TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

From memmap to minidump; Creating a Volatility Plugin to facilitate per-process memory analysis

---

Author:
Odysseas Stavrou

Thesis Committee:
Prof. Sotiris Ioannidis
Assoc Prof. Vasilis Samoladas
Assoc Prof. Davide Maiorca

A thesis submitted in fulfilment of the requirements
for the diploma of
Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering

July 21, 2025

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

From memmap to minidump; Creating a Volatility Plugin to facilitate per-process memory analysis

by Odysseas Stavrou

The Volatility Framework is the most advanced memory analysis framework in its category out there. It can effectively analyse Windows, Linux and Mac memory snapshots. However, Volatility deals explicitly with and in memory. It will give you a view into the memory snapshot for you to explore the low level system structures and venture at any (valid) address of any process physical or virtual, either in kernel-land or user-land. But what happens if we want to explore a tad bit higher level of memory, say for example the Objects in a process suspected of being a Command and Control (C2/CnC) agent? Or a red-teamer extracting secrets from a specific process? How do we go from a whole Windows memory snapshot into an isolated "user-friendly" Minidump that we can use with already made tools (`WinDBG`, `mimikatz`, and most notably their plugins) to analyse a single process? This work leverages some pre-existing research in an attempt to create a Volatility plugin that can extract and restructure a Process' memory into a Minidump snapshot taking advantage of Volatility's highly modular, and pluggable tooling and in the process, outlining how given access to the opaque structures (`_EPROCESS`, `_ETHREAD`, etc.) one can generate analysable Minidumps.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

From memmap to minidump; Creating a Volatility Plugin to facilitate per-process memory analysis

by Odysseas Stavrou

Το Volatility αποτελεί το πιο προηγμένο εργαλείο ανάλυσης μνήμης στην κατηγορία του. Μπορεί αποτελεσματικά να αναλύσει στιγμιότυπα μνήμης από Windows, Linux και Mac. Ωστόσο, το Volatility ασχολείται ρητά με τη μνήμη, επιτρέποντας την εξερεύνηση δομών χαμηλού επιπέδου και την πρόσβαση σε οποιαδήποτε έγκυρη διεύθυνση φυσικής ή εικονικής μνήμης, είτε στον χώρο του πυρήνα είτε του χρήστη. Τι συμβαίνει όμως όταν θέλουμε να διερευνήσουμε μνήμη υψηλότερου επιπέδου, όπως για παράδειγμα αντικείμενα σε μια διεργασία που θεωρείται ύποπτη ως πράκτορας Command and Control (C2/CnC); Ή όταν μια ομάδα επιθετικού ελέγχου (red team) θέλει να εξαγάγει ευαίσθητα δεδομένα από μια συγκεκριμένη διεργασία; Πώς μπορούμε από ένα πλήρες στιγμιότυπο μνήμης των Windows να δημιουργήσουμε ένα απομονωμένο και «φιλικό προς τον χρήστη» Minidump που να μπορεί να χρησιμοποιηθεί από ήδη διαθέσιμα εργαλεία (όπως `WinDBG`, `mimikatz`, και κυρίως τις επεκτάσεις τους) για την ανάλυση μιας μεμονωμένης διεργασίας; Η παρούσα εργασία αξιοποιεί υπάρχουσα έρευνα για τη δημιουργία μίας επέκτασης (plugin) στο Volatility που μπορεί να εξάγει και να αναδιαμορφώσει τη μνήμη μιας διεργασίας σε ένα στιγμιότυπο Minidump. Η προσέγγιση αυτή εκμεταλλεύεται την υψηλή αρθρωτότητα και επεκτασιμότητα του Volatility, και εξηγεί πώς, μέσω πρόσβασης σε αδιαφανείς δομές (όπως `_EPROCESS`, `_ETHREAD`), μπορούμε να παράγουμε Minidumps κατάλληλα για ανάλυση.

Acknowledgements

I would like to express my sincere gratitude to my thesis committee: Professor Sotiris Ioannidis and Vasilis Samoladas from TU Crete, and Davide Maiorca from the University of Cagliari, for their support, guidance, and thoughtful evaluation of my thesis. Special thanks to Davide Maiorca for offering me valuable exposure to real-life forensic investigations and opening doors for potential future collaborations.

Additionally, I deeply appreciate the invaluable teachings of my professors at TU Crete—Apostolos Dollas, Aggelos Bletsas, and Nikolaos Giatrakos—whose expertise and dedication significantly enriched my academic journey.

I am also grateful to my employer, Hack The Box, and all my coworkers, who allowed me to seamlessly integrate my research findings into practical applications and vice versa.

My heartfelt thanks go to my girlfriend, Olga, for her unwavering support, patience, and understanding throughout this endeavour. Furthermore, I am profoundly thankful to my family and close friends, George, Raphael, James, Vaggelis, and Stavros, for their constant support and patience in tolerating my many peculiarities.

Special appreciation goes to the CyberMouflons hacking community in Cyprus for their shared enthusiasm.

Finally, I would like to thank my music concert companions George, Odysseus, Panos, and Nikos, whose friendship and shared passion provided an essential escape from daily chaos.

Odysseus, Chania, 2025

Contents

Semper Curiosus, Semper Discens, Semper Crescens…

Chapter 1

Introduction

1.1 Forensics

Computer or Digital Forensics encompasses Computer Science primitives with legal Forensics to correctly handle and analyze a digital Artefact, whether this is a physical Hard Drive, a mobile phone, a complete Information System (IoT devices, SoCs) or a single text document. The legal part pertains to the handling of said evidence in a manner which conforms to the local laws and requirements of the state and/or certification authority, and their presentation in court if applicable.

Much like a medical forensic pathologist, Digital Forensics will often diagnose, and not be limited to, the "Who", "When", "What", and most importantly "How", a certain chain of events unravelled. If the event at hand is a computer infection, for example, it is the job of the Forensic Analyst to examine the Artefacts, and provide the Entry Point of the attack, which systems were affected **and** in what manner!

This work focuses **exclusively** on Memory Forensics, a subcategory where the artefact of interest is the Memory of a System, in our case a Windows (x86) machine.

1.1.1 Forensics in CTFs

The Forensic in a Capture The Flag (CTF) competitions is often a really misunderstood category and really under-represented due to their co-notation

with Steganography. A lot of CTFs feature steganography challenges or challenges that will have the player manually fix an Image. In both cases, damaging the concept of forensics because of the severe lack of cyber-security underpinnings and overly reliance on guesswork. Additionally, there has been an effort to 'modernize' the category with a push to include more Malware Analysis challenges, with some focus on Command and Control (C2/CnC) channel decryption. This is something I am actively trying to change with my work in Hack The Box, creating content that was inspired by real-life attacks, techniques, and threat actors.

1.2 The Idea

My interest in memory forensics began with the first Capture the Flag (CTF) competition I participated in, where I quickly realized how drawn I was to forensic analysis—particularly the intricacies of memory analysis and the techniques used to interpret it. Over the past five years, I have immersed myself in the field of memory forensics, both through personal research and professional work involving the creation of related content.

During this time, I observed that much of the analysis was focused on the entire memory space of the operating system, effectively treating the OS itself as the subject of investigation. This realization prompted a shift in my perspective: rather than analysing the operating system as a whole, could I target the user-level processes running atop it?

With some pre-existing familiarity in .NET programming and experience with the Empire C2 framework, I began to consider how one might transition from analysing full memory snapshots to examining .NET objects within the heap of a specific process—namely, the Empire agent. This line of inquiry led me to explore minidumps, a type of process-specific memory dump compatible with tools such as WinDBG and Mimikatz. These minidumps enable more focused and efficient analysis by isolating valuable process-level information, forming the foundation for the work presented in this dissertation.

1.3 Artefact Mediums and Acquisition types

As mentioned above, the artefact in question, for this work, is the Memory of the System. When a (full) Memory acquisition happens, it means the contents

of RAM (Random Access Memory) in their entirety, are being exported to a file with a structure depending on the acquisition method!

Since the contents of RAM are being exported, it means the resulting file contains the whole **physical address space** of the machine, and it's up to the analysing tool to figure out the Virtual Mapping (Chapter 2). Acquiring a sample from an 8GB Machine will always produce an 8GB file, regardless of how much memory the system is using.

Some of the most popular and open source acquisition tools (regardless of platform) are:

- LiME → LiME Artefact
- avml → LiME Artefact
- WinPmem → Raw Artefact

However, none of them were used for this work. Instead, the built-in dumping tooling of the Hypervisor (VirtualBox) was used:

- .pgmphystofile → Raw Artefact
- `$> VBoxManage debugvm [VM] dumpvmcore` → ELF Artefact

All the aforementioned memory acquisition types will produce structurally different artefacts, however, **all** of them are a container of the **same** data:

The physical address at the time of capturing!

The only difference is the way this data is accessed. In the same way both an `mp3` and a `wav` file can be played by the same Media Player (software-wise), all of these samples can be processed by the analysing tool and accessed in the manner (Volatility 3.3.1).

1.3.1 Thesis Organization

The thesis is organized as follows:

Chapter 2: Reviews previous work relevant to the research topic.

Chapter 3: Describes Volatility internals and associated primitives.

Chapter 4: Examines Windows internals.

Chapter 5: Discusses Minidump and export methodologies (and the PoC)

Chapter 6: Outlines derivative work and possible future research directions.

Chapter 2

Previous and Related Work

Volatility has a great deal of built-in tools that could identify possible malware techniques such as:

- Shellcode Injection
- Process Ghosting
- Process Hollowing
- Syscall Unhooking
- Direct/Indirect Syscalls

These plugins will give you an insight into the process itself, but not into the memory, since it is somewhat of unstructured. For example, they will identify the shellcode that has been injected, and the destination page, however, this will pinpoint the malicious process, and it is up to the analyst to examine the possible existence of any objects of interest, on top of the (now identified) process' memory.

If, however, there was a runtime on top of the process i.e. **.NET's CLR**, then a lot can be deduced about the objects and the memory as shown in this 2022 paper [1] by Modhuparna Manna et al.

As such, all the information that is needed to analyse a malicious process with a runtime (CLR, and possibly JVM) already exists within the process, and it is available!

Together with 1.2, presents the opportunity that if a Minidump is created from a CLR process then there can be further analysis, specific to the runtime of the process.

In addition to this, there exist some great PTE (Page Table Entries) research by Frack Block et al. that examines the PTE entries in a memory structure using

Volatility [2], [3], [4]. While not directly used, it is an area of improvement, as described in 6.1.2.

This work is directly and heavily influenced by the extraordinary work Ulf Frisk did on the MemProcFS[1] Project.

`MemProcFS` provides a very simple and convenient way to visualize physical memory as a File System structure from a memory snapshot or even from DMA devices using `PCILeech`[2]

`MemProcFS` effectively demonstrates how a Minidump can be exported from a memory snapshot. However, the process is somewhat convoluted—understandably so, as Minidump extraction is only one of the tool's many features, and duplicating this functionality in a more streamlined form would probably not align with its broader design goals.

This work should in no way be used as a replacement for `MemProcFS` as it is:

- Orders of Magnitude slower than `MemProcFS`
- Lower compatibility than `MemProcFS`
- Not Extensible

However, this work was done as an opportunity for me to dive into Windows Internals, Volatility development and specifically documenting the steps that are needed to go from having interactive access to the snapshot's memory to creating a Minidump, something that was identified as a gap both from online documentation and from Volatility, a tool I use almost every day.

[1] https://github.com/ufrisk/MemProcFS
[2] https://github.com/ufrisk/pcileech/

Chapter 3

Volatility

3.1 History

In 2006, Nick Petroni et al. released `FATKit` [5] (Forensics Analysis ToolKit), a collection of tools that, even back then, facilitated one of the most powerful concepts of Volatility to date; "Reconstructing and Viewing data in different levels of abstraction." In 2007, the first version of Volatility, "Volatility Framework v1.1.1" was released, having evolved from the immense research on `FATKit` and `VolaTools` [6].

The holistic framework that is known today emerged in 2011 with the release of Volatility 2.0. A proper and extensible framework written in Python 2.

However, challenges still remain. Being written in Python 2 meant that moving forward, a true rewrite had to take place due to the Python 2 EOL (End-Of-Life) of January 2020, among other needs like performance and features.

The rewrite was finalized in May 2025 with the Monumental release of Volatility 3 v2.26.0, reaching parity release with Volatility 2, thus deprecating it, almost 14 years later. This work focuses exclusively on Volatility 3, and thus, it will be mentioned as plain "Volatility".

3.2 Usage

From the end-user perspective, Volatility is a plugin runner and said bundled plugins have remarkably precise functionality, and yet they can be used as fine building blocks when developing a plugin, 3.6 Plugins. Volatility can also provide the user with an interactive view into the memory snapshot much like using a debugger, displaying structures and exploring memory

addresses and their content, using `Volshell` which it is itself a plugin with no functionality.

3.3 Internals

Since `aarch64` support is not official yet, everything from this point on will assume a `x86(_64)` platform. Volatility's future plans are covered in 3.7 Future.

Volatility encompasses a great deal of tools and techniques in order to make sense of the many aforementioned artefacts, and can be thought of as a really powerful and featureful Memory Mapper, knowing how to translate virtual addresses to physical and how to construct objects on any abstract memory location both virtual and physical.

However, in order for the Framework to be at a place where the aforementioned functionality is possible, some prerequisites must be met.

3.3.1 Initialization

When run, Volatility will use the bundled tool called `automagic`. This appropriately-named tool has many functionalities that set up the context on top of which the plugin will run. Notably it will:

- Figure out where and how the Physical Memory is structured in the provided artefact
- Figure out the Memory Map (Virtual Address -> Physical Address)
- Construct the appropriate Layers (3.4) and the rest of the layer hierarchy
- Create `config` and `context` which the plugins run of top of

`config` encapsulates all the configuration options, to name a few:

- Plugin Name
- Kernel Module Name and offsets
- Layer Names
- Symbol configuration (Linux banner or Windows PDB)

While the `context` variable contains all the object counterparts:

- Layers 3.4
- Modules
- Symbol Spaces 3.5

3.4 Layers

Volatility uses Layers to be able to access the same data, i.e.: contents of address `0x994a4d0c0000` in the same manner, regardless of Artefact type or acquisition method.

Layers are constructed from a bottom to top approach, each Layer depends on all the Layers below it, thus following a Tree-like structure rather than a single stack, because a Layer may have multiple Dependencies, i.e.: Physical Memory & Swap on Disk are both considered physical memory as far as the Operating System is concerned, yet accessed differently.

