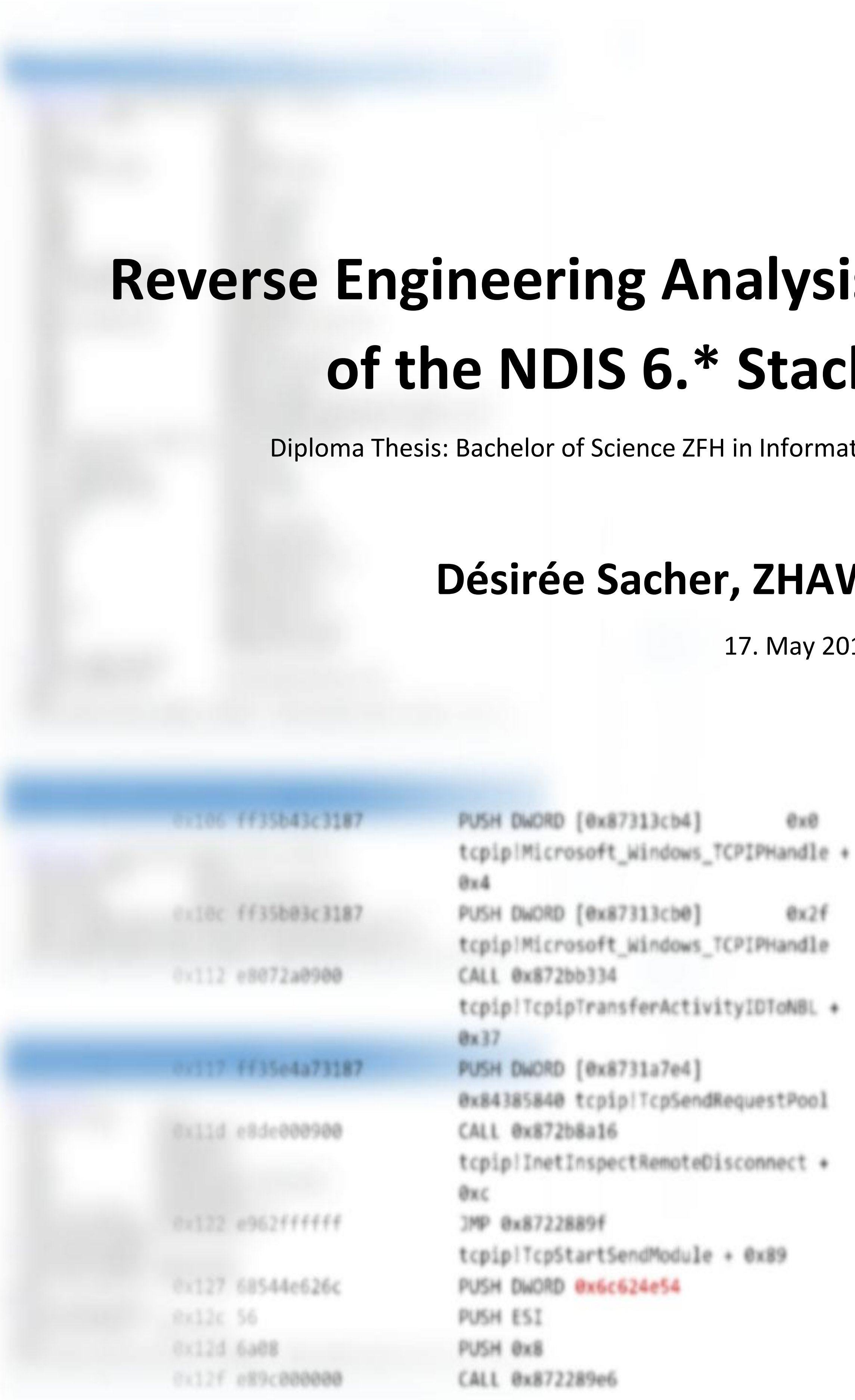

Reverse Engineering Analysis of the NDIS 6.* Stack

Diploma Thesis: Bachelor of Science ZFH in Informatik

Désirée Sacher, ZHAW

17. May 2016

Zurich University
of Applied Sciences

zh
aw

Supervisor: Peter Stadlin

English:

Hereby the signatory confirms, that this bachelor thesis with the subject

Reverse Engineering Analysis of the NDIS 6.* stack

according to the approved scope of work with clearance as of **February 24, 2016** (Design Review) was independently conducted without any external help and in line with valid regulations.

Deutsch:

Hiermit bestätigt die oder der Unterzeichnende, dass die Bachelorarbeit mit dem Thema

Reverse Engineering Analysis of the NDIS 6.* stack

gemäss freigegebener Aufgabenstellung mit Freigabe vom **24. Februar 2016** (Design Review) ohne jede fremde Hilfe im Rahmen der gültigen Reglements selbständig ausgeführt wurde.

Signed: …………………………………………………………………..(Desiree Sacher)

Date: ………………………………………………………………………………………..

Acknowledgements

I want to thank my supervisor Peter Stadlin for his advice and support during the last few months. Also I would like to thank Michael Cohen for his patience, understanding, and trust when working with me.

Thank you Chris, Patrick, and Sacha. Without your review feedback this documentation wouldn't have looked the same, and I definitely would have missed the Oxford comma.

At last I want to thank my whole family for being such a great support, and understanding and appreciating me for being different.

Table of Contents

Abstract

Microsoft's current Network kernel stack, called NDIS (Network Driver Interface Specification) version 6, is still not well supported by today's memory forensic tools. Instead, today's tools focus on identifying the location in memory where information from netstat.exe is stored and parse it. To get a better understanding of the NDIS stack, it was compared to the Linux socket stack, and the differences were analysed.

The goals were to analyse the NDIS 6.* stack of Windows systems, identify network connection structures in memory, and document state changes, and finally document all results in English.

The thesis showed that many basic concepts were implemented similar. Amongst these were the socket function itself, but also the basic structs for how the addresses are stored. This is most likely due to the standard, being defined by the Berkeley University of California, which also defined the BSD (Berkeley Software Distribution) UNIX operating system standards. For compatibility reasons over different operating systems, the similar implementation makes sense. Socket handling, buffering, packet queuing, and handling of the device is different though. Over all Microsoft implemented more layers of responsibilities and separation of duties. The buffering concept of Microsoft additionally considers allocating virtual memory pages.

As for the practical analysis of identifying structures in memory, interesting structures were indeed identified and even information assigned to functions were located in the disassembly of the memory image, this chosen structure however couldn't be identified in memory on time.

The thesis produced a big map of all publicly documented NDIS structs and functions and their relationship to each other, as a side product. This paper can be used as a standard reference for introduction to the topic and suits as a base collection of knowledge for further research in this field.

1. Introduction

With Windows Vista, Microsoft rewrote the whole operating system and introduced not only new user account control, and security features, but built a new architecture with better isolation of the kernel mode from the user mode. Also the NDIS stack, which is short for Network Driver Interface Specification, was rewritten to support, amongst other features, IPv6. It is Microsoft's kernel part that handles all network related traffic, by defining standard interfaces and the layers for interaction [1]. It also handles state information and controls parameters like pointers, and handles for the network drivers.

For forensic investigations, often memory analysis plays a key role today and will also be important in the future. Malware can try to hide, but it always needs to run. Memory holds all running information, and the bigger the available memory, the higher the chance to also find old information that has not been overwritten yet.

The information stored in memory images was well understood for older Windows versions, but for current versions plugins are still written and adjusted. To improve open source memory forensic tools and better understand the information in memory, this thesis focuses on analysing the NDIS 6.* stack and documenting the findings in a clear way. Most preferred are usually pictures [2].

The goals were defined as followed:

1. Analysing the NDIS stack of Windows systems that use NDIS 6.*
2. Identify network connection structures in memory and document state changes
3. The thesis shall be composed in English

The focus on the NDIS 6.* versioned stack was defined because of older versions becoming less used, and are already well understood and researched. Identifying the network connection structures in memory was the goal to be able to improve memory forensic capabilities of existing tools. Documenting state changes was defined a goal from the initial point of receiving packets and therefore changes in the memory need to be occurring while this is happening. The language was chosen to be English to easily relay the findings for further use.

From the beginning it was planned to focus on the theoretical understanding. Implementations of changes, if there were some, were planned to be performed by the tool developers. As little comprehensive documentation is available on this topic, the time appeared to be better invested focusing on the analytical research.

The interest in this topic came from following up on a forensic training by SANS (FOR508 - Advanced Computer Forensic Analysis and Incident Response) and having a technical background in IT networks and security. The missing capabilities were recognized when reflecting the learned material.

2. Realisation of the Thesis

This chapter covers the administrative thoughts and efforts for the planning of the thesis.

2.1. Project Definition

The project was based upon the requirements defined for a bachelor thesis of the course of studies for “Informatik” by ZHAW [3], defined in the document “a_Reglement-Bachelorarbeit_Studiengang-Informatik_V3.3.pdf”.

Relevant technical background to understand the results of this thesis that were not taught during the study at ZHAW are explained in chapter 3. Further definitions can be found in Appendix A: Glossary chapter.

2.2. Process Planning

Initially the project was planned in Kanban Sprints taking each 3 weeks, while the detailed work load was planned with Long Pomodoro Sessions [45], as already practiced well in former projects. But as progressing depended heavily on the outcome of former sessions, the iterations couldn’t be planned as long term as in past projects. Therefore, different possible methods of moving forward were discussed with the supervising professor, Mile stones were defined, and in agreement, future steps were planned. The mile stones and hours spent were tracked in a separate spreadsheet.

The goal was to understand the Microsoft NDIS 6.* stack better for detecting more network related artefacts in memory images and ultimately to improve the Memory Forensic tool Rekall[1]. Microsoft publishes a lot of information in the MSDN and was also publishing new information while this Thesis was being written. The main challenge though, is that Microsoft’s Code is traditionally not open source and therefore requires different Reverse Engineering tactics. To reach the defined goal, the following steps were initially planned:

1. Becoming acquainted with the TCP/IP or NDIS Stack 6.*
2. Becoming acquainted with reverse engineering
3. Evaluation of tools for Reverse Engineering of the NDSI 6.* structs from the memory
4. Definition of use cases for the Analyse over multiple Windows versions that have a NDIS 6.* stack
5. Analyse the NDIS struct in Memory as Proof of Concept with corresponding tools and scripts
6. (Optional) Feasibility study: Identification of recognition features of network connection elements
7. (Optional) Determination of behaviour of structs in memory by utilising defined use cases
8. Documentation of the results in appropriate form for future extensions/modifications of the existing Rekall framework

[1] See Chapter 3.3.3.

The evaluation of tools as mentioned in step 3 is documented in the chapter 3.3. It is also described specifically why those tools were chosen, and what key benefits and differentiates them.

The definition of the use cases is part of the practical analysis and therefore documented in chapter 5.4.1.

The reason to base much of the research on Rekall, and use it for analysis, was that Rekall allows for the importing of Microsoft PDB files[2], and can interpret memory image information by knowing exact locations of reference values and tables, and function names. It is therefore unique in its capabilities of interpreting otherwise unstructured information. The downside of only this tools being able to interpret such information was, that the presented information could neither be challenged nor scrutinized, as only a single source was available. To solve this, as a secondary approach, it was decided to examine if Linux Socket structs and its organisation could be reflected to the Microsoft NDIS stack, and find the same or similar struct information across multiple platforms. The following steps were defined for this:

1. A list of important structs in the Linux Socket stack was defined
2. Understanding of work and packet flows within the Linux Socket stack had to be achieved
3. A mapping of Linux Socket structs to Windows NDIS structs and functions were tested

Those steps were not a replacement for the initial set goals 6 and 7, but a theoretical addition to the goal that was planned to be achieved within step 5. Additionally, the following questions were planned to be answered by the end of the thesis:

- Is the Windows netstat implementation also inspired by the Open Source/Berkeley Version?
- How are IP addresses stored within memory images?
- Were pool tags[3] invented by Microsoft? Or are there similar features in UNIX?
- Are the structures in Memory stored in the same way in Windows as in UNIX?

All of these results are documented in chapter 4.

As building up this knowledge in itself took a significant amount of time, and could be considered a separate Thesis alone, steps 6 and 7 were defined to be optional at the Design Review meeting.

From early on in the planning process it was defined that earlier NDIS versions would be considered as out of scope, as they have already been well researched and current memory forensic tools easily interpret information in such images.

2.3. Project Implementation

As mentioned in the previous chapter, during the project the type of planning was changed. As tasks were defined as milestones, and reaching them required research that could not be directly estimated, the use of Kanban was stopped during the first two months. Instead next milestones were defined, communicated with the supervisor, and when reached, next steps were defined. For the total time of

[2] See chapter 3.2.2.

[3] See chapter 3.2.3.

the thesis (including setup, introduction to reverse engineering, meetings), 520 hours were allocated through the evenings, weekends and holidays of the length of time.

As at beginning, the understanding of the goal between supervisor and student wasn't identical, this had to be clarified in two meetings between the Kick-Off and the Design Review. This also influenced milestones and led to adjusting the expected results at the design review meeting. The milestones were as following.

No	Milestones	Estimated [h]	Estimated [date]	Actual [h]	Actual [date]
1	Basic understanding of reverse engineering	25	14.09.2015	25	14.09.2015
2	Setup test environment	8	04.10.2015	10	04.10.2015
3	Thesis Kick-Off readiness	20	14.10.2015	18	14.10.2015
4	Analysing "socket" structs	15	02.11.2015	17	02.11.2015
5	Understanding the Linux socket stack	40	03.12.2015	60	25.01.2015
6	Design Review readiness	10	24.02.2016	10	24.02.2014
7a	Analysing "related to socket" structs	40	21.02.2016	60	27.03.2016
7b	Finding matching struct information in NDIS for specific Linux structs	40	27.03.2016	80	17.04.2016
8	Visualising and documenting found information	50	27.03.2016	80	10.04.2016
9	Researching other defined goals from supervisor (locating IP, netstat comparison, pool tags in Linux)	20	27.03.2016	70	01.05.2016
10	Analysing material for practical approach	10	17.04.2016	10	04.05.2016
11	Defining and testing proof of concept	40	17.04.2016	20	08.05.2016
12	Finish writing & layout of documentation	50	15.05.2016	60	15.05.2016
	Total	368		520	

Table 1: Table of milestones

As visible in Table 1, first the understanding of the Linux socket stack was a little underestimated but especially more time was used, when correlating the defined Linux socket structs to the Windows NDIS structs and documenting the results. The Windows design turned out to be more complex than expected and a direct mapping wasn't possible anymore. Also the initial sub defined goals of understanding netstat and detecting IP addresses took longer as understanding of concepts was underestimated. Therefore, the proof of concept unfortunately had to be kept short, as time was running out.

3. Introduction to Memory Forensics

Memory forensic capabilities are today used by Incident Response teams on a regular basis, to analyse potentially infected systems. As today's malware started to not even touch disks anymore but only modifying systems in memory (for example registry settings) and systems are often running for a long time (for example servers), the actual state of the system can only be captured in memory.

3.1. Common Methodology Knowledge

A structured procedure for analysing systems is essential, especially as often during an attack, information is spread by invaders laterally moving from compromised systems to other systems. The most followed procedure is described in chapter 3.1.1. A suggested procedure to follow for analysing memory images is covered in chapter 3.1.2.

Once all evidence is captured, it is analysed with forensic tools. To analyse hard disk images, they are commonly mounted in read-only mode on a local system. Modern tools allow mapping of hard disks over the network on the physical drive level (described in chapter 3.3.2 where an Open Source hard disk forensic tool suite is briefly covered). As at the beginning, there was also time invested to get into reverse engineering and this method might not be common to everybody, chapter 3.1.3 covers a brief introduction.

3.1.1. Six-step Incident Response Process

For Incident Response, SANS advices to follow a six step process for a structured analysis, based on the recommendations of the "Computer Incident Response Guidebook" (pub #:5239-19) from the US Navy Staff Office [6].

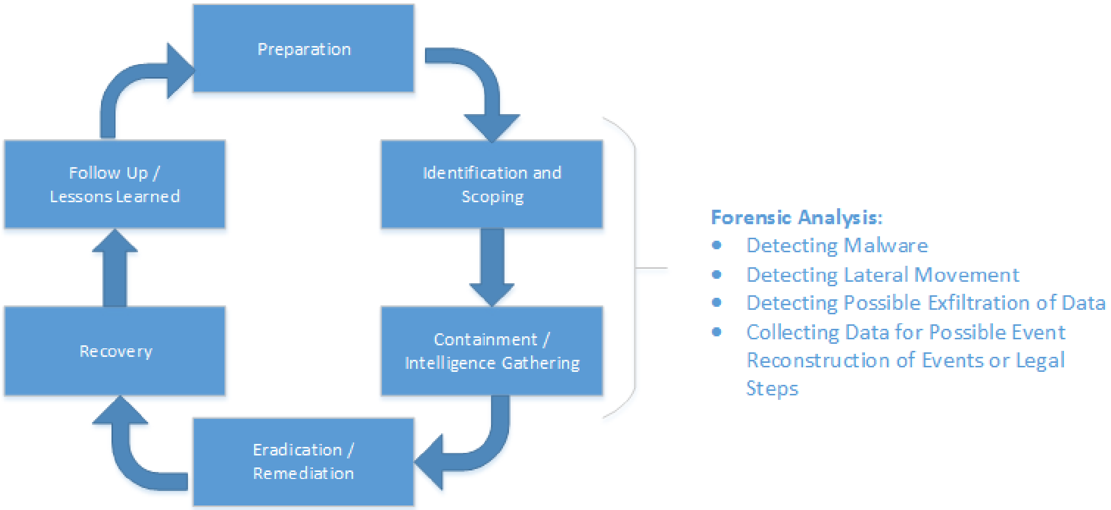

Figure 1: Six-step IR Process and Forensics from the SANS / US Navy Staff Office

Preparation includes establishing incident response capabilities but also general preparation measures such as patching, inventory, or secure design measurements.

Identification and scoping emphasises determination of all affected systems, even if they might not show the exact same symptoms. To do this well it's important to gather intelligence by analysing systems in depth.

Containment / Intelligence gathering is the understanding of what malware was used to compromise the systems (if any was used), how lateral movement took place, how an attacker first entered the network, and if possible, what and how data was ex-filtrated.

Eradication / Remediation should only be started once a complete picture (where possible) of the attack has been gained and no further information can be added by watching the attackers. This phase usually needs to happen quickly and firmly to close the doors on the compromised systems.

Recovery is the phase where future adjustments are decided upon to prevent a possible further similar attack.

The **Follow up / lessons learned** phase usually includes verifications that former vectors of attack have been closed off for example, by adjust monitoring settings, re-audit affected systems by using Penetration Testing, and checking on other possible modifications that were formerly decided on.

3.1.2. Memory Forensic Analysis Process

SANS also splits its analysis of memory images into six steps. Of course there are different tools that can be used to perform such analysis, but in the SANS FOR508 class Redline [7] (GUI tool by Mandiant) and Volatility are covered. SANS also provides free information [8] that is used in the class and can help in investigations after the training. The suggested process though is as followed:

1. Identifying rogue processes
2. Analyse process DLLs and handles
3. Review network artefacts
4. Look for evidence of code injection
5. Check for signs of a rootkit
6. Dump suspicious processes and drivers

Technical support for these steps is given in the cheat sheets [9] and posters [10]. This process is a good help if details of the compromise are not known yet. For example, starting with identifying rogue processes does make sense as it's the easiest way to understand what is currently happening on the system. The followed steps become more and more focused and help in identifying further artefacts and details.

3.1.3. Reverse Engineering

"Reverse engineering is the process of extracting knowledge or design information from anything man-made and re-producing it or re-producing anything based on the extracted information" [11]. Often programs are disassembled for this reason, by using specific tools. The most used tool for this is IDA [12], even though it's only free available version 5.0 was released in 2006 [13]. Basic assembly knowledge is needed to interpret and understand the displayed information. Assembly was covered in the school curriculum module "Informatik Grundlagen 3" and is therefore not further explained here. An introduction to disassembling programs with IDA pro can for example be found in the endnotes [14]. It can also be gained at workshops [15] or other direct trainings.

For extracting information about the NDIS stack, it would have been possible to disassemble the Windows kernel directly, but as this thesis focuses on the information in memory, reading the disassembly of the memory image was instead followed up on. Tools that use the python interpreter [16] can do this automatically when interpreting the image by using the `dis` plugin [17]. The usage is very simple but the plugin lacks visualisation. Therefore, assembly knowledge is indispensable.

3.2. Important Terms and Definitions

The following terms are important to know about and understand when analysing or debugging Windows systems.

3.2.1. Symbols

Microsoft stores all debugging information, including different types of symbol files, separate from the executables [18]. They include CodeView symbols, COFF, DBG, SYM, PDB, and also export tables are directly exported from binary files [19]. The most recent, and most preferred format is PDB Files. By default, they are generated when a project is compiled by using Visual Studio. Symbol files for all published Windows versions can be found on publicly available Symbol Servers. They also include the DLLs and executable files to debug after crashes or analyse of systems that aren't local. The symbol servers can be included in the Debug environment setup to automatically fetch the right files.

3.2.2. PDB Files

PDB stands for "program database". PDB files are automatically created by Windows when an executable with Debug information is built. When imported from the symbol server, they are referenced by their file name which is in the form <GUID>.pdb. Building of the PDB can be enforced by using the `/ZI`, `/Zi` or `/DEBUG` compiler linker switch.

As the exact source information about how the files are built were recently published [20] but the APIs to access the information have been public accessible for a while, it's known that the following information is provided [21]:

- Public, private, and static function addresses
- Global variable names and addresses
- Parameter and local variable names and offsets where to find them on the stack
- Type data consisting of class, structure and data definitions
- Frame pointer omission optimization (FPO) [22] data
- Source file names and line number information

If a developer wishes to remove specific information, they can strip the files by using the `/PDBSTRIPPED:filename` linker option. Only information about public symbols (non-static functions and global variables), a list of object files responsible for sections of code, and frame pointer optimization information will then be stored.

The information present in the PDB files helps a debugger to resolve function names, and parameters or local variables that are only stored on the stack. They are compressed with the GNU Zip (gz) format, as soon as they are extracted with matching software (like for example 7-Zip in Windows), the files can be read in for example a text editor.

Rekall uses PDB files to analyse memory images. It determines the right version by scanning the image for the GUID information. Volatility[4] on the other hand, manually specifies Windows version by referencing the profile when the tool is run, and therefore can't be as specific.

3.2.3. Pool Tags

A pool tag is a four byte character that is associated with a dynamically allocated part of a memory pool [23]. It is specified by the driver when the memory is allocated by using the routine `ExAllocatePoolWithTag`. It's also possible that this tag is stored in reversed order (little endian). On Windows, a file called Pooltag.txt is installed as part of the Debugging Tools for Windows. For everybody else, it can also be found on the internet [24]. The file includes the pool tag itself, the driver where it's used, and a short description. Network relevant pool tags are for example the ones listed in the following list. By using the pool_tracker [25] plugin included with Rekall or the pooltracker [26] plugin included in Volatility, additionally a statistic about all pool tags found in the image is displayed.

```
RhHi - tcpip.sys  - Reference History Pool
Rind - tcpip.sys  - Raw Socket Receive Indications
TcAR - tcpip.sys  - TCP Abort Requests
TcBW - tcpip.sys  - TCP Bandwidth Allocations
TcCM - tcpip.sys  - TCP Congestion Control Manager Contexts
TcCR - tcpip.sys  - TCP Connect Requests
TcCC - tcpip.sys  - TCP Create And Connect Tcb Pool
TcDD - tcpip.sys  - TCP Debug Delivery Buffers
TcDQ - tcpip.sys  - TCP Delay Queues
TcDR - tcpip.sys  - TCP Disconnect Requests
TcEW - tcpip.sys  - TCP Endpoint Work Queue Contexts
TcFR - tcpip.sys  - TCP FineRTT Buffers
TcHT - tcpip.sys  - TCP Hash Tables
TcIn - tcpip.sys  - TCP Inputs
TcLS - tcpip.sys  - TCP Listener SockAddrs
TcLW - tcpip.sys  - TCP Listener Work Queue Contexts
TcpA - tcpip.sys  - TCP DMA buffers
TcpB - tcpip.sys  - TCP Offload Blocks
TcDM - tcpip.sys  - TCP Delayed Delivery Memory Descriptor Lists
TcDN - tcpip.sys  - TCP Delayed Delivery Network Buffer Lists
TcpE - tcpip.sys  - TCP Endpoints
TcpI - tcpip.sys  - TCP ISN buffers
TcpL - tcpip.sys  - TCP Listeners
TcpM - tcpip.sys  - TCP Offload Miscellaneous buffers
```

Figure 2: Extraction of pool tag tcpip.sys list

3.2.4. Memory Descriptor List (MDL)

When an I/O buffer is bigger than a physical page or the number of pages are discontinuous, the operating system uses memory descriptor list (MDL) to describe the physical page for a virtual memory buffer [27]. An MDL consists of a fixed structure and has system routines to calculate its size, and macros to access it via a driver. Its structure is semi-opaque, which means that the driver should only access the visible members `Next` and the `MdlFlags` of the structure, the rest of the structure is hidden.

---

4 See chapter 3.3.1.

The `Next` member is used to chain MDLs which for example can be useful for managing an array of buffers, as it's used with network drivers, where one IP packet has its own buffer and each buffer in the array has its own MDL in the chain which can automatically be cleaned up after use. The `MdlFlags` member describes the MDLs content [28], it's a binary value that is set bitwise. The following flags are available.

```
#define MDL_MAPPED_TO_SYSTEM_VA     0x0001
#define MDL_PAGES_LOCKED            0x0002
#define MDL_SOURCE_IS_NONPAGED_POOL 0x0004
#define MDL_ALLOCATED_FIXED_SIZE    0x0008
#define MDL_PARTIAL                 0x0010
#define MDL_PARTIAL_HAS_BEEN_MAPPED 0x0020
#define MDL_IO_PAGE_READ            0x0040
#define MDL_WRITE_OPERATION         0x0080
#define MDL_PARENT_MAPPED_SYSTEM_VA 0x0100
#define MDL_FREE_EXTRA_PTES         0x0200
#define MDL_DESCRIBES_AWE           0x0400
#define MDL_IO_SPACE                0x0800
#define MDL_NETWORK_HEADER          0x1000
#define MDL_MAPPING_CAN_FAIL        0x2000
#define MDL_ALLOCATED_MUST_SUCCEED  0x4000
#define MDL_INTERNAL                0x8000
```

Figure 3: MdlFlags options (from `wdm.h` file)

To interact with the opaque or hidden members of the MDL, the macros `MmGetMdlVirtualAddress` for the virtual memory address of the I/O buffer, `MmGetMdlByteCount` for returning the size in bytes and `MmGetMdlByteOffset` for the offset within a physical page are used. The following diagrams displays the structure with its visible members and the macros used to interact with the whole structure and its opaque members.

