

Aleksander Østerud

Windows 10 Memory Compression in Digital Forensics

Uncovering Digital Evidence in Compressed Swap

Master's thesis in Information Security

Supervisor: André Årnes and Katrin Franke

December 2018

Aleksander Østerud

Windows 10 Memory Compression in Digital Forensics

Uncovering Digital Evidence in Compressed Swap

Master's thesis in Information Security
Supervisor: André Årnes and Katrin Franke
December 2018

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

 **NTNU**
Norwegian University of
Science and Technology

Abstract

Digital investigators and incident responders often rely on evidence residing in computer memory and page files on hard drives. Artifacts such as browsing history, image thumbnails and shell commands can answer important questions in digital investigations. Windows 10 introduces memory compression, which compresses inactive parts of computer memory, leading to obfuscation of potentially important artifacts.

In this thesis, the student proposes principles and investigates methods for decompressing the parts of memory compressed by the Windows 10 operating system. The goal of the thesis is to create a method for decompressing and de-obfuscating potentially important information from compressed data in memory samples and page files, and making it available to the forensics community. Memory compression in digital forensics of Windows 10 is a previously unsolved problem.

Through research and experiments, the student has created a proof-of-concept tool with these capabilities, called "MemoryDecompression". The tool is tested on data from two scenarios that involves recovering strings that has been compressed and obfuscated by the memory manager. The results show that strings are in fact being obfuscated through memory compression. The tool was submitted to the Volatility Plugin Contest as a contender, and ended up on 2nd place. This is presented as an indicator of quality and potential value. It also brings attention to the issues of memory compression, and makes the tool available to the forensics community.

The results, the impact and the weaknesses of the applied experiments are discussed. Finally, the thesis suggests future work in this subject, which includes further research on Windows memory manager, and further development of MemoryDecompression tool.

Sammendrag

Dataetterforskere og hendelseshåndterere baserer seg ofte digitale på bevis som er tilstede i dataminnet og i sidevekslingsfiler på harddisker under en hendelse. Artefakter som nettleserhistorikk, bilder og kommandoer kjrt i en konsoll kan svare på viktige spørsmål relatert til digital etterforskning. Windows 10 introduserer minnekomprimering, som komprimerer inaktive deler av dataminnet, noe som fører til obfuskering av potensielt viktige bevis.

I denne oppgaven avdekker studenten prinsipper og undersøker metoder for dekomprimere deler av dataminnet som er komprimert av Windows 10. Målet med avhandlingen er å skape en verdifull metode for dekomprimering og de-obfuskering av potensielt viktig informasjon fra komprimerte data i minnedumper og sidevekslingsfiler, samt tilgjengeliggjøre den for dataetterforskningsmiljet. Minnekomprimering i dataetterforskning er et problemet som hittil ikke er løst av noen.

Gjennom forskning og eksperimenter er det laget et "proof-of-concept"-verktøy med disse kapabilitetene, kalt "MemoryDecompression". Verktøyet testes på data fra to scenarier ved å forsøke gjenoppretting av strenger som har blitt komprimert og dermed obfusert. Resultatene viser at strenger faktisk blir obfusert gjennom minnekomprimering. Verktøyet ble innsendt som et bidrag til Volatility Plugin Contest, og endte på 2. plass. Dette presenteres som en indikator p kvalitet og potensiell verdi. Det retter ogsåoppmærksomhet mot problemene rundt minnekompresjon, og tilgjengeliggjør verktøyet til dataetterforskningsmiljøet.

Ved hjelp av resultatene diskuteres virkningen og svakhetene av de praktiske forsøkene. Til sist foreslår avhandlingen fremtidig arbeid innen emnet, som inkluderer videre forskning på Windows minnehåndtering og videreutvikling av MemoryDecompression-verktøyet.

Preface

This Masters thesis is a product of the authors Master studies at the Norwegian University of Science and Technology (NTNU) and written over two semesters in 2018. The topic of this thesis was inspired by a conversation with Michael Hale Ligh and Andrew Case, co-authors of *The Art of Memory Forensics* (Case and Walters, 2014b), during a memory forensics course in September 2017.