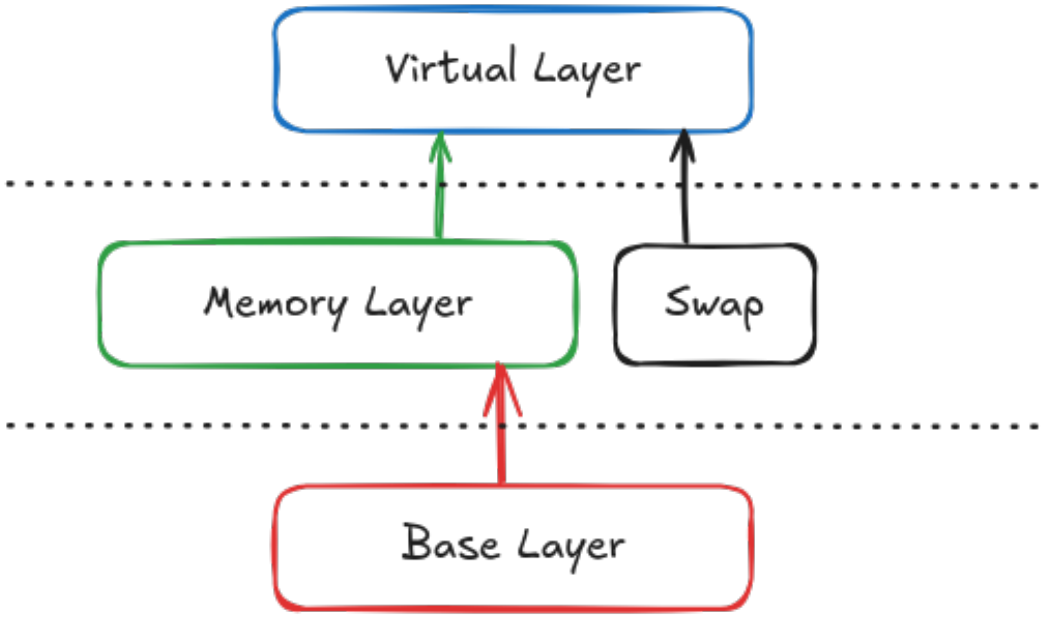

FIGURE 3.1: Volatility Layer Hierarchy [7]

While not every sample will contain all the layers, the most notable Layer types are:

- Memory Layer - The physical layer. Contains the physical memory of the system. Notable types (depending on acquisition method):
  - `FileLayer`

- Elf64Layer

- Virtual Layer - Contains the virtual memory of a process, **and** the Kernel. Of type: `Intel32e`
- Base Layer - The Layer responsible for operating on the snapshot file itself. Of type: `FileLayer`

The type `FileLayer` can be seen in both Base and Memory Layer. This happens when the snapshot in question is a RAW snapshot, meaning the physical address space **is** the artefact itself, i.e: an exact 1:1 mapping.

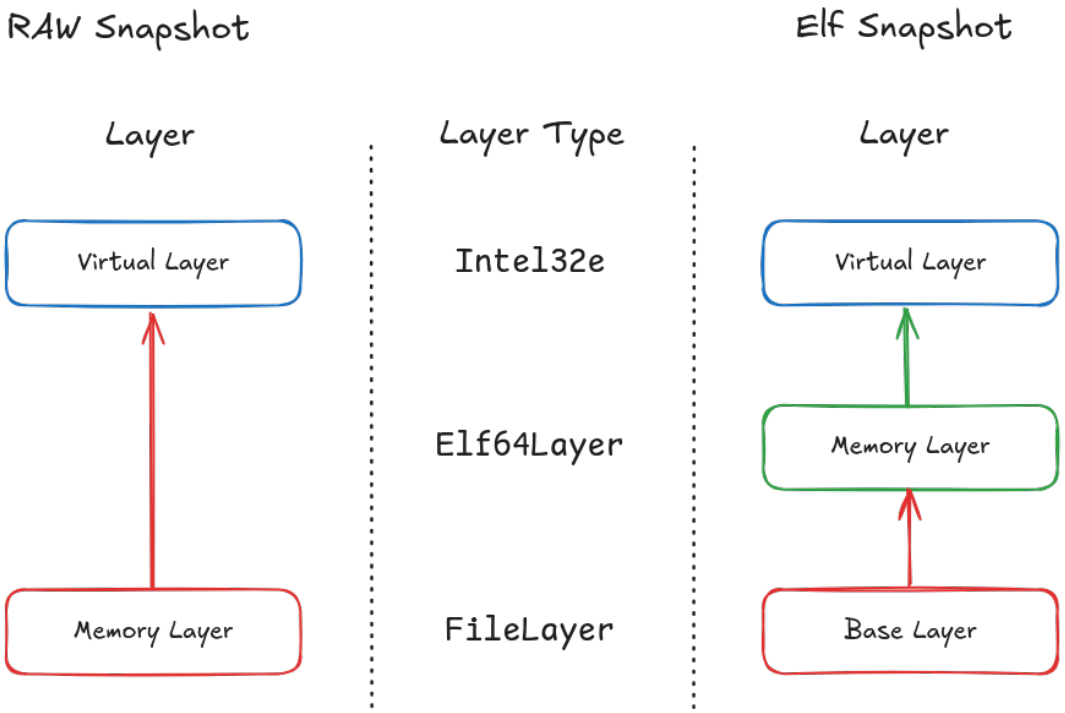

FIGURE 3.2: Differences in Layers between acquisition types [7]

The Kernel Virtual Layer is a core requirement for all plugins (there are some plugins that do not require it, like `banners` but these types of plugins only do static searching without needing memory and symbol mechanisms). Its existence also signifies that the `automagic` finished successfully and that a working kernel exists on the system (either Linux or Windows).

In addition, on top of the Memory Layer, the framework can create any number of Virtual Layers that each encapsulates the virtual address space of each process as shown below:

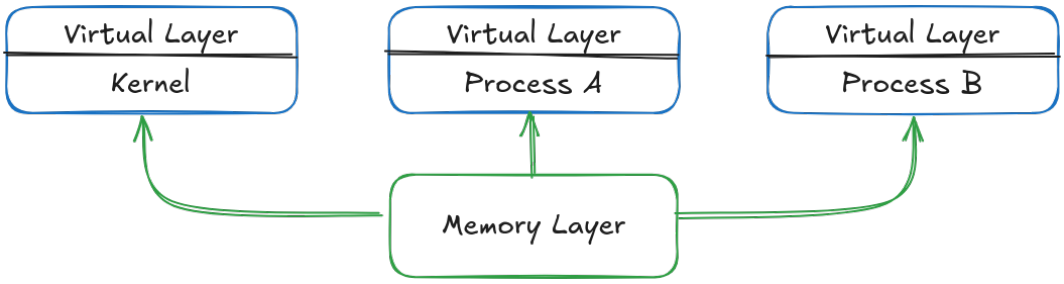

FIGURE 3.3: Virtual Layers Diagram [7]

Lastly, now that Volatility has a complete tree of Layers, it can access any type of address, at any Layer.

For example: When invoking the `read()` on a Virtual Layer trying to read `n` bytes from an address, the Virtual Layer will translate that address to the Layer below it and invoke `read()` on that Layer. The cycle continues until the data has been read and propagated up the calling stack:

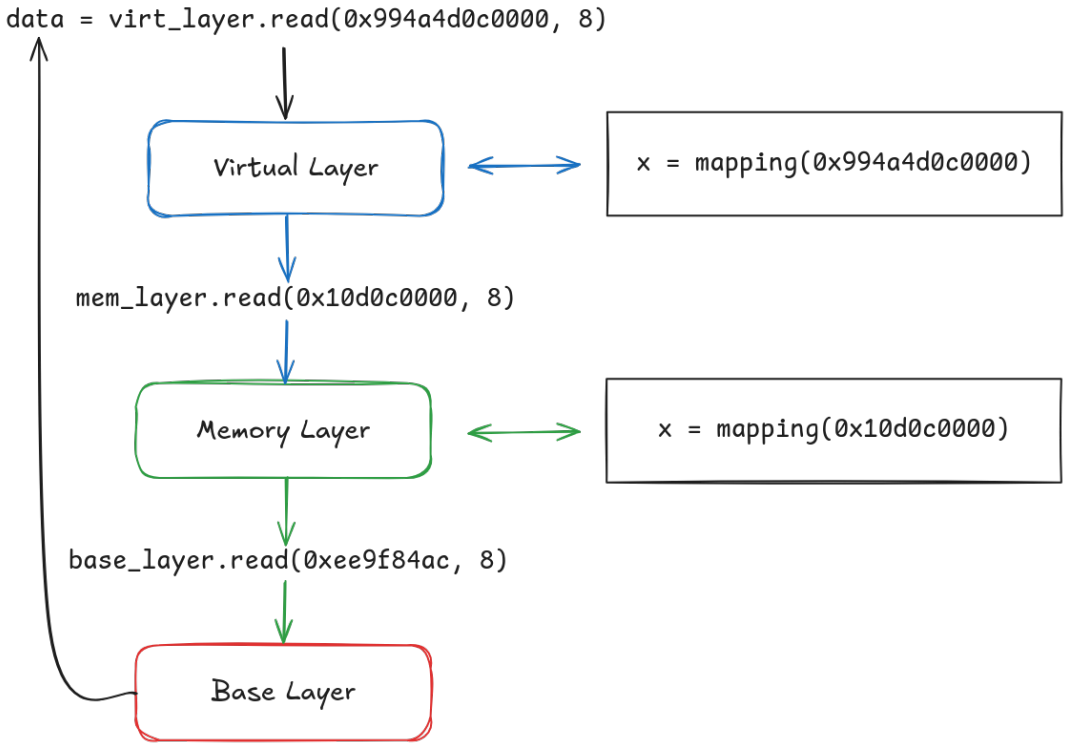

FIGURE 3.4: Reading Example [7]

3.5 Symbols

Up until this point, there was no distinction between operating systems (Windows, Linux) as the focus was purely into understanding how Volatility structures the Layers and parsing the physical memory. The goal now for the framework is to create the Kernel Layer mentioned above. Since the Kernel is specific to the Operating System, there is a divergent methodology the framework follows depending on the OS at hand.

A Symbol is a **named** location in memory that is described by both:

- Offset (Location inside a Virtual Layer)
- Template (Structure Layout)

Volatility needs this information to create the basic (Kernel's) objects in order to facilitate any further analysis. As already mentioned, the Kernel Layer is a hard requirement for almost all plugins, and the only way to construct it, is by having access to the Symbols.

As Mike Auty from the Volatility Foundation always tries to describe symbols to newcomers, using the cookie cutter analogy. The Template (Structure) is the cookie cutter, the Layer is the dough, and the Offset is where on the dough we place the cutter. Objects are the cookies cut from the dough.

With a snapshot taken from a Windows System, Volatility is able to automatically download all necessary symbols (`ntkrnlmp.pdb` for the kernel) from Microsoft themselves, since the symbol server [1] and the resulting symbols are public!

Using a Linux snapshot makes symbol acquisition more challenging. Due to the vast number of Linux distributions—each with different versions and functionalities—every release has its own unique kernel image. These kernel images differ not only in functionality, but also in the structure and availability of symbols. In fact, even identically named symbols can have different definitions across distributions or versions.

This is why the Volatility team has imposed some strict rules when checking the symbol package against the snapshot. The Linux banner (`Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc...)` of the Intermediate Symbol File (ISF) is expected to match the output of the `banners` plugin.

[1] https://msdl.microsoft.com/download/symbols

This is because the banner will contain the date of the kernel's compilation, and if they match, then it is certain to be compatible.

In order to create such an ISF, the debugging kernel is needed, because it contains the `DWARF` information that are exported into a Volatility compatible `json` file.

Needing the debug package for every possible kernel image create a huge logistic issue, both in version tracking and in storage needs. That being said, there are some attempts by Abyss W4tcher [2] to populate a repository [3] with ISFs from popular Linux Distributions.

In any case, this requires some manual labour from the user, and it certainly does not provide the same ease-of-use as the Windows counterpart. There is some research by Valentin Obst [8] on how the `.btf` (BPF Type Format) section inside the Kernel image can be used in order to produce a working ISF.

3.6 Plugins

As already mentioned in Initialization 3.3.1, plugins run on an already constructed context, and they have the Kernel Layer as a requirement.

All plugins have a `run()` method which serves as the entry point to the plugin, and also returns the Rendering Template (along with the data) for the renderer.

Plugins produce highly precise output, but to achieve this, they rely on many smaller methods that are also specific yet highly modular. These smaller methods function like puzzle pieces—they can be combined with methods from other plugins to accomplish the developer's intended functionality.

Take for example the `pslist.PsList` plugin. It is responsible for printing a list of the running processes and some of their info:

```
1 return renderers.TreeGrid([
2     ("PID", int),
3     ("PPID", int),
4     ("ImageFileName", str),
5     (f"Offset{offsettype}", format_hints.Hex),
```

[2] https://github.com/Abyss-W4tcher
[3] https://github.com/Abyss-W4tcher/volatility3-symbols

```
    ("Threads", int),
    ("Handles", int),
    [...]
],
self._generator(),
)
```

Other than that, and the output functionality, the plugin does not provide any additional functionality. However, in order to create the list of the running processes, it walks the `_LIST_ENTRY` object inside the Kernel:

```
@classmethod
def list_processes([...]):
    # [...]
    list_entry = kernel.object(
        object_type="_LIST_ENTRY",
        offset=ps_aph_offset
    )
    # [...]
    eproc = kernel.object(
        object_type="_EPROCESS",
        offset=list_entry.vol.offset - reloff,
        absolute=True
    )

    procs = eproc.ActiveProcessLinks.to_list(
        symbol_type=eproc.vol.type_name,
        member="ActiveProcessLinks",
        forward=forward,
    )
    # [...]
```

The `list_processes()` method is one of the most used methods across the entire framework and beyond, because anytime a plugin wants to operate on a process (or more) it has to invoke this method.

The project's high modularity provides plugin developers with significant flexibility, and promotes efficient and enjoyable development paradigms through the use and composition of small specialized methods that are defined in Classes for more broad usage, i.e.: `PsList`

3.7 Future

There are some experimental Pull Requests (PRs) for adding **some** `aarch64` support, however it has a long way to go. Adding Arm support to the project will enable analysing memory snapshot from Android devices as well.

For the Linux Symbol issues mentioned above, there is work on the experimental `BTF` approach with hopes of streamlining it, and as a result ISF creation would be somewhat automatic for Linux snapshots as well.

There is also some work being put into the coding standards of the project that would create and standardise the pre-existing APIs i.e.:

- Concrete naming scheme for Layers [4]
- An API for retrieving said Layers [5]
- Support for Windows Modern Hibernation [6]

3.8 Virtual Memory Translation

While not entirely in scope of this work, it is paramount there exist a fundamental understanding of address translation, especially focusing on the Page Table and its Entries, the PTEs (Page Table Entries) since section 4.4 mentions them in extent.

