatility 3`, see Listing 7.

```
$ cp ~/workspace/dwarf2json/linux-6.14.json.xz .venv/lib/python3.12/site-packages/
volatility3/symbols
```

Listing 7: Creating symbols table for Volatility 3 using dwarf2json.

## 2.3. Analysis of RAM Dump

### 2.3.1. Analysis of Running and Terminated Processes

With the setup complete I started the analysis of the RAM dump. First I executed the following two commands:

- `vol -f ram.dump linux.pslist`
- `vol -f ram.dump linux.psscan`

First I listed the interesting running processes using the `linux.pslist` plugin, see Figure 3. Here I could already identify the opened image viewer process `eog` (Eye of GNOME), as well as the open Vivaldi browser processes `vivaldi-bin`.

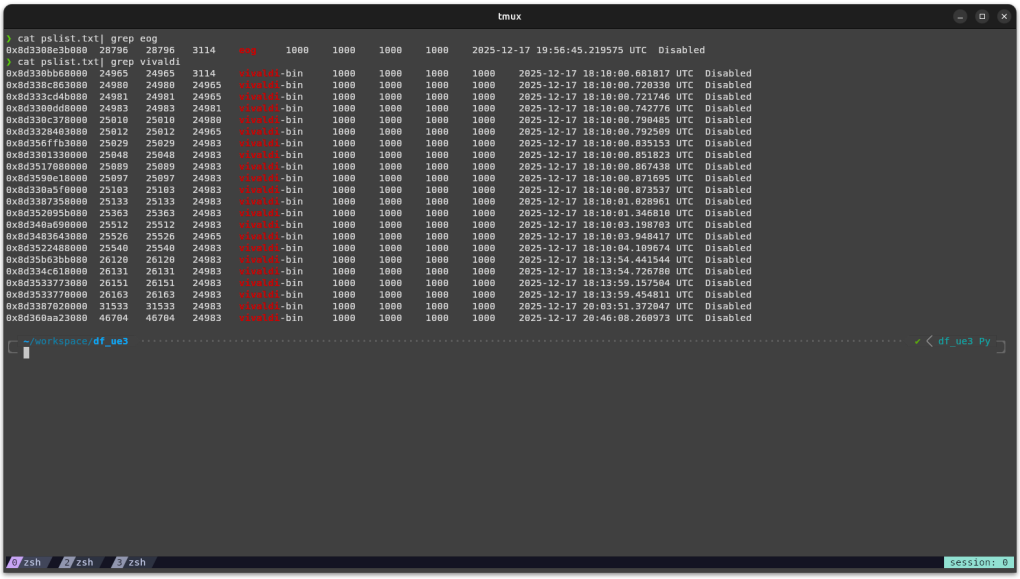


Figure 3: vol -f ram.dump linux.pslist output

Next I used the `linux.psscan` plugin to find terminated processes that are still present in memory, see Figure 4. Here I could again identify multiple `eog` and `vivaldi-bin` processes that were terminated or intentionally “unlinked” but still present in memory.

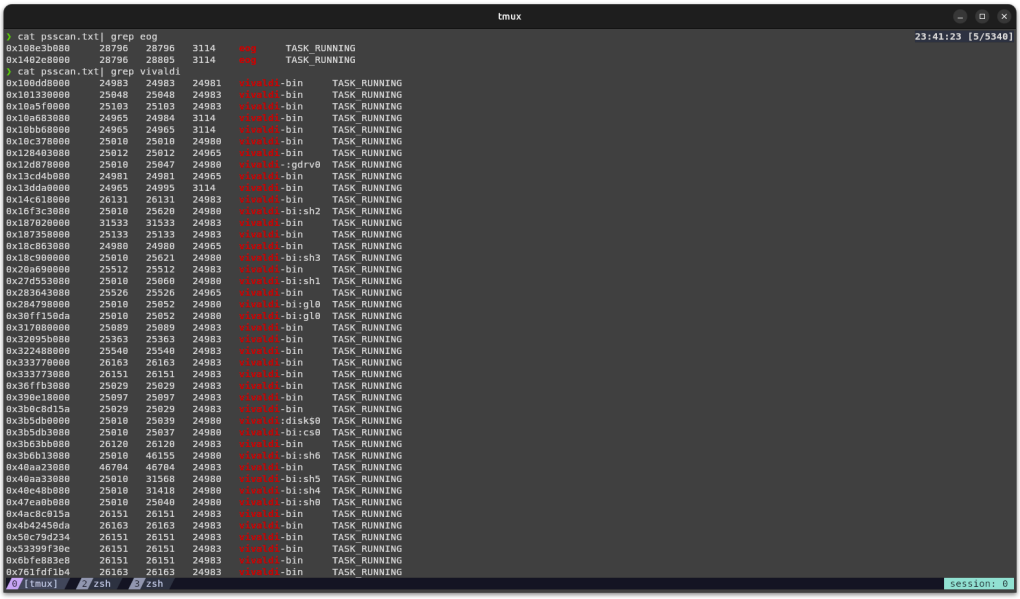


Figure 4: vol -f ram.dump linux.psscan output

All in all the `pslist` plugin found $489$ processes, while the `psscan` plugin found $2775$ processes.