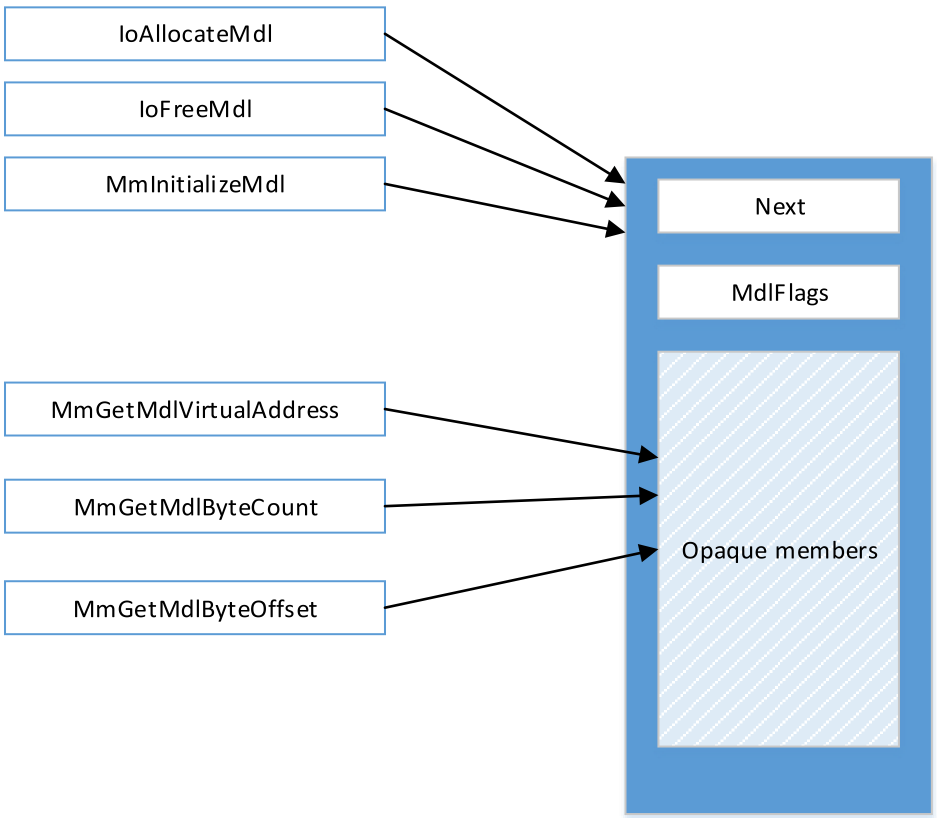

Figure 4: MDL layout with most important macros and routines

The routines `IoAllocateMdl` for allocating the MDL, `IoFreeMdl` for freeing the MDL, and `MmInitialzeMdl` for allocating a block of non-paged memory and formatting it exist as an MDL help in generally managing the MDL. There are more routines available that are well described in the MSDN [29].

3.2.5. Locating KPCR

The kernel processor control region (KPCR) is a data structure that contains a pointer to the Kernel Debugger Datablock (KDBG), which contains the information about the current processor and also contains a substructure called Processor Control Block (KPRCB), which contains information such as CPU step and a pointer to the thread object of the current thread. [30]. It is useful as it's the main orientation point in for example identifying all currently running processes in memory by reading the executive process block (`EPROCESS`), and identifying the `PsActivePRocessHeader` pointer. As each process afterwards contains information about used libraries, spawned processes, paths of process executables or pointers to Virtual Address Descriptors (VAD) [31] trees, it is essential to locate this structure. It used to be stored at a fixed kernel virtual address in Windows XP [32]. With randomizing the address space layout (ASLR), one of the big changes that came with Windows Vista, finding the KPCR became more difficult. Initial scans for that KCPR were extremely slow because the whole image was being scanned for the structure, as it's recognizable by pointing to its own address [33]. But it's also possible to find it by searching threads that are waiting to be run on the CPU, as the list head is stored in the KPRCB as well.

Rekall doesn’t rely on scanning the memory for the KDBG but instead uses Windows version related profiles that know the offset addresses of the most important structures from the PDB information.

3.3. Used Tools

The following tools were used for analysis. All of them are mentioned at the SANS FOR 508 Forensic training. Exercises during the training are performed by using Volatility and the Sleuth Kit. This section includes the expected results from the tools, and why they were used.

3.3.1. Volatility

Volatility [34] is an Open Source memory forensic toolkit written in Python. It is only being used for analysing memory images. The creation of images need to be done with other tools as Volatility does not offer this feature. It offers a different set of plugins for Windows as well as Linux systems. To analyse different operating systems, the profile needs to be applied by using the `--profile=` option. A list of Windows systems supported can be found in their GitHub [35] or be listed by using the command `vol.py –info`. Linux profiles need to be created on an example Linux workstation, and can afterwards be imported in the profiles section of volatility.

Volatility is the most used open source tool for memory forensic analysis and therefore the default tool for memory analysis. For this thesis, Volatility was used to challenge found results and as it’s capable of also analysing Linux images, using it to compare the different operating systems and their structures.

3.3.2. Sleuth Kit and File System Layers Introduction

The Sleuth kit is a collection of command line tools and a C library for analysing disk images and recover data [36]. The tools can be used on different layers of the file system. The layers are as displayed in the following picture.

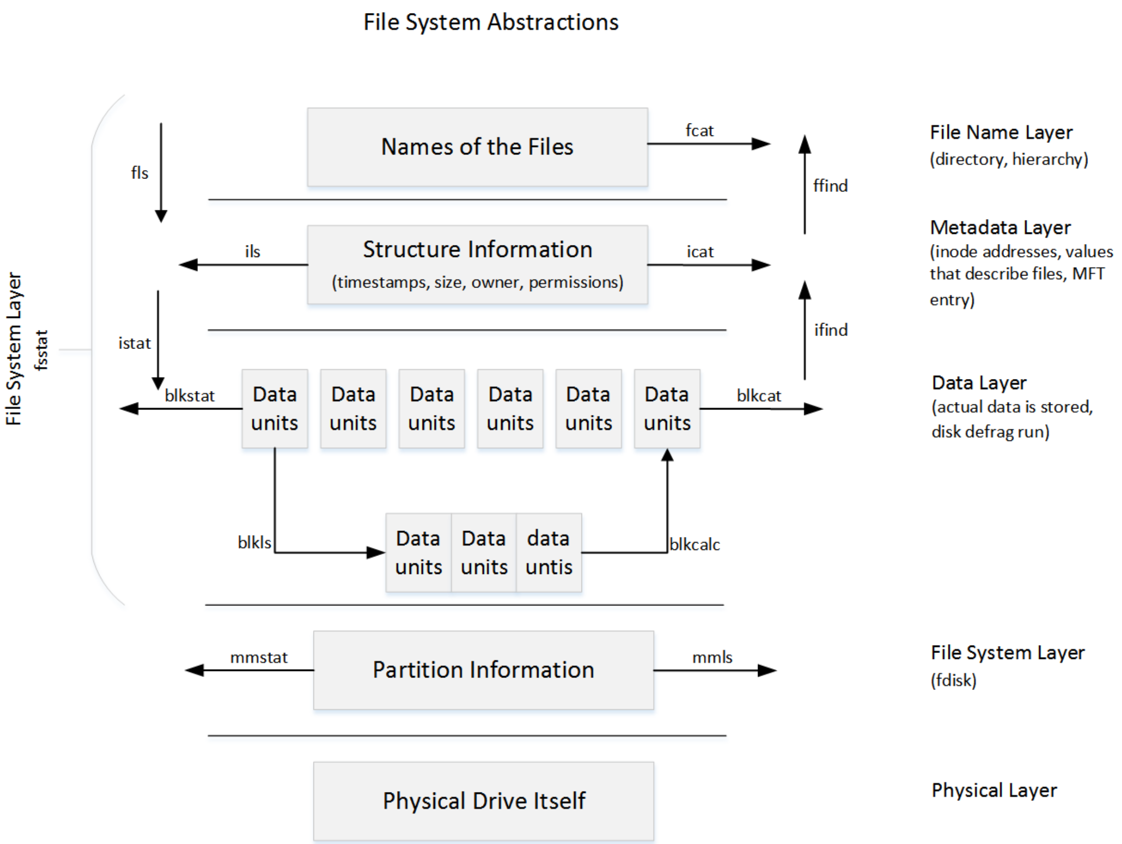

Figure 5: File System Abstractions and Sleuth kit tools for each Layers

The Sleuth kit is a standard toolkit used for detailed analysis on the File Name, the Metadata, the Data, and even on the File System Layer. It can also assist when automatic interpretation of images by Volatility and Rekall don't work or shall be challenged by accessing settings or files previously acquired from hard disks.

3.3.3. Rekall

Rekall [37] is developed by one of the original volatility developers, and is also written in Python. The main difference from Volatility is that it's not expecting the user to define the needed profile for analysing the image, but rather detects the needed profile by scanning for the GUID information and displaying information by parsing the PDB files created for debugging purposes.

This tool is used as the primary tool for this thesis as it combines interesting ideas and concepts such as using PDB files, but also working constantly in an interactive Python shell, which allows an easier analysis by self-writing quick scripts that can interact with the image being analysed. Also the author of Rekall was contacted before the Bachelor Thesis to discuss details as he was working for Google in Zurich.

3.3.4. SIFT Workstation

During the SANS forensic courses, a virtual machine called the SIFT Workstation [38] is used. It's an Ubuntu machine with all Linux forensic tools that are used during the class. It can be downloaded for free. The tools preinstalled are, amongst others, Volatility, the Sleuth kit, Rekall, log2timeline (used for creating timelines out of hard disc image files timestamps) and support for reading different file systems and image types.

3.3.5. Memory Imaging

To acquire a memory image, there are several possibilities [39]. We will cover the two most common open source solutions. The easiest to use is most likely DumpIt [40] by MoonSols, which can be downloaded for free in their webpage resources section. A tutorial for its use can be found in the endnotes [41]. WinPmem [42] is developed by the Rekall developer Michael Cohen and has been included in Rekall. A tutorial for its use can be also find on the Rekall-WinPmem website.

If, like in our case, the testing is performed in a virtual environment, a current memory snapshot can be acquired by suspending the virtual machine. For VMware the snapshot can be either converted from the vmss and vmem file of the machine [43], or Volatility [44] and Rekall [45] both also parse the files directly.

4. Theoretical Approach

For clarification, the theoretical analysis, and initial questions asked to understand basic concepts, are documented in this separate chapter. These questions were not initially defined in the thesis goals but instead determined in discussion with the supervisor, to create foundation for reaching the initially defined goals.

4.1. Explanation

To store data in the C programming language, or like in our case C++, in a physically grouped way it is common to use the complex data type “structs” [46]. One of the many bonuses of using structs is that it allows access via single pointer or the declared name. A struct can contain many other datatypes, and therefore be easily used for defining “data” that belongs together (in our case, network and IP information). As Microsoft has in the past let itself be inspired by Open Source projects, and also mentions it’s inspiration in the MSDN [47], it was first decided to check on socket structures defined under the Berkeley license and compare those to Windows Sockets.

4.2. Description of Material at Hand

To compare the structures, the first approach was to find websites with descriptions of both stacks to compare structs and find similarities. A great reference for UNIX socket programming was found in “Beej’s Guide to Network Programming” [48] or of course in the “Advanced Programming in the UNIX Environment” by W. Richard Stevens and Stephen A. Rago. For Windows structs, the MSDN was referenced as Microsoft documents its Operating System there in detail. Although the Operating System is not Open Source, a lot of debug information is available. One of the challenges that we face in memory forensics though is, that with every system update, the possibility exists, that structs are changed, and therefore the layout in memory is likely to change as well. For ease of operations, the UNIX code examples are taken from “Beej’s Guide to Network Programming”.

Apart from finding the differences between UNIX socket and Windows NDIS 6.* structs, the following questions are dealt with in this chapter:

- Is the Windows netstat implementation also inspired by the Open Source/Berkeley Version?
- How are IP addresses stored in memory images?
- Were pool tags invented by Microsoft or is there a similar feature in UNIX?
- Are the structures in Memory stored in the same way in Windows as in UNIX?

4.3. Result and Discussion of Material at Hand

The available material was mostly distributed on the internet. Information was spread over different websites, by different owners, and a lot of information wasn’t current anymore. Already identifying the best resource for explaining the current Linux socket stack was a challenge. It was decided to focus on current source code as often as possible and only using other resources for explaining more details. Although the core structure of the `sk_buff` remained the same, linked structs were changed and features were adjusted for current needs.

The Windows material centred on the MSDN references, which unfortunately wasn't very well-arranged. Also, the practical analysis showed, that the MSDN is not complete but, as expected, only documents the needed structures for interaction. For information about `NET_BUFFER_LIST` further information was found that was dated in 2010. This documentation was very throughout, but only specific to the buffering of packets in NDIS 6. A mapping of the information known from the Linux socket stack to the NDIS 6.* stack could still be achieved though.

For understanding the netstat differences, the documentation about how the Volatility plugin was reverse engineered turned out to be most useful. The display of IP addresses was directly approached in memory and interpreted with the information collected in chapter 3.

4.4. Own Contribution for Resolution of Problem

To get a throughout understanding of the NDIS stack, the structs of the Linux socket stack were compared to the NDIS stack structures. A direct mapping couldn't be achieved in all cases. The following chapters describe the details of the analysis.

4.4.1. Analysis of Structs

To better understand the used structs and their intentions in Linux and Windows, the following table should give a quick overview over the most important Linux structs for the network stack and the matching equals in Windows. Functions and system calls are identifiable by (), structs are only listed by their names.

Goal / Task	**Linux 4.5**	**Windows (MSDN Version v.85)**
Store socket information	socket() (system call)	socket() (function)
Store IP address information	sockaddr, sockaddr_in	sockaddr, sockaddr_in
Network layer representation of sockets (shared layout with inet_sock*)	sock_common	WSK_PROVIDER_BASIC_DISPATCH
Network layer representation of socket	sock *(includes sock_common)	WSK_SOCKET
Representation of INET sockets (IP based)	snet_sock *(includes sock)	WSK_PROVIDER_CONNECTION_DISPATCH WSK_PROVIDER_DATAGRAM_DISPATCH
Socket buffering	sk_buff	NET_BUFFER, points to NET_BUFFER_HEADER that points to NET_BUFFER_DATA
Linked list to buffered packets	sk_buff_head	NET_BUFFER_LIST

Goal / Task	Linux 4.5	Windows (MSDN Version v.85)
Backlog queue for received packets	sk_receive_queue() (struct sk_buff_head)	NdisMIndicateReceiveNetBufferLists() ProtocolReceiveNetBufferLists () FilterReceiveNetBufferLists ()
Backlog queue for about to be sent packets	sk_write_queue() (struct sk_buff_head)	NdisSendNetBufferLists () MiniportSendNetBufferLists ()
Backlog queue for error while sending or no confirmation received for (no ack)	sk_error_queue() (struct sk_buff_head)	
Store device information about state of interface and the device itself	net_device	NDIS_MINIPORT_ADAPTER_ATTRIBUTES (physical device) NDIS Protocol Drivers (binds miniport driver or intermediate driver to upper edge/NDIS to send and receive network data

Table 2: Linux structs and equivalent Windows structs and functions

4.4.1.1. Comparing the socket Structure

When comparing basic structs the initial focus was the `socket` [49] command (in UNIX). The socket system call is defined by the following arguments:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Figure 6: `socket` system call (UNIX)

The windows `socket` function [50] is defined as follows:

```C++
SOCKET WSAAPI socket(
  _In_ int af,
  _In_ int type,
  _In_ int protocol
);
```

Figure 7: `socket` function (Windows)

To easily compare the arguments, the following table is provided for reference:

UNIX argument	UNIX argument description[5]	Windows argument	Windows argument description
int domain	Nature of communication including address format, example: PF_INET(IPv4)/ PF_INET6(IPv6)	int af	Address family, example: AF_INET(IPv4), AF_NETBIOS, AF_INET6(IPv6)
int type	Type of socket including communication characteristics, example: SOCK_STREAM(stream) or SOCK_DGRAM(datagram)	int type	Type of socket, example: SOCK_STREAM(stream), SOCK_DGRAM(datagram), RAW
int protocol	Protocol type, example: 0 (any), getprotobyname()(TCP/UDP)	int protocol	Protocol type, example: ICMP, TCP, UDP

Table 3: Comparison socket UNIX/Windows arguments

As can be easily seen from the comparison table above, the layout is almost exactly the same, except that in UNIX the first argument is called "domain", and in Windows it's called "Address family". This is probably caused by standardisation and to improve and simplify communications over all Operating Systems.

4.4.1.2. Comparing the sockaddr and sockaddr_in Structure

As the `sockaddr` and the `sockaddr_in` structures are almost the same in UNIX as in Windows, they are both combined in the same chapter. First here the UNIX structures.

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

Figure 8: sockaddr structure (UNIX)

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)

struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

Figure 9: sockaddr_in structure (UNIX)

The Windows structure [51] mostly distinguishes from UNIX, as for the struct member `sa_family` the type is specified with ADDRESS_FAMILY (by using typedef). In previous versions [52] of the NDIS stack,

[5] Source: "Advanced Programming in the UNIX Environment", Section 16.2 Socket Descriptors.

the sockaddr sa_family type was also defined with ushort as it is in UNIX. The possible values for the address family and also the definitions of SOCKADDR, SOCKADDR_IN, and further structures, are listed in the Ws2def.h header file [53], which is automatically included in the Winsock2.h file.

```cpp
typedef struct sockaddr {
  ADDRESS_FAMILY sa_family;
  CHAR           sa_data[14];
} SOCKADDR, *PSOCKADDR;
```

Figure 10: SOCKADDR structure (Windows)

```cpp
typedef struct sockaddr_in {
  ADDRESS_FAMILY sin_family;
  USHORT         sin_port;
  IN_ADDR        sin_addr;
  CHAR           sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;
```

Figure 11: SOCKADDR_IN structure (Windows)

The definition of ADDRESS_FAMILY is therefore visible in the mentioned header file and is also a USHORT.

```
ws2def.h
//
// Address families.
//

typedef USHORT ADDRESS_FAMILY;
```

Figure 12: Definition of ADDRESS_FAMILY found in ws2def.h (Windows)

The known address family values are defined with "AF_NAME" or "PF_NAME", where the protocol family values are identical to the address family values. The following screenshot displays a range of (but not all) possible entries.

```
//
// Although AF_UNSPEC is defined for backwards compatibility, using
// AF_UNSPEC for the "af" parameter when creating a socket is STRONGLY
// DISCOURAGED.  The interpretation of the "protocol" parameter
// depends on the actual address family chosen.  As environments grow
// to include more and more address families that use overlapping
// protocol values there is more and more chance of choosing an
// undesired address family when AF_UNSPEC is used.
//
#define AF_UNSPEC       0               // unspecified
#define AF_UNIX         1               // local to host (pipes, portals)
#define AF_INET         2               // internetwork: UDP, TCP, etc.
#define AF_IMPLINK      3               // arpanet imp addresses
#define AF_PUP          4               // pup protocols: e.g. BSP
#define AF_CHAOS        5               // mit CHAOS protocols
#define AF_NS           6               // XEROX NS protocols
#define AF_IPX          AF_NS           // IPX protocols: IPX, SPX, etc.
#define AF_ISO          7               // ISO protocols
#define AF_OSI          AF_ISO          // OSI is ISO
```

Figure 13: Possible values of `ADDRESS_FAMILY` found in ws2def.h (Windows)

Both UNIX and Windows use for `sin_addr` a struct called `in_addr`. In UNIX this is defined like displayed in the following picture.

```
// (IPv4 only--see struct in6_addr for IPv6)

// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

Figure 14: `in_addr` structure (Windows)

The Windows definition in general can be found the `WinSock2.h` header file, and is the same union that was already mentioned in the UNIX reference that was used in previous versions. For Windows SDK released Windows Vista and later, the structures are now in the `Ws2def.h` header file though (automatically included in the `Winsock2.h` file). For IPv6, the definitions are in `Ws2ipdef.h` [54].

```
/*
 * Internet address (old style... should be updated)
 */
struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
#define s_addr  S_un.S_addr
                                /* can be used for most tcp & ip code */
#define s_host  S_un.S_un_b.s_b2
                                /* host on imp */
#define s_net   S_un.S_un_b.s_b1
                                /* network */
#define s_imp   S_un.S_un_w.s_w2
                                /* imp */
#define s_impno S_un.S_un_b.s_b4
                                /* imp # */
#define s_lh    S_un.S_un_b.s_b3
                                /* logical host */
};
#endif
```

Figure 15: `IN_ADDR` structure (Windows)

Our initial suspicion that Microsoft based these structs on the Berkeley versions was proven when seeing the License header of the `WinSock2.h` file, that clearly references Berkeley.

```
ws2def.h   WinSock2.h

//$TAG BIZDEV
//  $IPCategory:
//  $DealPointID:    118736
//  $AgreementName:  berkeley software distribution license
//  $AgreementType:  oss license
//  $ExternalOrigin: regents of the university of california
//$ENDTAG