Address translation on `x64` is very similar to `x86` but with the addition of an extra (4th) table, and all four of them exist in the system address space:

4. Page Map Level 4 Table (PML4T)
3. Page Directory Pointer Table (PDPT)
2. Page Directory Table (PDT)
1. Page Table (PT)

[4] https://github.com/volatilityfoundation/volatility3/pull/1381#issuecomment-2601075197
[5] https://github.com/volatilityfoundation/volatility3/issues/1351
[6] https://github.com/volatilityfoundation/volatility3/pull/1036

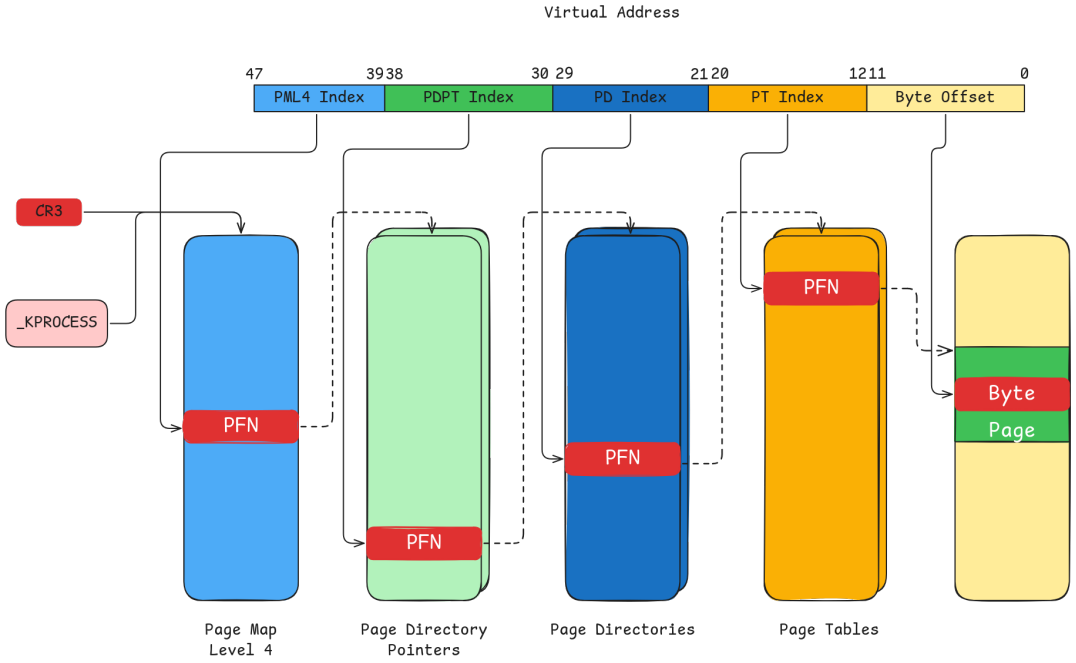

FIGURE 3.5: Address Translation on x64 [9]

As evident by the figure above, there are only 48 usable bits out of the total 64 bits in a virtual address, and they are divided into five parts, resembling an Index inside each of the aforementioned Tables or a PFN (Page Frame Number)

On a context switch (4.3.6) the CPU will load the value from the `DirectoryTableBase` a field in the `_KPROCESS` Structure inside the scheduled Process (`_EPROCESS`), into the `CR3` register, which points to the PML4 table

The subject of interest here, however, are the PTEs which is the currency the VMM (Virtual Memory Manager 4.2) and the VADs deal in. Each page of virtual address space, is associated with a PTE that lives in the system address space which contain the physical address the virtual counterpart is mapped:

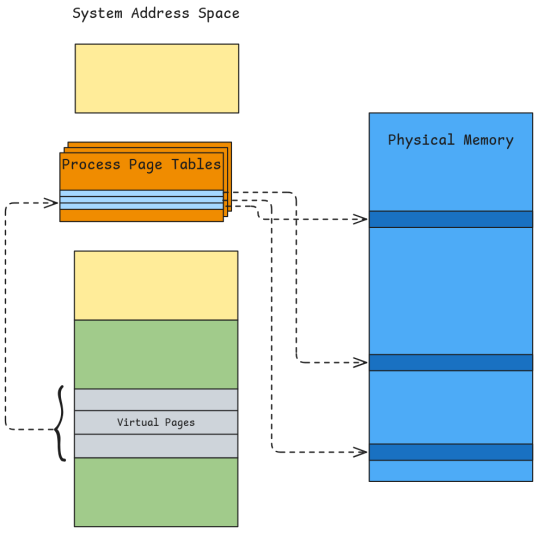

FIGURE 3.6: PTEs Mapping Example [9]

The above figure shows how three consecutive virtual pages can have a completely different arrangement in physical memory. There could be the case of no PTEs being available at all. This is because the memory region could be reserved and committed, but never accessed. In that case, the relevant PTEs will be constructed by the VMM after a page fault.

Volatility provides structured methods within its Layers that can perform these calculations when reading from a layer, in addition to partially translating an address to retrieve the PTE, so the protections of the memory page can be read, as will be shown in 5.2.8

Chapter 4

Windows

Windows Internals is the term used to describe the core mechanisms, architectural design, and organizational principles that underpin Windows NT–based operating systems. Because Windows is built as a highly modular platform, studying its internals reveals the intricate interactions among its components—from low-level kernel services to user-mode subsystems—and not only deepens the understanding of how the OS functions, but also opens avenues for implementing software in areas such as usability research, security analysis, and both offensive and defensive engineering.

In addition, some extraordinary and renowned projects are a direct result of Windows' Internals and its undocumented functionality reversal:

- ReactOS
  An open-source re-implementation of Windows NT
- Wine
  A compatibility layer that translates Windows Syscalls into Linux ones by implementing the `Win32 API` in userspace
- Offensive, EDR/Antivirus Bypassing Techniques
  By leveraging undocumented code (`Nt,Zw` functions) and more, there are techniques that can bypass Antivirus Software
- Volatility
  In order for Volatility to analyze Windows snapshot, a huge amount of effort has been put in by the maintainers of the project to analyze Windows Internals

4.1 Internals

The figure 4.1 displays a simplified view of the Windows Architecture. The red lines divide the components into User and Kernel Mode.

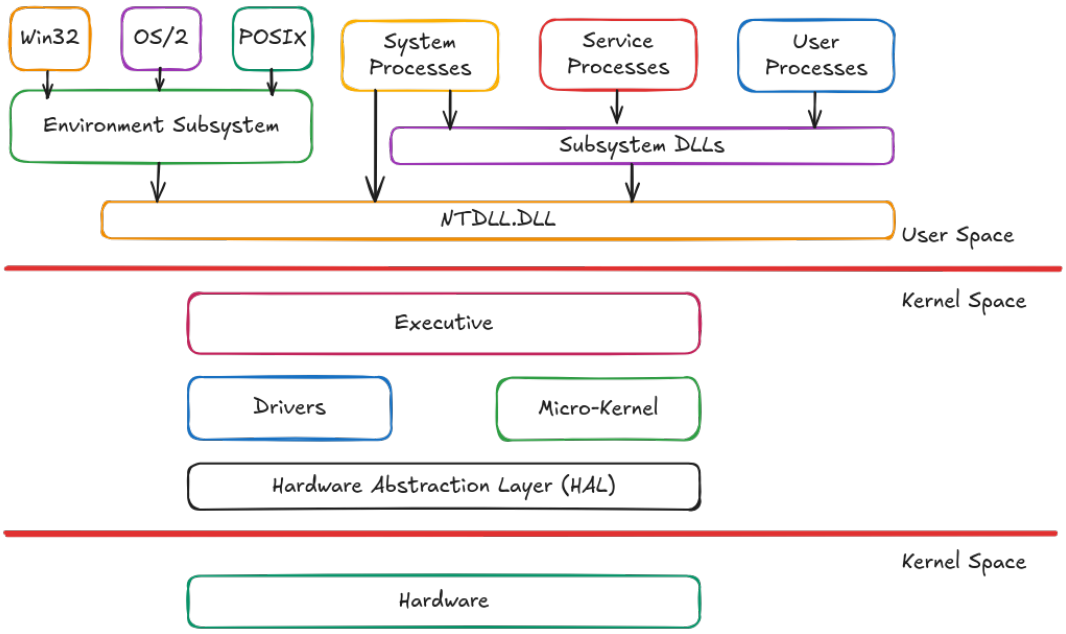

FIGURE 4.1: Windows Architecture [9]

The four core components of the User Space are as follows:

- **Environment Subsystem**: It emulates an operating environment for the application to run in. A great example is the **POSIX** subsystem that was later replaced with the **WSL** (Windows Subsystem for Linux)
- **System Processes**: Core System Processes that are launched by the Kernel on boot. They run under the **SYSTEM** account and in Session 0 and are not limited to:
  - `winlogon.exe`: Logon Manager
  - `smss.exe`: Session Manager
  - `ntoskrnl.exe`: System Process (PID 4)
- **Service Processes**: Processes started by the Task Scheduler, i.e.: Print Spooler, Exchange Server, etc.
- **User Processes**: Processes started by a user

User and **Service** Processes generally never call native Windows code directly. Instead, they rely on their subsystem libraries commonly known as

Windows API (WinAPI) which contain documented code. This said code is responsible to convert the calling code into the appropriate internal and native system calls, usually implemented in `NTDLL.DLL`.

The kernel mode components are:

- **Executive**: Responsible for Memory, Process and Thread Management, I/O operations, Networking, and IPC (inter-process communication). Contains the Executive Objects
- **Micro-Kernel**: Interrupt handling, Thread scheduling, Synchronization, contains the Kernel Objects
- **(Device) Drivers**: Non-Hardware drivers such as File system and Network Drivers, as well as Hardware Drivers that translate the user function calls to the device specific I/O requests
- **HAL**: The Hardware Abstraction Layer are a set of platform-specific abstractions that isolate the kernel from the hardware differences

Although not mentioned, there also exists:

- **Graphic System**: Implements the GUI (Graphical User Interface) and its functions
- **Hypervisor**: It contains its own services and layers for communicating with the Host system

The Executive subsystem is implemented within `NTOSKRNL.EXE` and it is comprised by the following components:

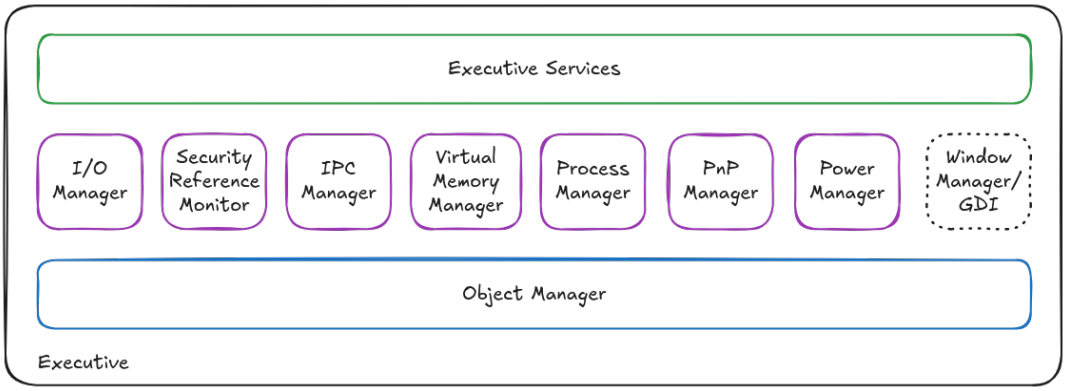

FIGURE 4.2: Executive Subsystem

4.2 Kernel Objects

All Kernel-mode components (Kernel, Drivers, Executive, etc) allocate objects from two distinct and dynamic-sized pools (Pool is the equivalent of User-mode Heap—the underlying memory used for the allocation). All requests are serviced and managed from the VMM (Virtual Memory Manager) Executive Component:

- **Paged Pool**: Virtual memory in the System address space that can the paged out. Paged in after a Page-Fault. Accessible from any process context
- **Non-Paged Pool**: Virtual memory guaranteed to reside in Physical Memory at all times. Can be accessed at any time without incurring a Page-Fault. Meaning that it can be accessed from any IRQL (Interrupt Request Level)

Both of these pools are part of system virtual memory and are mapped to in the virtual space of every process.

All kernel object are prepended with a `_POOL_HEADER` object that provides information about the Pool which allocated the Object:

```
>>> dt('_POOL_HEADER')
symbol_table_name1!_POOL_HEADER (16 bytes):
0x0 :   PoolIndex                   bitfield
0x0 :   PreviousSize                bitfield
0x0 :   Ulong1                      unsigned long
0x2 :   BlockSize                   bitfield
0x2 :   PoolType                    bitfield
0x4 :   PoolTag                     unsigned long
0x8 :   AllocatorBackTraceIndex     unsigned short
0x8 :   ProcessBilled               *_EPROCESS
0xa :   PoolTagHash                 unsigned short
```

FIGURE 4.3: `_POOL_HEADER` type

Note: When displaying a type using `dt()` in Volshell, attributes with the same offset are interpreted as being part of a `Union`.

As a result, by reading data at a negative (max 16 bytes) offset from the address of any non-executive kernel object will yield valuable information about the allocating Pool, especially the `PoolTag`.

4.2.1 Pool Tags

Pool Tags are 4-byte hardcoded identifiers that associate a Memory Request to the resulting object is allocated for. This mechanism allows for:

- **Identification of Purpose and Behaviour**: It allows developers to pinpoint which component is allocating what memory **and** the purpose of the resulting object
- **Identification of Memory Leaks**: Tools like **PoolMon** and **WinDBG** can monitor Pool allocations, thus tracking down any possible memory leaks

Some interesting Pool Tags include:

Pool Tag	Component	Description
`Proc`	Executive/Kernel	Process objects and management
`dFVE`	BitLocker Crash Dump Filter	Full Volume Encryption crash dump filter (BitLocker)
`VAD`	Memory Manager (nt!mm)	Virtual Address Descriptors (VADs)
`Thre`	Executive/Kernel	Thread objects and management
`File`	File System	File objects and file management

TABLE 4.1: Popular Windows Kernel Pool Tags

Extending the example from 4.3. Suppose an unknown (non-executive) object at address `0xe78c9828bf30`. To identify the underlying Pool Tag, and by extension residence in the Paged or Non-Pages Pools, one can read 4 bytes at `-0xc` to read the Pool Tag:

```
(layer_name) >>> virt_layer = self.context.layers['layer_name']
(layer_name) >>> virt_layer.read(0xe78c9828bf30 - 0xc, 4)
b'VadF'
```

Since the Pool Tag is `VadF` then the object at address `0xe78c9828bf30` must be of type `_MMVAD_SHORT` and such objects are serviced from the **Non-Paged Pool**:

```
(layer_name) >>> vad = self.context.object(
    'symbol_table_name1!_MMVAD_SHORT',
    offset=0xe78c9828bf30,
    layer_name='layer_name'
)
(layer_name) >>> dt(vad)
symbol_table_name1!_MMVAD_SHORT (64 bytes) @ 0xe78c9828bf30:
```

```
[...]
(layer_name) >>> vad.get_tag()
'VadF'
```

4.2.2 Executive Objects

Executive Objects are used by the Executive Subsystem and are managed by the Object Manager. They represent fundamental, high-level abstractions of system resources and are distinct from kernel objects.