### 2.3.2. Analysis of Open Network Connections

With the command shown in Listing 8 I listed all open network connections using the `linux.socket.sockstat` plugin.

```
$ vol -f ram.dump linux.sockstat.Sockstat
```

Listing 8: Listing open network connections using linux.sockstat.Sockstat plugin.

The output of the command is shown in Figure 5. Here I can see all the open sockets in different states, including `LISTEN`, `ESTABLISHED` and `CLOSE_WAIT`.

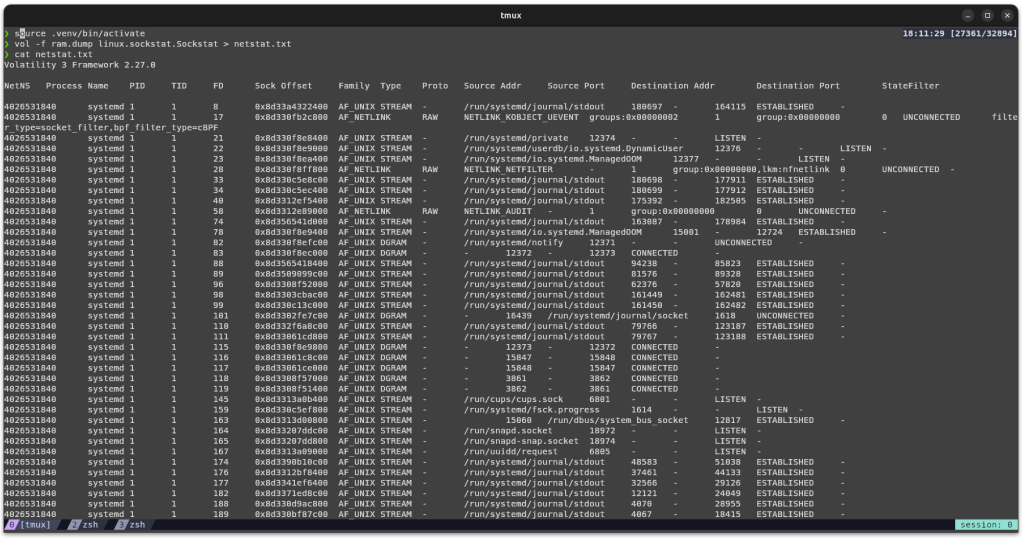


Figure 5: Sockstat output.

To narrow down the results I filtered for the sockets of the `vivaldi-bin` processes using the command shown in Figure 6. This shows that the process `vivaldi-bin` handles around 410 open sockets.

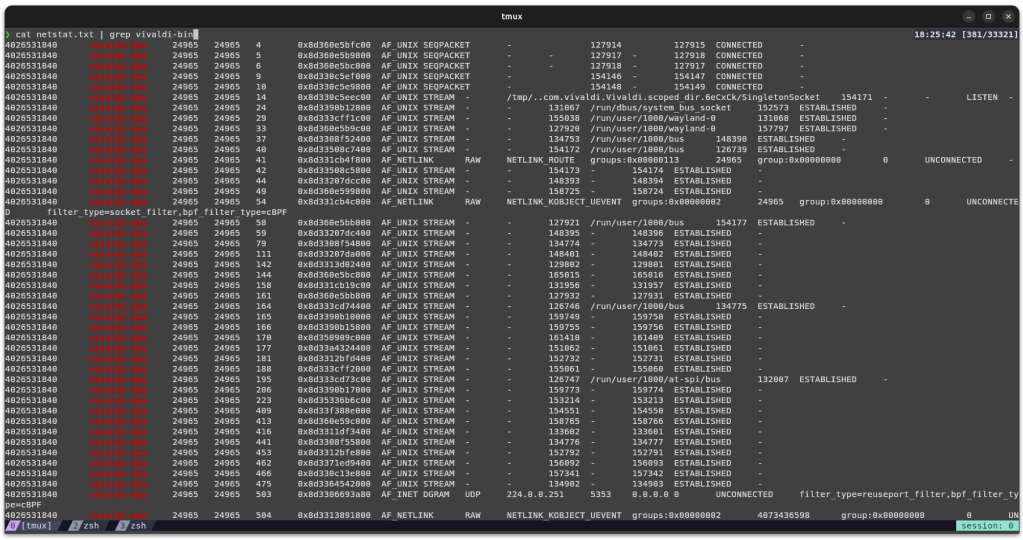


Figure 6: Sockstat output.

### 2.3.3. Finding the Unique Artefact in Memory

To find the unique artefact I searched for the string `giphy-3348127193.gif` in memory using the command shown in Listing 9.

```
$ strings ram.dump | grep giphy-3348127193.gif

  giphy-3348127193.gif
  /home/philip/Pictures/giphy-3348127193.gif
  [...]
```

Listing 9: Searching for unique artefact in memory.

The simple string search already revealed the full path of the opened image, see Listing 9. Opening the file path in the default image viewer application indeed shows the correct `gif` image, see Figure 7.