//$TAG ENGR
//  $Owner:    vadime
//  $Module:   published_inc
//
//$ENDTAG

/* Winsock2.h -- definitions to be used with the WinSock 2 DLL and
 *               WinSock 2 applications.
 *
 * This header file corresponds to version 2.2.x of the WinSock API
 * specification.
 *
 * This file includes parts which are Copyright (c) 1982-1986 Regents
 * of the University of California.  All rights reserved.  The
 * Berkeley Software License Agreement specifies the terms and
 * conditions for redistribution.
 */
```

Figure 16: `WinSock2.h` header file

4.4.1.3. Comparing the inet_sock, sock, and sock_common Structs

The inet_sock struct, located in the inet_sock.h file, is a representation of INET sockets [55] and consists of an element sk of type sock and an element inet of type inet_opt. In the collaboration diagram of the inet_sock struct reference you (see Figure 17 [56]), that it references the sock struct which again references the sock_common struct. All of these structs are described in this chapter, especially the relationship in between them.

Initially it was planned to map the related members in a direct table, as was done for the socket and the sockaddr structs. However, as research has shown, the NDIS stack completely splits up its settings into different structs and functions, and a direct mapping isn't possible. An attempt is shown in the following table with the sock_common struct.

The **sock_common** struct [57] (defined in the sock.h file) is only a minimal network layer representation of sockets, which is also defined when inet_sock is configured (referenced by sk, initialized with struct sock). The most important members can be mapped the following way.

Member socket	Description socket	Member NDIS
skc_daddr	Foreign IPv4 address	WskSocketConnect.RemoteAddress
skc_rcv_saddr	Bound local IPv4 address	WskSocketConnect.LocalAddress
skc_dport	placeholder for inet_dport/tw_dport (destination port)	MiniportSendNetBufferLists.PortNumber
skc_num	placeholder for inet_num/tw_num (local port)	NdisSendNetBufferLists.PortNumber
skc_state	Connection state	Wsk*.Irp

Table 4: sock_common member comparison with suitable representations in NDIS

The **sock** struct, which reflects the general network layer representation of sockets [58], defined in sock.h, is a general struct that includes all of the information needed to communicate over a socket. It includes information like the reference to sock_common and a lock sk_lock of type socket_local_t. The sk_receive_queue, and the sk_write_queue are bound to the socket here as well.

Inet_sock struct items are used to represent Internet sockets and are IP protocol based [59]. As there are different types of sockets, this specification is necessary to understand and relate the information to the right context.

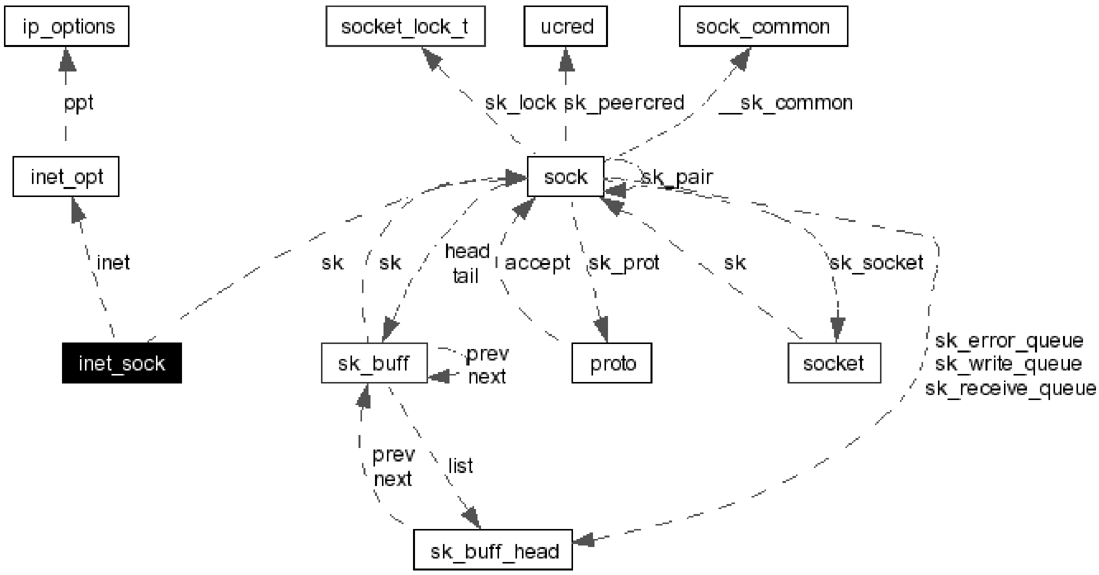

Figure 17: inet_sock struct collaboration diagram

Windows on the other hand knows WSK_* elements. The socket object for a Windows Socket Kernel socket (also called WSK) contains a pointer to a provider dispatch table structure [60]. Windows knows four different types of sockets (Basic, Listening, Datagram and Connection-oriented), and all of them support different socket functions, where the provider dispatch table structures for each category are defined by the WSK Network Programming Interface (NPI). It defines the interface between network modules and defines, that client modules can only be attached to provider modules and vice versa [61]. The NPIs define the following parameters.

Name	Description
NPI Identifier	Unique identifier used by the network module. Network modules can use multiple NPIs
Client characteristics (optional)	Specifies the NPI specific characteristics of each client module (versions, address family, protocol)
Provider characteristics (optional)	Specifies the NPI specific characteristics of each provider module (versions, address family, protocol)
Client module callback functions	>=0, can be used by the provider module to call directly, when successfully attached by provider module
Provider module functions	>=1, can be used by the client module to call directly, when successfully attached by a client module
Client dispatch table	Contains functions pointers to each of the client module callback functions. Not required when no client module callback functions defined
Provider dispatch table	Contains function pointers to each of the provider module functions

Table 5: Network Programming Interface (NPI) parameters

To simplify the following description, only Provider structures are focused on, which are used when WSK applications don't need event callback functions.

The Provider Dispatch table structures also have a shared part, called `WSK_PROVDER_BASIC_DISPATCH`, which reminds of the `sock_common` struct. Compared to its Linux counterpart though it doesn't hold information like IP addresses, port numbers, network address family information, or such, but rather control information about the socket, like pointers to buffers, definition of buffer sizes and the definition of socket type.

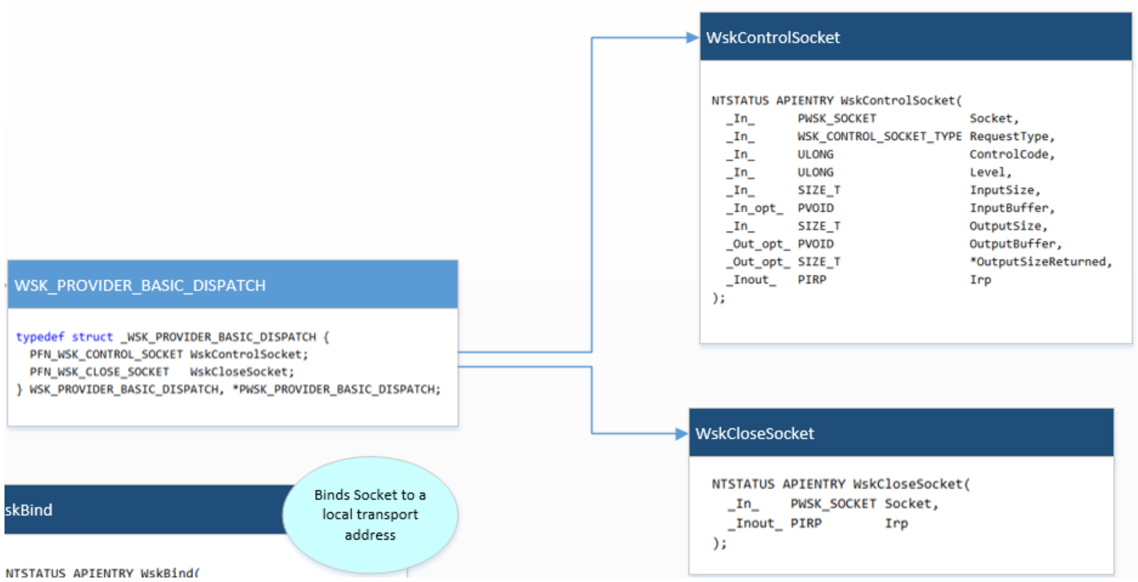

Figure 18: Extract of NDIS Struct Relation Map - `WSK_PROVIDER_BASIC_DISPATCH`

Information like IP addresses are linked by the `WSK_PROVIDER_CONNECTION_DISPATCH` struct in the `WskBind`, `WskConnect`, `WskGetLocalAddress`, and `WskGetRemoteAddress` functions.

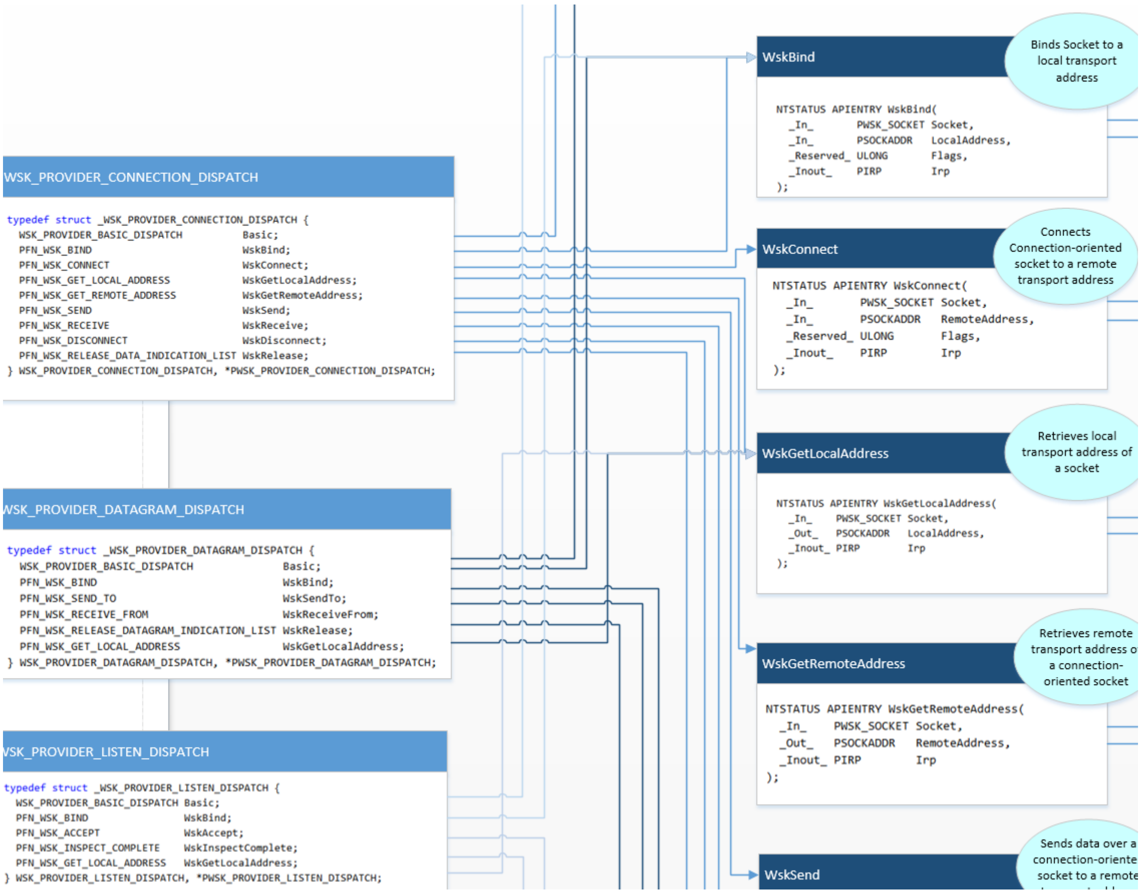

Figure 19: Extract of NDIS Struct Relation Map - `WSK_PROVIDER_CONNECTION_DISPATCH`

The `WSK_PROVIDER_DATAGRAM_DISPATCH` struct references its own functions. As UDP is stateless, it only cares about sending out the information, and doesn't track success or failure of delivery. The related functions are therefore also called `WskSendTo` and `WskReceiveFrom`. The functions store the local and the remote IP address information, as well as the `AcceptSocketContext` pointer.

When a WSK application wants to use functions, it calls the `WskSocket` (creates local socket), the `WskAccept` (incoming traffic) or the `WskSocketConnect` (creates local socket and binds it to a local address and remote address) function. They store type information such as address family, socket type, and a link to a `WSK_CLIENT` struct (which is unfortunately opaque). But it also has pointers to the before mentioned dispatch tables, and its sending functions and even tracks the parent process and the parent thread as well as the security descriptor.

To strictly delimit the different layers, information like the socket configuration is actually stored on the NDIS interface, which is only interacted through a protocol driver layer (further described in 4.4.1.6). This means a lot of the information stored in the `sock` struct cannot be directly mapped to just one struct of the NDIS stack.

4.4.1.4. Comparing the `sk_buff` and `sk_buff_head` Structure

For queuing and buffering network related information, a common data structure `sk_buff` [62] is used (in the Kernel Source Code also called "skb"). It includes all control information for a buffered packet and the buffer is organised as a double linked list [63]. A graphical view on it also describes the interconnection of this struct with other datatypes well [64] (see Figure 20). It shows that the struct includes pointers to the next element, to the previous element, to a list (type `sk_buff_head` [65]), the socket (named sk, type `sock`, not displayed in the below figure), timestamp (named tstamp), the device and a "real device" (named dev, type `net_device`), header info about the type (selection by choosing an option of a union), network header info (again chosen by an option of a union), mac address information (selection of union), and additionally control buffer info that can also be private [66].

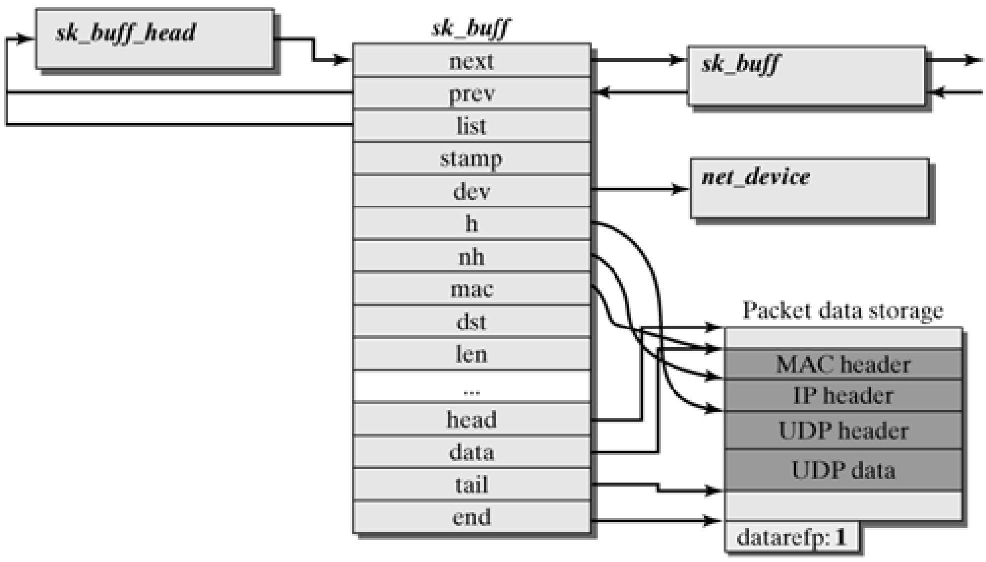

Figure 20: `sk_buff` structure according to flylib.com

The `NET_BUFFER` structure of Microsoft does have similarities, however is only linked via a single linked list. As displayed in this image from the MSDN [67], the `NET_BUFFER` structure links to the next `NET_BUFFER` object and to the associated memory section.

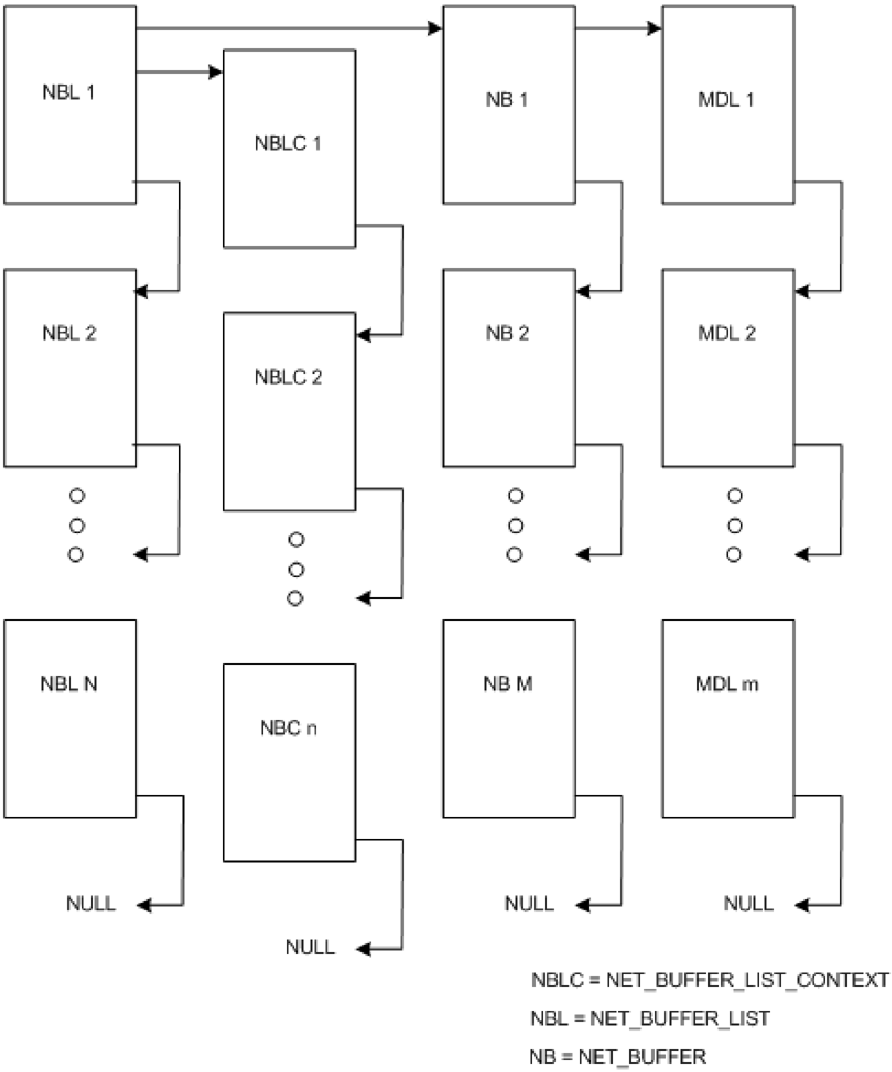

Figure 21: Relationship between `NET_BUFFER` structures, source MSDN

The pointer for this link to the next `NET_BUFFER` [68] structure is stored in the `Next` field of the `NET_BUFFER_DATA` structure, which is part of the `NET_BUFFER_HEADER` structure, that itself is part of the `NET_BUFFER` struct [69].

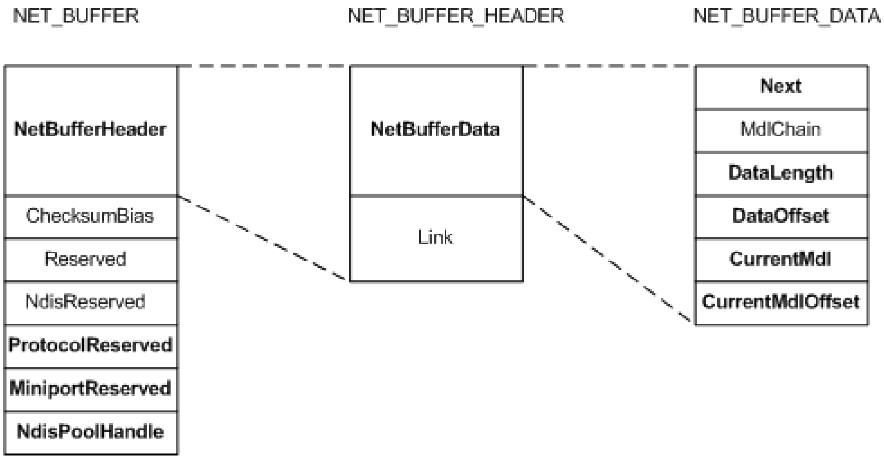

Figure 22: Expanded `NET_BUFFER` structure, source MSDN

The `NET_BUFFER_LIST` [70] struct is referenced from sending and receiving queues, and only the `NET_BUFFER_DATA` struct (linked by the `NET_BUFFER_HEADER`) actually holds the MDL[6] chain link to point to the actual data that is buffered.

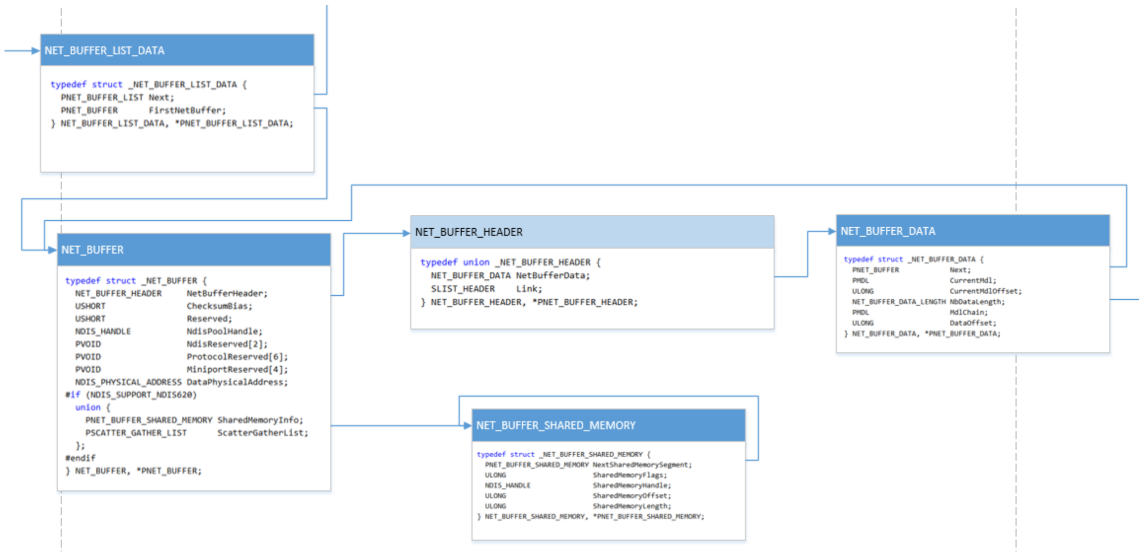

Figure 23: Extract of NDIS Struct Relation Map - `NET_BUFFER_DATA` struct

4.4.1.5. Queuing in Linux and Windows

Linux uses three queues, `sk_receive_queue`, `sk_write_queue` [71], and `sk_error_queue` for Packets incoming, sending, and for errors. All consist of `sk_buff` elements that are linked to another [72] by using the `sk_buff_head` struct that includes pointers to the previous und the next `sk_buff` element [73]. [74]. These queues are useful for the TCP protocol as every packet requires an acknowledgement and if not received, the packet must be able to be resent [75].

---

6 See chapter 3.2.4.

Windows on the other hand only knows the `NET_BUFFER_LIST` structure and related `NET_BUFFER` and MDL chains and different sort of drivers. For adding packets to the right chains, functions are used. Mapped to the right port number and the adapter handle, newly received packets are added by the Miniport `NdisMIndicateReceiveNetBufferLists` [76] function, to be sent packets on the other hand are managed by the Miniport `NdisMSendNetBufferListsComplete` [77] function that uses the Adapter Handle to assign the packets to the right `NET_BUFFER_LIST` queue. It is important to understand that NDIS has different drivers for different contexts. This is well explained in the following diagram from the Code Machine tutorial on "NDIS 6 Net Buffer Lists and Net Buffers" [78].

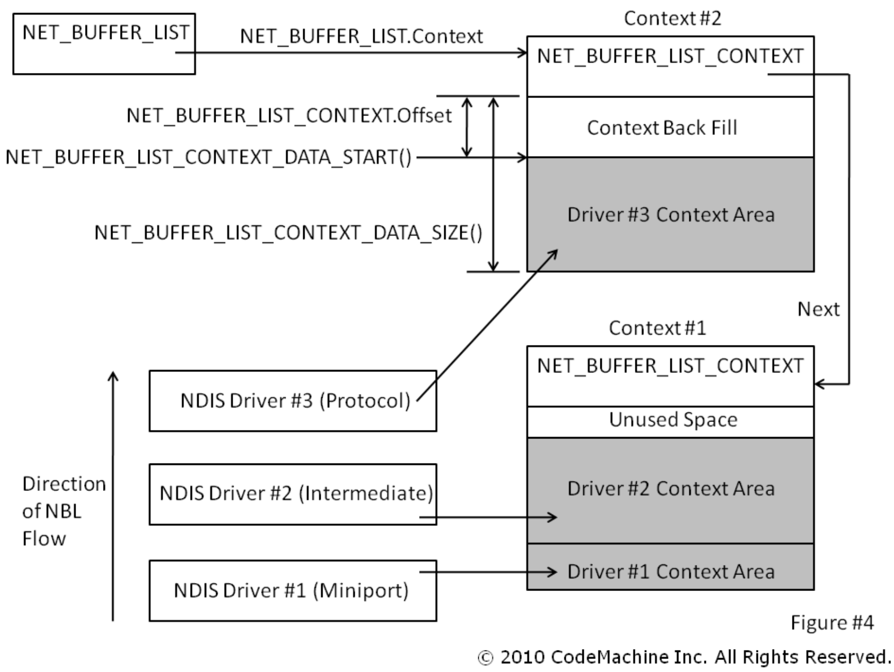

Figure 24: NDIS driver association within context areas when packets are received

As those different drivers also have different functions, and covering all of them would have been too much for this analysis, this thesis focuses, when reasonable, on the Miniport driver, as it is the standard hardware driver [79] for the interaction with the devices and device specific tasks [80].

The interaction with the protocol driver is only briefly described here, as this enables the developer to exchange underlying drivers (such as the Miniport driver). Sending on the Protocol driver level is generally performed by the `NdisSendNetBufferLists` function [81], receiving is performed by the `ProtocolReceiveNetBufferLists` [82] function.

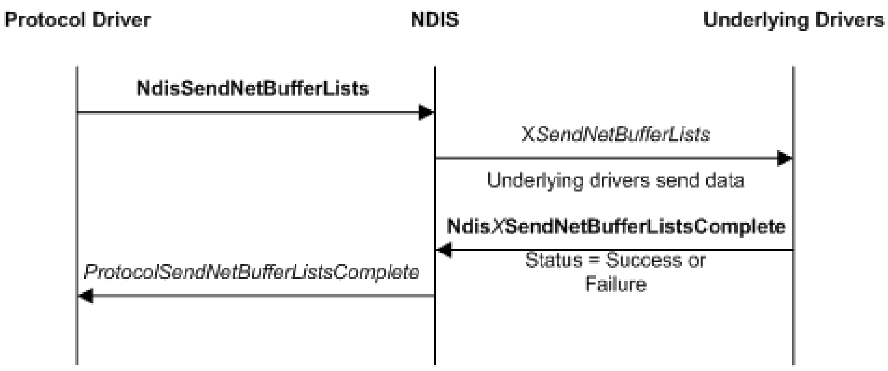

Figure 25: Sending data from a protocol driver

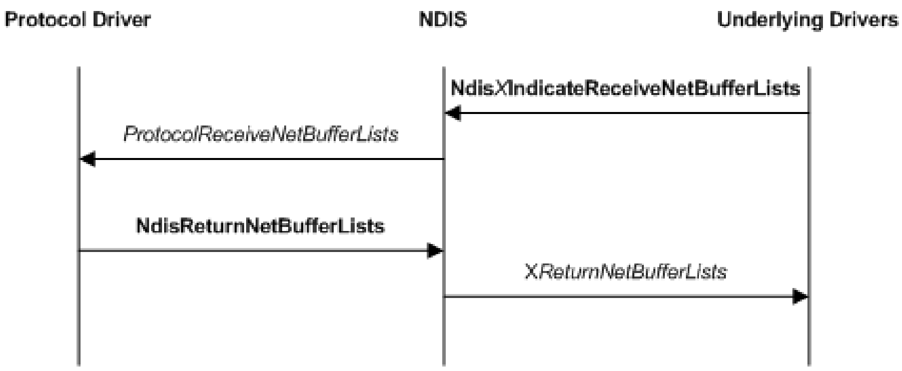

Figure 26: Receiving data in protocol drivers

As for the `sk_error_queue`, there was no such function or queue found in the NDIS stack. Most probably it's not needed, as the `NET_BUFFER_LIST` struct has a status field that tracks the completion status [83] on that level.

Another main difference is that what is defined as a struct in UNIX, is a function in Windows. The reason for this difference can only be guessed at, however most probably can be explained with functions returning values and therefore simplification of error handling is a possible explanation. Also from an object oriented programming point of view it makes sense to use a function instead of a structure, as this way instancing of objects with a standard memory handling is possible.

4.4.1.6. Comparing the `net_device` Struct

The `net_device` struct represents the physical device, and includes speed and connection settings, different queues, a spinlock[7] for active polling (when `CONFIG_NETPOLL` is set), an associated interface buffer, and the current state of the interface. Even the developers mention, that the design isn't optimal as it mixes I/O data with "high level" data. The struct header information can be found in the Appendix C: Struct Headers on page 78.

Windows took a different approach, and an equal can be best associated with the NDIS Miniport Driver [84], as this driver type reflects the device in the NDIS stack. The attributes of the interface can be set with a union called `NDIS_MINIPORT_ADAPTER_ATTRIBUTES` [85]. This union unifies all of the

[7] Glossary entry G1.

NDIS_MINIPORT type of structs that define the configuration of the interface. The syntax of it is as shown in the following picture.