The work is similar to previous research by Andrew Case and Golden G. Richard III on analyzing compressed RAM in Mac OS X and Linux (Richard, 2014a). The authors has also addressed the subject of this thesis in "Memory Forensics: The Path Forward" (Richard, 2016b).

Acknowledgements

- Co-Supervisor André Årnes for constructive feedback and his previous work on digital forensics.
- The Open Source and Digital Forensics community, for providing invaluable tools and by sharing knowledge and experiences.
- Alex Ionescu and Michael Ligh for taking the time to answer questions regarding the thesis.
- My employer, colleagues, and friends at NorCERT for providing a great environment for continuous learning and development. Your support and guidance has been greatly appreciated.
- Special thanks to my colleague and Windows internals guru Hans Kristian Brendmo for invaluable guidance and support in Windows reverse engineering and C++ programming.

Table of Contents

Abstract	i
Sammendrag	i
Preface	ii
Table of Contents	vi
List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Motivation	2
1.2 Problem Description	2
1.3 Hypothesis Statement	2
1.4 Related Work	3
1.4.1 Memory Forensics	3
1.4.2 Analyzing Page Files	3
1.4.3 Memory Compression in Digital Forensics	3
1.4.4 Memory Compression function	4
1.4.5 Decompressing Registry Keys	4
1.4.6 Previous Research affected by Memory Compression	4
1.5 Contribution	5
1.6 Limitations and Scope	5
1.7 Thesis Structure	6
1.8 Conventions	6
1.9 Tools	7
1.9.1 VMWare Workstation	7
1.9.2 Volatility	7
1.9.3 Windows Performance Toolkit	7

1.9.4	WinDbg	8
1.9.5	The Sleuth Kit	8
1.9.6	IDA Pro	8
2	Digital Forensics	9
2.1	Sources of Digital Evidence	9
2.2	The Digital Forensics Process	9
2.3	Forensic Soundness	10
2.4	Windows Forensics	10
3	Windows Memory Management	13
3.1	Virtual Memory	13
3.2	Demand Paging	15
3.3	Memory Compression	15
3.4	Microsoft's Xpress Algorithm	16
3.4.1	Plain LZ77	16
3.4.2	RtlCompressBuffer	16
3.4.3	RtlDecompressBufferEx	19
3.4.4	Essential Arguments - 64-bit Parameter Handling	20
4	Memory Forensics	21
4.1	Memory Acquisition	21
4.1.1	How to capture memory	21
4.1.2	When to capture memory	22
4.2	Memory Analysis	23
4.2.1	Processes	23
4.2.2	Hibernation Files	24
4.2.3	Page Files	24
5	Methodology	27
5.1	Development of a Proof of Concept Tool	27
5.2	Finding the Compression Algorithm	27
5.3	Kernel Debugging with WinDbg	30
5.3.1	Lab Setup	30
5.3.2	Inspecting RtlCompressBuffer	31
5.3.3	Finding the Decompression Function	33
5.3.4	Inspecting RtlDecompressBufferEx	34
5.3.5	Extracting Data for Testing	35
5.3.6	Decompressing Testing Data	37
5.3.7	Testing LZ77	38
5.4	Proof of Concept Tool: MemoryDecompression	40
5.4.1	Choice of Programming Language	40
5.4.2	Sample Output	40
5.4.3	How MemoryDecompression Works	42
5.5	Testing MemoryDecompression	44
5.5.1	Scenario: Console Activity	44

5.5.2	Scenario: Browser Activity	45
6	Results	47
6.1	Results: Console Activity	47
6.1.1	String search on compressed data	49
6.1.2	String search on the decompressed data	49
6.1.3	Discussion	51
6.2	Results: Browser Activity	53
6.2.1	Before TestLimit.exe	53
6.2.2	After TestLimit.exe	54
6.2.3	Pagefile.sys	55
6.2.4	Discussion	58
7	Discussion	59
7.1	Applied Experiments	59
7.1.1	Findings	59
7.1.2	Forensic Soundness	59
7.1.3	Impact	60
7.1.4	Weaknesses	60
7.2	Evaluation of MemoryDecompression	61
7.2.1	Integrity	61
7.2.2	Improvements	62
7.2.3	Volatility Plugin Contest	62
7.3	Strategies for Decompressing Memory	63
7.3.1	Strategy 1: Decompression using a Brute Force approach	63
7.3.2	Strategy 2: Reverse Engineering Windows kernel code	63
7.3.3	Evaluation	64
7.3.4	Conclusion	64
7.4	Future Work	64
7.4.1	Volatility Extension	64
7.4.2	Pattern Searching for compressed data	65
7.4.3	MemoryDecompression platform-agnostic	65
7.4.4	Pooltag Analysis	65
7.4.5	Reverse Engineering WinDbg libraries	66
7.4.6	Statistical Verification	66
7.4.7	Further Development of MemoryDecompression	66
7.5	Hypothesis Discussion	66
7.5.1	Hypothesis 1	67
7.5.2	Hypothesis 2	67
8	Conclusion	69
	Bibliography	70
	Appendix A	75