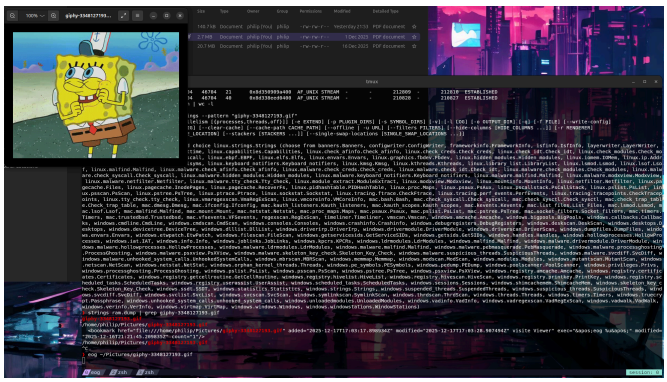

Figure 7: Unique artefact found in memory.

# 3. Assignment - Part 2

First we downloaded and unzipped the provided memmory dump file. The Sha256 checksum of the file is: `fee4a87527509ed8a67c51a2b3e21a74ae52739e0d69020312180339cfd79e3b`.

## 3.1. Basic Memory Analysis

I first collected some basic information about the memory dump using the `windows.info` plugin, see Listing 10.

```
$ vol -f physmem.raw windows.info

  Volatility 3 Framework 2.27.0
  Progress:  100.00               PDB scanning finished
  Variable        Value

  Kernel Base     0xf80420a00000
  DTB     0x1ae000
  Symbols symbols/windows/ntkrnlmp.pdb/7C85537A944BEF2014AE251FDEA1C590-1.json.xz
  Is64Bit True
  IsPAE   False
  layer_name      0 WindowsIntel32e
  memory_layer    1 FileLayer
  KdVersionBlock  0xf804216099a0
  Major/Minor     15.22621
  MachineType     34404
  KeNumberProcessors      2
  SystemTime      2023-01-09 22:17:11+00:00
  NtSystemRoot    C:\Windows
  NtProductType   NtProductWinNt
  NtMajorVersion  10
  NtMinorVersion  0
  PE MajorOperatingSystemVersion  10
  PE MinorOperatingSystemVersion  0
  PE Machine      34404
  PE TimeDateStamp        Mon Jul  5 20:20:35 2100
```

Listing 10: vol -f physmem.raw windows.info

The output shows that the memory dump is from a Windows 11 system (Build `22621`) running on a 64-bit architecture with 2 processors. The system time at the time of the memory dump was `2023-01-09 22:17:11+00:00`.

## 3.2. Finding SIDs and Credentials

Using the command shown in Listing 11 I extracted all SIDs found in the memory dump.

```
$ vol -f physmem.raw windows.getsids.GetSIDs > sids.txt
```

Listing 11: vol -f physmem.raw windows.getsids.GetSIDs

The output file `sids.txt` contained multiple SIDs, see Figure 8.

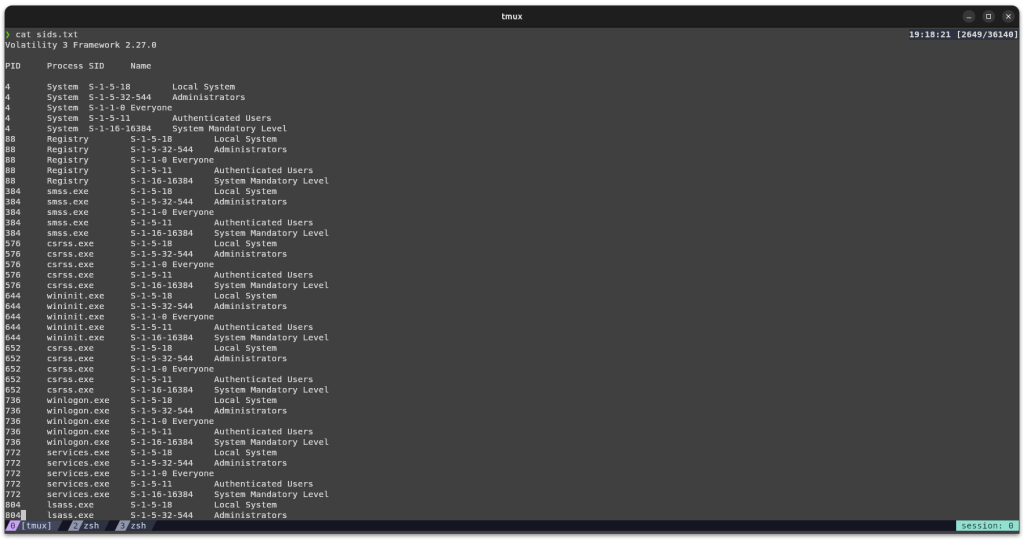


Figure 8: Extracted SIDs from memory dump

Next I tried to extract credential hashes from the dump, see Listing 12.

```
$ vol -f physmem.raw windows.registry.hashdump

Volatility 3 Framework 2.27.0
Progress:  100.00               PDB scanning finished
User    rid     lmhash  nthash
WARNING  volatility3.plugins.windows.registry.hashdump: Hbootkey is not valid
```

Listing 12: vol -f physmem.raw windows.registry.hashdump

As shown in the output, I was not able to extract any credential hashes from the memory dump since the `Hbootkey` is not valid. We were provided a valid Hbootkey in the assignment description, so I added an early return statement to the plugins `Python` file, as shown in Listing 13, returning the valid `Hbootkey`. The plugin can be found in the Volatility 3 installation directory under `volatility3/plugins/windows/registry/hashdump.py`.

```
@classmethod
def get_hbootkey(
    cls, samhive: registry_layer.RegistryHive, bootkey: bytes
) -> Optional[bytes]:
    sam_account_path = "SAM\\Domains\\Account"
return b"\xBC\xF8\x54\x8E\xAE\x42\x90\x0B\xED\xA0\xF1\x50\xE1\x65\x04\xB5"
```