```
C++

typedef union _NDIS_MINIPORT_ADAPTER_ATTRIBUTES {
  NDIS_OBJECT_HEADER                               Header;
  NDIS_MINIPORT_ADD_DEVICE_REGISTRATION_ATTRIBUTES AddDeviceRegistrationAttributes;
  NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES    RegistrationAttributes;
  NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES         GeneralAttributes;
  NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES         OffloadAttributes;
  NDIS_MINIPORT_ADAPTER_NATIVE_802_11_ATTRIBUTES   Native_802_11_Attributes;
#if  (NDIS_SUPPORT_NDIS61)
  NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES HardwareAssistAttributes;
#endif
#if (NDIS_SUPPORT_NDIS630)
  NDIS_MINIPORT_ADAPTER_NDK_ATTRIBUTES             NDKAttributes;
#endif
} NDIS_MINIPORT_ADAPTER_ATTRIBUTES, *PNDIS_MINIPORT_ADAPTER_ATTRIBUTES;
```

Figure 27: NDIS_MINIPORT_ADAPTER_ATTRIBUTES union structure

This union directly references many other configuration settings, but also tries to differentiate between different levels and hand over information from one level to the next (as shown in the previous chapter in Figure 24).

The Miniport adapter settings are for example most set in the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES struct. Figure 28 shall give a hint of the many connections of the NDIS_MINIPORT_ADAPTER_ATTRIBUTES union to other structs. For a more readable view please refer to the big diagram in reference [0].

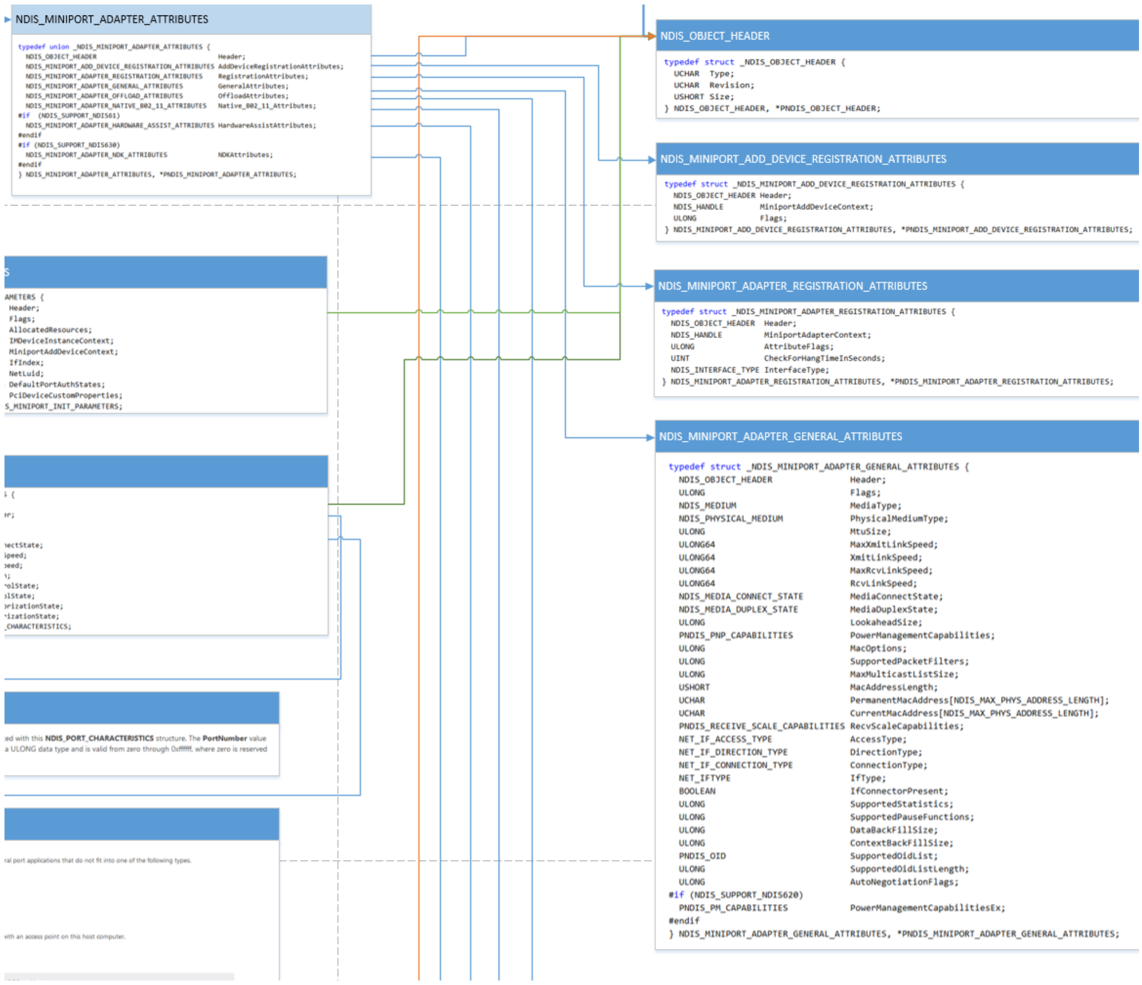

Figure 28: Extract from NDIS Struct Relation Map – `NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES`

To remember the settings on a protocol driver level, they are also stored in the struct `NDIS_BIND_PARAMETERS` (see Figure 29).

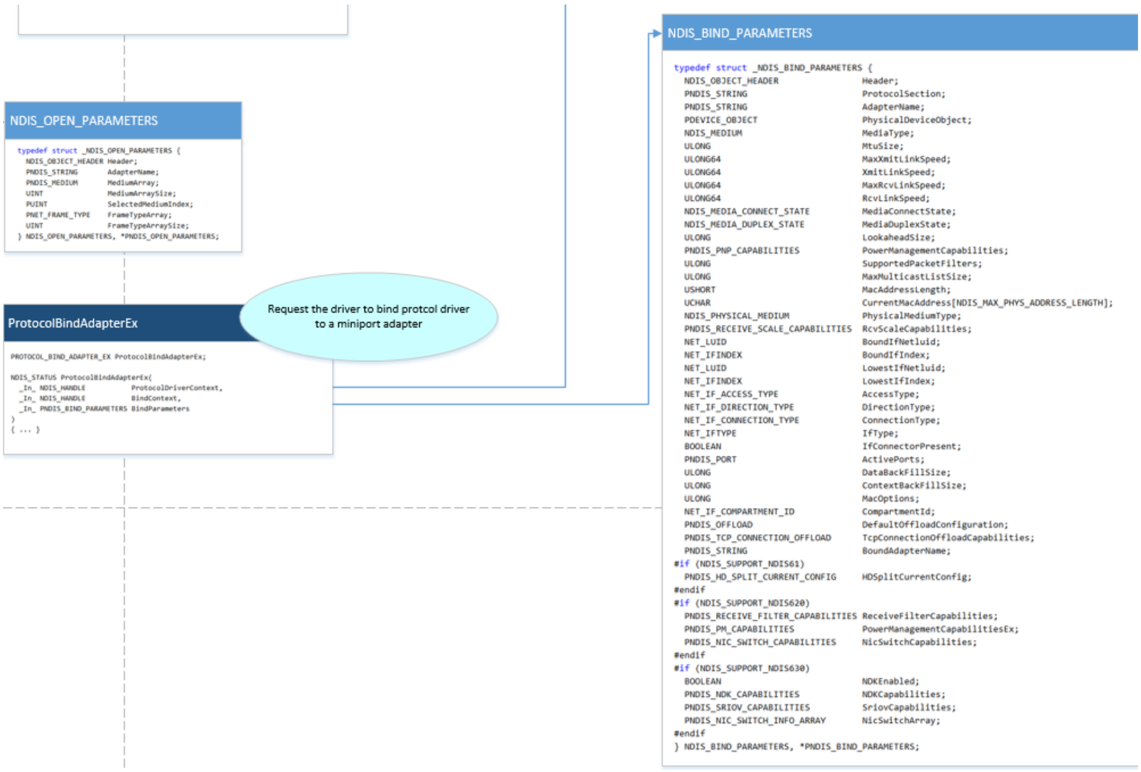

Figure 29: Extract from NDIS Struct Relation Map – NDIS_BIND_PARAMETERS

4.4.2. Analysis of netstat

As described in the blog post about Volatility's new `netscan` module [86], netstat in Windows reads its information from a hash table (`RtlInitWeakEnumerationHashTable`) where connections and sockets are stored in a bitmap port pool. The single TCP connections can be found in pools with the tag `TcpE` (`tcpip.sys` – TCP Endpoints [87]), the TCP sockets are stored in pools tagged with `TcpL` (`tcpip.sys` – TCP Listeners), and for UDP endpoints and sockets the pool tag `UdpA` is used (`tcpip.sys` – UDP Endpoints). The address of that hash table is read by using the function `InternalGetTcpTable2`, which is a wrapper of the `GetTcpTable2` API. `GetTcpTable2` holds a pointer to the Ipv4 TCP connection table [88]. This means a plugin to interpret this information needs to follow the same paths as netstat does or tries to simulate the steps.

Because of availability of the source code of the implementation in Linux, it's worth considering also having a look at this implementation. When checking the source code of netstat under Linux [89] (see Figure 30), in function `tcp_do_one` (line 1051, called from `tcp_info()` which was called from within the main function) it becomes visible that the information is searched by using `sscanf` on a submitted `line`. `Line` is read from a file descriptor (line 458) which ultimately points to subdirectories of the directory `/proc`. TCP information, for example, is read from the file `/proc/net/tcp`.

```
    If (lnr == 0)
       return;

    num = sscanf(line, “%d: %64[0-9A-Fa-f]:%X %64[0-9A-Fa-f]:%X %X %lX:%lX %X:%lX
                 %lX %d %d %lu %*s\n”, &d, local_addr, &local_port, rem_addr,
                 &rem_port, &state, &txq, &rxq, &timer_run, &time_len, &retr, &uid,
```

```
                  &timeout, &inode);

    if (num < 11) {
       fprintf(stderr, _(“warning, got bogus tcp line.\n”));
       return;
    }

    if (!flag_all && ((flag_lst && rem_port) || (!flag_lst &&
        !rem_port)))
      return;

    if (strlen(local_addr) > 8) {
#if HAVE_AFINET6
.
.
.
#endif
    } else {
       sscanf(local_addr, “%X”, &localaddr->sin_addr.s_addr);
       sscanf(rem_addr, “%X”, &remaddr->sin_addr.s_addr);
       localsas.ss_family = AF_INET;
       remsas.ss_family = AF_INET;
}
```

Figure 30: Extract of netstat source code (Debian)

As this is a complete different approach than in Windows, where hash tables are used, also the `netscan` plugins for identifying open connection elements was done a little differently.

The path that Volatility’s and Rekall’s plugin `netscan` choses is to build several pool scanners for the different pool tags and scan all of the memory for it [90] (see Figure 31).

```
#---------------------------------------------------------------------
# netscan plugin
#---------------------------------------------------------------------

class Netscan(common.AbstractWindowsCommand):
“””Scan a Vista, 2008 or Windows 7 image for connections and sockets”””

@staticmethod
def is_valid_profile(profile):
return (profile.metadata.get(‘os’, ‘unknown’) == ‘windows’ and
        profile.metadata.get(‘major’, 0) == 6)

@cache.CacheDecorator(“tests/netscan”)
def calculate(self):

# Virtual kernel space for dereferencing pointers
kernel_space = utils.load_as(self._config)

# Physical space for scanning
flat_space = utils.load_as(self._config, astype = ‘physical’)
```

```
if not self.is_valid_profile(kernel_space.profile):
  debug.error("This command does not support the selected profile.")

# Scan for TCP listeners also known as sockets
for offset in PoolScanTcpListener().scan(flat_space):
  tcpentry = obj.Object('_TCP_LISTENER', offset = offset, vm = flat_space,
                        native_vm = kernel_space)

# Only accept Ipv4 or Ipv6
if tcpentry.AddressFamily not in (AF_INET, AF_INET6):
  continue

# For TcpL, the state is always listening and the remote port is zero
for ver, laddr, raddr in tcpentry.dual_stack_sockets():
  yield tcpentry, "TCP" + ver, laddr, tcpentry.Port, raddr, 0, "LISTENING"

# Scan for TCP endpoints also known as connections
for offset in PoolScanTcpEndpoint().scan(flat_space):
  tcpentry = obj.Object('_TCP_ENDPOINT', offset = offset, vm = flat_space,
                        native_vm = kernel_space)

if tcpentry.AddressFamily == AF_INET:
  proto = "TCPv4"
elif tcpentry.AddressFamily == AF_INET6:
  proto = "TCPv6"
else:
  continue

# These are our sanity checks
if (tcpentry.State.v() not in tcpip_vtypes.TCP_STATE_ENUM or
    (not tcpentry.LocalAddress and (not tcpentry.Owner or
    tcpentry.Owner.UniqueProcessId == 0 or tcpentry.Owner.UniqueProcessId >
    65535))):
  continue

yield tcpentry, proto, tcpentry.LocalAddress, tcpentry.LocalPort,
  tcpentry.RemoteAddress, tcpentry.RemotePort, tcpentry.State
```

Figure 31: Extract of netscan plugin code of Volatility

Currently `netscan` in Volatility supports the pool tags `UdpA`, `TcpL`, and `TcpE` only, which is enough to interpret netstat though.

Scanning for pool tags can lead to planted or manipulated evidence in a forensic assessment, as pool tags can be renamed or otherwise be manipulated by malware. This was also pointed out in a recent talk at ShmooCon "ADD – Complicating Memory Forensics through Memory Disarray" [91]. To avoid easy manipulation of data, simple checks regarding the pool size, type and index are performed (see Figure 32).

```
Class PoolScanTcpListener(PoolScanUdpEndpoint):
  """PoolScanner for Tcp Listeners"""
```

```
checks = [ ('PoolTagCheck', dict(tag = "TcpL")),
           # Seen as 0x120 on Win7 SP0 x64
           ('CheckPoolSize', dict(condition = lambda x: x >= 0xa8)),
           ('CheckPoolType', dict(non_paged = True, free = True)),
           ('CheckPoolIndex', dict(value = 0)),
            ]
```

Figure 32: Extract of `PoolScanTcpListener` class code of the netscan plugin of Volatility

This unfortunately isn't proof that the found "evidence" is correct historical information. For previous versions of Windows (for example Windows XP or Windows 2000), a plugin called `connscan [92]` exists to find closed connections that are no longer linked in netstat. In former Windows versions the pool tag name TCPT was used to identify the TCP objects [93].

The netscan plugin for Linux on the other hand carves[8] for network connection structures and builds its list of possible old connections from that information. [94]

So former reverse engineering of the NDIS stack showed, that the connection elements in Windows are read from a table, which is eventually found by following the starting point of netstat.exe, and therefore the plugin to read that information identifies connection elements and interprets that information by the known pool tags. In the socket stack, everything is a file and so the connections are stored in a file as well. In Linux the identifying connection elements is based on the structs only. This all leads to the conclusion that netstat in Windows is definitely no copy of the Berkely socket.

4.4.3. Display of IP Address in Memory

One basic question that we wanted to get answered was, how IP addresses are displayed in memory. From the background research it was known, that for TCP Endpoints, the pool tag `TcpE` is used, so it made sense to look for IP addresses close to these pool tags. Also creating the Map of the NDIS struct relations, it was clear that IP addresses are usually stored by type `PSOCKADDR`. Reading about the struct `sockaddr` on MSDN [95] it is made clear, that the structure can vary depending on the used protocol in use and the context of the address families. As such it doesn't necessarily need to hold a pointer to the actual address, and also the way of the information being stored isn't completely standardized.

Now, how do you find something where you don't know what it looks like? As a simple `yarascan`[9] didn't return any output, a look at the Volatility and Rekall Python code to view how network information is interpreted sounded like a good idea. However, first it made sense to check on existing documentation in this area. In "The Art of Memory Forensics" (B1) in Chapter 11, Networking – Network Artefacts – Windows Sockets API (Winsock) – Data Structures (XP and 2003) the reversed structures are described.

8 Glossary entry G2.

9 Glossary entry G3.

```
>>> dt("_ADDRESS_OBJECT")
'_ADDRESS_OBJECT'
0x0  : Next                ['pointer', ['_ADDRESS_OBJECT']]
0x58 : LocalIpAddress          ['IpAddress']
0x5c : LocalPort           ['unsigned be short']
0x5e : Protocol            ['unsigned short']
0x238 : Pid               ['unsigned long']
0x248 : CreateTime            ['WinTimeStamp', {'is_utc': True}]
>>> dt("_TCPT_OBJECT")
'_TCPT_OBJECT'
0x0  : Next                ['pointer', ['_TCPT_OBJECT']]
0x14 : RemoteIpAddress          ['IpAddress']
0x18 : LocalIpAddress          ['IpAddress']
0x1c : RemotePort           ['unsigned be short']
0x1e : LocalPort           ['unsigned be short']
0x20 : Pid               ['unsigned long']
```

Figure 33: Screenshot of network artefacts structures from the digital edition of "The Art of Memory Forensics", page 314

Now to automatically interpret such information in Volatility or Rekall, overlays are created. This specific overlay can be found in the `tcpip_vtypes.py` file, located in the subfolder `plugins/overlays/windows`. The overlay in python looks as in the Figure 34.

```
# Structures used by connections, connscan, sockets, sockscan.
# Used by x64 XP and x64 2003 (all service packs).
tcpip_vtypes_2003_x64 = {
    '_ADDRESS_OBJECT' : [ None, {
    'Next' : [ 0x0, ['pointer', ['_ADDRESS_OBJECT']]],
    'LocalIpAddress' : [ 0x58, ['IpAddress']],
    'LocalPort' : [ 0x5c, ['unsigned be short']],
    'Protocol'  : [ 0x5e, ['unsigned short']],
    'Pid' : [ 0x238, ['unsigned long']],
    'CreateTime' : [ 0x248, ['WinTimeStamp', dict(is_utc = True)]],
  }],
    '_TCPT_OBJECT' : [ None, {
    'Next' : [ 0x0, ['pointer', ['_TCPT_OBJECT']]],
    'RemoteIpAddress' : [ 0x14, ['IpAddress']],
    'LocalIpAddress' : [ 0x18, ['IpAddress']],
    'RemotePort' : [ 0x1c, ['unsigned be short']],
    'LocalPort' : [ 0x1e, ['unsigned be short']],
    'Pid' : [ 0x20, ['unsigned long']],
    }],
}
```

Figure 34: Network artefact structure overlay (Volatility source code, `tcpip_vtypes.py`)

The hex values are the offset location where in the struct the next values can be found. This file also contains overlay information of newer or other versions of Windows. Especially with Windows 8 it becomes visible, on how "small" changes can influence the structure of the overlay structs, as every new version seems to have new offset addresses or even new fields entirely.

As with the new version of the NDIS stack, many basic areas of struct information and their interaction was changed, the overlay for Windows Vista and newer systems does look quite different. The `TcpE` pool tag marks the `_TCP_ENDPOINT` structures and has the following overlay for it.

```
class Win7x64Tcpip(obj.ProfileModification):
    before = ['Win7Vista2008x64Tcpip']
    conditions = {'os': lambda x: x == 'windows',
                  'memory_model': lambda x: x == '64bit',
                  'major': lambda x : x == 6,
                  'minor': lambda x : x == 1}
    def modification(self, profile):
        profile.merge_overlay({
            '_TCP_ENDPOINT': [ None, {
                'State' : [ 0x68, ['Enumeration', dict(target = 'long', choices = TCP_STATE_ENUM)]],
                'LocalPort' : [ 0x6c, ['unsigned be short']],
                'RemotePort' : [ 0x6e, ['unsigned be short']],
                'Owner' : [ 0x238, ['pointer', ['_EPROCESS']]],
                }],
            '_TCP_SYN_ENDPOINT': [ None, {
                'InetAF' : [ 0x48, ['pointer', ['_INETAF']]],
                'LocalPort' : [ 0x7c, ['unsigned be short']],
                'RemotePort' : [ 0x7e, ['unsigned be short']],
                'LocalAddr' : [ 0x50, ['pointer', ['_LOCAL_ADDRESS']]],
                'RemoteAddress' : [ 0x68, ['pointer', ['_IN_ADDR']]],
                'Owner' : [ 0x58, ['pointer', ['_SYN_OWNER']]],
                }],
            '_TCP_TIMEWAIT_ENDPOINT': [ None, {
                'InetAF' : [ 0x30, ['pointer', ['_INETAF']]],
                'LocalPort' : [ 0x48, ['unsigned be short']],
                'RemotePort' : [ 0x4a, ['unsigned be short']],
                'LocalAddr' : [ 0x50, ['pointer', ['_LOCAL_ADDRESS']]],
                'RemoteAddress' : [ 0x58, ['pointer', ['_IN_ADDR']]],
                }],
            })
```

Figure 35: Windows 7 x64 _TCP_ENDPOINT Overlay (Volatility Source Code, tcpip_vtypes.py)

```
# Structures for netscan on x86 Windows 7 (all service packs).
tcpip_vtypes_7 = {
    '_TCP_ENDPOINT': [0x210, { # TcpE
        'InetAF' : [0xC, ['pointer', ['_INETAF']]],
        'AddrInfo' : [0x10, ['pointer', ['_ADDRINFO']]],
        'ListEntry': [0x14, ['_LIST_ENTRY']],
        'State' : [0x34, ['Enumeration', dict(
            target='long', choices=TCP_STATE_ENUM)]],
        'LocalPort' : [0x38, ['unsigned be short']],
        'RemotePort' : [0x3A, ['unsigned be short']],
        'Owner' : [0x174, ['pointer', ['_EPROCESS']]],
        'CreateTime' : [0, ['WinFileTime', {}]],
    }],
    '_TCP_SYN_ENDPOINT': [None, {
        'ListEntry': [8, ['_LIST_ENTRY']],
        'InetAF' : [0x24, ['pointer', ['_INETAF']]],
        'LocalPort' : [0x48, ['unsigned be short']],
        'RemotePort' : [0x4a, ['unsigned be short']],
        'LocalAddr' : [0x28, ['pointer', ['_LOCAL_ADDRESS']]],
        'RemoteAddress' : [0x34, ['pointer', ['_IN_ADDR']]],
        'Owner' : [0x2c, ['pointer', ['_SYN_OWNER']]],
        'CreateTime' : [0, ['WinFileTime', {}]],
    }],
    '_TCP_TIMEWAIT_ENDPOINT': [None, {
        'ListEntry': [0, ['_LIST_ENTRY']],
        'InetAF' : [0x18, ['pointer', ['_INETAF']]],
        'LocalPort' : [0x28, ['unsigned be short']],
        'RemotePort' : [0x2a, ['unsigned be short']],
        'LocalAddr' : [0x2c, ['pointer', ['_LOCAL_ADDRESS']]],
        'RemoteAddress' : [0x30, ['pointer', ['_IN_ADDR']]],
        'CreateTime' : [0, ['WinFileTime', {}]],
    }],
}
```

Figure 36: Windows 7 x86 _TCP_ENDPOINT overlay (Rekall Source Code, tcpip_vtypes.py)

Now to find the association from pool tag to the struct name, this is done directly in the `netscan.py` (located in the `plugins` folder) plugin.

```
class PoolScanTcpEndpoint(poolscan.PoolScanner):
    """PoolScanner for TCP Endpoints"""

    def __init__(self, address_space):
        poolscan.PoolScanner.__init__(self, address_space)

        self.pooltag = "TcpE"
        self.struct_name = "_TCP_ENDPOINT"

        self.checks = [('CheckPoolSize', dict(condition = lambda x: x >= 0x1f0)),
                       ('CheckPoolType', dict(non_paged = True, free = True)),
                       ('CheckPoolIndex', dict(value = lambda x : x < 5)),
                       ]
```

Figure 37: `PoolScanTcpEndpoint` class for interpreting the `TcpE` pool tag (Volatility Source Code, `netscan.py`)

To follow a concrete example now in memory, first the layout in memory should be explained. As the IP address value is stored in a `_IN_ADDR` struct, the following picture should give a quick overview of how we find it in memory.

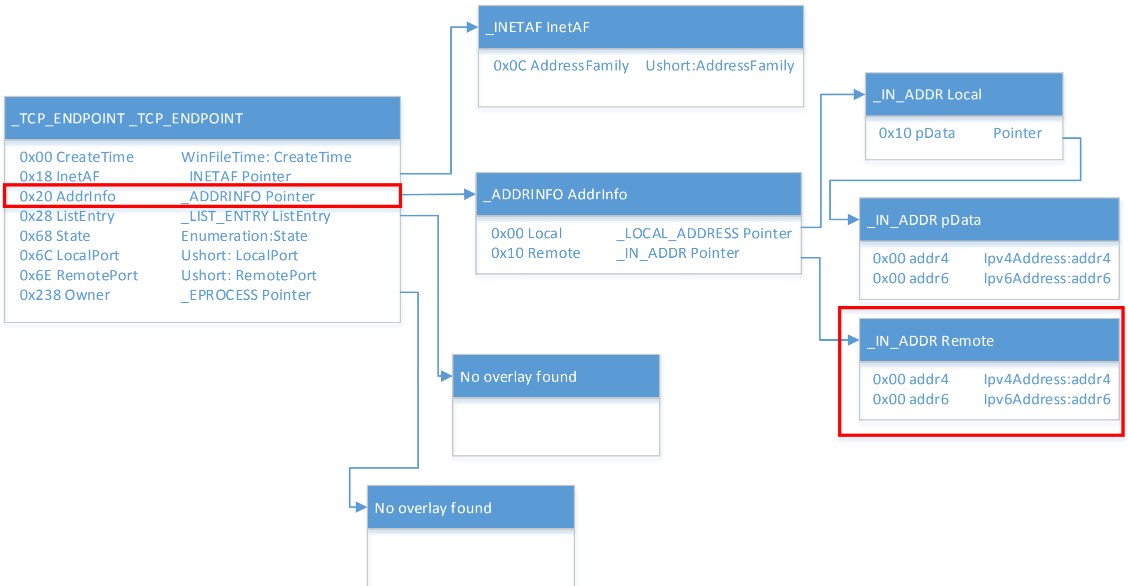

Figure 38: Overlay relationship to locate IP address value in memory in Rekall

As in Rekall, all those overlays are treated in an object oriented fashion, the method of accessing it over the included python shell is actually quite beautiful.

For this case, a Windows 7 x86 image was used. At first, `TcpE` occurrences were searched for by using `yarascan [96]` on the image. This search now is done within Rekall.