List of Tables

3.1	Table showing registers used for passing parameters in 64-bit architecture from Microsoft (2018b).	20
6.1	Shows the amount of strings related to ”fareniv” that were found in the different locations.	58

List of Figures

3.1	Illustration showing how virtual addresses can be mapped to physical from Case and Walters (2014e).	14
3.2	Parameters of RtlCompressBuffer from Microsoft (2018c).	17
3.3	Alternatives of the CompressionFormatAndEngine-parameter from Microsoft (2018c).	18
3.4	Parameters of RtlDecompressBufferEx. Figure is copied from Microsoft's documentation (Microsoft, 2018d)	19
3.5	RtlDecompressBufferEx syntax with explanation of how arguments are handled, based on Microsoft (2017c).	20
4.1	Memory acquisition flow chart from Case and Walters (2014c).	22
5.1	Screenshot from Windows Performance Recorder.	28
5.2	Screenshot from Windows Performance Analyzer, navigating to the Flame by Process view.	28
5.3	Screenshot from Windows Performance Analyzer, stack of the MemCompression process in the Flame by Process view.	29
5.4	Windows 10 version on the host.	30
5.5	Windows 10 version on the guest.	31
5.6	Screen shot from Visual Studio showing contents from winnt.h.	33
5.7	Example from the Microsoft's Xpress Algorithm documentation.	39
5.8	Screenshot from the web browser activity.	45

Introduction

During the past decade, researchers have made memory forensics a powerful utility in digital investigations. By creating a number of tools that is now available to the analyst, memory forensics has evolved from simple string searches to a deeper, more structured analysis. A wealth of information can be found, exposing activity that only resides in memory (Richard, 2016b).

A relatively new development in malware evolution is malware that leaves no traces on the hard drive of the system it infects. Considering this development, being able to perform memory forensics is critical. Other volatile artifacts that is lost when the computer shuts down includes running processes, network connections, key material for full hard drive encryption, console commands and chat messages (Richard, 2016b).

Microsoft introduced memory compression in Windows 10 as a feature for increased performance. This enables parts of process memory to be compressed and swapped out to a designated memory store (Yosifovich, 2017a). Compressing these parts of memory potentially obfuscates the data. This process weakens the unstructured searching methods traditionally applied on this type of data (Richard, 2016a). Memory compression in digital forensics of Windows 10 is a previously unsolved problem. It was mentioned on a Forensic Lunch episode (Cowen, 2015) in 2015 by Andrew Case, and described as a "huge problem".

This chapter contains the introduction to the thesis. It provides the background and motivation of the research, leading to the description of the research problem and the hypotheses. Previous work on memory forensics and memory compression is covered, as well as how this research will serve as a contribution to the forensics community. The limitations and scope of the thesis are presented, discussing the area of focus for the research and the applied experiments.

1.1 Motivation

"There is currently no research that documents the algorithms used by compressed store¹, which means that the compressed pages are essentially inaccessible to memory forensics investigators. " Richard (2016b)

The motivation behind the thesis is giving the forensics community a proof-of-concept tool that de-obfuscates compressed pages, and provide research that can be further built upon. Malware hunting in digital forensics is about being one step ahead of the adversaries. As modern operating systems creates opportunities for security analysts to have more data at their disposal, the forensics community should make use of this opportunity, and develop tools strengthening the capability of the good guys. Windows 10 is already widely used, and will likely continue to grow over the coming years. As a consequence, it is likely to be targeted by attackers, and thereby ending up being analyzed by investigators.