Listing 13: Fixing Hbootkey retrieval in windows.registry.hashdump plugin

With the correct Hbootkey in place I re-ran the `windows.registry.hashdump` plugin, see Listing 14.

```
$ vol -f physmem.raw windows.registry.hashdump

Volatility 3 Framework 2.27.0
Progress:  100.00               PDB scanning finished

User    rid     lmhash  nthash
```

```
Administrator 500 aad3b435b51404eeaad3b435b51404ee 31d6cfe0d16ae931b73c59d7e0c089c0
Guest 501 aad3b435b51404eeaad3b435b51404ee 31d6cfe0d16ae931b73c59d7e0c089c0
DefaultAccount 503 aad3b435b51404eeaad3b435b51404ee 31d6cfe0d16ae931b73c59d7e0c089c0
WDAGUtilityAccount 504 aad3b435b51404eeaad3b435b51404ee 46cbf54c64b31b778c5019c7a4c90970
Spongebob 1001 aad3b435b51404eeaad3b435b51404ee d8ce5e07ae6dd698222c75def3dc23f6
```

Listing 14: vol -f physmem.raw windows.registry.hashdump after Hbootkey fix

The table below lists the extracted credential hashes from the memory dump in a more organized format.

| User | RID | LM Hash | NT Hash |
|---|---|---|---|
| Administrator | 500 | aad3b435b51404eeaad3b435b51404ee | 31d6cfe0d16ae931b73c59d7e0c089c0 |
| Guest | 501 | aad3b435b51404eeaad3b435b51404ee | 31d6cfe0d16ae931b73c59d7e0c089c0 |
| DefaultAccount | 503 | aad3b435b51404eeaad3b435b51404ee | 31d6cfe0d16ae931b73c59d7e0c089c0 |
| WDAGUtilityAccount | 504 | aad3b435b51404eeaad3b435b51404ee | 46cbf54c64b31b778c5019c7a4c90970 |
| Spongebob | 1001 | aad3b435b51404eeaad3b435b51404ee | d8ce5e07ae6dd698222c75def3dc23f6 |

The first three NT hashes all the well-known blank value. My attempts to recover Spongebob's password using hashcat with different dictionaries or brute-forcing mode were not successful.

## 3.3. Processes and Network Connections

Using the `windows.pslist` plugin I listed all running processes found in the memory dump, see Figure 9.

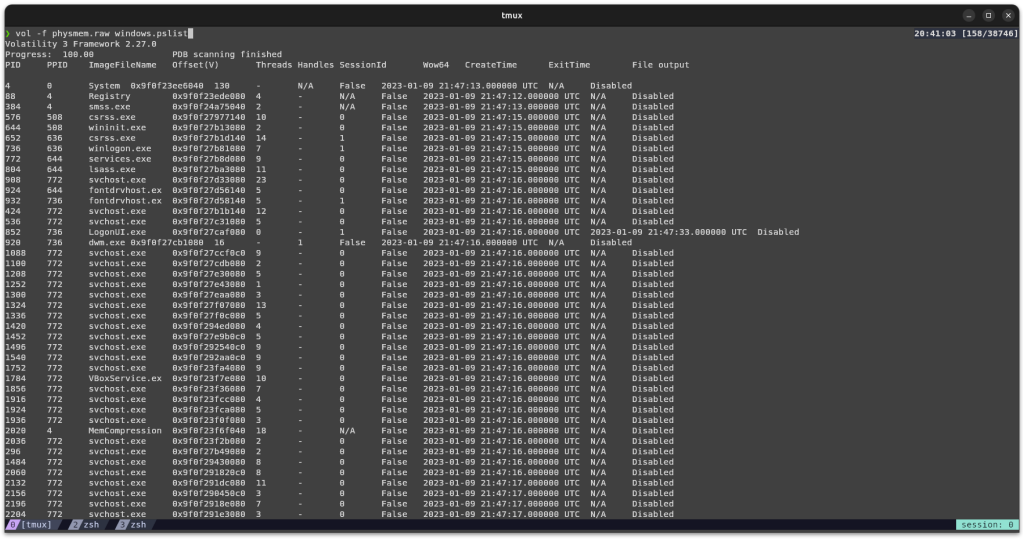


Figure 9: vol -f physmem.raw windows.pslist

The output shows the first process was startet at `023-01-09 21:47:13.000000 UTC`, which is approximately 30 minutes before the memory dump was aquired at `2023-01-09 22:17:11+00:00`, as shown in the `windows.info` output. The list of processes also contains multiple active `firefox.exe`, `tor.exe` and `msedge.exe` processes. There are also an open `notepad.exe` process, `msteams.exe` and `Lwinpmem_mini_x` which is likely the memory acquisition tool.