Any Kernel or User-mode component that wants to interact with a system resource will have to do so through an Executive Object. In addition, they encapsulate the relevant Kernel Object that manages the synchronization and other relevant Kernel functionalities. Some of the most notable and used Objects in this work are:

- _EPROCESS
- _ETHREAD

Executive Objects are also allocated from the two Kernel Pools mentioned above, and as such, they not only have the _POOL_HEADER object prepended to them but, also the _OBJECT_HEADER [10]:

```
(layer_name) >>> dt('_OBJECT_HEADER')
symbol_table_name1!_OBJECT_HEADER (56 bytes):
0x0 :   PointerCount             long long
0x8 :   HandleCount              long long
0x8 :   NextToFree               *void
0x10 :   Lock                     _EX_PUSH_LOCK
0x18 :   TypeIndex                unsigned char
0x19 :   DbgRefTrace              bitfield
0x19 :   DbgTracePermanent        bitfield
0x19 :   TraceFlags               unsigned char
0x1a :   InfoMask                 unsigned char
0x1b :   Flags                    unsigned char
0x1c :   Reserved                 unsigned long
0x20 :   ObjectCreateInfo         *_OBJECT_CREATE_INFORMATION
0x20 :   QuotaBlockCharged        *void
0x28 :   SecurityDescriptor       *void
0x30 :   Body                     _QUAD
```

This header directly precedes the object in question, and it is extremely useful because given the address of the object one can arrive to the Header, and vice versa. Additionally, this header provides valuable information about the object itself:

- `TypeIndex`: An Index inside the `ObTypeIndexTable` array that points to `_OBJECT_TYPE`
- `InfoMask`: The header is itself preceded by additional optional headers. This masking value can help in the identification of such headers
- `Body`: A pointer to the Object

```
(layer_name) >>> dt('_OBJECT_TYPE')
symbol_table_name1!_OBJECT_TYPE (224 bytes):
   0x0 :   TypeList                    _LIST_ENTRY
  0x10 :   Name                        _UNICODE_STRING
  0x20 :   DefaultObject               *void
  0x28 :   Index                       unsigned char
  0x2c :   TotalNumberOfObjects        unsigned long
  0x30 :   TotalNumberOfHandles        unsigned long
  0x34 :   HighWaterNumberOfObjects    unsigned long
  0x38 :   HighWaterNumberOfHandles    unsigned long
  0x40 :   TypeInfo                    _OBJECT_TYPE_INITIALIZER
  0xb8 :   TypeLock                    _EX_PUSH_LOCK
  0xc0 :   Key                         unsigned long
  0xc8 :   CallbackList                _LIST_ENTRY
  0xd8 :   SeMandatoryLabelMask        unsigned long
  0xdc :   SeTrustConstraintMask       unsigned long
```

FIGURE 4.4: `_OBJECT_TYPE` Structure

Interesting attributes from this structure include:

- `Name`: The 'canonical' name of the object (i.e.: Thread, Process, etc)
- `Key`: The allocating Pool Tag
- `TypeInfo.PoolType`: Allocated from the Paged Pool or from the Non-Paged one

Using everything from above there is a clear structural distinction between Executive, and non-Executive Kernel objects:

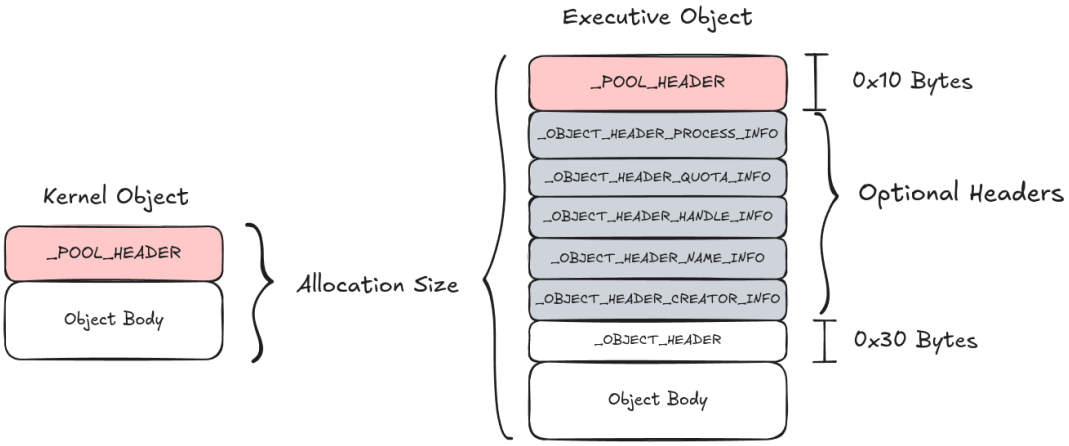

FIGURE 4.5: Executive and non-Executive Kernel Objects [7]

Making this distinction and understanding the preceded Structures was paramount in categorizing the VADs (4.4) based on their backing storage's Pool Tag.

4.3 Process

This section will focus on how the Executive and Kernel components organise their structures, and what do those structures represent. The specific usage of each structure (and its attributes) will be showcased in 5.2 Exporting.

4.3.1 EPROCESS

Each and every Windows Process is represented in the Executive subsystem with an _EPROCESS structure. This structure is one of the most important structures in this work because of its core functionality and attributes it encapsulates, as well as its frequency in the Volatility environment and plugins.

Note: For any executive structure mentioned here or in the below subsections, not all fields are displayed due to the size of the structure.

Some of the most important attributes of the _EPROCESS structure are:

```
(layer_name) >>> dt('_EPROCESS')
symbol_table_name1!_EPROCESS (2112 bytes):
    0x0 :   Pcb                         _KPROCESS
  0x1d0 :   UniqueProcessId             *void
  0x1d8 :   ActiveProcessLinks          _LIST_ENTRY
  0x2e0 :   Peb                         _PEB
```

```
0x370 :   ThreadListHead                  _LIST_ENTRY
0x558 :   VadRoot                         _RTL_AVL_TREE
```

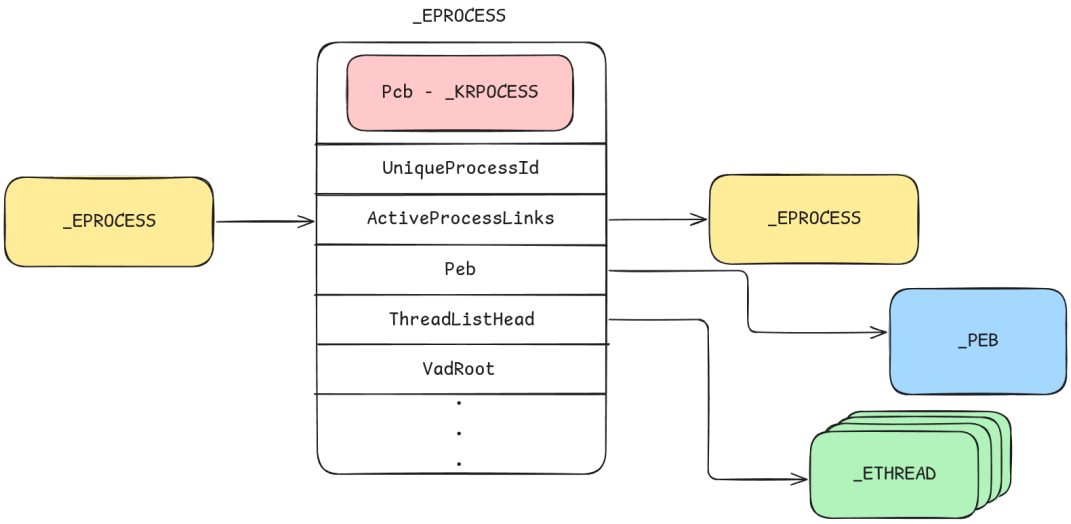

FIGURE 4.6: _EPROCESS [9]

- `Pcb` - Process Control Block: Of type `_KPROCESS`, explained in 4.3.2
- `UniqueProcessId`: The PID (Process Identifier) of the process. Since processes are allocated in the same way that Kernel allocates Handles, the PID is always divisible by 4
- `ActiveProcessLinks`: An entry in a doubly Linked-List of active processes pointed by `PsActiveProcessHead`
- `Peb` - Process Environment Block: Explained in 4.3.3
- `ThreadListHead`: The first entry in a doubly Linked-List of `_ETHREAD`s, explained in 4.3.4
- `VadRoot`: The root `VAD` (Virtual Address Descriptor), explained in 4.4

4.3.2 KPROCESS

The `_KPROCESS` structure is used by the Kernel component for scheduling. It holds information about time spent in Kernel/User mode, Context Switches, CPU Affinity, etc:

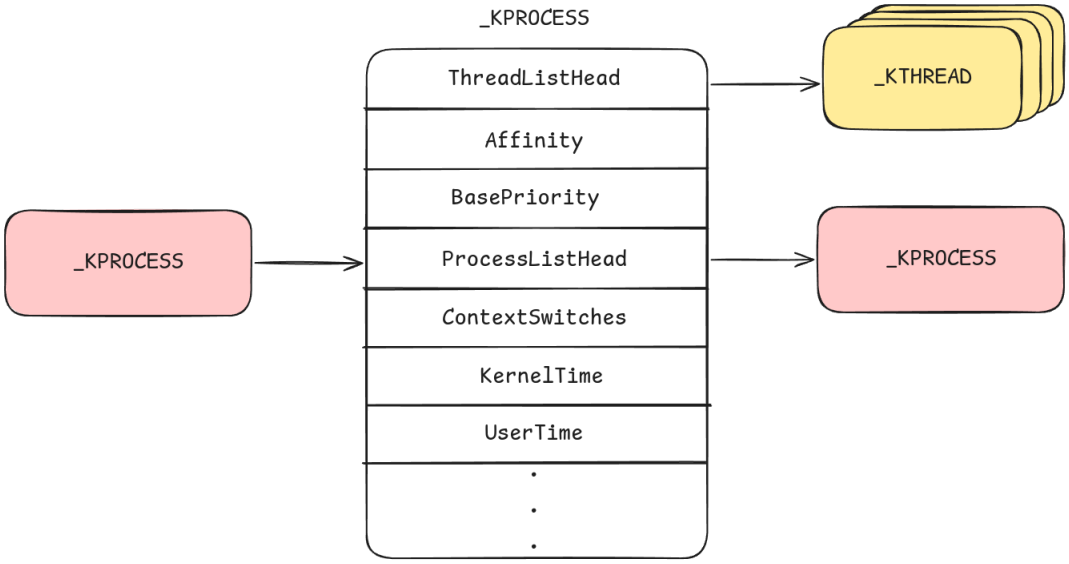

FIGURE 4.7: _KPROCESS [9]

4.3.3 PEB

The _PEB structure is a user space structure; while _EPROCESS.Peb exists in system address space, it points to the structure in user space. It is responsible for storing information which other programs need access to it from user space. Information that would be too expensive to lock behind system calls.

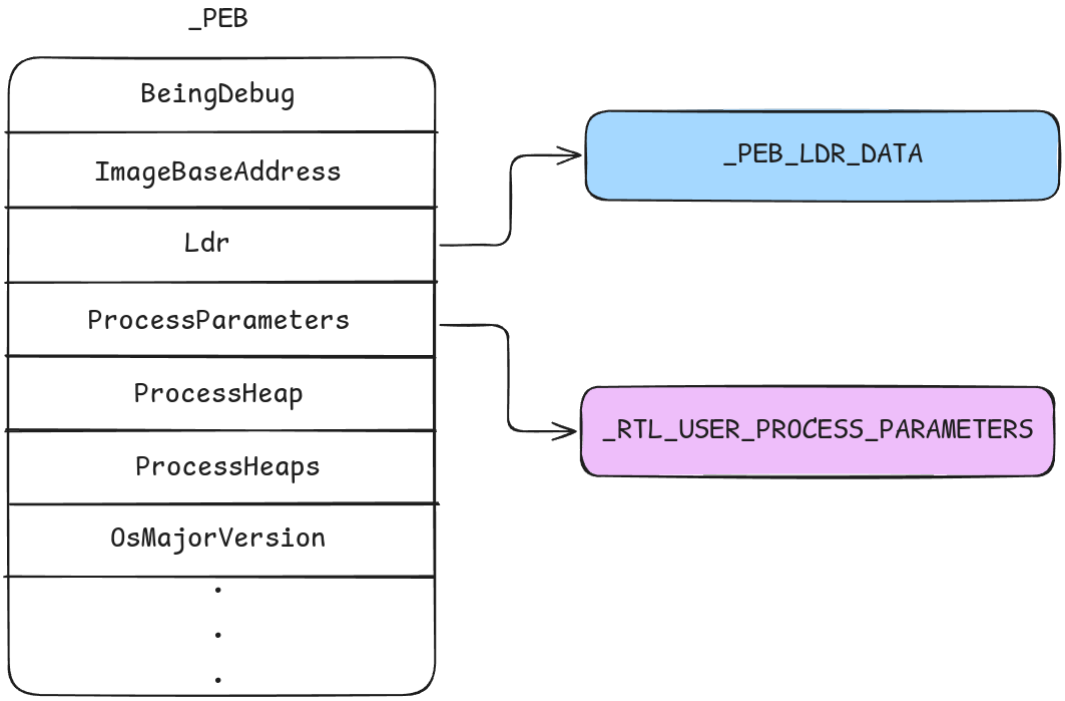

FIGURE 4.8: _PEB Structure [9]

```
(layer_name) >>> dt('_PEB')
symbol_table_name1!_PEB (2000 bytes):
    0x2 :   BeingDebugged                unsigned char
   0x10 :   ImageBaseAddress             *void
   0x18 :   Ldr                          *_PEB_LDR_DATA
   0x20 :   ProcessParameters            *_RTL_USER_PROCESS_PARAMETERS
   0x30 :   ProcessHeap                  *void
   0xf0 :   ProcessHeaps                 **void
  0x118 :   OSMajorVersion               unsigned long
  0x11c :   OSMinorVersion               unsigned long
  0x120 :   OSBuildNumber                unsigned short
```