1.2 Problem Description

Digital investigators and incident responders often rely on evidence residing in computer memory, and in page files on hard drives. Artifacts such as browsing history, image thumbnails and shell commands can reveal important information in digital investigations. Windows 10 introduces memory compression, which compresses parts of computer memory, leading to obfuscation of potentially important artifacts.

In this project, the student will study principles and investigate methods for decompressing the parts of memory compressed by the Windows 10 operating system. The student will attempt to use this knowledge to create a proof-of-concept tool for decompressing memory, and thus de-obfuscating potentially vital artifacts in memory samples and page files.

1.3 Hypothesis Statement

To solve the issue of memory compression, the thesis will be based on two research hypotheses. These will be supported or refuted through practical experiments and evaluations using existing literature.

- **Hypothesis 1:** *By analyzing operating system behaviour, it is possible to create a method for decompressing, and thus de-obfuscating, data from memory samples or page files that has been compressed by the Windows 10 memory manager.*
- **Hypothesis 2:** *De-obfuscating the data compressed by Windows 10 memory manager can uncover digital evidence for computer forensics investigators.*

¹Compressed store is where the memory compression performed by the memory manager in Windows 10 stores its data.

1.4 Related Work

This section contains related work on the subject.

1.4.1 Memory Forensics

The acquisition and analysis of computer memory is increasingly common within forensics the last five years. As memory analysis enables a wealth of information, it is an invaluable addition to digital forensics. Frameworks for acquiring and analyzing computer memory has become increasingly available to the investigators, who typically had to rely on manual string searches in the past. In addition, the literature on the subject of memory forensics has become much better with the release of *The Art of Memory Forensics* in 2014 (Case and Walters, 2014b).

1.4.2 Analyzing Page Files

In addition to covering most parts of memory forensics, *The Art of Memory Forensics* contains some literature on analysis of page files (Case and Walters, 2014a). As memory compression in Windows 10 serves as a virtual page file, the same techniques will apply for analysis of the data once decompressed. The page file can only be analyzed using unstructured techniques. Strings, grep, anti-virus scans, or Yara signatures are some examples of tools that are utilized. *The Art of Memory Forensics* also mentions a tool called "page-brute" that splits a page file into chunks and scans it with Yara rules. Unstructured analysis can be used to find e-mail headers, console commands, HTTP requests and responses, etc. By searching for something directly related to the suspected activity, it is possible to find actionable evidence. For example, URL-history or search terms typed in the browser that relates to illegal activities can be used to as pivot points to further investigate and find evidence of a crime.

1.4.3 Memory Compression in Digital Forensics

Memory compression in digital forensics has been addressed on Mac OS/OSX and Linux. The paper "In Lieu of Swap" (Richard, 2014a) presents the challenges and opportunities as well as a method and tool for decompressing memory compressed by the two operating systems. The authors predicted that the same problem would probably apply to a future Windows version. This paper also presents how swap files can be used as a source of evidence. In addition to the unstructured analysis techniques mentioned in Section 1.4.2, the paper addresses the use of data carving tools as technique for extracting valuable information from page files. Tools such as bulk-extractor, photorec and foremost (AccessData, 2018a).

Another research paper by the same authors also addresses the topic of memory compression, specifically on Windows 10 (Richard, 2016a). The paper generally discusses current issues and future directions for memory forensics. It also presents memory compression as a big challenge for forensic investigators, as they are highly dependent on unstructured methods for finding in-memory data. The authors mention that there is no current research that documents the algorithms used to perform memory compression and

that discovering a means for decompressing this data will enable unstructured analysis to become useful.

The topic of memory compression in Windows 10 has been addressed in the latest Windows Internals book (Yosifovich, 2017a). There is a section on the topic that contains information about how the memory manager performs memory compression works, but unfortunately it does not present any details on the methodology of the research. The Memory Compression process is also mentioned in book (Yosifovich, 2017c). The memory manager uses its user-mode address space to store the compressed pages that would otherwise go to a page file. Unlike other System processes, the Memory Compression process stores data in user-mode space, and is therefore subject to trimming. This means that its data can be paged out to a page file on disk. The process also host a number of system threads, usually seen as `SmKmStoreHelperWorker` and `SmStReadThread`. "Sm" is the prefix for Store Manager, which manages memory compression in Windows 10.