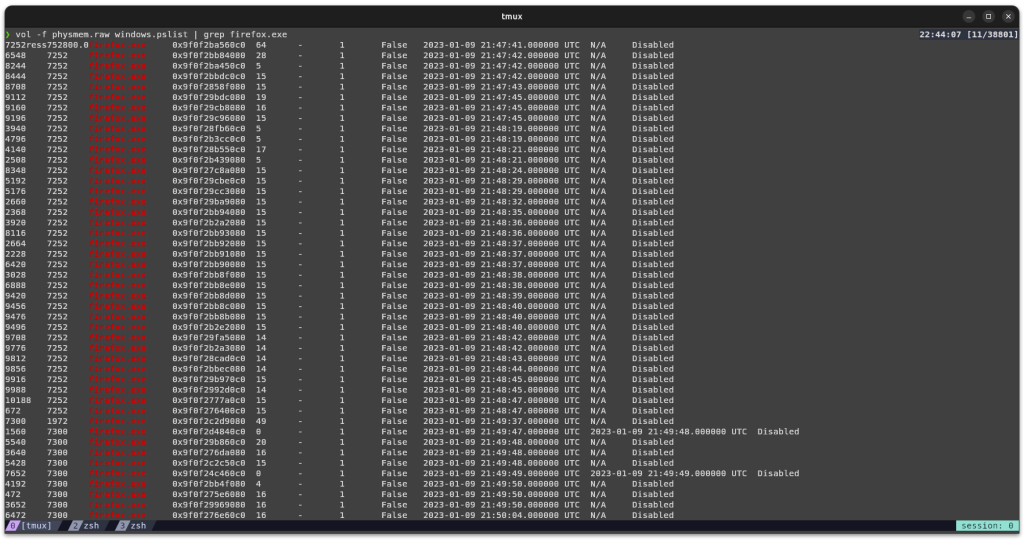


Figure 10: Example firefox.exe processes

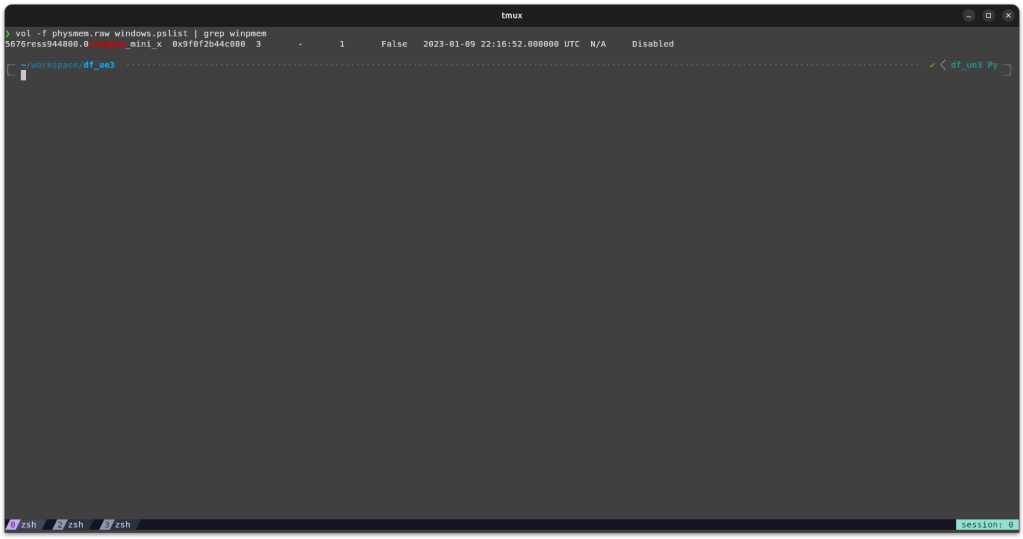


Figure 11: WinPmem process

Running the `windows.psscan.psscan` plugin did not reveal any additional information.

Running `windows.netstat.Netstat` and `windows.netscan.Netscan` listed all open network connections, see Figure 12 and Figure 13.