- `BeingDebugged`: Indicates whether the process is currently being debugged
- `ImageBaseAddress`: Base address where the process is loaded in memory
- `Ldr`: A structure that holds lists of loaded modules in different ordering:
  - Loading Order
  - In-Memory Order
  - Initialization Order
- `ProcessParameters`: Contain Process information
- `ProcessHeap{,s}`: Self Explanatory
- `Os{Major,Minor}Version`: Windows Version numbers
- `OsBuildNumber`: Windows Build Number

4.3.4 ETHREAD

In operating systems, a thread represents the smallest type of schedulable code that anyone can run, and much like `_EPROCESS` a thread to is represented as an executive object named `_ETHREAD`. It also contains a `_KTHREAD` at the base address for scheduling operations and again, much like the process counterpart it exists in system address space apart from the `_TEB` which exists in user space:

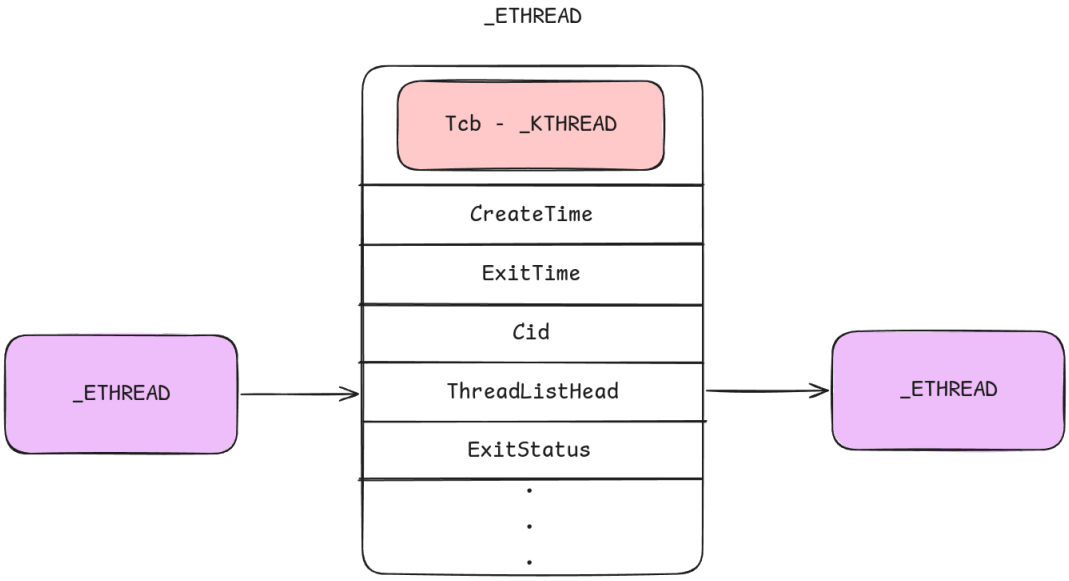

FIGURE 4.9: _ETHREAD Structure [9]

```
(layer_name) >>> dt('_ETHREAD')
symbol_table_name1!_ETHREAD (1928 bytes):
    0x0 :   Tcb                             _KTHREAD
  0x4c0 :   CreateTime                      _LARGE_INTEGER
  0x4c8 :   ExitTime                        _LARGE_INTEGER
  0x508 :   Cid                             _CLIENT_ID
  0x578 :   ThreadListEntry                 _LIST_ENTRY
  0x5d8 :   ExitStatus                      long
```

- `Tcb`: Of Type `_KTHREAD`, described in 4.3.5
- `CreateTime`: Time when the Thread was instantiated
- `ExitTIme`: Time the Thread exited, negative if still running
- `Cid`: A structure holding the Thread ID and the Owner Process PID
- `ThreadListEntry`: An entry in a doubly Linked-List of `_ETHREAD`s
- `ExitStatus`: The exit status of the Thread (if exited)

4.3.5 KTHREAD

The `_KTHREAD` structure hold data related to scheduling activities, such as the Context of the Thread, and unlike the `_EPROCESS`, it holds the pointer to the `_TEB` instead.

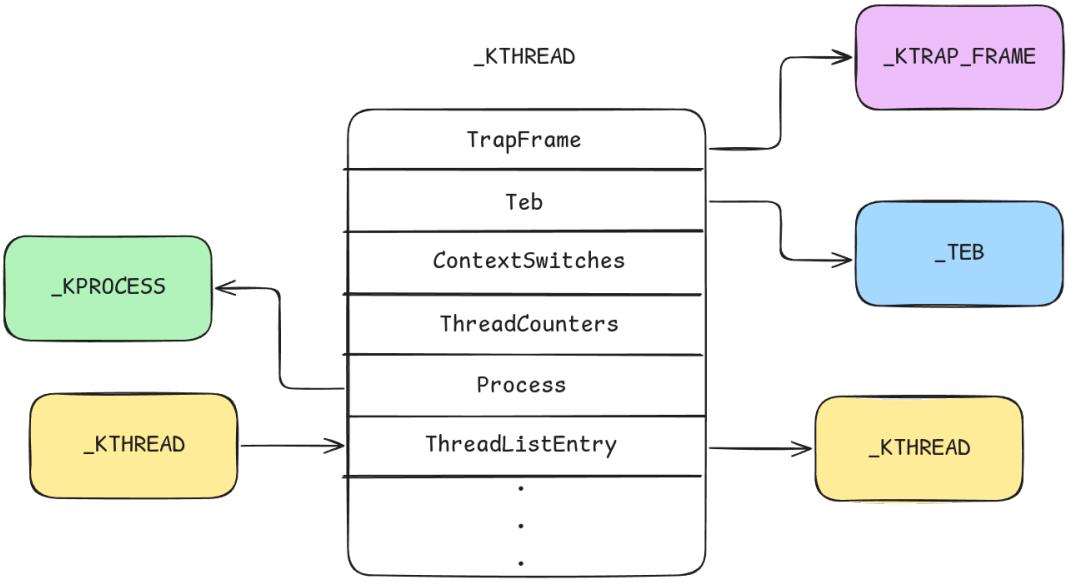

FIGURE 4.10: _KTHREAD Structure [9]

```
(layer_name) >>> dt('_KTHREAD')
symbol_table_name1!_KTHREAD (1216 bytes):
   0x90 :    TrapFrame                      *_KTRAP_FRAME
   0xf0 :    Teb                            *void
  0x154 :    ContextSwitches                unsigned long
  0x168 :    ThreadCounters                 *_KTHREAD_COUNTERS
  0x220 :    Process                        *_KPROCESS
  0x2c8 :    WaitPrcb                       *_KPRCB
  0x2f8 :    ThreadListEntry                _LIST_ENTRY
```

- `TrapFrame`: Holds the CPU Context, described in 4.3.6
- `Teb`: A pointer to the user space `_TEB` structure, described in 4.3.7
- `ContextSwitches`: Number of Context Switches
- `ThreadListEntry`: An entry in a doubly Linked-List of `_KTHREAD`s

4.3.6 KTRAP FRAME

One notable structure for this work, `_KTRAP_FRAME` holds the Context of the CPU Registers. A `trap` is an exception produced by software, for example a division by zero or a `syscall` invocation, rather than a hardware interrupt like a mouse movement or a Key press on a keyboard. When a `trap` occurs, the CPU switches to Kernel mode and takes a copy of the context in order

to properly handle the exception and once finished, the kernel restores the context and continues normal thread scheduling.

The important register here is the value of the RSP register which points to the top of the Stack, and in conjunction with the StackBase found in _NT_TIB (described in 4.3.7, the whole of the Stack can be recovered.

4.3.7 TEB & NtTib

Similarly to the _PEB structure, the _TEB (Thread Environment Block) holds thread information for other components that need access from user space. The only part of the _TEB that pertains to this work is the _NT_TIB.

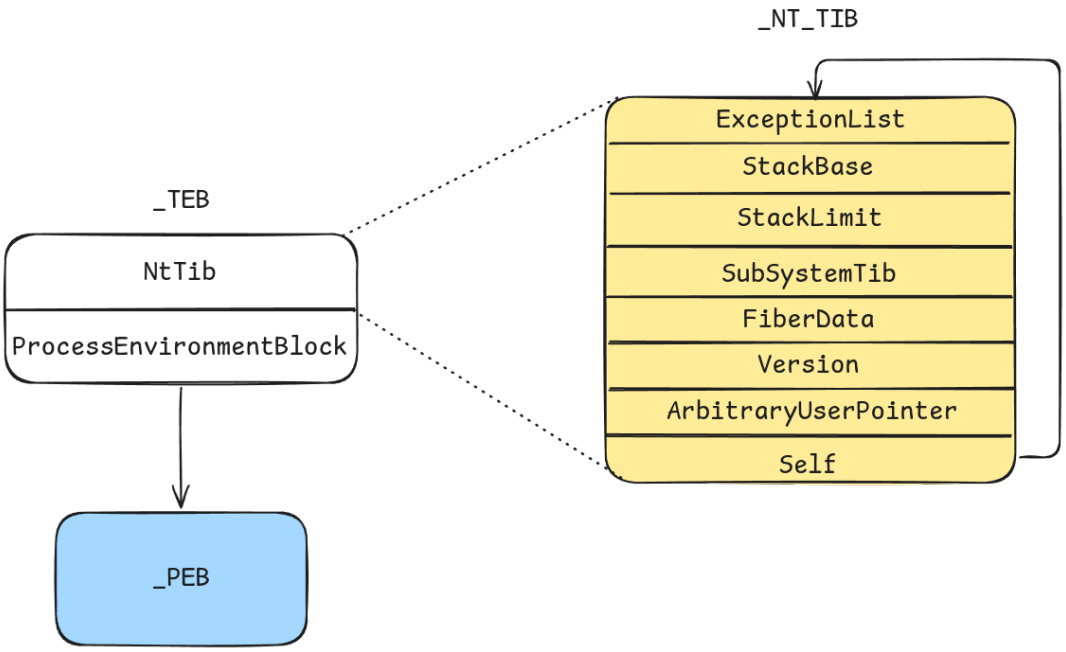

FIGURE 4.11: _TEB & _NT_TIB Structures [9]

4.3.8 Complete Diagram of Process Components

Based on all the aforementioned Structures and Relations, a complete diagram of the organisation of a Process' Executive, Kernel and User objects is shown below:

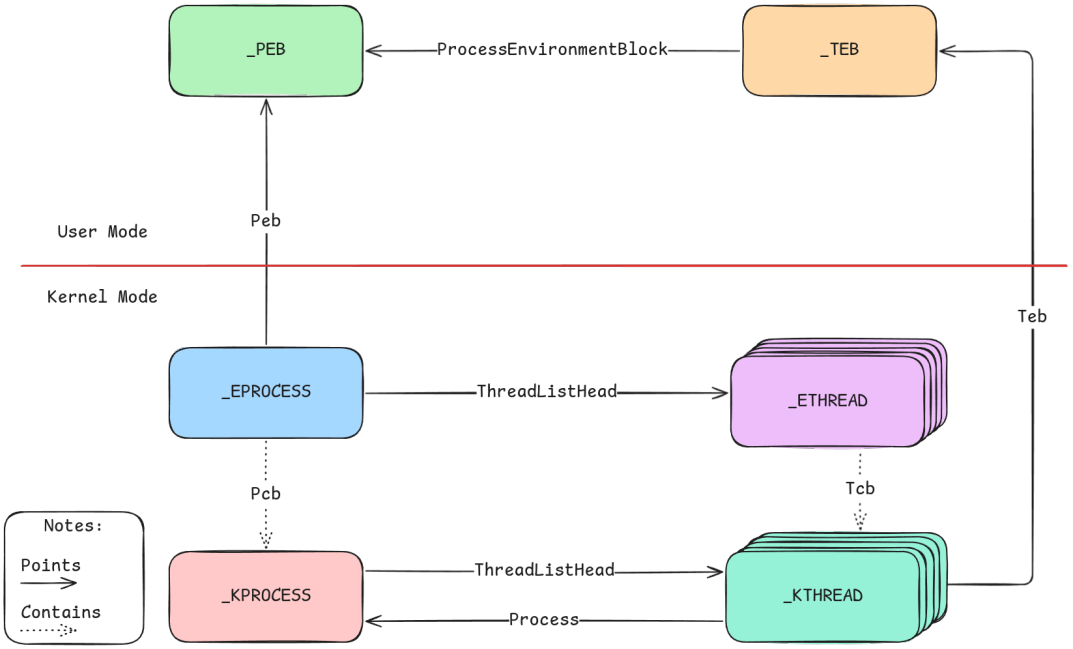

FIGURE 4.12: Complete Diagram of Core Process Components

Note: While not directly visible in the diagram, since `_KPROCESS` is the first field of `_EPROCESS` if one needs to retrieve the `_EPROCESS` object, then they need only to cast the address of the `_KPROCESS` object to an `_EPROCESS` object since they have the same address. The same applies for `_KTHREAD/_ETHREAD` and `_TEB/_NT_TIB`.

Note 2: While specifically Kernel space structures, the `_EPROCESS` (and more) Structure can be mapped into the User space address space as well.

4.4 Virtual Address Descriptors (VADs)

VADs are a meta-data-like structure that describe virtually continuous memory allocations made through `VirtualAlloc()` that share the same characteristics. When a program tries to allocate memory, either by User invocation or for loading modules, a call to `VirtualAlloc()` is made and a VAD will be constructed in the system address space that will be used by the Memory Manager to track the Memory Allocation.

Suppose a thread requests a large block of memory. The memory manager could immediately reserve and commit it—allocating all the necessary page tables (see Section 3.8). But if the thread never touches most of that memory, or delays accessing it, the OS's work in mapping and zeroing every page is wasted. [10]

By creating a VAD to track this allocation, Windows can stagnate the allocation of those PTEs. If the program tries to access an address within the allocated region and the Memory Manager has not allocated the memory yet, a Page Fault is triggered. The Memory Manager will then look for a VAD that describes the specific request of the user. When said VAD is found, the Memory Manager will allocate and commit any (or part) of the remaining data.

If, however, there is no VAD that describes that request then this means no allocation (no call to `VirtualAlloc`) took place, and that constitutes an Address Violation!