1.4.4 Memory Compression function

A blog post by Alois Kraus addresses some aspects of memory compression, but not in relation to digital forensics (Kraus, 2016). The topic is how the Memory Compression process performs when a certain memory pressure is applied. This work includes the use of Windows Performance Kit, which reveals what function `MemCompression` uses for compressing and decompressing memory pages. This type of work has been done before, but to the knowledge of the student, this blog post was the only source of information that contained traces of the correct function.

1.4.5 Decompressing Registry Keys

During the work on this thesis, two other blog posts were written on the topic of memory compression. They both address some of the issues mentioned in Section 1.2. The first blog post addresses the compression algorithm, and a use-case concerning compressed registry keys in memory samples and page files (MSUHANOV, 2018). The author suggests a method for recognizing LZ77 compressed data, which is based upon finding a certain string after a 32 bit integer. This is how the author finds compressed registry keys in memory and page file. As most compressed pages does not follow this pattern, it does not help much in this thesis, but it is a very strong technique for finding compressed data with known structures.

The second blog post shows how to find the function used for compression and decompression. Though the the data is interpreted incorrectly, ending up with the wrong algorithm, the means to get the information is correct and similar to the methodology in this thesis (TSS, 2018).

1.4.6 Previous Research affected by Memory Compression

Any previous work that has researched methods for unstructured memory analysis is likely affected by memory compression. The search patterns that are utilized to find specific data become obsolete. An example of this is the previous work performed at NTNU on

search strategies for finding and extracting cryptographic keys from memory (Maartman-Moe, 2009). If the memory pages that contains the cryptographic keys are compressed, finding them will likely be impossible with the same techniques. By decompressing and de-obfuscating the data, it should be possible to find the keys using the same techniques.

1.5 Contribution

The main contribution of this thesis is the proof-of-concept tool "MemoryDecompression". The tool was submitted to the Volatility Plugin Contest 2018 and ended up on 2nd place. As a result, it is now publicly available online (Hale Ligh, 2018b), as discussed in section 7.2.3. By making this tool available, everyone who performs computer forensics on Windows 10 workstations, and likely servers in the future, will have more potential evidence data at their disposal. As it works on both page files and memory dumps, it is a valuable contribution to both memory and hard disk analysis. It is also possible that the tool would work on Windows Phones and X-Box memory images, however that is not within the scope of this thesis.

MemoryDecompression is a proof-of-concept tool, and should be treated as such. Though the tool has been verified through the Volatility Plugin Contest, it is still a risk that it could produce faulty results (further discussed in Section 7.1.2). The source code is openly available online and in Appendix A, so the investigator has the opportunity to examine the code before using the tool.

The secondary contribution is the research covered in Chapter 5 that can be further built upon to get a better understanding of how memory compression works. The research covers algorithm for decompression and how it works, but does not contain how the memory compression store handles its meta data. Further research on the subject can make it possible to create a more sophisticated decompression-tool for digital forensics. The thesis opens several paths to further research. As the algorithm is known, it is possible to create a platform-agnostic tool for decompressing memory from Windows 10. Pooltag-analysis is suggested as a means to find the meta data belonging to the Memory Compression process. Reverse engineering of the Windows debugger libraries is also an area of future research. Section 7.4 discusses some more alternatives for future work.

1.6 Limitations and Scope

This subject has a lot of potential areas for research, this makes limiting the research essential. The research in this thesis will focus on de-obfuscating contents of Windows 10 memory samples and page files. Memory compression is also available for Windows Phone and X-Box, but these platforms will not be covered. Hibernation files are mentioned in this thesis, but not covered in the practical experiments. This is because the forensics value of hibernation files has drastically decreased in later versions of Windows, as they are zeroed out after the system wakes (Richard, 2016c).

The students limitations in this thesis are the complexity of the topic and the time at his disposal. Completing the objectives of the thesis requires deep knowledge of the Windows

operating systems. As part of the process, a significant portion of time is spent acquiring the necessary competence and knowledge for accomplishing the goals of this thesis.