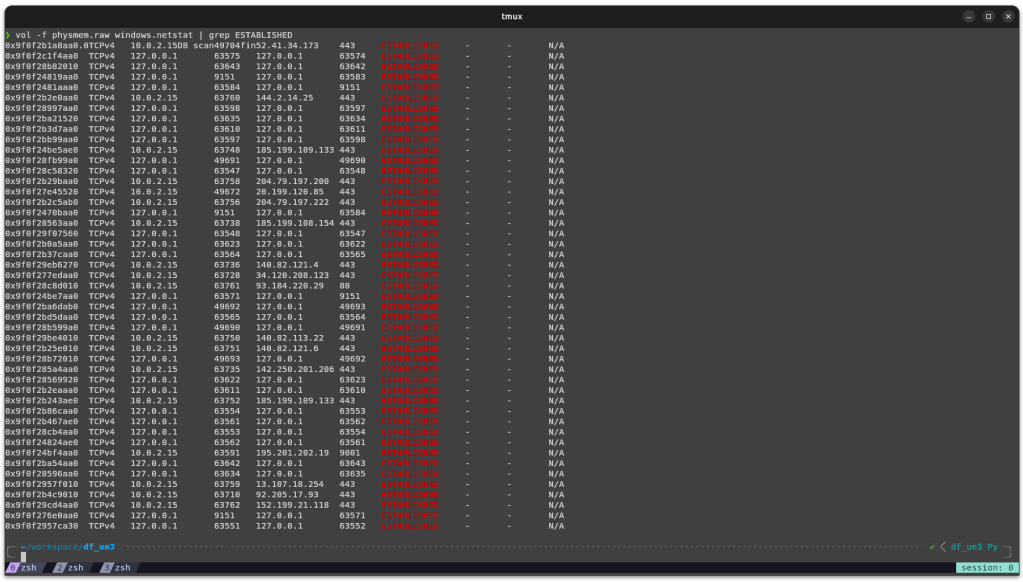


Figure 12: vol -f physmem.raw windows.netstat output

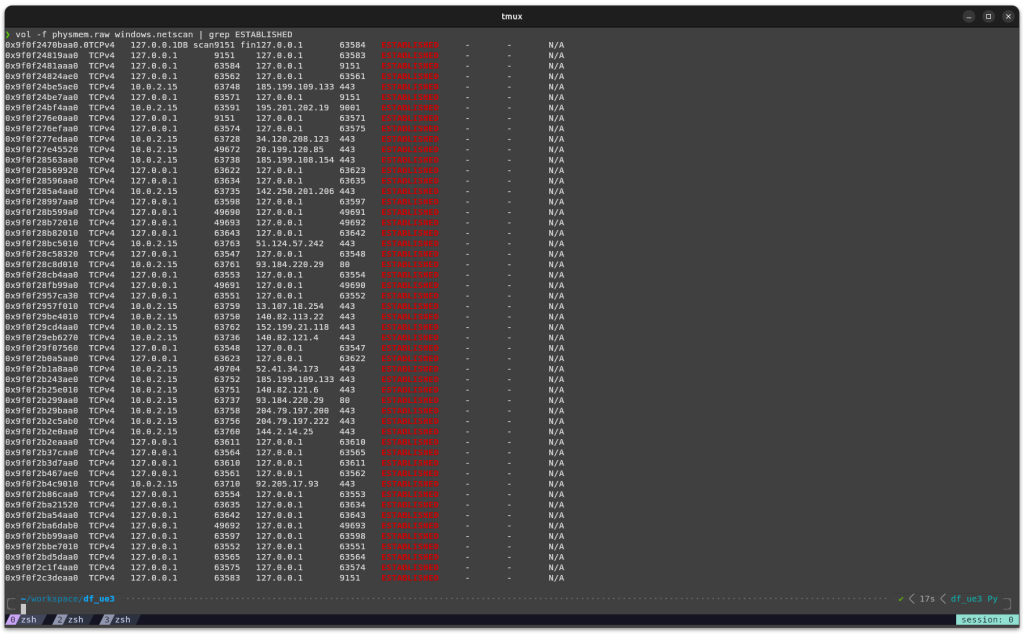


Figure 13: vol -f physmem.raw windows.netscan output

For better readability I only show the connections with the state `ESTABLISHED` in Figure 12 and Figure 13.

Next I looked at the commandline arguments of the running processes using the `windows.cmdline.CmdLine` plugin, see Listing 15.

```
$ vol -f physmem.raw windows.cmdline.CmdLine
```

Listing 15: vol -f physmem.raw windows.cmdline.CmdLine

As shown in the output, I was able to see a variety of running processes with different commandline arguments, see Figure 14.

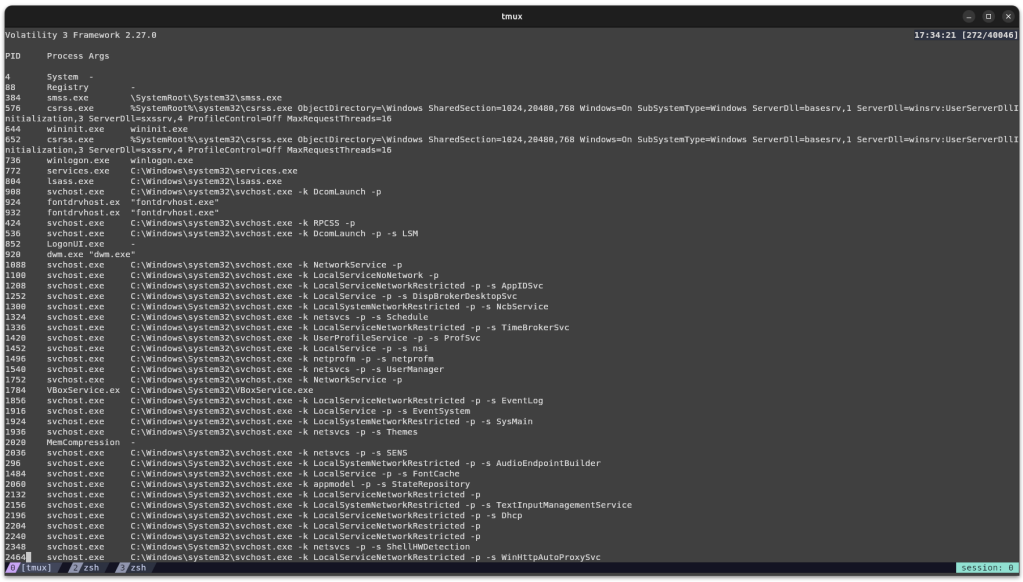


Figure 14: vol -f physmem.raw windows.cmdline.CmdLine output

Filtering the output let me link some `msedge.exe` processes to running `msteams.exe` processes, Edge seems to be used as an embeeded browser to display Teams content.