4.4.1 Memory States

There are three possible Memory States (that pertain to its physical availability):

1. **Free**: Free Memory is memory that has not been reserved nor committed and it is available
2. **Reserved**: Memory that has been reserved, but no commitments have been made. It must be committed before use
3. **Committed**: Memory that when translated will result in valid physical memory addresses $\Leftrightarrow$ PTEs have been allocated for this region

As well as attributes not limited to:

1. **Private**: Memory for use in a single process, never shared. Private committed memory pages that have not been accessed yet, are zero initialized on first access: "On demand Zero"
2. **Image**: Memory Mapped into the view of an executable Image (EXE/DLL)
3. **Mapped**: Memory that is mapped to files using memory mapping APIs

All of the above 'types' of memory allocations are tracked by VADs.

VADs are allocated from the Non-Paged Pool and, as such:

- Exist in Kernel Space
- They are preceded by the `_POOL_HEADER` object

Although there exist more than the following VAD types, the two most predominant types on which this work focusses and their associated Pool Tag are:

Pool Tag	VAD Type	Description
VadS	_MMVAD_SHORT	Private (Anonymous) Region allocated by the VMM
VadF	_MMVAD_SHORT	VAD Created by FreeVM splitting
Vad*	_MMVAD	VMM Allocations (long)

TABLE 4.2: Notable VAD Types and associated Pool Tags

VADs are organised in a self-balanced AVL Tree (named after its creators Adelson-Velsky-Landis), where the heights of the two child subtrees of any node differ by at most one. A node describing a memory region lower than its parent goes to the left, whilst the node that describes a higher region to the right. This makes searching, removing and adding memory regions more efficient than a single and even a doubly linked list.

As already mentioned, the _MMVAD_SHORT (with a Tag of VadS) is used for Private memory allocations. This covers one of the three cases of memory attributes and leaves **Image** and **Mapped** to be differentiated (since they share the same VAD Tag and Type).

```
(layer_name) >>> dt('_MMVAD')
symbol_table_name1!_MMVAD (136 bytes):
 0x0 :   Core                  _MMVAD_SHORT
0x40 :   VadFlags2             _MMVAD_FLAGS2
0x48 :   Subsection            *_SUBSECTION
0x50 :   FirstPrototypePte     *_MMPTE
0x58 :   LastContiguousPte     *_MMPTE
0x60 :   ViewLinks             _LIST_ENTRY
0x70 :   VadsProcess           *_EPROCESS
0x78 :   u4                    __unnamed_38ba
0x80 :   FileObject            *_FILE_OBJECT
```

FIGURE 4.13: _MMVAD

```
(layer_name) >>> dt('_MMVAD_SHORT')
symbol_table_name1!_MMVAD_SHORT (64 bytes):
 0x0 :   NextVad                *_MMVAD_SHORT
 0x0 :   VadNode                _RTL_BALANCED_NODE
 0x8 :   ExtraCreateInfo        *void
0x18 :   StartingVpn            unsigned long
0x1c :   EndingVpn              unsigned long
0x20 :   StartingVpnHigh        unsigned char
0x21 :   EndingVpnHigh          unsigned char
0x22 :   CommitChargeHigh       unsigned char
0x23 :   SpareNT64VadUChar      unsigned char
0x24 :   ReferenceCount         long
0x28 :   PushLock               _EX_PUSH_LOCK
0x30 :   u                      __unnamed_20b2
0x34 :   CommitCharge           unsigned long
0x38 :   u5                     __unnamed_20b6
```

FIGURE 4.14: _MMVAD_SHORT

As evident, the two structures vary quite a bit, with most important the presence of the Subsection field in the _MMVAD type, which in turn has a pointer to the _CONTROL_AREA object:

```
(layer_name) >>> dt('_CONTROL_AREA')
symbol_table_name1!_CONTROL_AREA (128 bytes):
 0x0 :   Segment                        *_SEGMENT
 0x8 :   AweContext                     *void
 0x8 :   ListHead                       _LIST_ENTRY
0x18 :   NumberOfSectionReferences      unsigned long long
0x20 :   NumberOfPfnReferences          unsigned long long
0x28 :   NumberOfMappedViews            unsigned long long
0x30 :   NumberOfUserReferences         unsigned long long
0x38 :   u                              __unnamed_207c
0x3c :   u1                             __unnamed_207f
0x40 :   FilePointer                    _EX_FAST_REF
0x48 :   ControlAreaLock                long
0x4c :   ModifiedWriteCount             unsigned long
0x50 :   WaitList                       *_MI_CONTROL_AREA_WAIT_BLOCK
0x58 :   u2                             __unnamed_2087
0x68 :   FileObjectLock                 _EX_PUSH_LOCK
```

```
0x70 :   LockedPages                    unsigned long long
0x78 :   u3                             __unnamed_208b
```

A VAD having a `_SUBSECTION` field means that it is backed by a section object (such as a mapped file or image) and `_CONTROL_AREA` is the object responsible for managing the underlying section object. While their functionality is crucial, they are out of scope since there is a much more simple method to differentiate between a file backed VAD one or an image backed one.

`_CONTROL_AREA` is a Kernel object, and thus, as already mentioned, it is allocated from a Pool! And if 4.2 is recalled, there is an associated Tag which can be used to identify if the current VAD is backed by a image or a file!

By reading the Tag (4-byte value located at `-0xc`) one can identify two distinct Tags:

- `MmCa`: Control Area for Mapped Files
- `MmCi`: Control Area for Images

There is more information one can get from the VADs, but it is self-explanatory and will be covered in 5.2, along with the relevant implementation details.

Chapter 5

Minidump Analysis & Export

5.1 Minidump

A Minidump is a compact binary that contains a snapshot of a process at one specific point in time. They can be exported per user's request or configured in the operating system to be exported when a crash happens. They are widely used in Windows to debug and diagnose application failures, however, their memory contents density is useful for security auditors (for security engagements) and forensic analysts (for post-mortem analysis) without the overhead of the entirety of the (physical) memory. Minidumps are usually analysed using `WinDBG` which can be extended with plugins for extra functionality.

5.1.1 Structure

While the structure of a Minidump is rather simple, it can create some interesting avenues for exploration:

```
typedef struct _MINIDUMP_HEADER {
  ULONG32 Signature;
  ULONG32 Version;
  ULONG32 NumberOfStreams;
  RVA     StreamDirectoryRva;
  ULONG32 CheckSum;
  union {
    ULONG32 Reserved;
    ULONG32 TimeDateStamp;
  };
```

```
  ULONG64 Flags;
} MINIDUMP_HEADER, *PMINIDUMP_HEADER;
```

Some takeaways from the header described above:

- `Signature`: Always `b'MDMP'/0x504d444d`
- `NumberOfStreams`: Number of (Minidump) Streams contained
- `StreamDirectoryRva`: Of type `RVA:ULONG32`. A pointer to the streams directory, an array of `MINIDUMP_DIRECTORY`
- `Flags`: Define what streams should the parsing tool except to exist in the Minidump:
  - `MiniDumpNormal`: Minimal info for stack traces.
  - `MiniDumpWithFullMemory`: Captures all accessible memory, making the dump much larger.
  - `MiniDumpWithHandleData`: Includes information about handles.
  - `MiniDumpWithUnloadedModules`: Tracks recently unloaded modules.
  - `MiniDumpWithThreadInfo`: Adds thread state info.
  - `MiniDumpWithFullMemoryInfo`: Detailed memory region info.

The Minidump file provides the parsing tool with the whole of the User space memory of the process, so effectively every possible VAD (4.4). This is accomplished with the use of two Minidump structures:

- `MINIDUMP_LOCATION_DESCRIPTOR`: The simplest form of memory containment, just a size and location field

```
typedef struct _MINIDUMP_LOCATION_DESCRIPTOR {
  ULONG32 DataSize;
  RVA     Rva;
} MINIDUMP_LOCATION_DESCRIPTOR;
```

- `MINIDUMP_MEMORY_DESCRIPTOR`: Builds on `MINIDUMP_LOCATION_DESCRIPTOR` by providing the `StartOfMemoryRange` field. The parsing tool will use this to construct the allocated Virtual Memory space!

```
typedef struct _MINIDUMP_MEMORY_DESCRIPTOR {
  ULONG64                      StartOfMemoryRange;
```

```
  MINIDUMP_LOCATION_DESCRIPTOR Memory;
} MINIDUMP_MEMORY_DESCRIPTOR, *PMINIDUMP_MEMORY_DESCRIPTOR;
```

5.2 Exporting Minidump

As already mentioned, a Minidump consists of a series of Streams, and this work describes how using Volatility, one can export said streams from the memory artefact and construct the Minidump from them. This work attempts to export nine streams in this order:

1. `ThreadListStream`
2. `ThreadInfoListStream`
3. `ModuleListStream`
4. `UnloadedModuleListStream`
5. `MiscInfoStream`
6. `SystemInfoStream`
7. `HandleDataStream`
8. `MemoryInfoListStream`
9. `Memory64ListStream`

Below there are the stream structure definitions and implementation details for the exporting of each stream individually

5.2.1 `ThreadListStream`

An array of `MINIDUMP_THREAD` Structures describing the active Threads of the process. As already mentioned in 4.3.4, if the `ExitTime` field in the `_ETHREAD` structure is positive, then the thread as exited:

```
typedef struct _MINIDUMP_THREAD {
  ULONG32                     ThreadId;
  ULONG32                     SuspendCount;
  ULONG32                     PriorityClass;
  ULONG32                     Priority;
  ULONG64                     Teb;
  MINIDUMP_MEMORY_DESCRIPTOR  Stack;
  MINIDUMP_LOCATION_DESCRIPTOR ThreadContext;
} MINIDUMP_THREAD, *PMINIDUMP_THREAD;
```

Exporting:

1. **ThreadId**: `_ETHREAD.Cid.UniqueThread`
2. **SuspendCount**: `_ETHREAD.Tcb.SuspendCount`
3. **PriorityClass**: Hardcoded to 32
4. **Teb**: `_ETHREAD.Tcb.Teb`
5. **Stack**: Points to the stack data, see 5.2.1
6. **ThreadContext**: Points to a `_CONTEXT` Structure, constructed from the data of the `_KTRAP_FRAME` located at `_ETHREAD.Tcb.TrapFrame`

Stack

In order to retrieve the thread's stack, one can read the `RSP` value from the Trap Frame, and the Stack Base from the `_NT_TIB` structure. Creating the `_NT_TIB` object in volatility is really simple:

```
nt_tib = self.context.object(
    object_type=f"{symbol_table_name}{constants.BANG}_NT_TIB",
    layer_name=proc_layer_name,
    offset=thread.Tcb.Teb,
)
```

FIGURE 5.1: Object Creation in Volatility

Arguments Takeaway:

1. `object_type`: The type of the object that is being created
2. `layer_name`: The layer the object exists. In this case, `_TEB` (and `_NT_TIB`) exist in the virtual layer of the process
3. `offset`: Where to create the object
4. `native_layer_name`: Not used here, the layer to which the pointers of the current object point to

By creating the objects in their layer, one uses Volatility's built-in mechanisms for accessing data, rather than python generic functions, i.e.: `struct.unpack`.

5.2.2 `ThreadInfoListStream`

Contains information about all threads of the process. An array of `MINIDUMP_THREAD_INFO` structures.

```
typedef struct _MINIDUMP_THREAD_INFO {
  ULONG32 ThreadId;
  ULONG32 DumpFlags;
  ULONG32 DumpError;
  ULONG32 ExitStatus;
  ULONG64 CreateTime;
  ULONG64 ExitTime;
  ULONG64 KernelTime;
  ULONG64 UserTime;
  ULONG64 StartAddress;
  ULONG64 Affinity;
} MINIDUMP_THREAD_INFO, *PMINIDUMP_THREAD_INFO;
```

Exporting:

1. **ThreadId**: `_ETHREAD.Cid.UniqueThread`
2. **DumpFlags**: `0x4` if the thread has exited, else 0
3. **DumpError**: Hardcoded to `0`. Since acquisition already happen (memory artefact), there were no errors to begin with
4. **ExitStatus**: `_ETHREAD.ExitStatus`
5. **ExitTime**: `0` if the thread is running, else `_ETHREAD.ExitTime.QuadPart`
6. **KernelTime**: `_ETHREAD.Tcb.KernelTime`
7. **UserTime**: `_ETHREAD.Tcb.UserTime`
8. **StartAddress**: `_ETHREAD.StartAddress`
9. **Affinity**: `_ETHREAD.Tcb.Affinity.Bitmap[0]`, `0` if non-existent

5.2.3 ModuleListStream

An array of `MINIDUMP_MODULE` structures, describing the loaded modules (DLLs, EXEs, etc.).

```
typedef struct _MINIDUMP_MODULE {
  ULONG64                     BaseOfImage;
  ULONG32                     SizeOfImage;
  ULONG32                     CheckSum;
  ULONG32                     TimeDateStamp;
  RVA                         ModuleNameRva;
```

```
  VS_FIXEDFILEINFO             VersionInfo;
  MINIDUMP_LOCATION_DESCRIPTOR CvRecord;
  MINIDUMP_LOCATION_DESCRIPTOR MiscRecord;
  ULONG64                      Reserved0;
  ULONG64                      Reserved1;
} MINIDUMP_MODULE, *PMINIDUMP_MODULE;
```

In order to export the modules' information, the list of the processes modules need to be traversed:

```
(layer_name) >>> ldr_modules = list(proc.load_order_modules())
(layer_name) >>> dt(ldr_modules[0])
symbol_table_name1!_LDR_DATA_TABLE_ENTRY (312 bytes) @ 0x29511105900
```

From here, every index of the `ldr_modules` will be referred as plain `module` for the exporting details:

1. **BaseOfImage**: `module.DllBase`
2. **SizeOfImage**: `module.SizeOfImage`
3. **Checksum**: `module.Checksum`
4. **TimeDateStamp**: `module.TimeDateStamp`
5. **ModuleNameRva**: Pointer to `MINIDUMP_STRING`
6. **VersionInfo/MiscRecord**: Unneeded, zero-filled
7. **CvRecord**: Code View record. A really important structure that defines the PDB (Portable Debugger) symbol name of the module. It is used by WinDBG to download appropriate symbols for this module, see 5.2.3

Code View

The Code View information lies in the Debug Directory, inside the optional header of the file. In order to arrive to this, there are some steps one must take (using Volatility mechanisms)

Firstly, the `IMAGE_DOS_HEADER` object needs to be created on the base address of the module, and from there the `IMAGE_NT_HEADERS64`:

```
(layer_name) >>> dos_hdr = module.DllBase.dereference().cast(
    f"{symbol_table}{constants.BANG}_IMAGE_DOS_HEADER"
)
```

```
(layer_name) >>> nt_hdr = proc_layer.context.object(
    object_type=f"{symbol_table}{constants.BANG}_IMAGE_NT_HEADERS64",
    layer_name=proc_layer_name,
    offset=module.DllBase + dos_hdr.e_lfanew,
)
```