```
[1] memorydumpMSSite151002.dmp 21:43:56> yarascan string=”TcpE”
------------------------------------→ yarascan(string=”TcpE”)
```

```
.
.
Rule: r1
Owner: (Unknown Kernel Memory)
  Offset                          Hex                               Data
---------- ------------------------------------------------ ----------------
0x8547ed64 54 63 70 45 00 00 00 00 02 00 00 00 d0 60 ca 84  TcpE…......`..
0x8547ed74 80 c4 b8 84 38 ca 4f 84 70 9e b7 84 70 9e b7 84  ….8.O.p…p…
0x8547ed84 b4 33 60 b9 60 66 b7 84 60 66 b7 84 b9 33 60 b9  .3`.`f..`f…3`.
0x8547ed94 28 ba 5a 87 58 9b 65 84 04 00 00 00 c0 9d 00 50  (.Z.X.e…......P
```

Figure 39: yarascan for `TcpE` in Rekall

It's important to know, that the pool tag is only at position 0x04. This can be verified by checking the header of the pool, as shown in the following picture.

```
[1] memorydumpMSSite151002.dmp 22:18:34> print
session.profile._POOL_HEADER(0x8547ed64-0x4)
[_POOL_HEADER _POOL_HEADER] @ 0x8547ED60
  0x00 PoolIndex                [BitField(9-16):PoolIndex]: 0x00000000
  0x00 PoolType                 [Enumeration:Enumeration]: 0x00000002
                                (NonPagedPoolMustSucceed)
  0x00 PreviousSize             [BitField(0-9):PreviousSize]: 0x00000039
  0x00 Ulong1                   [unsigned long:Ulong1]: 0x04420039
  0x02 BlockSize                [BitField(0-9):BlockSize]: 0x00000042
  0x04 AllocatorBackTraceIndex  [unsigned short:AllocatorBackTraceIndex]:
                                0x00006354
  0x04 PoolTag                  [unsigned long:PoolTag]: 0x45706354
  0x06 PoolTagHash              [unsigned short:PoolTagHash]: 0x00004570
```

Figure 40: Checking the pool header assignments for the current session profile in Rekall

The pool tag value is the offset address that we can work with. The size of _POOL_HEADER on x64 is 0x10, on x32 systems it's 0x08. The _TCP_ENDPOINT struct starts after the _POOL_HEADER end, therefore the point where the overlay for _TCP_ENDPOINT needs to be pointed to is "0x8547ed64-0x4+0x8" in this example.

```
[1] memorydumpMSSite151002.dmp 20:13:20> print
tcpip_profile._TCP_ENDPOINT(0x8547ed64-0x4+0x8)
 [_TCP_ENDPOINT _TCP_ENDPOINT] @ 0x8547ED68
  0x00  CreateTime  [WinFileTime:CreateTime]: 0x00000000 (-)
  0x0C  InetAF      <_INETAF Pointer to [0x84B8C480] (InetAF)>
  0x10  AddrInfo    <_ADDRINFO Pointer to [0x844FCA38] (AddrInfo)>
  0x14  ListEntry   [_LIST_ENTRY ListEntry] @ 0x8547ED7C
  0x34  State       [Enumeration:State]: 0x00000004 (ESTABLISHED)
  0x38  LocalPort   [unsigned be short:LocalPort]: 0x0000C09D
  0x3A  RemotePort  [unsigned be short:RemotePort]: 0x00000050
  0x174 Owner      <_EPROCESS Pointer to [0x00000000] (Owner)>
```

Figure 41: Overlaying offset address with `_TCP_ENDPOINT` overlay in Rekall

To now access the `AddrInfo` overlay, we use the `AddrInfo` function on this overlay.

```
[1] memorydumpMSSite151002.dmp 19:26:33> print
tcpip_profile._TCP_ENDPOINT(0x8547ed64-0x4+0x8).AddrInfo
Pointer to [_ADDRINFO AddrInfo] @ 0x844FCA38
  0x00 Local  <_LOCAL_ADDRESS Pointer to [0x84B95230] (Local)>
  0x08 Remote <_IN_ADDR Pointer to [0x844FCAC8] (Remote)>
```

Figure 42: Displaying the `AddrInfo` overlay in Rekall

Now following the last pointer gives us finally the wanted IP address.

```
[1] memorydumpMSSite151002.dmp 19:26:46> print
tcpip_profile._TCP_ENDPOINT(0x8547ed64-0x4+0x8).AddrInfo.Remote
Pointer to [_IN_ADDR Remote] @ 0x844FCAC8
  0x00 addr4  [Ipv4Address:addr4]: '217.163.21.35'
  0x00 addr6  [Ipv6Address:addr6]: 'd9a3:1523::d8c7:4f84:100:e0'
```

Figure 43: Display of Remote IP address by using overlays in Rekall

If we want to find this information without the overlays, we can do this directly. The `yarascan-TcpE` output from Figure 39 outputs `TcpE` being at the offset address 0x8547ed64. Now adding 0x04 to move to the end of the header and again 0x10 (+ 0x14 in total, which means 20 blocks further) to reach the location where the pointer to the `AddrInfo` is stored.

```
Rule: r1
Owner: (Unknown Kernel Memory)
  Offset                          Hex                              Data
---------- ------------------------------------------------ ----------------
0x8547ed64 54 63 70 45 00 00 00 00 02 00 00 00 d0 60 ca 84  TcpE…......`..
0x8547ed74 80 c4 b8 84 38 ca 4f 84 70 9e b7 84 70 9e b7 84  ….8.O.p…p…
0x8547ed84 b4 33 60 b9 60 66 b7 84 60 66 b7 84 b9 33 60 b9  .3`.`f..`f…3`.
0x8547ed94 28 ba 5a 87 58 9b 65 84 04 00 00 00 c0 9d 00 50  (.Z.X.e…......P
```

Figure 44: Finding the IP address in hex in Rekall

The offset address that it pointed to is stored in little endian (marked in red and bold in Figure 44).

```
[1] memorydumpMSSite151002.dmp 21:27:03> analyse_struct 0x844fca38
-------------------------------------→ analyse_struct(0x844fca38)
2016-04-26 21:41:42,581:DEBUG:rekall.1:Running plugin (analyse_struct) with args
((2219821624,)) kwargs ({})
  Offset     Content
----------  -------
       0x0  Data:0x84b95230
       0x4  Data:0xe0000001
       0x8  Data:0x844fcac8
       0xc  Data:0x22
       0x7c Data:0x0
                                  Out<3> Plugin: analyse_struct
```

Figure 45: `analyse_struct` display of offset address of `_ADDRINFO` struct in Rekall

Now that we know that the Remote address is stored at 0x8, we can display the value that is stored at the offset it's pointing to.

```
[1] memorydumpMSSite151002.dmp 21:42:43> analyse_struct 0x844fcac8
-------------------------------------→ analyse_struct(0x844fcac8)
2016-04-26 21:46:33,434:DEBUG:rekall.1:Running plugin (analyse_struct) with args
((2219821768,)) kwargs ({})
  Offset   Content
---------- -------
       0x0 Data:0x2315a3d9
       0x4 Data:0x0
       0x8 Data:0x844fc7d8
                                  Out<4> Plugin: analyse_struct
```

Figure 46: `analyse_struct` display of offset address of the Remote `_IN_ADDR` struct in Rekall

The value at 0x0 is the calculated IP address. It's again stored in little endian:

```
2315a3d9 => 0xd9 (=217).0xa3(=163).0x15(=21).0x23(=35)
```

Figure 47: Calculation of IP address from hex value

You can also gain this value by using an online hex to IP address calculator [97], or using the scheme described on stack overflow [98].

4.4.4. Pool Tags in Linux

According to experts who developed Volatility and Rekall, Linux doesn't use anything akin to pool tags [99]. This is why memory forensic tools for Linux even more than Windows, need to have a profile created before the needed analysis, to interpret the available information effectively. Also, as struct structures can vary with different patch levels, the matching versions of profiles shouldn't be underestimated.

Pool Tags are a very useful feature in Windows, but should not be kept as the only information of proof, as they can be easily changed by malicious code. A second verification with other layout information is therefore preferable.

4.5. Discussion of Own Contribution

To be able to actually draw a comparison from the socket stack to the NDIS 6.* stack, it was important to get an overview of the different setups first. As the NDIS stack was mostly described on MSDN only, it was attempted to put that information into a more graphical form.

The full picture with better resolution can be found on the project GitHub page [0].

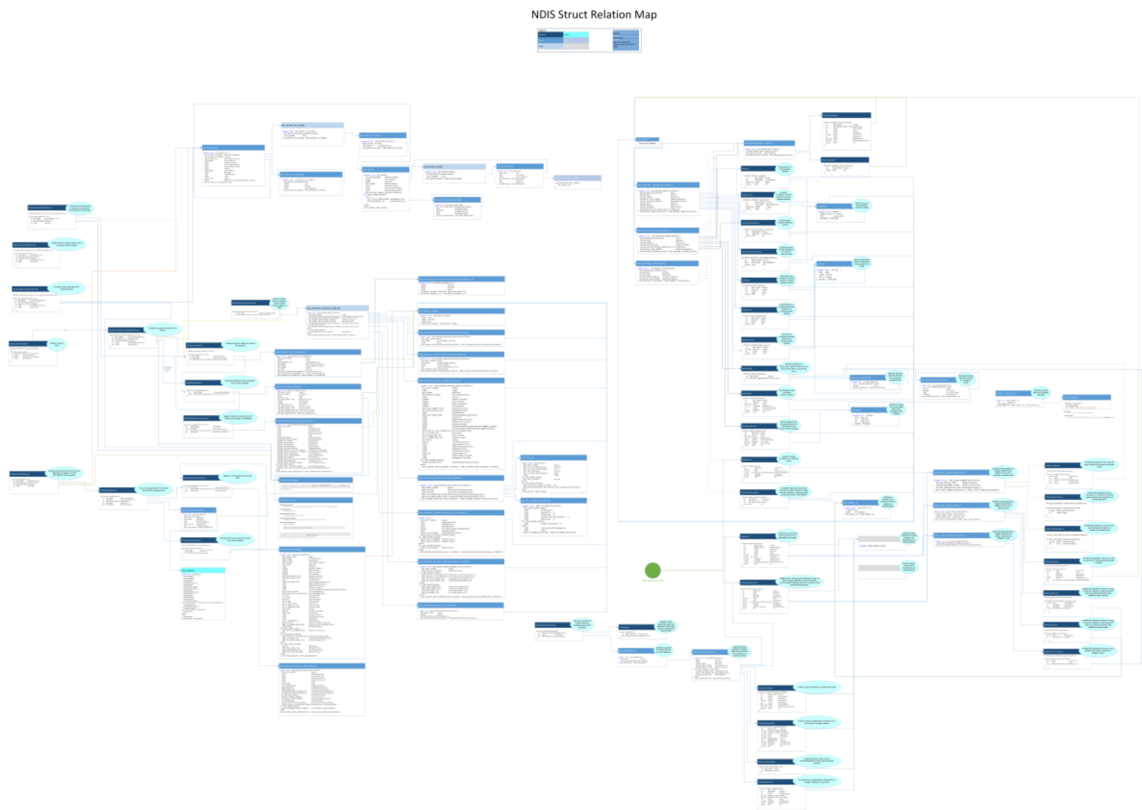

Figure 48: Relation between NDIS structs

The “NDIS Struct Relation Map” helped in identifying the different features, options and settings of storage, and in what structs or functions they are located. This also helped making obvious that the storage of interface related information, including the `NET_BUFFER_LIST`, were used and associated differently.

As the memory always is a snapshot of a point in memory, a second version was created to add the return values of the functions as well. It can eventually also be found on the GitHub site.

This diagram displays how the information in memory can be found, and how the structs could possibly be put together. It contains all NDIS structs and its member’s relations to other NDIS structs, functions, unions or other types found during this timeframe. The basic values like integer, string, ulong weren’t described. They are well described on the internet [100][101] and not specific for the NDIS related research. As the NDIS stack is constantly developed and drawing the scope was challenging, the map might not be complete though.

It was attempted to cover also opaque or partially opaque structures. They are marked by different colours (grey) and only visibility of the known parts is included.

For overall comparison between Microsoft NDIS 6.* and Linux socket stack, the following findings were conducted:

Basic concepts, like the socket function or how IP addresses are stored, are almost identical in the Windows NDIS 6 stack and Linux socket stack. This is most likely due to the standard being defined by the Berkeley University of California, which standard is used for BSD operating systems.

The socket structures in Linux, which hold the network socket configuration, for possible network connections, are split over different structs in Windows that collect the dispatch state of the sockets depending on the protocol used. The information in these structs is fetched by functions.

Buffering is similar by being a double linked list in Linux and Windows only using a single linked list. On the other hand, Linux only knows a "buffer header" (`sk_buff_head`) and links the buffered elements directly. Windows on the other hand knows `NET_BUFFER_LIST` which point to other `NET_BUFFER_LIST` and `NET_BUFFER` structures. A `NET_BUFFER` object contains a separate network packet. This `NET_BUFFER` has a pointer to the `NET_BUFFER_DATA` section that points to a Memory Descriptor List, a virtual memory allocation, where the data is actually stored.

Queuing is directly done in `NET_BUFFER_LIST` structures that are assigned to the interfaces. There is a receiving and a sending queue possible per interface, assigned depending on its needs. Error tracking is directly done in the `NET_BUFFER_LIST` structure.

The physical device settings are stored in different layers of device configuration, as software interaction is done over the protocol driver, and only that protocol driver talks to the physical interface (which is called Miniport). Settings for the interface are collected in unions, and are stored in a struct that stores the information for binding the NDIS part to the interface.

The originally asked questions to better understand network and forensic tools capabilities could be answered by finding the storage of the IP address in memory, and comparing netstat of Windows and Linux, and the forensic tools that parse that information.

5. Practical Approach

To document the results this thesis initially reached for, which is based on the understanding gained from analysis of the previous chapter, they were unfolded in this separate chapter. It is based on practical analysis in a memory image.

5.1. Explanation

Memory forensic is already a discipline that heavily depends on reverse engineering tactics, especially for Microsoft images. As a memory image is only a snapshot of a computer, reflecting a moment in time when many things were executed, it's hard to use for reconstructing complete execution flows. It's possible to find out former actions if the memory hasn't been overwritten in the meantime, in general this can't be guaranteed however. Knowledge in how memory is allocated in an operating system is therefore essential to effective memory forensics.

5.2. Description of Material at Hand

There are only few suggested methods on how to approach finding new information within image files. A good overview can be found in "The Art of Memory Forensics" (B1) chapter 5 – Windows Objects and Pool allocations. They propose pool-tag scanning in general but also mention the limitations and suggest dispatcher header scans and robust signature scans as alternatives.

Another suggested method was described in the paper "Network Connections Information Extraction of 64-Bit Windows 7 Memory Images" (B2). It suggests locating the network information based on a proposed algorithm that first starts with locating the KPCR (see chapter 3.2.5), afterwards finding the `tcpip.sys` driver by using the current active thread, and following the KTHREAD structure.

The paper "Windows Memory Analysis Based on KPCR" (B3) is more focused on building memory forensic analysis in general. As the KPCR is the location where also Volatility and Rekall first look for information, this seems to be well considered and is already used by today's memory forensic tools.

In "Characterization of the windows kernel version variability for accurate memory analysis" (B4), the problem with changing struct structures is described, that can differ between different kernel versions. It's a good explanation why Rekall uses PDB files and locates the accurate versions to interpret the found information. This paper, in its conclusion, also brings to attention that for example that the `tcpip.sys` driver is largely undocumented and "matching known binaries to the exact running binary in memory image is a critical first step to memory analysis of all operating systems".

"Forensic Analysis of Windows User space Applications through Heap allocations" (B5) describes tools to analyse page files and the allocated sections in memory. The tools discussed are plugins added in Rekall. This paper also describes Michael's suggested path to follow for analysis, when this thesis was initially started and the project goals were initially defined.

5.3. Result and Discussion of Material at Hand

All of the resources mentioned in the previous chapter were a good help in understanding the problem better. They explained previous attempts, although scrutinizing was difficult because there was not enough time to test all suggested solutions. “Characterization of the windows kernel version variability for accurate memory analysis” (B4) and “Forensic Analysis of Windows User space Applications through Heap allocations” (B5) turned out to be the most helpful, as they were most recent and as written by the author of Rekall, most fit for the further usage with the tool suite.

As meetings with the Google developer of Rekall, Michael Cohen, were hold already before this thesis was started, the selection of papers was possibly biased. When questioning the method with the understanding gained from reading “The Art of Memory Forensics” (B1), it appeared reasonable though. Also, first building the knowledge from the theoretical research, a systematic decision for further analysis led to follow up on different structs and pool tags, instead the ones previously known.

An initial idea was to further elaborate on the `_TCP_ENDPOINT` structure. In the first meeting with Michael, the hash table where netstat.exe information is stored in was found in memory with the pool tag `TcHT`, referencing another hash table with the pool tag `Htab`. This was not followed up on because of focusing the thesis on the analysis and comparison of possible new structs.

5.4. Own Contribution for Resolution of Problem

For analysing the found structures in memory, a systematic approach was decided on, which consisted of deciding for a use case that analysis could be built on. A proof of concept was planned to use the theoretically built up knowledge and interpret information in memory. The feasibility study and it’s follow up action was decided to be optional at the design review meeting and are therefore only mentioned in chapters as a reference of completion and to clearly illustrate the coverage of the initial set goals.

5.4.1. Definition of Use Cases for the Analyse over Multiple Windows Versions that have a NDIS 6.* Stack

The use case for the analysis was defined as something simple that could be easily replicated on all different versions of Windows. It had three goals, which are graphically described in the following picture.

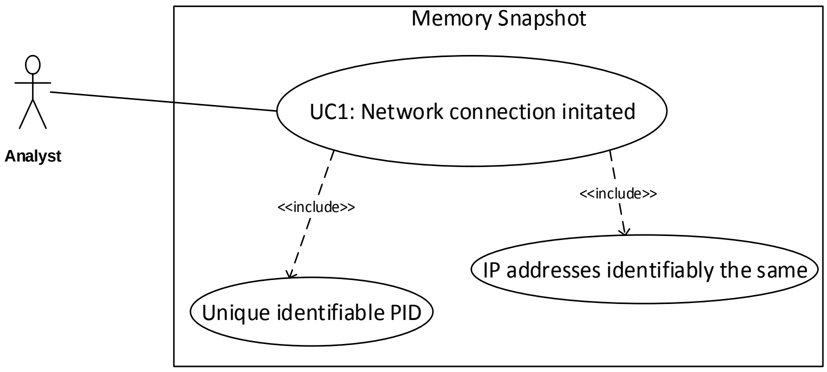

Figure 49: Use case for memory analysis

Network connection initiated

This step could have been easily covered by just starting a `ping` or `nslookup` to a system on the internet, for easy analysis it was discarded though to instead choose a noisier connection. It should also be noted that an `nslookup` connection is relatively quick, and therefore taking a memory snapshot at the right moment could potentially be a challenge, and lead to unpredictable results.

Unique identifiable PID

The first thought was to just start Windows Update and let it initiate a long lived connection. This is done over background processes using Windows Background Intelligent Transfer Service – BITS [102] and therefore not clearly identifiable.

IP addresses identifiable the same

It's possible that connections opened to DNS names are load balanced and therefore connect to different end systems of the same cloud service. This was considered less of a problem if the IP address could be associated with the same organization or type of connection.

To have an easy simple example, it was chosen to open Internet Explorer and let it connect to www.microsoft.com. This way, a separate PID was available for tracing in the analysis, and several IP addresses are available to be located in the memory image when the connection to the website is initiated. Also as downloading the website without cached content takes some moments, there should be enough evidence available in memory after the connection ends.

The backup use case to have a simpler connection would have been to just create a `ping -t` connection while performing the snapshot of memory.

For the sake of completeness, the complete description of the use case is found in the following table.

Identification	UC1: Network Connection initiated
Name	Network Connection initiated
Description	A network connection is initiated that has unique identifiable PID and a predictable IP address for easy identification.
Responsible(s)	Analyst
Causing event	Starting the Internet Explorer Application and opening a connection to www.microsoft.com
Actor	Analyst
Precondition	Running Windows system in a virtual machine that can create network connections.
Postcondition	Memory image taken while the connection was initiated
Result	Memory image of a normal installed and patched Windows system, initiating a connection to the website of www.microsoft.com and its artefacts can be found in memory.

Identification	UC1: Network Connection initiated
Main Scenario	1. Analyst starts Windows VM and logs into Windows 2. Analyst opens the Internet Explorer 3. Analyst goes to www.microsoft.com 4. Analyst either creates the memory image by using a tool like DumpIt [103], Pmem [104] or hibernating the VM image and using the hibernation file [105]
Alternate scenario	-
Exception scenario	- The creation of the image file is interrupted my errors - The created memory image can't be parsed by the common tools, a different tool should be used
Priority	100%

Table 6: Use Case description UC1: Network connection initiated

5.4.2. Analyse the NDIS Structs in Memory as Proof of Concept with Corresponding Tools and Scripts

To analyse the NDIS structs in memory, first the association with the known structs had to be done. The tool of choice for this analysis was Rekall, as it displayed, and interpreted information such as kernel data structures based on the associated PDB files and it included features to show memory allocations as described in the bibliography reference (B5).

5.4.2.1. Introduction

When following the path to locate the storage of the IP address in chapter 4.4.3, we followed the `_TCP_ENDPOINT` struct. It was attempted to match the known parts of this struct to one of the structs of the map, but a match couldn't be found. There are two possible reasons for this.

1. Microsoft changes structs when the information is copied into memory
2. The `_TCP_ENDPOINT` struct is not part of the publicly documented NDIS 6.* stack, and instead specific to the netstat context.

As the state of knowledge presents itself now, the second option is considered more reasonable. This leads to the conclusion, that for identifying other structs, the `TcpE` pool tag can be ignored and instead other pool tags shall be attempted to be identified and matched to possible other structs.

A good start for finding more pool tags is given by the `pool_tracker` plugin which does show the amount of occurrences of each pool tag. The output was merged with the pool tag list by a self-written `awk` script [106] to easily filter for `tcpip.sys` related pool tags only. The full list can be found in the Appendix E on page 90.

Another approach would be by scanning the image for hex values matching the found IP addresses and see what and if pool tags are near them.

To further analyse the _TCP_ENDPOINT struct and its exact usage, next steps could be to disassemble the kernel from the memory image. This can also be useful when researching pool tags in general. Instructions for the concrete TcpE example can be found in Appendix F on page 95.

5.4.2.2. Proof of Concept

From the analysis of the relation map, the function NdisSendNetBufferLists looked like an interesting place to find in memory. It is used to send network data that is contained in a NET_BUFFER_LIST. It uses information like the NdisBindingHandle, the NetBufferLists, the NDIS PortNumber and SendFlags for this. The possibility would be to find a place where the NET_BUFFER_LIST is associated to the port and therefore can be associated to a service.

When checking the pool tag list, the tag TNbl looked like an interesting tag to follow up on, as from its description, is about "TCP Send NetBufferLists". Converting the tag in the test memory image returned the hex value 0x6c624e54. Searching the previously exported disassembly of the tcpip.sys module part[10], the hex value was found in the tcpip!TcpStartSendModule function. Near the end of the disassembly section, the pool tag seems to be written (at offset address 0x8722893d).