Most interesting was the running `notepad.exe` process, which had the commandline argument `C:\Users\Spongebob\Desktop\password.txt.txt`, see Figure 15. This indicates that the user `Spongebob` had a text file named `secret.txt` open on his desktop at the time of the memory dump.

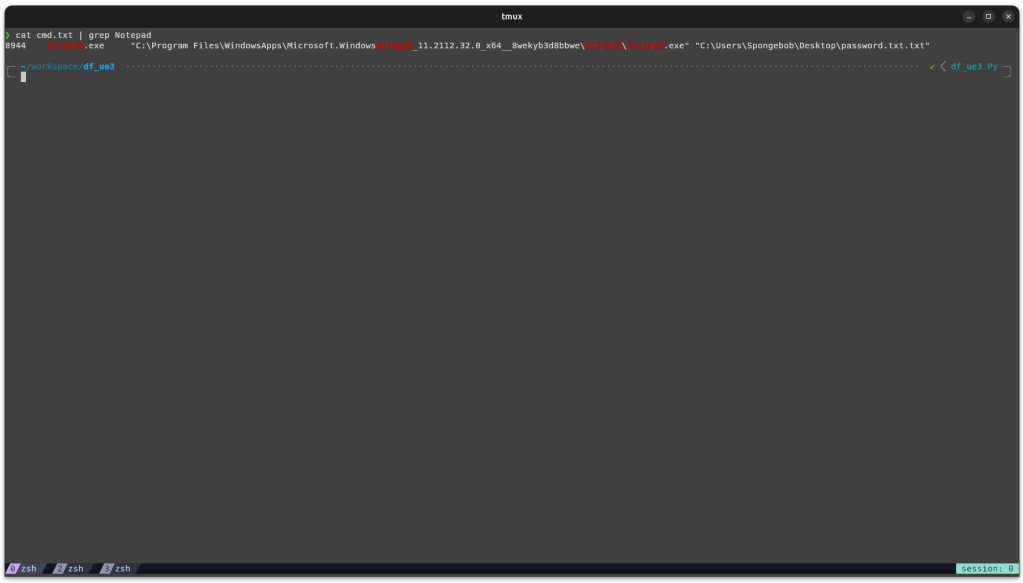


Figure 15: Notepad command line argument

## 3.4. Analyzing Files in the Dump

To analyze files present in the memory dump I used the `windows.filescan.Filescan` plugin to list all file objects found in memory, see Figure 16.

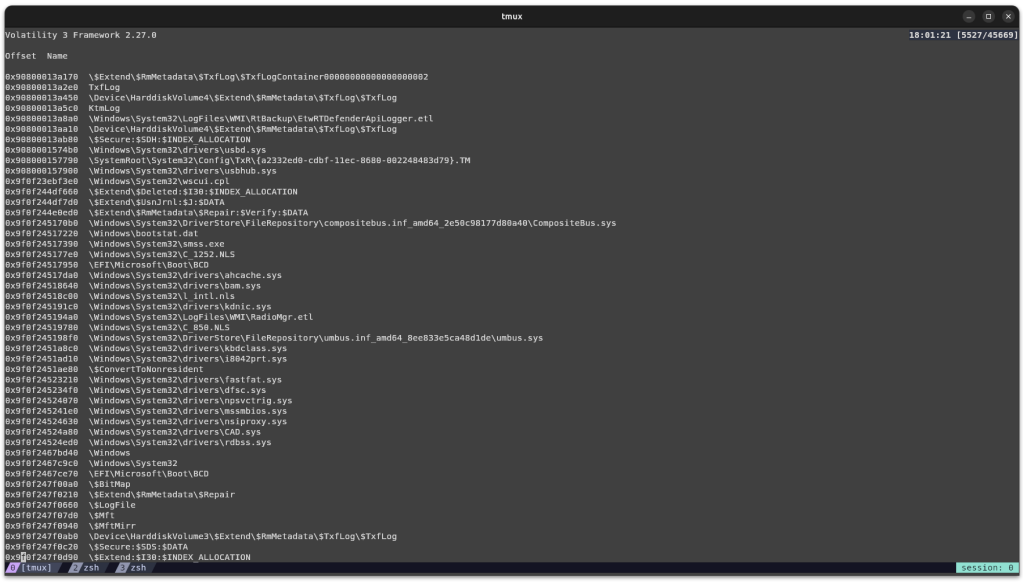


Figure 16: Filescan output

I dumped the files with the command shown in Listing 16.

```
$ vol -f physmem.raw -o files/ windows.dumpfiles.DumpFiles
```

Listing 16: Dumping files from memory dump

The dumped files contain quite a lot of cookie databases and `-wal` files which are used by modern browsers to store session data, visited domains can be viewed this way.

Looking at those dumped `sqlite` databases I was able to see some of the databases containing cookies, see Figure 17.


Figure 17: Dumped cookie databases

Looking at all the dumped databases I could also see databases containing visited URLs, see Figure 18.