Afterwards, the debug directory in the Optional Header is retrieved and the Code View record is read:

```
dbg_dir_va = nt_hdr.OptionalHeader.DataDirectory[
    DEBUG_DIRECTORY_ENTRY
].VirtualAddress

dbg_dir = proc_layer.context.object(
    object_type=f"{symbol_table}{constants.BANG}_IMAGE_DEBUG_DIRECTORY",
    layer_name=proc_layer_name,
    offset=module.DllBase + dbg_dir_va,
)
[...]
minidump_module._code_view = proc_layer.read(
    offset=module.DllBase + dbg_dir.AddressOfRawData,
    length=dbg_dir.SizeOfData,
)
```

5.2.4 UnloadedModuleListStream

An array of `MINIDUMP_UNLOADED_MODULE` structures. The symbol `RtlpUnloadEventTrace` is an array of `_RTL_UNLOAD_EVENT_TRACE` structure that holds module unload events residing in `NTDLL.DLL`'s memory section (in the current process). This symbol is retrieved using Volatility's internal tooling.

```
typedef struct _RTL_UNLOAD_EVENT_TRACE {
    PVOID BaseAddress;   // Base address of dll
    SIZE_T SizeOfImage;  // Size of image
    ULONG Sequence;      // Sequence number for this event
    ULONG TimeDateStamp; // Time and date of image
    ULONG CheckSum;      // Image checksum
    WCHAR ImageName[32]; // Image name
} RTL_UNLOAD_EVENT_TRACE, *PRTL_UNLOAD_EVENT_TRACE;
```

```
typedef struct _MINIDUMP_UNLOADED_MODULE {
  ULONG64 BaseOfImage;
  ULONG32 SizeOfImage;
  ULONG32 CheckSum;
  ULONG32 TimeDateStamp;
  RVA     ModuleNameRva;
} MINIDUMP_UNLOADED_MODULE, *PMINIDUMP_UNLOADED_MODULE;
```

Exporting:

Iterating over the array with a window size of `0x68` byte (due to the padding of `_RTL_UNLOAD_EVENT_TRACE`'s structure) and parsing its data, the stream can be created.

There exists some checks in place that invalidate an unload event:

- `BaseAddress == 0`
- `SizeOfImage == 0`
- `SizeOfImage > 0x10000000`

5.2.5 MiscInfoStream

Contains miscellaneous information about the System and the Process; Creation Time, User and Kernel Time, etc:

```
typedef struct _MINIDUMP_MISC_INFO_2 {
  ULONG32 SizeOfInfo;
  ULONG32 Flags1;
  ULONG32 ProcessId;
  ULONG32 ProcessCreateTime;
  ULONG32 ProcessUserTime;
  ULONG32 ProcessKernelTime;
  ULONG32 ProcessorMaxMhz;
  ULONG32 ProcessorCurrentMhz;
  ULONG32 ProcessorMhzLimit;
  ULONG32 ProcessorMaxIdleState;
  ULONG32 ProcessorCurrentIdleState;
} MINIDUMP_MISC_INFO_2, *PMINIDUMP_MISC_INFO_2;
```

The platform specific details (Frequency and CPU States) are not of much importance.

5.2.6 SystemInfoStream

Contains information about the CPU. Populated by the data residing in the KUSER_SHARED_DATA structure, which always exists in the same address, inside the Kernel Layer. This structure is retrieved by Volatility's info.Info.get_kuser_structure and the resulting fields are almost identical with the following, apart from the ProcessorLevel, which is retrieved using the KPCRs structures.

```
typedef struct _MINIDUMP_SYSTEM_INFO {
  USHORT          ProcessorArchitecture;
  USHORT          ProcessorLevel;
  USHORT          ProcessorRevision;
  UCHAR           NumberOfProcessors;
  ULONG32         MajorVersion;
  ULONG32         MinorVersion;
  ULONG32         BuildNumber;
  ULONG32         PlatformId;
  RVA             CSDVersionRva;
  USHORT          SuiteMask;
  CPU_INFORMATION Cpu;
} MINIDUMP_SYSTEM_INFO, *PMINIDUMP_SYSTEM_INFO;
```

5.2.7 HandleDataStream

Describes the handles of the process at the time the Minidump was written, an array of MINIDUMP_HANDLE_DESCRIPTOR.

```
typedef struct _MINIDUMP_HANDLE_DESCRIPTOR {
  ULONG64 Handle;
  RVA     TypeNameRva;
  RVA     ObjectNameRva;
  ULONG32 Attributes;
  ULONG32 GrantedAccess;
  ULONG32 HandleCount;
  ULONG32 PointerCount;
} MINIDUMP_HANDLE_DESCRIPTOR, *PMINIDUMP_HANDLE_DESCRIPTOR;
```

In order to retrieve the open handles, the Type Map and Cookie need to be acquired:

```
type_map = handles.Handles.get_type_map(
    context=self.context, kernel_module_name=self.config["kernel"]
)

cookie = handles.Handles.find_cookie(
    context=self.context, kernel_module_name=self.config["kernel"]
)
```

Then, by iterating over the open handles of the process, the type of the handle can be retrieved:

```
for ent in handles.Handles.handles(
    self.context, self.config["kernel"], proc.ObjectTable
):
    try:
        obj_type = ent.get_object_type(type_map, cookie)
        if obj_type is None:
            continue

        if obj_type == "File":
            item = ent.Body.cast("_FILE_OBJECT")
            obj_name = item.file_name_with_device()
        elif obj_type == "Process":
            item = ent.Body.cast("_EPROCESS")
            obj_name = utility.array_to_string(item.ImageFileName)
        elif obj_type == "Thread":
            item = ent.Body.cast("_ETHREAD")
            obj_name = f"Tid {item.Cid.UniqueThread}"
        elif obj_type == "Key":
            item = ent.Body.cast("_CM_KEY_BODY")
            obj_name = item.get_full_key_name()
        else:
            try:
                obj_name = ent.NameInfo.Name.String
            except (ValueError, exceptions.InvalidAddressException):
                obj_name = None
    except exceptions.InvalidAddressException:
        vollog.log(
            constants.LOGLEVEL_VVV,
```

```
                f"Cannot access _OBJECT_HEADER at {ent.vol.offset:#x}",
            )
            continue
        [...]
```

After finding the object type, then everything else is straight forward, since the handle object has the same fields with the stream structure.

5.2.8 `MemoryInfoListStream` & `Memory64ListStream`

This stream was by far the most difficult and cumbersome to extract, however it was also the most wholesome because it ties everything from the above Chapters together in almost magical cohesion.

The `MemoryInfoListStream` is an array of `MINIDUMP_MEMORY_INFO` structs with the following fields:

```
typedef struct _MINIDUMP_MEMORY_INFO {
  ULONG64 BaseAddress;
  ULONG64 AllocationBase;
  ULONG32 AllocationProtect;
  ULONG32 __alignment1;
  ULONG64 RegionSize;
  ULONG32 State;
  ULONG32 Protect;
  ULONG32 Type;
  ULONG32 __alignment2;
} MINIDUMP_MEMORY_INFO, *PMINIDUMP_MEMORY_INFO;
```

As already mentioned in 5.2.10, since Minidump objects are represented as `ctypes.Structure` objects, the data represented by the above structure will reside in `self._memory` and will be used later when building `Memory64ListStream`.

Before beginning to exporting the process memory, the export method sets up some prerequisite variables:

- Retrieves and sorts the VADs of the process and creates a parallel array of the starting addresses of them:

```
vad_list = proc.get_vad_root().traverse()
vad_list: List[extensions.MMVAD] = sorted(
    vad_list,
    key=lambda x: x.get_start()
```

```
)
vad_starts = [v.get_start() for v in vad_list]
```

- Retrieves **all** valid Memory Regions:

```
addresses = list(
    proc_layer.mapping(
    0x0,
    proc_layer.maximum_address,
    ignore_errors=True)
)
```

Then it begins by iterating over each valid memory range, which is described as the tuple of (**virtaddr**, **size**, **physaddr**, **size**, **layer**) and the values of interest are the:

- Virtual Address
- Size

As mentioned in 4.4, VADs track all Virtual Allocation for a process, so the current address must reside in a VAD. Having the VADs sorted, and incidentally the sorted array of their starting values, allows for performing a binary search to find the specific VAD and as a result, significantly drops the Time Complexity from $O(n^2) \rightarrow O(nlogn)$ and slashing the run times from 9 minutes to 1.5 (for a specific process), compared to a nested linear search.

Having now identified the 'parent' VAD, information can now be extracted from it to populate the above Minidump Structure.

- **BaseAddress** is the Virtual Address of the beginning of the returned Memory Region
- **RegionSize** is the Size returned from `.mapping()`
- **AllocationBase** is the beginning of the VAD (could not have a valid PTE associated)
- **AllocationProtect** is taken from `MMVAD_SHORT.Protection` with the help of `VadInfo.protect_values()`
- **State** is hardcoded to `0x1000 / MEM_COMMIT` since all addresses are returned from `.mapping()` $\Leftrightarrow$ there exists a physical address $\Leftrightarrow$ there exists a valid PTE

- **Type** is differentiated from the existence of the `Subsection` field in the VAD and if so the Pool Tag of `CONTROL_AREA` as described in 4.4.1.

This leaves the `Protect` to be populated, which unlike the `AllocationProtect` depicts the current protections and the value can be determined by the PTE address i.e. the **PFN** of the PT (Page Table) in diagram 3.5. Volatility can easily provide this address using the `Intel._translate_entry()` which returns the full PTE that will later be reconstructed to the proper physical address:

```
[layer_name]> pl._translate_entry(0x7ffebd650000)
Out[14]: (9367487233579171845, 11)

[layer_name]> hex(9367487233579171845)
Out[15]: '0x82000002037e4005'

[layer_name]> pl.translate(0x7ffebd650000)
Out[16]: (8648540160, 'memory_layer')

[layer_name]> hex(8648540160)
Out[17]: '0x2037e4000'
```

As such, the current page protections can be calculated as so:

```
nx = bool(pte_value & NX_MASK)
rw = bool(pte_value & RW_MASK)

if nx:
    mem_info.Protect =
    MEM_PROTECT.PAGE_READWRITE.value if rw
    else MEM_PROTECT.PAGE_READONLY.value
else:
    mem_info.Protect =
    MEM_PROTECT.PAGE_EXECUTE_READWRITE.value if rw
    else MEM_PROTECT.PAGE_EXECUTE_READ.value
```

5.2.9 MEM_IMAGE Expansion

MemProcFS enumerates Memory by iterating over the PTEs of each process, by doing so it also iterates over the Prototype-PTEs as well. These are non-Committed PTEs but can still (sometimes) translate properly because the Image is physically there, just not mapped into the process because it was not

requested. By expanding the Memory Region boundaries, a more complete Module (VAD Export) can be captured from parts that are not in the Process Memory but still exist physically. But this would only help create a more portable Minidump because the image itself, resides within the full memory snapshot which is also accessible. The approach is to extend upwards until the previous memory region or VAD starting point (whichever comes first) and extend downwards at most `0x400000` (`4MB`) or until the VAD ending point.

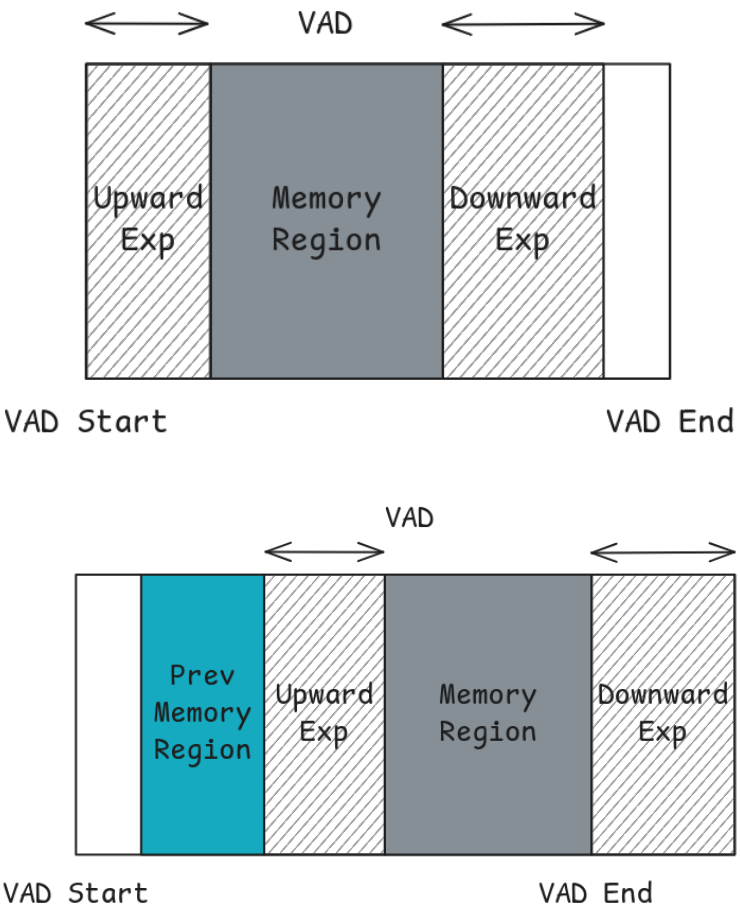

FIGURE 5.2: Memory Region Expansion

The resulting memory is then read into the `._memory` field of the current (modelled) object.

Finally, neighbouring regions that share the same Protection and Type and they are continuous are merged.

5.2.10 Packaging

Unlike other projects that deal with Minidump Files, this project does not interact with the Minidump itself, therefore, there is no need for a Minidump Python library. What is needed, however, is a correct and simple method to store the retrieved data into a proper structure without the need to manually define the Struct size nor the size of each field (when exporting to raw bytes). Since the only concern is 'easily storing' the retrieved data and quickly exporting the whole of the structure into the correct size, Python's built-in `ctypes.Structure` was used as a base Object for all the (repeatable) structures to inherit from.

Take for example the `MINIDUMP_MODULE` Struct:

```
typedef struct _MINIDUMP_MODULE {
  ULONG64                      BaseOfImage;
  ULONG32                      SizeOfImage;
  ULONG32                      CheckSum;
  ULONG32                      TimeDateStamp;
  RVA                          ModuleNameRva;
  VS_FIXEDFILEINFO             VersionInfo;
  MINIDUMP_LOCATION_DESCRIPTOR CvRecord;
  MINIDUMP_LOCATION_DESCRIPTOR MiscRecord;
  ULONG64                      Reserved0;
  ULONG64                      Reserved1;
} MINIDUMP_MODULE, *PMINIDUMP_MODULE;
```

It can easily be represented in Python using a custom class that inherits from `ctypes.Structure`:

This allows one to set the size of the fields, so when exporting the structure will be automatically packed into the correct ranges just by doing `bytes(MINIDUMP_MODULE()` which also contains unpopulated fields zeroed out. For some structures, the `_pack_` field needed to be set to `1` to avoid any alignment optimizations from Python's side.