```
------ tcpip!TcpngpReferenceDriver ------
  0x87228804        0x0 33c9            XOR ECX, ECX
  0x87228806        0x2 b8007d3187      MOV EAX, 0x87317d00
                                        tcpip!TcpngpReferenceCount
  0x8722880b        0x7 41              INC ECX
  0x8722880c        0x8 f00fc108        LOCK XADD [EAX], ECX
  0x87228810        0xc c3              RET
  0x87228811        0xd 90              NOP
  0x87228812        0xe 90              NOP
  0x87228813        0xf 90              NOP
  0x87228814       0x10 90              NOP
  0x87228815       0x11 90              NOP
------ tcpip!TcpStartSendModule ------
  0x87228816        0x0 8bff            MOV EDI, EDI
  0x87228818        0x2 56              PUSH ESI
  0x87228819        0x3 57              PUSH EDI
  0x8722881a        0x4 6854635352      PUSH DWORD 0x52536354
  0x8722881f        0x9 6a38            PUSH 0x38
  0x87228821        0xb e8f8d2ffff      CALL 0x87225b1e
                                        tcpip!InetAcquireFsbPool
  0x87228826       0x10 a3e4a73187      MOV [0x8731a7e4], EAX
                                        0x84385840 tcpip!TcpSendRequestPool
  0x8722882b       0x15 85c0            TEST EAX, EAX
  0x8722882d       0x17 757a            JNZ 0x872288a9
                                        tcpip!TcpStartSendModule + 0x93
  0x8722882f       0x19 833ddc3c318701  CMP DWORD [0x87313cdc], 0x1    0x0
                                        tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                        + 0x24
  0x87228836       0x20 7567            JNZ 0x8722889f
                                        tcpip!TcpStartSendModule + 0x89
  0x87228838       0x22 a0e03c3187      MOV AL, [0x87313ce0]           0x0
                                        tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                        + 0x28
```

[10] Instructions in Appendix F

```
0x8722883d      0x27 3c02             CMP AL, 0x2
0x8722883f      0x29 7304             JAE 0x87228845
                                      tcpip!TcpStartSendModule + 0x2f
0x87228841      0x2b 84c0             TEST AL, AL
0x87228843      0x2d 755a             JNZ 0x8722889f
                                      tcpip!TcpStartSendModule + 0x89
0x87228845      0x2f 8b0dc83c3187     MOV ECX, [0x87313cc8]         0x0
                                      tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                      + 0x10
0x8722884b      0x35 8b15cc3c3187     MOV EDX, [0x87313ccc]         0x0
                                      tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                      + 0x14
0x87228851      0x3b be80000000       MOV ESI, 0x80
0x87228856      0x40 b840000080       MOV EAX, 0x80000040
0x8722885b      0x45 23ce             AND ECX, ESI
0x8722885d      0x47 23d0             AND EDX, EAX
0x8722885f      0x49 0bca             OR ECX, EDX
0x87228861      0x4b 743c             JZ 0x8722889f
                                      tcpip!TcpStartSendModule + 0x89
0x87228863      0x4d 8b0dd03c3187     MOV ECX, [0x87313cd0]         0x0
                                      tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                      + 0x18
0x87228869      0x53 8b3dd43c3187     MOV EDI, [0x87313cd4]         0x0
                                      tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                      + 0x1c
0x8722886f      0x59 8bd1             MOV EDX, ECX
0x87228871      0x5b 23d6             AND EDX, ESI
0x87228873      0x5d 8bf7             MOV ESI, EDI
0x87228875      0x5f 23f0             AND ESI, EAX
0x87228877      0x61 3bd1             CMP EDX, ECX
0x87228879      0x63 7524             JNZ 0x8722889f
                                      tcpip!TcpStartSendModule + 0x89
0x8722887b      0x65 3bf7             CMP ESI, EDI
0x8722887d      0x67 7520             JNZ 0x8722889f
                                      tcpip!TcpStartSendModule + 0x89
0x8722887f      0x69 6846612487       PUSH DWORD 0x87246146
                                      tcpip!str:send9request_pool_(TCP) +
                                      0x10
0x87228884      0x6e 685c083087       PUSH DWORD 0x8730085c
                                      tcpip!MICROSOFT_TCPIP_PROVIDER
0x87228889      0x73 68400f3087       PUSH DWORD 0x87300f40
                                      tcpip!TCPIP_MEMORY_FAILURES
0x8722888e      0x78 ff35b43c3187     PUSH DWORD [0x87313cb4]       0x0
                                      tcpip!Microsoft_Windows_TCPIPHandle +
                                      0x4
0x87228894      0x7e ff35b03c3187     PUSH DWORD [0x87313cb0]       0x2f
                                      tcpip!Microsoft_Windows_TCPIPHandle
0x8722889a      0x84 e8952a0900       CALL 0x872bb334
                                      tcpip!TcpipTransferActivityIDToNBL +
                                      0x37
0x8722889f      0x89 b8170000c0       MOV EAX, 0xc0000017
0x872288a4      0x8e e935010000       JMP 0x872289de
                                      tcpip!TcpStartSendModule + 0x1c8
0x872288a9      0x93 6854534e62       PUSH DWORD 0x624e5354
0x872288ae      0x98 be80000000       MOV ESI, 0x80
0x872288b3      0x9d 56               PUSH ESI
```

```
0x872288b4       0x9e e85b1d0000          CALL 0x8722a614
                                          tcpip!
                                          NetioAllocateNetBufferMdlAndDataPool
0x872288b9       0xa3 a3e8a73187          MOV [0x8731a7e8], EAX
                                          0x849f6b80 tcpip!TcpSendNetBufferPool
0x872288be       0xa8 85c0                TEST EAX, EAX
0x872288c0       0xaa 757b                JNZ 0x8722893d
                                          tcpip!TcpStartSendModule + 0x127
0x872288c2       0xac 833ddc3c318701      CMP DWORD [0x87313cdc], 0x1    0x0
                                          tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                          + 0x24
0x872288c9       0xb3 7562                JNZ 0x8722892d
                                          tcpip!TcpStartSendModule + 0x117
0x872288cb       0xb5 a0e03c3187          MOV AL, [0x87313ce0]           0x0
                                          tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                          + 0x28
0x872288d0       0xba 3c02                CMP AL, 0x2
0x872288d2       0xbc 7304                JAE 0x872288d8
                                          tcpip!TcpStartSendModule + 0xc2
0x872288d4       0xbe 84c0                TEST AL, AL
0x872288d6       0xc0 7555                JNZ 0x8722892d
                                          tcpip!TcpStartSendModule + 0x117
0x872288d8       0xc2 8b0dc83c3187        MOV ECX, [0x87313cc8]          0x0
                                          tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                          + 0x10
0x872288de       0xc8 8b15cc3c3187        MOV EDX, [0x87313ccc]          0x0
                                          tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                          + 0x14
0x872288e4       0xce b840000080          MOV EAX, 0x80000040
0x872288e9       0xd3 23ce                AND ECX, ESI
0x872288eb       0xd5 23d0                AND EDX, EAX
0x872288ed       0xd7 0bca                OR ECX, EDX
0x872288ef       0xd9 743c                JZ 0x8722892d
                                          tcpip!TcpStartSendModule + 0x117
0x872288f1       0xdb 8b0dd03c3187        MOV ECX, [0x87313cd0]          0x0
                                          tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                          + 0x18
0x872288f7       0xe1 8b3dd43c3187        MOV EDI, [0x87313cd4]          0x0
                                          tcpip!MICROSOFT_TCPIP_PROVIDER_Context
                                          + 0x1c
0x872288fd       0xe7 8bd1                MOV EDX, ECX
0x872288ff       0xe9 23d6                AND EDX, ESI
0x87228901       0xeb 8bf7                MOV ESI, EDI
0x87228903       0xed 23f0                AND ESI, EAX
0x87228905       0xef 3bd1                CMP EDX, ECX
0x87228907       0xf1 7524                JNZ 0x8722892d
                                          tcpip!TcpStartSendModule + 0x117
0x87228909       0xf3 3bf7                CMP ESI, EDI
0x8722890b       0xf5 7520                JNZ 0x8722892d
                                          tcpip!TcpStartSendModule + 0x117
0x8722890d       0xf7 6808612487          PUSH DWORD 0x87246108
                                          tcpip!str:
                                          aggregate9NetBuffer_pool_(TCP) + 0x10
0x87228912       0xfc 685c083087          PUSH DWORD 0x8730085c
                                          tcpip!MICROSOFT_TCPIP_PROVIDER
0x87228917      0x101 68400f3087          PUSH DWORD 0x87300f40
                                          tcpip!TCPIP_MEMORY_FAILURES
```

0x8722891c	0x106	ff35b43c3187	PUSH DWORD [0x87313cb4] 0x0 tcpip!Microsoft_Windows_TCPIPHandle + 0x4
0x87228922	0x10c	ff35b03c3187	PUSH DWORD [0x87313cb0] 0x2f tcpip!Microsoft_Windows_TCPIPHandle
0x87228928	0x112	e8072a0900	CALL 0x872bb334 tcpip!TcpipTransferActivityIDToNBL + 0x37
0x8722892d	0x117	ff35e4a73187	PUSH DWORD [0x8731a7e4] 0x84385840 tcpip!TcpSendRequestPool
0x87228933	0x11d	e8de000900	CALL 0x872b8a16 tcpip!InetInspectRemoteDisconnect + 0xc
0x87228938	0x122	e962ffffff	JMP 0x8722889f tcpip!TcpStartSendModule + 0x89
0x8722893d	0x127	68544e626c	PUSH DWORD **0x6c624e54**
0x87228942	0x12c	56	PUSH ESI
0x87228943	0x12d	6a08	PUSH 0x8
0x87228945	0x12f	e89c000000	CALL 0x872289e6

Figure 50: Disassembly of `tcpip.sys` `TcpStartSendModule` section

Now if we recall the `NdisSendNetBufferLists` function, we are looking for a NDIS handle, a pointer to a NET_BUFFER_LIST, a NDIS_PORT_NUMBER and the flags formatted in ULONG.

C++

```
VOID NdisSendNetBufferLists(
  _In_ NDIS_HANDLE       NdisBindingHandle,
  _In_ PNET_BUFFER_LIST NetBufferLists,
  _In_ NDIS_PORT_NUMBER PortNumber,
  _In_ ULONG             SendFlags
);
```

Figure 51: `NdisSendNetBufferLists` function from MSDN

If we check the disassembly output from Figure 50, we see that right before the pool tag is being pushed to the stack, a value for the `Microsoft_Windows_TCPIPHandle` function is pushed as well as a value for the `TcpipTransferActivityIDtoNBL` function, where NBL most likely stands for NET_BUFFER_LIST.

In a next step, those offset addresses could be further followed up on. If we `yarascan` the memory image for the pool tag to see if these information are further stored in memory, no direct link between the hex values from the disassembly and the followed values in the memory could be found. A further check to analyse was done by using `analyse_struct(offsetaddress-0x60)` to also check the address space before the tags.

```
Owner
Symbol
Role   Offset       HexDump
- --   ----------   ------------------------------------------------------------------
- r1   0x84115264   54 4e 62 6c 39 00 00 34 00 00 10 09 00 00  TNbl9…4……...
                    00 00 00 00 00 00 00 00 00 00 00 00 50 58 61 00  …........Pxa.
                    05 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00  …............
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
- r1   0x84483e34   54 4e 62 6c 78 14 37 85 08 b0 85 85 bc ca 55 84  TNblx.7……...U.
                    00 00 00 00 03 00 12 08 43 63 42 63 40 64 37 85  ……..CcBc@d7.
                    00 10 00 00 00 20 01 00 00 00 00 00 78 28 ab 85  …........x(..
                    80 42 64 84 00 30 01 00 00 00 00 00 00 00 00 00  .Bd..0……......
- r1   0x844acdb4   54 4e 62 6c 13 00 39 04 41 6c 65 45 00 00 00 00  TNbl..9.AleE….
                    00 00 00 00 06 00 00 00 00 00 00 00 01 13 00 00  …............
                    04 00 00 00 02 00 c0 80 00 00 00 00 00 00 00 00  …............
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
- r1 0x8455cab4     54 4e 62 6c c8 d7 87 85 d0 92 40 84 ac b8 d3 84  TNbl…..@.....
                    00 00 00 00 03 00 12 04 43 63 42 63 fd 02 01 00  ……..CcBc….
                    00 10 00 00 00 40 9f 02 00 00 00 00 58 8b aa 85  ….@......X...
                    D0 72 55 84 00 50 9f 02 00 00 00 00 61 aa 7c 4b  .rU..P……..a.|K
- r1 0x8456532c     54 4e 62 6c 28 27 8f 85 20 25 44 85 3c 6e 68 84  TNbl(,…%D.<nh.
                    00 00 00 00 00 00 00 00 e0 53 56 84 40 54 56 84  ……...SV.@TV.
                    00 00 00 00 80 fb 9f 84 9a 70 00 00 00 00 00 00  ……..p…...
                    00 00 00 00 00 00 00 00 00 00 00 00 aa ce 27 87  …..........’.
- r1 0x84686c64     54 4e 62 6c 80 fb 9f 84 3c 6e 68 84 5c 25 8f 85  TNbl….<nh.\%..
                    00 00 00 00 68 25 8f 85 18 6d 68 84 78 6d 68 84  ….h%...mh.xmh.
                    00 00 00 00 80 fb 9f 84 00 00 00 00 00 00 00 00  …............
                    00 00 00 00 00 00 00 00 00 00 00 00 aa ce 27 87  …..........‘.
- r1 0x84686e34     54 4e 62 6c 80 fb 9f 84 88 fb 9f 84 6c 6c 68 84  TNbl……...llh.
                    00 00 00 00 40 68 4a 85 e8 6e 68 84 48 6f 68 84  ….@hJ..nh.Hoh.
                    00 00 00 00 80 fb 9f 84 70 00 00 00 00 00 00 00  ……..p……...
                    00 00 00 00 00 00 00 00 00 00 00 00 aa ce 27 87  …..........‘.
- r1 0x849ffb3c     54 4e 62 6c 00 00 00 00 00 00 00 00 00 00 00 00  TNbl…........
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
- r1 0x849ffba0     54 4e 62 6c 07 00 00 00 00 00 00 00 ac 5b 9a 84  TNbl……....[..
                    ac 1b a1 84 02 00 00 00 80 00 00 00 24 00 00 00  …........$...
                    48 6e 68 84 04 00 00 00 04 00 00 04 d9 0e 00 00  Hnh….........
                    38 00 00 00 d9 0e 00 00 34 00 00 00 00 00 00 00  8……..4……...
- r1 0x849ffbe0     54 4e 62 6c b4 01 00 00 fb 5d 09 87 1c 33 09 87  TNbl…..]…3..
                    f0 1b a1 84 f0 6b 9f 84 d9 0e 00 00 38 00 00 00  ….k…...8…
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  …............
```

Figure 52: yarascan “TNbl” output

As these tests were only run at the very end of the thesis, this could not be followed up on more during the official time. Instead it will be further analysed after closing date.

Structs, which through this kind of research can be identified in memory, are planned to be added to for example the `netscan` plugin, to include such information in future analysis automatically.

5.4.3. (Optional) Feasibility Study: Identification of Recognition Features of Network Connection Elements

Tests for identifying further recognition features would have a similar approach as described in chapter 5.4.2. For example, interface settings are stored in the struct `NDIS_BIND_PARAMETERS` when the driver requests to bind the protocol driver to the miniport adapter (when sending packets). Also on a miniport level, most of these settings are located in the `NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES` struct. On the other side, when receiving a packet, the struct `NDIS_PORT_CHARACTERISTICS` is used to associate an NDIS port with a miniport adapter.

As visible in the full list in Appendix E, the pool tag “IP Interfaces” `Ipif` does exist and is found in the image. It’s possible that it can be related to the storage options of one of the above structs. Although it’s currently not clear, if in this context pool tags are used at all.

5.4.4. (Optional) Determination of Behaviour of Structs in Memory by Utilising Defined Use Cases

The scenario laid out in chapter 5.4.3 could be tested with further images, if associations are possible to make. This would be especially important as memory handling is not only different in the main Windows versions like Windows 7, Windows 8, and so forth, but also in between different editions like Windows Professional, Ultimate, Home and such. This is caused by different features and ways of implementations used. Also for example new features like the just recently announced memory compression [107] for Windows 10 has already been pushed to some Windows 8 systems.

5.5. Discussion of Own Contribution

The main learning of this analysis was, that the `_TCP_ENDPOINT` structure, used in the netstat context to store the local and remote address of connections, is not used in other parts of the NDIS stack. This was first assumed when comparing the structure to the officially documenters structures on the map, and afterwards confirmed when learning that only one function uses this pool tag.

As for the proof of concept, a first result was, that the information couldn’t be found in memory. Unfortunately, this can’t be taken as a full proof, as it’s possibly that with more time, the information can be better understood and be found after all. As the state presents itself now, the proof of concept failed, further tests will be conducted after the delivery of this paper.

6. Conclusion

The work on this thesis was fascinating and challenging from the beginning until the end. The biggest lesson though was, that right when initially the problems appeared to be understood, the realization afterwards came that it wasn't understood after all.

To compare the NDIS stack to the socket stack, helped in building a solid basic understanding of its tasks, and also made it easier to interpret the information. For example, the comments in the socket stack `net_device` structure, about covering too many levels and potentially being a security risk, explained the approach Microsoft chose when going for different levels of device drivers. Also it was a good guidance of what structures were not found yet, when building the big "NDIS Struct Relation Map". A basic understanding of the composition and structure of the NDIS 6 stack could be achieved.

The initial goal of finding more structs in memory couldn't be reached, as in retrospect, this goal was a little naïve with the background knowledge available at that time. Fortunately, this thesis turned out as a collection of all learnings achieved over the past few months, and can now be used as a good introduction to start further research. As the topic is still interesting, and there is still too little understanding, and more structs and behaviours to be recognized in memory, further research will most likely continue in the next few months.

A big lesson learned was the difference of collecting knowledge, and transforming the gained knowledge to a level of understanding. This was hard to estimate time for, as it depended on the form on the day and could not be predicted. It's one of the reasons why further analysis after the thesis completion sounds reasonable, as this way identifying new information under time pressure can be avoided.

References

The thesis related documents are stored on the following GitHub site:

[0] https://github.com/d3sre/Understanding_the_NDIS_6_stack

This is a list of all Internet references, mentioned throughout the paper.

[1] https://msdn.microsoft.com/en-us/library/windows/hardware/ff565448%28v=vs.85%29.aspx
[2] https://github.com/corkami/pics
[3] https://www.zhaw.ch/de/engineering/studium/bachelorstudium/informatik/
[4] http://d3sre.livejournal.com/1150.html
[5] http://pomodorotechnique.com/
[6] https://www.csirt.org/publications/navy.htm
[7] https://www.fireeye.com/services/freeware/redline.html
[8] https://digital-forensics.sans.org/community/cheat-sheets
[9] https://digital-forensics.sans.org/media/volatility-memory-forensics-cheat-sheet.pdf
[10] https://digital-forensics.sans.org/media/Poster-2015-Memory-Forensics2.pdf
[11] https://en.wikipedia.org/wiki/Reverse_engineering
[12] https://www.hex-rays.com/products/ida/
[13] https://www.hex-rays.com/products/ida/news.shtml
[14] http://securityxploded.com/reversing-basics-ida-pro.php
[15] http://0x1338.blogspot.ch/2015/07/blackhoodie-reversing-workshop-for.html
[16] https://docs.python.org/2/tutorial/interpreter.html
[17] https://docs.python.org/2/library/dis.html
[18] https://msdn.microsoft.com/en-us/library/windows/desktop/aa363368%28v=vs.85%29.aspx
[19] https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588%28v=vs.85%29.aspx
[20] https://blogs.msdn.microsoft.com/vcblog/2016/02/08/whats-inside-a-pdb-file/
[21] http://www.wintellect.com/devcenter/jrobbins/pdb-files-what-every-developer-must-know
[22] http://www.nynaeve.net/?p=91
[23] http://blogs.technet.com/b/askperf/archive/2008/04/11/an-introduction-to-pool-tags.aspx
[24] http://blogs.technet.com/b/yongrhee/archive/2009/06/24/pool-tag-list.aspx
[25] https://github.com/google/rekall/blob/master/rekall-core/rekall/plugins/windows/pool.py
[26] https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/pooltracker.py
[27] https://msdn.microsoft.com/en-us/library/windows/hardware/ff565421%28v=vs.85%29.aspx
[28] https://www.sysnative.com/forums/bsod-kernel-dump-analysis-debugging-information/269-fun-mdls.html
[29] https://msdn.microsoft.com/en-us/library/windows/hardware/ff565421%28v=vs.85%29.aspx
[30] https://en.wikipedia.org/wiki/Processor_Control_Region
[31] https://msdn.microsoft.com/en-us/library/ms810627.aspx
[32] http://www.schatzforensic.com.au/insideout/2010/07/finding-object-roots-in-vista-kpcr/
[33] http://scudette.blogspot.ch/2012/10/finding-kpcr-in-memory-images.html
[34] http://www.volatilityfoundation.org/
[35] https://github.com/volatilityfoundation/volatility
[36] http://www.sleuthkit.org/
[37] http://www.rekall-forensic.com/
[38] http://digital-forensics.sans.org/community/downloads
[39] http://www.forensicswiki.org/wiki/Tools:Memory_Imaging
[40] http://www.moonsols.com
[41] https://zeltser.com/memory-acquisition-with-dumpit-for-dfir-2/
[42] http://www.rekall-forensic.com/docs/Tools/pmem.html
[43] https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2003941
[44] https://github.com/volatilityfoundation/volatility/wiki/VMware-Snapshot-File
[45] http://www.rekall-forensic.com/posts/2014-10-03-vms.html

[46] https://en.wikipedia.org/wiki/Struct_%28C_programming_language%29

[47] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740673%28v=vs.85%29.aspx

[48] http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

[49] http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#socket

[50] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740506%28v=vs.85%29.aspx

[51] https://msdn.microsoft.com/en-us/library/windows/hardware/ff570822%28v=vs.85%29.aspx

[52] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740496%28v=vs.85%29.aspx

[53] https://msdn.microsoft.com/en-us/library/windows/desktop/aa814468%28v=vs.85%29.aspx

[54] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740496%28v=vs.85%29.aspx

[55] http://lxr.free-electrons.com/source/include/net/inet_sock.h

[56] http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structinet__sock.html

[57] http://lxr.free-electrons.com/source/include/net/sock.h#L148

[58] http://lxr.free-electrons.com/source/include/net/sock.h#L306

[59] http://stackoverflow.com/questions/2305465/inet-socket-and-socket

[60] https://msdn.microsoft.com/en-us/library/windows/hardware/ff571086%28v=vs.85%29.aspx

[61] https://msdn.microsoft.com/en-us/library/windows/hardware/ff568373%28v=vs.85%29.aspx

[62] http://lxr.free-electrons.com/source/include/linux/skbuff.h#L626

[63] http://www.linuxfoundation.org/collaborate/workgroups/networking/sk_buff

[64] http://flylib.com/books/en/3.475.1.29/1/

[65] http://lxr.free-electrons.com/source/include/linux/skbuff.h#L279

[66] http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/skbuff_8h-source.html

[67] https://msdn.microsoft.com/en-us/library/windows/hardware/ff568355%28v=vs.85%29.aspx

[68] https://msdn.microsoft.com/en-us/library/windows/hardware/ff568376%28v=vs.85%29.aspx

[69] https://msdn.microsoft.com/en-us/library/windows/hardware/ff568728%28v=vs.85%29.aspx

[70] https://msdn.microsoft.com/en-us/library/windows/hardware/ff568388%28v=vs.85%29.aspx

[71] http://vger.kernel.org/~davem/tcp_output.html

[72] http://www.haifux.org/lectures/217/netLec5.pdf

[73] http://osxr.org:8080/linux/source/include/linux/skbuff.h

[74] http://osxr.org:8080/linux/source/include/net/sock.h

[75] http://vger.kernel.org/~davem/tcp_output.html

[76] https://msdn.microsoft.com/en-us/library/windows/hardware/ff563598%28v=vs.85%29.aspx

[77] https://msdn.microsoft.com/en-us/library/windows/hardware/ff563668%28v=vs.85%29.aspx

[78] http://codemachine.com/article_ndis6nbls.html

[79] https://en.wikipedia.org/wiki/Miniport

[80] https://msdn.microsoft.com/en-us/library/windows/hardware/hh439643%28v=vs.85%29.aspx

[81] https://msdn.microsoft.com/en-us/library/windows/hardware/ff570753%28v=vs.85%29.aspx

[82] https://msdn.microsoft.com/en-us/library/windows/hardware/ff570449%28v=vs.85%29.aspx

[83] https://msdn.microsoft.com/en-us/library/windows/hardware/ff568388%28v=vs.85%29.aspx

[84] https://msdn.microsoft.com/en-us/library/windows/hardware/ff565969%28v=vs.85%29.aspx

[85] https://msdn.microsoft.com/en-us/library/windows/hardware/ff565920%28v=vs.85%29.aspx

[86] http://mnin.blogspot.ch/2011/03/volatilitys-new-netscan-module.html

[87] http://blogs.technet.com/b/yongrhee/archive/2009/06/24/pool-tag-list.aspx

[88] https://msdn.microsoft.com/en-us/library/bb408406%28v=vs.85%29.aspx

[89] https://sourceforge.net/p/net-tools/code/ci/master/tree/netstat.c

[90] https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/netscan.py

[91] https://www.youtube.com/watch?v=DTGLmuEYHBw

[92] https://github.com/volatilityfoundation/volatility/wiki/Command%20Reference#connscan

[93] https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/connscan.py

[94] https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/linux/netscan.py

[95] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740496%28v=vs.85%29.aspx

[96] http://www.rekall-forensic.com/docs/Manual/Plugins/Linux/LinYaraScan.html

[97] http://sami.on.eniten.com/hex2ip/?

[98] http://stackoverflow.com/questions/11581914/converting-ip-address-to-hex

[99] http://lists.volatilesystems.com/pipermail/vol-users/2010-January/000150.html

[100] http://wiki.pinguino.cc/index.php/Data_types

[101] https://developer.mbed.org/handbook/C-Data-Types

[102] https://msdn.microsoft.com/en-us/library/windows/desktop/bb968799(v=vs.85).aspx
[103] http%3A%2F%2Fwww.moonsols.com%2Fwp-content%2Fplugins%2Fdownload-monitor%2Fdownload.php%3Fid%3D7&usg=AFQjCNEzuO4RC0zecDqQZO9DTVHIXzFOmw&cad=rja
[104] http://www.rekall-forensic.com/docs/Tools/
[105] https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2003941
[106] https://github.com/d3sre/Understanding_the_NDIS_6_stack/blob/master/PooltaglistToPooltrackerMapping.awk
[107] http://www.tenforums.com/windows-10-news/17993-windows-10-memory-compression.html

Appendices

Appendix A: Glossary

The following terms and expressions are further explained.

ID	Term	Description
G1	Spinlock	A lock which causes a thread that tries to access it, to wait in a loop while checking the availability of the lock repeatedly. More information can be found here: https://en.wikipedia.org/wiki/Spinlock
G2	Carving	Process of re-assembling computer files from fragments when no file system metadata is available. More information can be found here: http://forensicswiki.org/wiki/File_Carving
G3	Yarascan	Scanning for YARA formatted allocation. YARA is a description language for malware families based on textual or binary patterns. More information can be found here: https://plusvic.github.io/yara/
G4	Lookaside Lists	Pre-allocated buffers to avoid the HEAP from taking the global lock and running lock-free. More information can be found here: http://stackoverflow.com/questions/21491625/lookaside-lists-vs-low-fragmentation-heap

Table 7: Glossary

Appendix B: Directories

Bibliography

The following books and papers were the core foundation this thesis was built on:

No	Reference
B1	Michael Hale Ligh, Andrew Case, Jamie Levy, AAron Walters: *The Art of Memory Forensics*. Wiley-publishing, 2014 - ISBN: 978-1-118-82509-9
B2	Lianhai Wang, Lijuan Xu, und Shuhui Zhang: *Network Connections Information Extraction of 64-Bit Windows 7 Memory Images*, 2010 http://link.springer.com/chapter/10.1007%2F978-3-642-23602-0_8#page-1
B3	Ruichao Zhang, Lianhai Wang, Shuhui Zhang: *Windows Memory Analysis Based on KPCR*, 2009 https://www.researchgate.net/publication/220793628_Windows_memory_analysis_based_on_KPCR

B4	Michael Cohen: *Characterization of the windows kernel version variability for accurate memory analysis*, 2015 http://www.sciencedirect.com/science/article/pii/S1742287615000109
B5	Michael Cohen: Forensic Analysis of Windows User space Applications through Heap allocations, 2015 http://www.rekall-forensic.com/docs/References/Papers/p1138-cohen.pdf

Table 8: Bibliography

List of Figures

List of Tables

Appendix C: Struct Headers

The net_device Struct Header

As mentioned in the thesis it would take too much space, the referenced header of the `net_device` struct options can be found here. It's relevant for understanding members of the struct and its possible tasks. The source is:

http://lxr.free-electrons.com/source/include/linux/netdevice.h#L1560