The research will be limited to finding the compression algorithm for the memory compression mechanism, and using it to create a brute-force tool. The other alternative was finding and incorporating the meta data of the Memory Compression process into the decompression tool. This alternative will not be covered in this thesis, but it will be further discussed in Section 7.3.2 and 7.3.3.

Taking these limitations into consideration, it is necessary to focus on certain areas. The main priority will be to create software that can decompress the obfuscated data. Analyzing the decompressed data will be covered, but not prioritized. The experiments requires careful planning, and even with relatively small memory samples it can take up to 7 hours from beginning to end. The focus is delivering an enhanced means of pre-processing memory samples and page files for the forensic community. This means that answering hypothesis 1 is prioritized over hypothesis 2.

Finally, The students lack of experience with the desired programming language to create the decompression tool is a limitation,. The consequence is that the process of creating the tool will take a significant amount of time.

1.7 Thesis Structure

The remainder of this thesis is structured as follows.

- **Chapter 2** contains background information of relevant topics from digital forensics in general.
- **Chapter 3** explains the relevant components of the Windows Memory Management.
- **Chapter 4** contains the relevant existing literature within memory forensics.
- **Chapter 5** explains how the decompression algorithm was found, and how the tool was developed. It also explains how the tool works, and presents the scenarios for the practical experiments.
- **Chapter 6** presents the results from the practical experiments performed with the MemoryDecompression tool.
- **Chapter 7** discusses strategies considered for solving the problem of memory compression.
- **Chapter 8** concludes the thesis.

1.8 Conventions

There are a number of conventions used throughout this paper.

All console commands are written in a console font, like `this`. To highlight interesting parts of the console command output, it is shown in **bold console text**. Both

Linux and Windows consoles are being used to produce results in this paper.

Windows console:

```
PS> echo "Hello Windows!"  
Hello Windows!
```

Linux console:

```
# echo "Hello Linux!"  
Hello Linux!
```

Some console outputs are trimmed for the sake of brevity. This is mentioned in the context of the output. If the output has a large number of lines, . . . is used to account for the missing lines.

1.9 Tools

This section contains the tools used in research and experiments in this thesis. The tools that were used are all highly recognized, and have been around for a long time. Choosing tools with this criteria reduces the risk of errors produced by the tools during the practical experiments.

1.9.1 VMWare Workstation

VMWare Workstation is a virtualization program that enables the use of virtual machines running on a desktop computer (VMWare, 2018). In this thesis it has been used as a lab environment. By hosting a virtual machine that is being debugged by the Windows debugger, it is possible to control and monitor the behaviour of the guest operating system.

A possible alternative to VMWare Workstation is VirtualBox by Oracle. VMWare Workstation was chosen for the lab environment as the student has it at his disposal, generally considers it a better tool and has a lot of experience using it.

1.9.2 Volatility

Volatility is a memory forensics framework written in Python. It is open source, and generally considered to be the best and most extensive memory forensics tool available. It supports Microsoft Windows, Mac OS X, and Linux. In this thesis, Volatility is mainly used for extracting compressed contents from a memory dump.

An alternative to Volatility is Rekall. Rekall was not considered, as it does not work on Windows 10 memory samples from VMWare virtual machines.

1.9.3 Windows Performance Toolkit

The Windows Performance Toolkit consists of two components, Windows Performance Recorder and Windows Performance Analyzer. The recorder saves the relevant operating system data to a file that can be parsed by the analyzer. In this thesis, the tool is used to

analyze desired parts of Windows. The student was made aware of this tool through an article on the internet (Kraus, 2016).

1.9.4 WinDbg

The Windows Debugger can be used to analyze kernel and user mode code, and analyze memory dump files created through system crashes. It can also analyze Windows executable files. In this thesis, WinDbg is used extensively to analyze the actions of the operating system.

1.9.5 The Sleuth Kit

The Sleuth Kit is a bundle of command line interface forensic analysis tool that can analyze volume and file systems. The framework works on several types of file formats and file systems, the VMWare .vmdk file being one of them (Eriberto, 2014).

There are a number of other tools with similar functionality, such as AccessData FTK Imager (AccessData, 2018b) and Autopsy (GuidanceSoftware, 2018). The Sleuth Kit was chosen as it produces the desired result with a few operations, and the output has a presentable structure.