In addition, the usage of a custom class allows for storing the data that will be later referenced by `RVA` pointers. The data is written when exporting in a serial manner for each stream and the `RVA`s are assigned dynamically.

```
class MINIDUMP_MODULE(ctypes.Structure):
_pack_ = 1
_fields_ = [
    ("BaseOfImage",    ctypes.c_uint64),
    ("SizeOfImage",    ctypes.c_uint32),
    ("CheckSum",       ctypes.c_uint32),
    ("TimeDateStamp",  ctypes.c_uint32),
    ("ModuleNameRva",  ctypes.c_uint32),
    ("VersionInfo",    VS_FIXEDFILEINFO),
    ("CvRecord",       MINIDUMP_LOCATION_DESCRIPTOR),
    ("MiscRecord",     MINIDUMP_LOCATION_DESCRIPTOR),
    ("Reserved0",      ctypes.c_uint64),
    ("Reserved1",      ctypes.c_uint64),
]

_module_name = b""
_code_view = b""

def size(self) -> int:
    return len(bytes(self)) + len(self._module_name) + len(self._code_view)
```

5.2.11 Real Life Proof of Concept

While experimenting with the Empire C2 framework, it was made apparent, that the session key that's being exchanged between the Agent and the C2 Server exists in the form of a **.NET** object inside memory and can be retrieved by examining the heap.

5.2.12 Empire EKE

Disclaimer: This work focuses on an older version of Empire where the Key can be retrieved with this (exact) same method, but the same principle should apply in theory to newer versions as well.

Empire uses a simple yet effective Diffie-Hellman-like EKE (Encrypted Key Exchange) to exchange a shared **AES256** bit key between the Victim and the C2 Server:

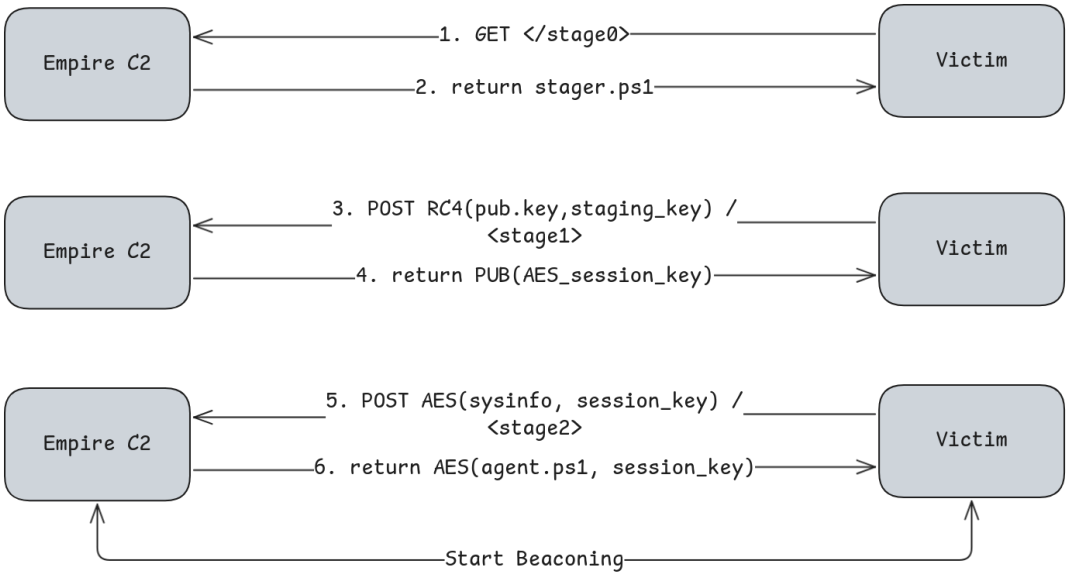

FIGURE 5.3: Empire's EKE

1. Victim runs the initial entry point program
2. C2 Server responds with the `stager.ps1` which contains a plaintext staging key
3. Stager generates a pair of `RSA` keys on the client and sends the Public key encrypted with the staging key over to the C2 Server
4. C2 Server generates an **AES256** key that is to be used for the shared communication later. It encrypts the session key using the client's Public Key and sends it over
5. Now Victim and C2 share a secret session key. The victim will send an AES encrypted packet containing some system information
6. C2 Server sends `agent.ps1` to the victim and Beaconing is started

Inside the `agent.ps1`'s code, there exist two objects of interest:

1. `System.Security.Cryptography.HMACSHA256`
2. `System.Security.Cryptography.AesCryptoServiceProvider`

There exists tools that given a Minidump (or a memory snapshot) can inspect the Heap for **.NET** objects, and by extension their members:

- `netext`
- `SOS.dll`
- Memory analysis of .NET and .Net Core applications [1]

For this work, `netext` was chosen for its simplicity of installation and usage, and also because it does not depend on the `sos.dll` libraries. Additionally, the session key Empire generates is by default, printable, so a custom, patched (and private) version was used.

A logging line was added after the Key generation to print out the session key:

```
[INFO]: http: Sending POWERSHELL stager (stage 1) to 172.17.0.1
[INFO]: Agent 283LEFTA from 172.17.0.1 posted public key
[INFO]: Agent 283LEFTA from 172.17.0.1 posted valid PowerShell RSA key
[INFO]: Generated Key: 5bed9977e60c276ee689c082cfb3a911337c2071dc003b410197b782d20fe2e5
[INFO]: New agent 283LEFTA checked in
[INFO]: Initial agent 283LEFTA from 172.17.0.1 now active (Slack)
[INFO]: http: Sending agent (stage 2) to 283LEFTA at 172.17.0.1
```

FIGURE 5.4: Debug Print of AES Session Key

Now, by opening the exported Minidump into WinDBG, one can load the `netext` plugin with `.load netext`:

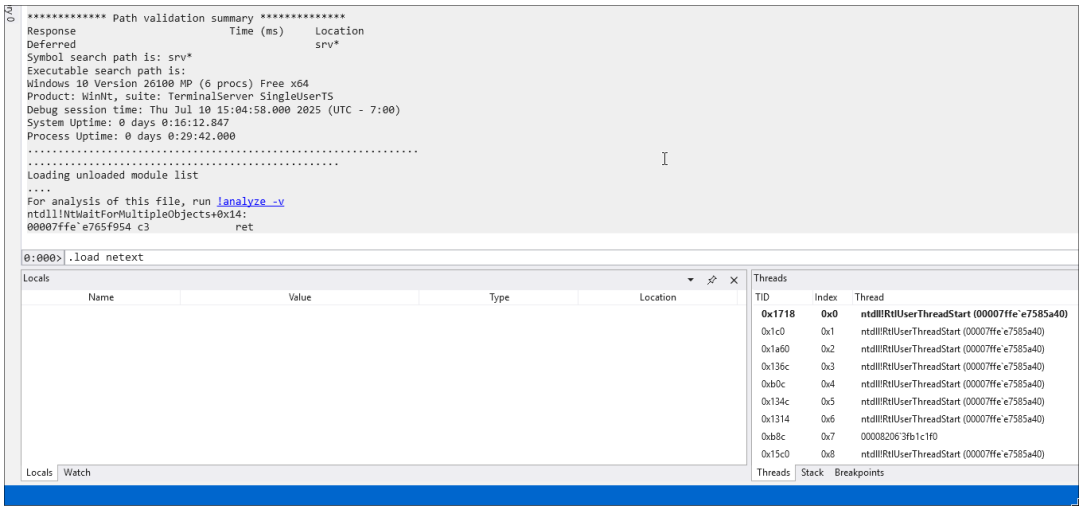

FIGURE 5.5: Loading `netext`

After loading the index table needs to be filled using `!windex -tree` and after indexing the found heap types can be displayed using `!windex -enumtypes`:

```
00007ffec2943480 System.Security.Cryptography.CapiNative_SafeNativeMethods_<>c (1)
00007ffe63932268 System.Security.Cryptography.CipherMode[] (1)
00007ffec16ee5f0 System.Security.Cryptography.CspParameters (2)
00007ffec28a9160 System.Security.Cryptography.CspProviderFlags (1)
00007ffec16e53b0 System.Security.Cryptography.HMACSHA256 (2)
00007ffec291c1f8 System.Security.Cryptography.KeySizes (6)
00007ffec28d60f8 System.Security.Cryptography.KeySizes[] (5)
00007ffec291cba8 System.Security.Cryptography.NativeHmac (2)
00007ffec16ee650 System.Security.Cryptography.RNGCryptoServiceProvider (2)
00007ffec1707300 System.Security.Cryptography.RSACryptoServiceProvider (1)
00007ffec291c0c8 System.Security.Cryptography.SafeCspHandle (1)
00007ffec291c9e8 System.Security.Cryptography.SafeKeyHandle (1)
00007ffec1701ea0 System.Security.Cryptography.SafeProvHandle (2)
```

FIGURE 5.6: .NET Heap Objects

There is no mention of an `AES` object, but there is for `HMAC256`:

```
0:000> !windex -mt 00007ffec16e53b0
Index is up to date
 If you believe it is not, use !windex -flush to force reindex
Address           MT                 Size Heap Gen Type Name
0000025c1c8a78d8 00007ffec16e53b0      104   0   2 System.Security.Cryptography.HMACSHA256
0000025c1cb00b40 00007ffec16e53b0      104   0   2 System.Security.Cryptography.HMACSHA256

2 Objects listed
```

FIGURE 5.7: `HMAC256` Objects

And it just so happens the first being the object of the Agent. To inspect an object, the command `!wdo 0000025c1c8a78d8` can be used:

```
9\mscorlib.dll
stem.Security.Cryptography.HashAlgorithm System.Object (00007FFEC291CB30 00007FFEC291CC20 00007FFEC16E2360 00007FFEC16F0BB0)
        HashValue 0000025c1c9aa2c0 33 5f 22 a9 a3 1c 07 3b 58 1a 64 af 6b 43 c7 66 b6 97 1f 8d 84 12 8b cc 65 5d 55 06 48 24 a4 11
    HashSizeValue 100 (0n256)
            State 0 (0n0)
      m_bDisposed 0 (False)
         KeyValue 0000025c1c9aa268 5b ed 99 77 e6 0c 27 6e e6 89 c0 82 cf b3 a9 11 33 7c 20 71 dc 00 3b 41 01 97 b7 82 d2 0f e2 e5
       m_hashName 0000025c1c2f6348 SHA256
          m_hash1 0000000000000000
          m_hash2 0000000000000000
           m_impl 0000025c1c8a7960
          m_inner 0000000000000000
          m_outer 0000000000000000
   blockSizeValue 40 (0n64)
        m_hashing 0 (False)
```

FIGURE 5.8: **AES256** Session Key

Chapter 6

Conclusion

6.1 Conclusion

This work serves as PoC on how Volatility can be used to identify, isolate, extract and rearrange really specific data in order to produce a Minidump File that can be further analysed by more targeted tools such as WinDBG and its plugins.

6.1.1 Spin-off & Derived Work

While the restructuring of the full memory dump was already achieved by the immaculate work of Ulf Frisk on `MemProcFS`, this work allowed for a more in-depth look in Volatility development and Windows Internals.

Specifically, the following contributions and publications were a direct result of this work:

Volatility3:

- Pull Request: #1381 - Volshell: Add Dedicated Method to retrieve EPROCESS/Task object
- Pull Request: #1729 - Volshell: Add byteorder argument for display_* functions
- Pull Request: #1816 - Add Stack Plugin
- Issue: #1705 - Volshell `display_type()` issue on `.write` attribute
- Issue: #1732 - PESymbols: Discrepancy because of symbol order
- Issue: #1761 - Bleeding Edge Kernel versions drop module_sect_attr

- Issue: #1808 - linux.kallsyms Backtrace

pypykatz:

- Pull Request: #177 - Fix Volatility reader

pypykatz-volatility3:

- Pull Request: #9 - Fix Requirements and plugin version

Publications:

- Memory dump analysis with Signal decryption
- Creating Linux Symbol Tables for Volatility: Step-by-step guide

6.1.2 Future Work

With the steps needed to go from a full memory snapshot to a Minidump clearly outlined, this work paves the way for a Volatility Plugin/Tool to be built in order to provide more compatibility and even merged with the framework.

In addition, as already mentioned in 5.2.8, and in conjunction with the PTE plugins [4], a more portable Minidump can be created, capturing more of the linked Module and requiring less extraction from the original Memory Snapshot.

Bibliography

[1] M. Manna, A. Case, A. Ali-Gombe, and G. G. Richard, "Memory analysis of .net and .net core applications," *Forensic Science International: Digital Investigation*, vol. 42, p. 301404, 2022, proceedings of the Twenty-Second Annual DFRWS USA. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281722000853

[2] F. Block and A. Dewald, "Windows memory forensics: Detecting (un)intentionally hidden injected code by examining page table entries," *Digital Investigation*, vol. 29, pp. S3–S12, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287619301574

[3] F. Block, "Windows memory forensics: Identification of (malicious) modifications in memory-mapped image files," *Forensic Science International: Digital Investigation*, vol. 45, p. 301561, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281723000707

[4] ——. (2021) Release of pte analysis plugins for volatility 3. [Online]. Available: https://insinuator.net/2021/12/release-of-pte-analysis-plugins-for-volatility-3

[5] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, no. 4, pp. 197–210, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287606001228

[6] A. Walters and N. Petroni, "Volatools: Integrating volatile memory forensics into the digital investigation process," *Digital Investigation*, 01 2007.

[7] O. Stavrou. (2025) Creating linux symbol tables for volatility: Step-by-step guide. [Online]. Available: https://www.hackthebox.com/blog/how-to-create-linux-symbol-tables-volatility

[8] V. Obst. (2024) Towards utilizing btf information in linux memory forensics. [Online]. Available: https://blog.eb9f.de/2024/11/10/btf2json.html

[9] P. Yosifovich, A. Ionescu, D. Solomon, and M. Russinovich, *Windows Internals: System architecture, processes, threads, memory management, and more, Part 1*, ser. Developer Reference. Pearson Education, 2017. [Online]. Available: https://books.google.gr/books?id=y83LDgAAQBAJ

[10] M. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014. [Online]. Available: https://books.google.gr/books?id=U1jOAwAAQBAJ