```
/**
 *      struct net_device - The DEVICE structure.
 *              Actually, this whole structure is a big mistake.  It mixes I/O
 *              data with strictly "high-level" data, and it has to know about
 *              almost every data structure used in the INET module.
 *
 *      @name:  This is the first field of the "visible" part of this structure
 *              (i.e. as seen by users in the "Space.c" file).  It is the name
 *              of the interface.
 *
 *      @name_hlist:    Device name hash chain, please keep it close to name[]
 *      @ifalias:       SNMP alias
 *      @mem_end:       Shared memory end
 *      @mem_start:     Shared memory start
 *      @base_addr:     Device I/O address
 *      @irq:           Device IRQ number
 *
 *      @carrier_changes:       Stats to monitor carrier on<->off transitions
 *
 *      @state:         Generic network queuing layer state, see netdev_state_t
 *      @dev_list:      The global list of network devices
 *      @napi_list:     List entry, that is used for polling napi devices
 *      @unreg_list:    List entry, that is used, when we are unregistering the
 *                      device, see the function unregister_netdev
 *      @close_list:    List entry, that is used, when we are closing the device
 *
 *      @adj_list:      Directly linked devices, like slaves for bonding
 *      @all_adj_list:  All linked devices, *including* neighbours
 *      @features:      Currently active device features
 *      @hw_features:   User-changeable features
 *
 *      @wanted_features:       User-requested features
 *      @vlan_features:         Mask of features inheritable by VLAN devices
 *
 *      @hw_enc_features:       Mask of features inherited by encapsulating devices
 *                              This field indicates what encapsulation
 *                              offloads the hardware is capable of doing,
 *                              and drivers will need to set them appropriately.
 *
 *      @mpls_features: Mask of features inheritable by MPLS
 *
 *      @ifindex:       interface index
 *      @group:         The group, that the device belongs to
 *
```

```
 *	@stats:		Statistics struct, which was left as a legacy, use
 *			rtnl_link_stats64 instead
 *
 *	@rx_dropped:	Dropped packets by core network,
 *			do not use this in drivers
 *	@tx_dropped:	Dropped packets by core network,
 *			do not use this in drivers
 *
 *	@wireless_handlers:	List of functions to handle Wireless Extensions,
 *				instead of ioctl,
 *				see <net/iw_handler.h> for details.
 *	@wireless_data: Instance data managed by the core of wireless extensions
 *
 *	@netdev_ops:	Includes several pointers to callbacks,
 *			if one wants to override the ndo_*() functions
 *	@ethtool_ops:	Management operations
 *	@header_ops:	Includes callbacks for creating,parsing,caching,etc
 *			of Layer 2 headers.
 *
 *	@flags:		Interface flags (a la BSD)
 *	@priv_flags:	Like 'flags' but invisible to userspace,
 *			see if.h for the definitions
 *	@gflags:	Global flags ( kept as legacy )
 *	@padded:	How much padding added by alloc_netdev()
 *	@operstate:	RFC2863 operstate
 *	@link_mode:	Mapping policy to operstate
 *	@if_port:	Selectable AUI, TP, ...
 *	@dma:		DMA channel
 *	@mtu:		Interface MTU value
 *	@type:		Interface hardware type
 *	@hard_header_len: Hardware header length, which means that this is the
 *			  minimum size of a packet.
 *
 *	@needed_headroom: Extra headroom the hardware may need, but not in all
 *			  cases can this be guaranteed
 *	@needed_tailroom: Extra tailroom the hardware may need, but not in all
 *			  cases can this be guaranteed. Some cases also use
 *			  LL_MAX_HEADER instead to allocate the skb
 *
 *	interface address info:
 *
 *	@perm_addr:		Permanent hw address
 *	@addr_assign_type:	Hw address assignment type
 *	@addr_len:		Hardware address length
 *	@neigh_priv_len;	Used in neigh_alloc(),
 *				initialized only in atm/clip.c
 *	@dev_id:		Used to differentiate devices that share
 *				the same link layer address
 *	@dev_port:		Used to differentiate devices that share
 *				the same function
 *	@addr_list_lock:	XXX: need comments on this one
 *	@uc_promisc:		Counter, that indicates, that promiscuous mode
 *				has been enabled due to the need to listen to
 *				additional unicast addresses in a device that
 *				does not implement ndo_set_rx_mode()
 *	@uc:			unicast mac addresses
 *	@mc:			multicast mac addresses
 *	@dev_addrs:		list of device hw addresses
 *	@queues_kset:		Group of all Kobjects in the Tx and RX queues
 *	@promiscuity:		Number of times, the NIC is told to work in
 *				Promiscuous mode, if it becomes 0 the NIC will
 *				exit from working in Promiscuous mode
 *	@allmulti:		Counter, enables or disables allmulticast mode
 *
 *	@vlan_info:	VLAN info
```

```
 *      @dsa_ptr:       dsa specific data
 *      @tipc_ptr:      TIPC specific data
 *      @atalk_ptr:     AppleTalk link
 *      @ip_ptr:        Ipv4 specific data
 *      @dn_ptr:        DECnet specific data
 *      @ip6_ptr:       Ipv6 specific data
 *      @ax25_ptr:      AX.25 specific data
 *      @ieee80211_ptr: IEEE 802.11 specific data, assign before registering
 *
 *      @last_rx:       Time of last Rx
 *      @dev_addr:      Hw address (before bcast,
 *                      because most packets are unicast)
 *
 *      @_rx:                   Array of RX queues
 *      @num_rx_queues:         Number of RX queues
 *                              allocated at register_netdev() time
 *      @real_num_rx_queues:    Number of RX queues currently active in device
 *
 *      @rx_handler:            handler for received packets
 *      @rx_handler_data:       XXX: need comments on this one
 *      @ingress_queue:         XXX: need comments on this one
 *      @broadcast:             hw bcast address
 *
 *      @rx_cpu_rmap:   CPU reverse-mapping for RX completion interrupts,
 *                      indexed by RX queue number. Assigned by driver.
 *                      This must only be set if the ndo_rx_flow_steer
 *                      operation is defined
 *      @index_hlist:           Device index hash chain
 *
 *      @_tx:                   Array of TX queues
 *      @num_tx_queues:         Number of TX queues allocated at alloc_netdev_mq() time
 *      @real_num_tx_queues:    Number of TX queues currently active in device
 *      @qdisc:                 Root qdisc from userspace point of view
 *      @tx_queue_len:          Max frames per queue allowed
 *      @tx_global_lock:        XXX: need comments on this one
 *
 *      @xps_maps:      XXX: need comments on this one
 *
 *      @offload_fwd_mark:      Offload device fwding mark
 *
 *      @trans_start:           Time (in jiffies) of last Tx
 *      @watchdog_timeo:        Represents the timeout that is used by
 *                              the watchdog ( see dev_watchdog() )
 *      @watchdog_timer:        List of timers
 *
 *      @pcpu_refcnt:           Number of references to this device
 *      @todo_list:             Delayed register/unregister
 *      @link_watch_list:       XXX: need comments on this one
 *
 *      @reg_state:             Register/unregister state machine
 *      @dismantle:             Device is going to be freed
 *      @rtnl_link_state:       This enum represents the phases of creating
 *                              a new link
 *
 *      @destructor:            Called from unregister,
 *                              can be used to call free_netdev
 *      @npinfo:                XXX: need comments on this one
 *      @nd_net:                Network namespace this network device is inside
 *
 *      @ml_priv:       Mid-layer private
 *      @lstats:        Loopback statistics
 *      @tstats:        Tunnel statistics
 *      @dstats:        Dummy statistics
 *      @vstats:        Virtual 80thernet statistics
```

```
 *
 *      @garp_port:     GARP
 *      @mrp_port:      MRP
 *
 *      @dev:           Class/net/name entry
 *      @sysfs_groups:  Space for optional device, statistics and wireless
 *                      sysfs groups
 *
 *      @sysfs_rx_queue_group:  Space for optional per-rx queue attributes
 *      @rtnl_link_ops: Rtnl_link_ops
 *
 *      @gso_max_size:  Maximum size of generic segmentation offload
 *      @gso_max_segs:  Maximum number of segments that can be passed to the
 *                      NIC for GSO
 *      @gso_min_segs:  Minimum number of segments that can be passed to the
 *                      NIC for GSO
 *
 *      @dcbnl_ops:     Data Center Bridging netlink ops
 *      @num_tc:        Number of traffic classes in the net device
 *      @tc_to_txq:     XXX: need comments on this one
 *      @prio_tc_map    XXX: need comments on this one
 *
 *      @fcoe_ddp_xid:  Max exchange id for FcoE LRO by ddp
 *
 *      @priomap:       XXX: need comments on this one
 *      @phydev:        Physical device may attach itself
 *                      for hardware timestamping
 *
 *      @qdisc_tx_busylock:     XXX: need comments on this one
 *
 *      @proto_down:    protocol port state information can be sent to the
 *                      switch driver and used to set the phys state of the
 *                      switch port.
 *
 *      FIXME: cleanup struct net_device such that network protocol info
 *      moves out.
 */
```

The sk_buff Struct Header

For reference and as in the thesis it would take too much space and influence the read flow, the referenced header of the sk_buff struct options can be found here. It's relevant for understanding members of the struct and its possible tasks. The source is:

http://lxr.free-electrons.com/source/include/linux/skbuff.h#L558

```
/**
 *      struct sk_buff - socket buffer
 *      @next: Next buffer in list
 *      @prev: Previous buffer in list
 *      @tstamp: Time we arrived/left
 *      @rbnode: RB tree node, alternative to next/prev for netem/tcp
 *      @sk: Socket we are owned by
 *      @dev: Device we arrived on/are leaving by
 *      @cb: Control buffer. Free for use by every layer. Put private vars here
 *      @_skb_refdst: destination entry (with norefcount bit)
 *      @sp: the security path, used for xfrm
 *      @len: Length of actual data
 *      @data_len: Data length
 *      @mac_len: Length of link layer header
 *      @hdr_len: writable header length of cloned skb
 *      @csum: Checksum (must include start/offset pair)
 *      @csum_start: Offset from skb->head where checksumming should start
```

```
 *      @csum_offset: Offset from csum_start where checksum should be stored
 *      @priority: Packet queueing priority
 *      @ignore_df: allow local fragmentation
 *      @cloned: Head may be cloned (check refcnt to be sure)
 *      @ip_summed: Driver fed us an IP checksum
 *      @nohdr: Payload reference only, must not modify header
 *      @nfctinfo: Relationship of this skb to the connection
 *      @pkt_type: Packet class
 *      @fclone: skbuff clone status
 *      @ipvs_property: skbuff is owned by ipvs
 *      @peeked: this packet has been seen already, so stats have been
 *              done for it, don't do them again
 *      @nf_trace: netfilter packet trace flag
 *      @protocol: Packet protocol from driver
 *      @destructor: Destruct function
 *      @nfct: Associated connection, if any
 *      @nf_bridge: Saved data about a bridged frame - see br_netfilter.c
 *      @skb_iif: ifindex of device we arrived on
 *      @tc_index: Traffic control index
 *      @tc_verd: traffic control verdict
 *      @hash: the packet hash
 *      @queue_mapping: Queue mapping for multiqueue devices
 *      @xmit_more: More SKBs are pending for this queue
 *      @ndisc_nodetype: router type (from link layer)
 *      @ooo_okay: allow the mapping of a socket to a queue to be changed
 *      @l4_hash: indicate hash is a canonical 4-tuple hash over transport
 *              ports.
 *      @sw_hash: indicates hash was computed in software stack
 *      @wifi_acked_valid: wifi_acked was set
 *      @wifi_acked: whether frame was acked on wifi or not
 *      @no_fcs:  Request NIC to treat last 4 bytes as Ethernet FCS
 *      @napi_id: id of the NAPI struct this skb came from
 *      @secmark: security marking
 *      @offload_fwd_mark: fwding offload mark
 *      @mark: Generic packet mark
 *      @vlan_proto: vlan encapsulation protocol
 *      @vlan_tci: vlan tag control information
 *      @inner_protocol: Protocol (encapsulation)
 *      @inner_transport_header: Inner transport layer header (encapsulation)
 *      @inner_network_header: Network layer header (encapsulation)
 *      @inner_mac_header: Link layer header (encapsulation)
 *      @transport_header: Transport layer header
 *      @network_header: Network layer header
 *      @mac_header: Link layer header
 *      @tail: Tail pointer
 *      @end: End pointer
 *      @head: Head of buffer
 *      @data: Data head pointer
 *      @truesize: Buffer size
 *      @users: User count - see {datagram,tcp}.c
 */
```

The sock Struct Header

For reference and as in the thesis it would take too much space and influence the read flow, the referenced header of the sock struct options can be found here. It's relevant for understanding members of the struct and its possible tasks. The source is:

http://lxr.free-electrons.com/source/include/net/sock.h#L306

```
/**
  *     struct sock – network layer representation of sockets
```

```
 *	@__sk_common: shared layout with inet_timewait_sock
 *	@sk_shutdown: mask of %SEND_SHUTDOWN and/or %RCV_SHUTDOWN
 *	@sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
 *	@sk_lock:	synchronizer
 *	@sk_rcvbuf: size of receive buffer in bytes
 *	@sk_wq: sock wait queue and async head
 *	@sk_rx_dst: receive input route used by early demux
 *	@sk_dst_cache: destination cache
 *	@sk_policy: flow policy
 *	@sk_receive_queue: incoming packets
 *	@sk_wmem_alloc: transmit queue bytes committed
 *	@sk_write_queue: Packet sending queue
 *	@sk_omem_alloc: "o" is "option" or "other"
 *	@sk_wmem_queued: persistent queue size
 *	@sk_forward_alloc: space allocated forward
 *	@sk_napi_id: id of the last napi context to receive data for sk
 *	@sk_ll_usec: usecs to busypoll when there is no data
 *	@sk_allocation: allocation mode
 *	@sk_pacing_rate: Pacing rate (if supported by transport/packet scheduler)
 *	@sk_max_pacing_rate: Maximum pacing rate (%SO_MAX_PACING_RATE)
 *	@sk_sndbuf: size of send buffer in bytes
 *	@sk_no_check_tx: %SO_NO_CHECK setting, set checksum in TX packets
 *	@sk_no_check_rx: allow zero checksum in RX packets
 *	@sk_route_caps: route capabilities (e.g. %NETIF_F_TSO)
 *	@sk_route_nocaps: forbidden route capabilities (e.g NETIF_F_GSO_MASK)
 *	@sk_gso_type: GSO type (e.g. %SKB_GSO_TCPV4)
 *	@sk_gso_max_size: Maximum GSO segment size to build
 *	@sk_gso_max_segs: Maximum number of GSO segments
 *	@sk_lingertime: %SO_LINGER l_linger setting
 *	@sk_backlog: always used with the per-socket spinlock held
 *	@sk_callback_lock: used with the callbacks in the end of this struct
 *	@sk_error_queue: rarely used
 *	@sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
 *			  IPV6_ADDRFORM for instance)
 *	@sk_err: last error
 *	@sk_err_soft: errors that don't cause failure but are the cause of a
 *		      persistent failure not just 'timed out'
 *	@sk_drops: raw/udp drops counter
 *	@sk_ack_backlog: current listen backlog
 *	@sk_max_ack_backlog: listen backlog set in listen()
 *	@sk_priority: %SO_PRIORITY setting
 *	@sk_type: socket type (%SOCK_STREAM, etc)
 *	@sk_protocol: which protocol this socket belongs in this network family
 *	@sk_peer_pid: &struct pid for this socket's peer
 *	@sk_peer_cred: %SO_PEERCRED setting
 *	@sk_rcvlowat: %SO_RCVLOWAT setting
 *	@sk_rcvtimeo: %SO_RCVTIMEO setting
 *	@sk_sndtimeo: %SO_SNDTIMEO setting
 *	@sk_txhash: computed flow hash for use on transmit
 *	@sk_filter: socket filtering instructions
 *	@sk_timer: sock cleanup timer
 *	@sk_stamp: time stamp of last packet received
 *	@sk_tsflags: SO_TIMESTAMPING socket options
 *	@sk_tskey: counter to disambiguate concurrent tstamp requests
 *	@sk_socket: Identd and reporting IO signals
 *	@sk_user_data: RPC layer private data
 *	@sk_frag: cached page frag
 *	@sk_peek_off: current peek_offset value
 *	@sk_send_head: front of stuff to transmit
 *	@sk_security: used by security modules
 *	@sk_mark: generic packet mark
 *	@sk_cgrp_data: cgroup data for this cgroup
 *	@sk_memcg: this socket's memory cgroup association
 *	@sk_write_pending: a write to stream socket waits to start
 *	@sk_state_change: callback to indicate change in the state of the sock
```

```
 *      @sk_data_ready: callback to indicate there is data to be processed
 *      @sk_write_space: callback to indicate there is bf sending space available
 *      @sk_error_report: callback to indicate errors (e.g. %MSG_ERRQUEUE)
 *      @sk_backlog_rcv: callback to process the backlog
 *      @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
 *      @sk_reuseport_cb: reuseport group container
 */
```

The sock_common Struct Header

For reference and as in the thesis it would take too much space and influence the read flow, the referenced header of the `sock_common` struct options can be found here. It's relevant for understanding members of the struct and its possible tasks. The source is:

http://lxr.free-electrons.com/source/include/net/sock.h#L148

```
/**
 *      struct sock_common - minimal network layer representation of sockets
 *      @skc_daddr: Foreign Ipv4 addr
 *      @skc_rcv_saddr: Bound local Ipv4 addr
 *      @skc_hash: hash value used with various protocol lookup tables
 *      @skc_u16hashes: two u16 hash values used by UDP lookup tables
 *      @skc_dport: placeholder for inet_dport/tw_dport
 *      @skc_num: placeholder for inet_num/tw_num
 *      @skc_family: network address family
 *      @skc_state: Connection state
 *      @skc_reuse: %SO_REUSEADDR setting
 *      @skc_reuseport: %SO_REUSEPORT setting
 *      @skc_bound_dev_if: bound device index if != 0
 *      @skc_bind_node: bind hash linkage for various protocol lookup tables
 *      @skc_portaddr_node: second hash linkage for UDP/UDP-Lite protocol
 *      @skc_prot: protocol handlers inside a network family
 *      @skc_net: reference to the network namespace of this socket
 *      @skc_node: main hash linkage for various protocol lookup tables
 *      @skc_nulls_node: main hash linkage for TCP/UDP/UDP-Lite protocol
 *      @skc_tx_queue_mapping: tx queue number for this connection
 *      @skc_flags: place holder for sk_flags
 *              %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPALIVE,
 *              %SO_OOBINLINE settings, %SO_TIMESTAMPING settings
 *      @skc_incoming_cpu: record/match cpu processing incoming packets
 *      @skc_refcnt: reference count
 *
 *      This is the minimal network layer representation of sockets, the header
 *      for struct sock and struct inet_timewait_sock.
 */
```

The inet_sock Struct Header

For reference and as in the thesis it would take too much space and influence the read flow, the referenced header of the inet_sock struct options can be found here. It's relevant for understanding members of the struct and its possible tasks. The source is:

http://lxr.free-electrons.com/source/include/net/inet_sock.h#L172

```
/** struct inet_sock - representation of INET sockets
 *
 * @sk - ancestor class
 * @pinet6 - pointer to Ipv6 control block
 * @inet_daddr - Foreign Ipv4 addr
 * @inet_rcv_saddr - Bound local Ipv4 addr
 * @inet_dport - Destination port
```