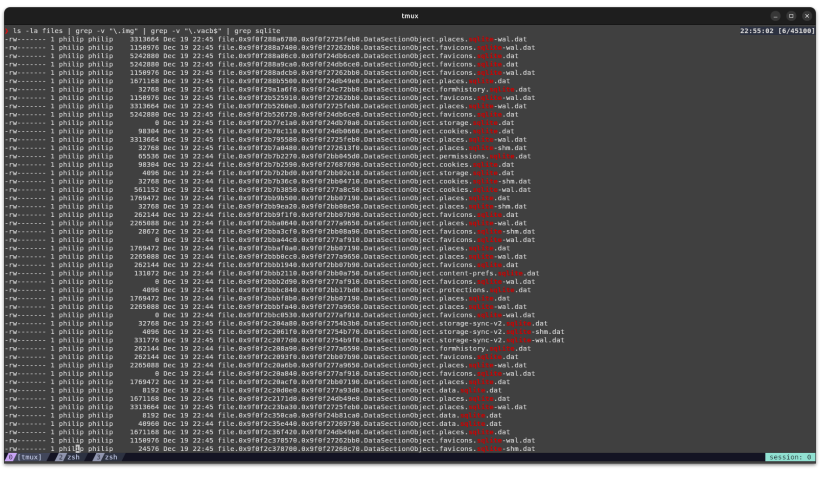

Figure 18: Dumped sqlite databases

Using the `strings` command I searched the databases for interesting URLs, I could see that close to the memory dump the user:

- Searched for `WinPmem`
- Visited the `WinPmem` Github
- Downloaded the `Tor Browser`
- Visited Hacker News
- And watched a YouTube video `Rick Astley - Never Gonna Give You Up`

See Figure 19 for a screenshot displaying the console output.

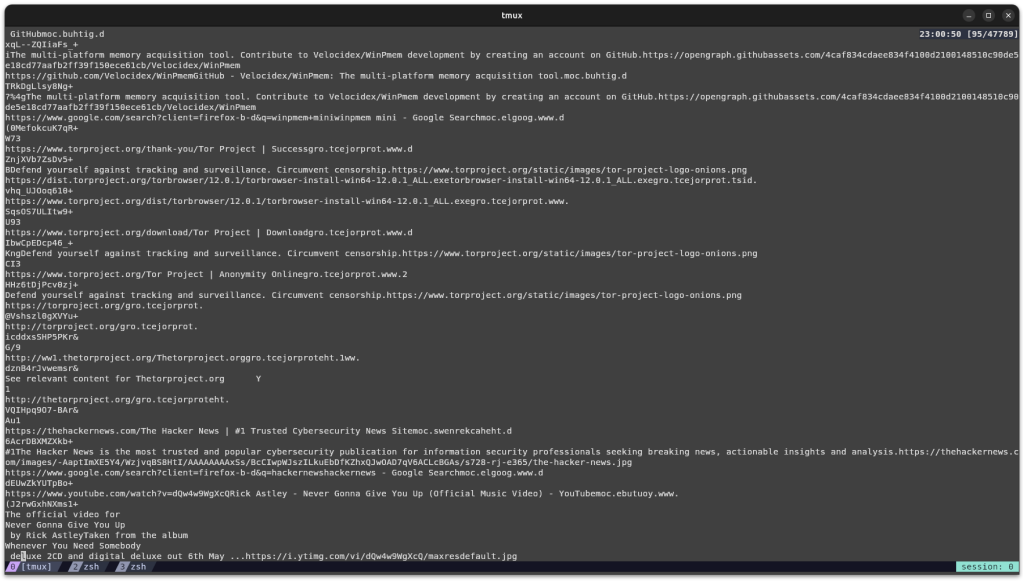


Figure 19: Strings output showing visited URLs

Similar outputs exist for the `Tor Browser` and `Microsoft Edge`.

I was not able to find the contents of the `password.txt.txt` file. I was able to extract its contents with a more basic approch using `strings`, see Listing 17.

```
$ strings physmem.raw | grep "password.txt.txt" -A 5 | less
```

Listing 17: grepping for password.txt.txt in memory dump

I was able to extract the following password from the output: `SuP3rS3crEt2023!`, see Figure 20.

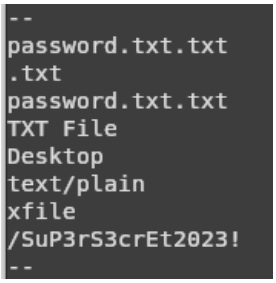


Figure 20: Extracted password from password.txt.txt

I was not able to confirm that this is the password related to the NT hash of user `Spongebob`.

## 3.5. Questions

1. What information can you extract about the operating system?
   - The memory dump is from a Windows 11 system (Build `22621`) running on a 64-bit architecture. The system time at the time of the memory dump was `2023-01-09 22:17:11+00:00`.
2. What happened at the time of the RAM dump?
   - The system showed several browsers running at the same time, notably Firefox and Tor Browser.
   - Temporary files indicate that web pages such as GitHub, Hacker News, and YouTube were likely still open.
   - Multiple network connections were active, including an active Tor network session.
   - A Notepad window was open containing a file named `password.txt.txt`.
   - Additional applications were running, including Microsoft Teams, along with various background services (for example, OneDrive).
   - A command-line session was active, and a memory dump utility was running.
3. What is the user SID?
   - `S-1-5-21-2607170198-3457296929-47938352-1001`
4. Can you find/crack the user password (and get a hint who sent you the RAM dump)?
   - The NTLM password hash could be extracted; however, it was not successfully cracked, and the contents of the `password.txt.txt` file could not be reliably recovered.
   - Memory dump analysis identified the string SuP3rS3crEt2023! as a potential password.
   - It is likely that the memory dump was created by the user Spongebob.