```
 * @inet_num - Local port
 * @inet_saddr - Sending source
 * @uc_ttl - Unicast TTL
 * @inet_sport - Source port
 * @inet_id - ID counter for DF pkts
 * @tos - TOS
 * @mc_ttl - Multicasting TTL
 * @is_icsk - is this an inet_connection_sock?
 * @uc_index - Unicast outgoing device index
 * @mc_index - Multicast device index
 * @mc_list - Group array
 * @cork - info to build ip hdr on each ip frag while socket is corked
 */
```

Appendix D: List of `tcpip.sys` Pool Tags

As the state of 22.04.2016, from https://blogs.technet.microsoft.com/yongrhee/2009/06/23/pool-tag-list/. Originally published on June 23, 2009.

Pool tag	Binary Name	Description
AleD	tcpip.sys	ALE remote endpoint
Ala4	tcpip.sys	ALE remote endpoint IPv4 address
Ala6	tcpip.sys	ALE remote endpoint IPv6 address
Alei	tcpip.sys	ALE arrival/nexthop interface cache
AleU	tcpip.sys	ALE pend context
AleE	tcpip.sys	ALE endpoint context
AlLl	tcpip.sys	ALE remote endpoint LRU
AlCi	tcpip.sys	ALE credential info
AlSP	tcpip.sys	ALE secure socket policy
AlPU	tcpip.sys	ALE secure socket policy update
AlPi	tcpip.sys	ALE peer info
AlP4	tcpip.sys	ALE peer IPv4 address
AlP6	tcpip.sys	ALE peer IPv6 address
AlPT	tcpip.sys	ALE peer target
Alep	tcpip.sys	ALE process info
AleS	tcpip.sys	ALE token info
AleP	tcpip.sys	ALE process image path
AleK	tcpip.sys	ALE audit
AleA	tcpip.sys	ALE connection abort context
AlDN	tcpip.sys	ALE endpoint delete notify
AleW	tcpip.sys	ALE enum filter array
AleN	tcpip.sys	ALE notify context
AlSs	tcpip.sys	ALE socket security context
AlPF	tcpip.sys	ALE policy filters
AleL	tcpip.sys	ALE LRU
AleI	tcpip.sys	ALE token ID
AlP5	tcpip.sys	ALE 5 - tuple state
AlE5	tcpip.sys	ALE 5 - tuple temp entry
Aric	tcpip.sys	ALE route inspection context

Pool tag	Binary Name	Description
Adnc	tcpip.sys	ALE endpoint deactivation notification context
Acrc	tcpip.sys	ALE connect request inspection context
Acrl	tcpip.sys	ALE connect redirect layer data
Abrc	tcpip.sys	ALE bind request inspection context
Abrl	tcpip.sys	ALE bind redirect layer data
FlmC	tcpip.sys	Framing Layer Client Contexts
FlmP	tcpip.sys	Framing Layer Provider Contexts
Flng	tcpip.sys	Framing Layer Generic Buffers (Tunnel/Port change notifications, ACLs)
FlpC	tcpip.sys	Framing Layer Client Contexts
FlpI	tcpip.sys	Framing Layer Interfaces
FlpM	tcpip.sys	Framing Layer Multicast Groups
FlpS	tcpip.sys	Framing Layer Serialized Requests
Fl6D	tcpip.sys	FL6t DataLink Addresses
Fl4D	tcpip.sys	FL4t DataLink Addresses
FlSB	tcpip.sys	Framing Layer Stack Block
FwSD	tcpip.sys	WFP security descriptor
Ic4c	tcpip.sys	ICMP IPv4 Control data
Ic4h	tcpip.sys	ICMP IPv4 Headers
Ic6c	tcpip.sys	ICMP IPv6 Control data
Ic6h	tcpip.sys	ICMP IPv6 Headers
IBbf	tcpip.sys	IP BVT Buffers
InAD	tcpip.sys	Inet Ancillary Data
IneI	tcpip.sys	Inet Inspects
InF0	tcpip.sys	Inet Generic Fixed Size Block pool 0
InF1	tcpip.sys	Inet Generic Fixed Size Block pool 1
InF2	tcpip.sys	Inet Generic Fixed Size Block pool 2
InIS	tcpip.sys	Inet Inspect Streams
InNP	tcpip.sys	Inet Nsi Providers
InPA	tcpip.sys	Inet Port Assignment Arrays
InPa	tcpip.sys	Inet Port Assignments
InPE	tcpip.sys	Inet Port Exclusions
InPP	tcpip.sys	Inet Port pool
InSB	tcpip.sys	Inet stack block
InSC	tcpip.sys	Inet Queued Send Contexts
Ipas	tcpip.sys	IP Buffers for Address Sort
IPbw	tcpip.sys	IP Path Bandwidth information
IPdc	tcpip.sys	IP Destination Cache
IPfg	tcpip.sys	IP Fragment Groups
IPfp	tcpip.sys	IP PreValidated Receives
IPif	tcpip.sys	IP Interfaces
IPlo	tcpip.sys	IP Loopback buffers
IPmf	tcpip.sys	IP Multicast Forwarding Entry pool
IPpo	tcpip.sys	IP Offload buffers
IPpa	tcpip.sys	IP Path information
IPrq	tcpip.sys	IP Request Control data

Pool tag	Binary Name	Description
IPss	tcpip.sys	IP Session State
IPsi	tcpip.sys	IP SubInterfaces
Ipng	tcpip.sys	IP Generic buffers (Address, Interface, Packetize, Route allocations)
IpOl	tcpip.sys	IP Offload Log data
Ippp	tcpip.sys	IP Prefix Policy information
IPre	tcpip.sys	IP Reassembly buffers
Iptt	tcpip.sys	IP Timer Tables
Iptc	tcpip.sys	IP Transaction Context information
Ipur	tcpip.sys	IP Unicast Routes
Ipwi	tcpip.sys	IP Work Item allocations
I4ai	tcpip.sys	IPv4 Local Address Identifiers
I4ba	tcpip.sys	IPv4 Local Broadcast Addresses
I4bf	tcpip.sys	IPv4 Generic Buffers (Source Address List allocations)
I4e	tcpip.sys	IPv4 Echo data
I4ma	tcpip.sys	IPv4 Local Multicast Addresses
I4nb	tcpip.sys	IPv4 Neighbors
I4rd	tcpip.sys	IPv4 Receive Datagrams Arguments
I4ua	tcpip.sys	IPv4 Local Unicast Addresses
I6ai	tcpip.sys	IPv6 Local Address Identifiers
I6aa	tcpip.sys	IPv6 Local Anycast Addresses
I6bf	tcpip.sys	IPv6 Generic Buffers (Source Address List allocations)
I6e	tcpip.sys	IPv6 Echo data
I6ma	tcpip.sys	IPv6 Local Multicast Addresses
I6nb	tcpip.sys	IPv6 Neighbors
I6rd	tcpip.sys	IPv6 Receive Datagrams Arguments
I6ua	tcpip.sys	IPv6 Local Unicast Addresses
ItoM	tcpip.sys	IPsec task offload interface
ItoD	tcpip.sys	IPsec task offload delete SA
ItoS	tcpip.sys	IPsec task offload add SA
ItoC	tcpip.sys	IPsec task offload context
ItoO	tcpip.sys	IPsec task offload paramters
Itht	tcpip.sys	IPsec hashtable
ISLe	tcpip.sys	IPsec SA list entry
IsRc	tcpip.sys	IPsec rebalance context
Ikmb	tcpip.sys	IPsec key module blob
Ithp	tcpip.sys	IPsec throttle param
Ipfl	tcpip.sys	IPsec flow handle
Ipap	tcpip.sys	IPsec pend context
Inlc	tcpip.sys	IPsec NL complete context
Iprc	tcpip.sys	IPsec RPC context
Ipis	tcpip.sys	IPsec inbound sequence info
Itok	tcpip.sys	IPsec token
Iser	tcpip.sys	IPsec inbound sequence range
IKeO	tcpip.sys	IPsec key object
I4sa	tcpip.sys	IPsec SADB v4

Pool tag	Binary Name	Description
I4s6	tcpip.sys	IPsec SADB v6
IHaO	tcpip.sys	IPsec hash object
Ipnc	tcpip.sys	IPsec negotiation context
Icse	tcpip.sys	IPsec NS connection state
Itro	tcpip.sys	IPsec outbound session security context
Itri	tcpip.sys	IPsec inbound packet security context
Ituo	tcpip.sys	IPsec outbound tunnel session security context
Itui	tcpip.sys	IPsec inbound packet tunnel security context
Ipft	tcpip.sys	IPsec filter
Ipwi	tcpip.sys	IPsec work item
Idqf	tcpip.sys	IPsec DOS protection QoS flow
Idpc	tcpip.sys	IPsec DOS protection pacer create
Idst	tcpip.sys	IPsec DOS protection state entry
Ifws	tcpip.sys	IPsec forward state
LeGe	tcpip.sys	Legacy Registry Mapping Module Buffers
Net	tcpip.sys	NetIO Generic Buffers (iBFT Table allocations)
NeWQ	tcpip.sys	NetIO WorkQueue Data
Nhfs	tcpip.sys	NetIO Hash Function State Data
Navl	tcpip.sys	Network Layer AVL Tree allocations
NLbd	tcpip.sys	Network Layer Buffer Data
NLcc	tcpip.sys	Network Layer Client Contexts
NLpd	tcpip.sys	Network Layer Client Requests
NLcp	tcpip.sys	Network Layer Compartments
NLap	tcpip.sys	Network Layer Netio Helper Function allocations
NLNa	tcpip.sys	Network Layer Network Address Lists
NMRb	tcpip.sys	Network Module Registrar Bindings
NMRc	tcpip.sys	Network Module Registrar Arrays
NMRf	tcpip.sys	Network Module Registrar Filters
NMRg	tcpip.sys	Network Module Registrar Generic Buffers
NMRm	tcpip.sys	Network Module Registrar Modules
NMRn	tcpip.sys	Network Module Registrar Network Protocol Identifiers
OlmC	tcpip.sys	Offload Manager Connections
OlmI	tcpip.sys	Offload Manager Interfaces
RaDA	tcpip.sys	Raw Socket Discretionary ACLs
RaEW	tcpip.sys	Raw Socket Endpoint Work Queue Contexts
RaJP	tcpip.sys	Raw Socket Join Path Contexts
RaMI	tcpip.sys	Raw Socket Message Indication Tags
RaPM	tcpip.sys	Raw Socket Partial Memory Descriptor List Tag
RaSM	tcpip.sys	Raw Socket Send Messages Requests
RaSL	tcpip.sys	Raw Socket Send Message Lists
RawE	tcpip.sys	Raw Socket Endpoints
RawN	tcpip.sys	Raw Socket Nsi
RhHi	tcpip.sys	Reference History Pool
Rind	tcpip.sys	Raw Socket Receive Indications
TcAR	tcpip.sys	TCP Abort Requests

Pool tag	Binary Name	Description
TcBW	tcpip.sys	TCP Bandwidth Allocations
TcCM	tcpip.sys	TCP Congestion Control Manager Contexts
TcCR	tcpip.sys	TCP Connect Requests
TcCC	tcpip.sys	TCP Create And Connect Tcb Pool
TcDD	tcpip.sys	TCP Debug Delivery Buffers
TcDQ	tcpip.sys	TCP Delay Queues
TcDR	tcpip.sys	TCP Disconnect Requests
TcEW	tcpip.sys	TCP Endpoint Work Queue Contexts
TcFR	tcpip.sys	TCP FineRTT Buffers
TcHT	tcpip.sys	TCP Hash Tables
TcIn	tcpip.sys	TCP Inputs
TcLS	tcpip.sys	TCP Listener SockAddrs
TcLW	tcpip.sys	TCP Listener Work Queue Contexts
TcpA	tcpip.sys	TCP DMA buffers
TcpB	tcpip.sys	TCP Offload Blocks
TcDM	tcpip.sys	TCP Delayed Delivery Memory Descriptor Lists
TcDN	tcpip.sys	TCP Delayed Delivery Network Buffer Lists
TcpE	tcpip.sys	TCP Endpoints
TcpI	tcpip.sys	TCP ISN buffers
TcpL	tcpip.sys	TCP Listeners
TcpM	tcpip.sys	TCP Offload Miscellaneous buffers
TcpN	tcpip.sys	TCP Name Service Interfaces
TcOD	tcpip.sys	TCP Offload Devices
TcpO	tcpip.sys	TCP Offload Requests
TcpP	tcpip.sys	TCP Processor Arrays
TcPt	tcpip.sys	TCP Partitions
Tcpt	tcpip.sys	TCP Timers
TcRA	tcpip.sys	TCP Reassembly Data
TcRB	tcpip.sys	TCP Reassembly Buffers
TcRD	tcpip.sys	TCP Receive DPC Data
TcRe	tcpip.sys	TCP Recovery Buffers
TcRH	tcpip.sys	TCP Reassembly Headers
TcRL	tcpip.sys	TCP Create And Connect Tcb Rate Limit Pool
TcRR	tcpip.sys	TCP Receive Requests
TcRW	tcpip.sys	TCP Receive Window Tuning Blocks
TcSa	tcpip.sys	TCP Sack Data
TcSR	tcpip.sys	TCP Send Requests
TcST	tcpip.sys	TCP Syn TCBs
TcTW	tcpip.sys	TCP Time Wait TCBs
TcUD	tcpip.sys	TCP Urgent Delivery Buffers
TcWQ	tcpip.sys	TCP TCB Work Queue Contexts
TcWS	tcpip.sys	TCP Window Scaling Diagnostics
Tedd	tcpip.sys	TCP/IP Event Data Descriptors
TNbl	tcpip.sys	TCP Send NetBufferLists
TQoS	tcpip.sys	TL QoS Client Data

Pool tag	Binary Name	Description
Ttnc	tcpip.sys	WFP tunnel nexthop context
Tsmp	tcpip.sys	TCP Send Memory Descriptor Lists
TSNb	tcpip.sys	TCP Send NetBuffers
TTsp	tcpip.sys	TCP TCB Sends
TWTa	tcpip.sys	Echo Request Timer Table
UdAE	tcpip.sys	UDP Activate Endpoints
UdJP	tcpip.sys	UDP Join Path Contexts
UdEW	tcpip.sys	UDP Endpoint Work Queue Contexts
UdMI	tcpip.sys	UDP Message Indications
UDNb	tcpip.sys	UDP NetBuffers
UdpA	tcpip.sys	UDP Endpoints
UdpH	tcpip.sys	UDP Headers
UdPM	tcpip.sys	UDP Partial Memory Descriptor Lists
UdpN	tcpip.sys	UDP Name Service Interfaces
UdSM	tcpip.sys	UDP Send Messages Requests
UNbl	tcpip.sys	UDP NetBufferLists

Appendix E: List of `tcpip.sys` Pool Tags Mapped to the `pool_tracker` Plugin

For analysis purposes the `pool_tracker` plugin of Rekall was mapped with the pool tag list to easily filter relevant pool tags occurring in the image that are related to `tcpip.sys`. The following output was generated on a Windows 7 Professional x86 system.

Tag	NP Alloc	NP	Bytes P	Alloc	Pool tag binary	Pool tag description
AlCi	4 (4)	832	0 (0)	0	tcpip.sys	ALE credential info
AlLl	4 (4)	832	0 (0)	0	tcpip.sys	ALE remote endpoint LRU
AlP4	4 (4)	832	0 (0)	0	tcpip.sys	ALE peer IPv4 address
AlP6	4 (4)	832	0 (0)	0	tcpip.sys	ALE peer IPv6 address
AlPT	27 (0)	0	0 (0)	0	tcpip.sys	ALE peer target
AlPU	4 (4)	832	0 (0)	0	tcpip.sys	ALE secure socket policy update
AlPi	10 (10)	1792	0 (0)	0	tcpip.sys	ALE peer info
AlSP	6 (6)	1088	0 (0)	0	tcpip.sys	ALE secure socket policy
AlSs	9 (0)	0	0 (0)	0	tcpip.sys	ALE socket security context
Ala4	78 (20)	1088	0 (0)	0	tcpip.sys	ALE remote endpoint IPv4 address
Ala6	72 (20)	1216	0 (0)	0	tcpip.sys	ALE remote endpoint IPv6 address

Tag	NP Alloc	NP	Bytes P	Alloc	Pool tag binary	Pool tag description
AleD	61 (21)	7632	0 (0)	0	tcpip.sys	ALE remote endpoint
AleE	476 (444)	201472	0 (0)	0	tcpip.sys	ALE endpoint context
AleL	4 (4)	1032	0 (0)	0	tcpip.sys	ALE LRU
AleP	58 (53)	24936	0 (0)	0	tcpip.sys	ALE process image path
AleS	19 (19)	32992	0 (0)	0	tcpip.sys	ALE token info
AleU	4 (4)	832	0 (0)	0	tcpip.sys	ALE pend context
Alei	4 (4)	832	0 (0)	0	tcpip.sys	ALE arrival/nexthop interface cache
Alep	54 (49)	3712	0 (0)	0	tcpip.sys	ALE process info
Fl6D	1 (1)	4096	0 (0)	0	tcpip.sys	FL6t DataLink Addresses
FlSB	4 (4)	256	0 (0)	0	tcpip.sys	Framing Layer Stack Block
FlmC	3 (3)	288	0 (0)	0	tcpip.sys	Framing Layer Client Contexts
Flng	38 (14)	8528	0 (0)	0	tcpip.sys	Framing Layer Generic Buffers (Tunnel/Port change notifications, ACLs)
FlpC	8 (8)	256	0 (0)	0	tcpip.sys	Framing Layer Client Contexts
FlpI	96 (48)	960	0 (0)	0	tcpip.sys	Framing Layer Interfaces
FlpM	2 (2)	8192	0 (0)	0	tcpip.sys	Framing Layer Multicast Groups
FlpS	3 (3)	12288	0 (0)	0	tcpip.sys	Framing Layer Serialized Requests
FwSD	0 (0)	0	8 (0)	0	tcpip.sys	WFP security descriptor
I4ai	1 (1)	4096	0 (0)	0	tcpip.sys	IPv4 Local Address Identifiers
I4ba	1 (1)	4096	0 (0)	0	tcpip.sys	IPv4 Local Broadcast Addresses
I4e	6 (0)	0	0 (0)	0	tcpip.sys	IPv4 Echo data
I4ma	1 (1)	4096	0 (0)	0	tcpip.sys	IPv4 Local Multicast Addresses
I4nb	2 (2)	8192	0 (0)	0	tcpip.sys	IPv4 Neighbors
I4ua	1 (1)	4096	0 (0)	0	tcpip.sys	IPv4 Local Unicast Addresses
I6ai	1 (1)	4096	0 (0)	0	tcpip.sys	IPv6 Local Address Identifiers
I6ma	1 (1)	4096	0 (0)	0	tcpip.sys	IPv6 Local Multicast Addresses

Tag	NP Alloc	NP	Bytes P	Alloc	Pool tag binary	Pool tag description
I6nb	3 (3)	12288	0 (0)	0	tcpip.sys	IPv6 Neighbors
I6ua	1 (1)	4096	0 (0)	0	tcpip.sys	IPv6 Local Unicast Addresses
IPdc	2 (2)	2064	0 (0)	0	tcpip.sys	IP Destination Cache
IPif	20 (16)	5376	0 (0)	0	tcpip.sys	IP Interfaces
IPmf	2 (2)	80	0 (0)	0	tcpip.sys	IP Multicast Forwarding Entry pool
IPpa	8 (7)	28672	0 (0)	0	tcpip.sys	IP Path information
IPrq	6 (6)	27296	0 (0)	0	tcpip.sys	IP Request Control data
IPsi	20 (16)	1280	0 (0)	0	tcpip.sys	IP SubInterfaces
IPss	2 (2)	8192	0 (0)	0	tcpip.sys	IP Session State
InAD	505 (0)	0	0 (0)	0	tcpip.sys	Inet Ancillary Data
InF0	1 (1)	4096	0 (0)	0	tcpip.sys	Inet Generic Fixed Size Block pool 0
InF1	2 (2)	8192	0 (0)	0	tcpip.sys	Inet Generic Fixed Size Block pool 1
InF2	1 (1)	4096	0 (0)	0	tcpip.sys	Inet Generic Fixed Size Block pool 2
InNP	3 (3)	96	0 (0)	0	tcpip.sys	Inet Nsi Providers
InPA	512 (512)	16384	0 (0)	0	tcpip.sys	Inet Port Assignment Arrays
InPE	17 (1)	40	0 (0)	0	tcpip.sys	Inet Port Exclusions
InPP	2 (2)	20176	0 (0)	0	tcpip.sys	Inet Port pool
InPa	184 (16)	57504	0 (0)	0	tcpip.sys	Inet Port Assignments
InSB	1 (1)	64	0 (0)	0	tcpip.sys	Inet stack block
Ipas	1428 (0)	0	0 (0)	0	tcpip.sys	IP Buffers for Address Sort
Ipng	7456 (8)	624	0 (0)	0	tcpip.sys	IP Generic buffers (Address, Interface, Packetize, Route allocations)
Iptt	66 (54)	10320	0 (0)	0	tcpip.sys	IP Timer Tables
Ipur	2 (2)	8192	0 (0)	0	tcpip.sys	IP Unicast Routes
Itht	6 (6)	144	0 (0)	0	tcpip.sys	IPsec hashtable
ItoM	10 (5)	720	0 (0)	0	tcpip.sys	IPsec task offload interface
Ke	5 (2)	392	2 (2)	10248	tcpip.sys	IPsec key object
LeGe	1558 (13)	16104	0 (0)	0	tcpip.sys	Legacy Registry Mapping Module Buffers

Tag	NP Alloc	NP	Bytes P	Alloc	Pool tag binary	Pool tag description
NLNa	3 (0)	0	0 (0)	0	tcpip.sys	Network Layer Network Address Lists
NLap	9 (2)	640	0 (0)	0	tcpip.sys	Network Layer Netio Helper Function allocations
NLcc	14 (14)	672	0 (0)	0	tcpip.sys	Network Layer Client Contexts
NLcp	2 (2)	2192	0 (0)	0	tcpip.sys	Network Layer Compartments
NMRb	67 (59)	4248	0 (0)	0	tcpip.sys	Network Module Registrar Bindings
NMRc	42 (0)	0	0 (0)	0	tcpip.sys	Network Module Registrar Arrays
NMRg	8 (0)	0	0 (0)	0	tcpip.sys	Network Module Registrar Generic Buffers
NMRm	62 (62)	3968	0 (0)	0	tcpip.sys	Network Module Registrar Modules
NMRn	12 (12)	672	0 (0)	0	tcpip.sys	Network Module Registrar Network Protocol Identifiers
NeWQ	53 (51)	2040	0 (0)	0	tcpip.sys	NetIO WorkQueue Data
Net	10 (0)	0	0 (0)	0	tcpip.sys	NetIO Generic Buffers (iBFT Table allocations)
Nhfs	2 (2)	4664	0 (0)	0	tcpip.sys	NetIO Hash Function State Data
OlmI	7 (5)	2920	0 (0)	0	tcpip.sys	Offload Manager Interfaces
RaDA	1 (1)	104	0 (0)	0	tcpip.sys	Raw Socket Discretionary ACLs
RaMI	1 (1)	4096	0 (0)	0	tcpip.sys	Raw Socket Message Indication Tags
Raw	60 (0)	0	0 (0)	0	tcpip.sys	Raw Socket Endpoints
RawE	1 (0)	0	0 (0)	0	tcpip.sys	Raw Socket Endpoints
RawN	2 (2)	80	0 (0)	0	tcpip.sys	Raw Socket Nsi
RhHi	1 (1)	20496	0 (0)	0	tcpip.sys	Reference History Pool
TNbl	57 (5)	2320	0 (0)	0	tcpip.sys	TCP Send NetBufferLists
TQoS	2 (2)	64	0 (0)	0	tcpip.sys	TL QoS Client Data
TSNb	2 (2)	752	0 (0)	0	tcpip.sys	TCP Send NetBuffers
TWTa	2 (2)	288	0 (0)	0	tcpip.sys	Echo Request Timer Table

Tag	NP Alloc	NP	Bytes P	Alloc	Pool tag binary	Pool tag description
TcCC	1 (1)	72	0 (0)	0	tcpip.sys	TCP Create And Connect Tcb Pool
TcCM	1 (1)	48	0 (0)	0	tcpip.sys	TCP Congestion Control Manager Contexts
TcCR	1 (1)	4096	0 (0)	0	tcpip.sys	TCP Connect Requests
TcDM	59 (12)	33600	0 (0)	0	tcpip.sys	TCP Delayed Delivery Memory Descriptor Lists
TcDN	7 (5)	17968	0 (0)	0	tcpip.sys	TCP Delayed Delivery Network Buffer Lists
TcDQ	1 (1)	264	0 (0)	0	tcpip.sys	TCP Delay Queues
TcEW	1 (1)	4096	0 (0)	0	tcpip.sys	TCP Endpoint Work Queue Contexts
TcHT	1 (1)	712	0 (0)	0	tcpip.sys	TCP Hash Tables
TcPt	1 (1)	80	0 (0)	0	tcpip.sys	TCP Partitions
TcRA	39 (22)	1056	0 (0)	0	tcpip.sys	TCP Reassembly Data
TcRB	158 (0)	0	0 (0)	0	tcpip.sys	TCP Reassembly Buffers
TcRD	1 (1)	32	0 (0)	0	tcpip.sys	TCP Receive DPC Data
TcRH	164 (6)	912	0 (0)	0	tcpip.sys	TCP Reassembly Headers
TcRL	1 (1)	1032	0 (0)	0	tcpip.sys	TCP Create And Connect Tcb Rate Limit Pool
TcRe	7 (7)	392	0 (0)	0	tcpip.sys	TCP Recovery Buffers
TcST	4 (4)	832	0 (0)	0	tcpip.sys	TCP Syn TCBs
TcTW	1 (1)	4096	0 (0)	0	tcpip.sys	TCP Time Wait TCBs
TcWS	10 (1)	968	0 (0)	0	tcpip.sys	TCP Window Scaling Diagnostics
TcpB	4 (4)	832	0 (0)	0	tcpip.sys	TCP Offload Blocks
TcpE	424 (369)	193552	0 (0)	0	tcpip.sys	TCP Endpoints
TcpI	4 (4)	1904	0 (0)	0	tcpip.sys	TCP ISN buffers
TcpL	32 (15)	2520	0 (0)	0	tcpip.sys	TCP Listeners
TcpM	1 (1)	40	0 (0)	0	tcpip.sys	TCP Offload Miscellaneous buffers
TcpN	165 (47)	4840	0 (0)	0	tcpip.sys	TCP Name Service Interfaces
Tcpt	2 (2)	4344	0 (0)	0	tcpip.sys	TCP Timers
UDNb	1 (1)	464	0 (0)	0	tcpip.sys	UDP NetBuffers

Tag	NP Alloc	NP	Bytes P	Alloc	Pool tag binary	Pool tag description
UNbl	4 (4)	1736	0 (0)	0	tcpip.sys	UDP NetBufferLists
UdEW	1 (1)	4096	0 (0)	0	tcpip.sys	UDP Endpoint Work Queue Contexts
UdMI	1 (1)	4096	0 (0)	0	tcpip.sys	UDP Message Indications
UdpA	1017 (20)	3680	0 (0)	0	tcpip.sys	UDP Endpoints
UdpN	2 (2)	80	0 (0)	0	tcpip.sys	UDP Name Service Interfaces

Appendix F: Example of Disassembling Image in Rekall

The following instructions where received from Michael Cohen to show the disassembling capabilities of Rekall, example is the `TcpE` pool tag.

1. Getting size of kernel.

```
[1] win10_20160425.aff4 12:58:34> peinfo "nt"
          Attribute                                      Value
------------------------------ -----------------------------------------------
-------------
Machine                        IMAGE_FILE_MACHINE_AMD64
TimeDateStamp                  2016-01-27 04:38:01Z
Characteristics                IMAGE_FILE_EXECUTABLE_IMAGE,
IMAGE_FILE_LARGE_ADDRESS_AWARE
GUID/Age                       D03C5CF7862E48FE84A06333F1CFA5981
PDB                            ntkrnlmp.pdb
MajorOperatingSystemVersion    10
MinorOperatingSystemVersion    0
MajorImageVersion              10
MinorImageVersion              0
MajorSubsystemVersion          10
MinorSubsystemVersion          0

Sections (Relative to 0xf80210089000):
Perm   Name       Raw Off          VMA             Size
---- -------- -------------- -------------- --------------
xr-   .text    0x000000000600 0x000000001000 0x00000021d400
xr-   ERRATA   0x00000021da00 0x00000021f000 0x000000000200
xr-   INITKDBG 0x00000021dc00 0x000000220000 0x000000010000
xr-   POOLCODE 0x00000022dc00 0x000000230000 0x000000002800
-r-   .rdata   0x000000230400 0x000000233000 0x000000090200
....
```

Figure 53: Read kernel size form image in Rekall

Kernel code starts at address 0xf80210089000+ 0x000000001000 and it is of size 0x00000021d400.

2. Disassemble entire kernel into a file.

```
[1] win10_20160425.aff4 13:04:51> dis offset=0xf80210089000+0x1000,
end=0xf80210089000+0x1000+0x00000021d400, output="ntkrnlmp_20160425.dis"
```

Figure 54: Disassembling kernel into file in Rekall

The same steps can also be done for parts of the kernel like the `tcpip.sys` by using `peinfo` "`tcpip`". This step can take a while.

3. Find allocating functions by looking for pool tags, by the example of `TcpE`.

```
[1] Default session 13:10:10> import struct
[1] Default session 13:10:24> hex(struct.unpack("<L", "TcpE")[0])
Out<3> '0x45706354'
```

Figure 55: Importing structs and finding `TcpE` referencing function

4. Search file (by using `less` or `vi`) to find the occurrence of the hex value.

```
------ tcpip!TcpStopEndpointModule ------
0xf880016e5ef8 0x58 488b0d39460a00 mov rcx, qword ptr [rip + 0xa4639]
                                   0xfffffa800135db40 tcpip!TcpEndpointPool
0xf880016e5eff 0x5f ba54637045 mov edx, 0x45706354
0xf880016e5f04 0x64 e8f738feff call 0xf880016c9800
                               tcpip!PplDestroyLookasideList
```

Figure 56: Found occurrence of `TcpE` in disassembly file

```
------ tcpip!TcpStartEndpointModule ------
0xf88001713ab4 0x14 c740f054637045   mov dword ptr [rax - 0x10], 0x45706354
0xf88001713abb 0x1b 33ff             xor edi, edi
0xf88001713abd 0x1d 4533c9           xor r9d, r9d
0xf88001713ac0 0x20 668978e8         mov word ptr [rax - 0x18], di
0xf88001713ac4 0x24 c740e054637045   mov dword ptr [rax - 0x20], 0x45706354
0xf88001713acb 0x2b 4533c0           xor r8d, r8d
0xf88001713ace 0x2e 33d2             xor edx, edx
0xf88001713ad0 0x30 33c9             xor ecx, ecx
0xf88001713ad2 0x32 48c740d810030000 mov qword ptr [rax - 0x28], 0x310
0xf88001713ada 0x3a e89162fdff       call 0xf880016e9d70
                                     tcpip!PplCreateLookasideList
0xf88001713adf 0x3f 488905526a0700   mov qword ptr [rip + 0x76a52], rax
                                     0xfffffa800135db40 tcpip!TcpEndpointPool
```

Figure 57: Found 2nd occurrence of `TcpE` in disassembly file

The interpretation here is that all allocations with this pool tag are done by a function called Endpoint Module which creates an entire pool for it with a lookaside list (see Glossary entry G4). As actually this pool tag is related to netstat, an existing list in memory referencing all TCP Endpoint addresses does sound reasonable, and proofs the assumed reason from chapter 5.4.2.

It's further possible to convert between the ASCII and hex value of a pool tag by using the following commands:

```
[1] Default session 13:10:10> import struct
```

```
[1] Default session 13:10:24> hex(struct.unpack("<L", "TcpE")[0])
                              Out<3> '0x45706354'
[1] Default session 13:10:52> struct.pack("<L", 0x45706354)
                              Out<4> 'TcpE'
```

Figure 58: Converting pool tag between hex and ASCII in Rekall