1.9.6 IDA Pro

IDA Pro is the Interactive Disassembler, a highly recognized tool for reverse engineering software. It is a very popular tool amongst malware analysts (Hex-Rays, 2017). In this research it was used to reverse engineer kernel code.

Digital Forensics

Digital forensics or digital investigation is a process where hypotheses that answer questions about digital events are developed and tested. This is done by using the scientific method on digital evidence to look for traces of malicious activity. Digital evidence is a digital object that contains reliable information that supports or refutes a hypothesis in a digital investigation (Carrier, 2005).

2.1 Sources of Digital Evidence

Digital evidence can be found everywhere. It can be a hard disk from a laptop or a log file from a Cisco-router. Any device that has information that can prove some activity happened or not. When a computer attack occurs, there are a lot of potential sources of evidence that could be of interest. From the computer memory of the attackers computer to the memory of the victim. In-between, the data could travel over a network, potentially leaving traces of information on network equipment. Smartphones and tablets are also potential sources of evidence (Arnes, 2018a).

2.2 The Digital Forensics Process

The Digital Forensics Process refers to a set of principals that is followed during cyber crime investigations.

- The process starts with **identifying** the incident or crime to investigate.
- The necessary data can then be determined and collected in the **collection** phase. This phase can typically include acquiring physical RAM from a running computer, and pulling the hard drive to make a copy for analysis.
- The next step is initial **examination** and preprocessing. This includes running appropriate tools on the sources of digital evidence, making the data human readable.

De-obfuscating and de-crypting evidence is a necessary part of this phase. Other activities include carving relevant data from deleted parts of the evidence, filtering out known good data and searching for known bad data. *Known good* can be legitimate operating system files, and *known bad* can be known strings relating to websites with criminal content.

- Next step is to **analyze** the data. The investigator processes the information data to answer the information requirements based on the nature of the specific case. If the goal is to find out whether or not a corporation is compromised, the analysis could involve looking for evidence of malware executed on the network.
- The last step is to make the final **report** of the incident or crime (Arnes, 2018a).

2.3 Forensic Soundness

Forensic soundness refers to the quality and completeness of methods or tool used in investigations (Arnes, 2018b). As the results of digital investigations in law-enforcement cases can lead to a person being imprisoned or exonerated, it is important that the tools and techniques can be trusted. There is a vast number of digital forensics tools available, some with overlapping functionality. One way of ensuring forensic soundness is to use similar tools to see if they produce the same results. This is referred to as Dual Tool Verification.

By choosing tools that have been extensively tested, preferably with the source code open to the public, the examiner can be relatively certain that the results from the tools are correct. Having the source code available makes it possible to see what the tool actually does (Arnes, 2018a).

2.4 Windows Forensics

Computer forensics on the Windows operating system is by far the most common skill digital investigators possess. According to netmarketshare.com, the share of Desktop computers with Windows operating system is over 85% in 2017 (NetApplications.com, 2018). As Windows is so widely used, it is also a natural target for computer network attacks by cyber criminals. As cyber criminals choose the path of least resistance and biggest gain, Windows is a great operating system to attack.

The Windows operating system is closed source, and several parts of it is not documented publicly by Microsoft. In some cases, the creator of a tool has to reverse engineer Windows components to uncover its functionality (Luttgens, 2014a).

Some core forensics artifacts found in Windows include:

- NTFS file system analysis
- Windows prefetch (evidence of program execution)
- Event logs
- Scheduled tasks

-
- The registry
 - Page file and hibernation file

Windows Memory Management

The Windows Memory Management refers to the operating subsystem that handles or organization of memory. Memory management includes a lot of components, like translating or mapping process's virtual addresses to physical addresses, paging memory contents to disk when physical memory becomes overloaded, memory compression as a virtual page file and making sure each process only has access to their permitted information (Yosifovich, 2017b).

3.1 Virtual Memory

The memory manager is responsible for handling virtual memory. Virtual memory gives each process its own private virtual address space (Yosifovich, 2017d). This creates a separation between the logical memory as seen by the processes, and the physical memory that contains the data. As shown in Figure 3.1, the virtual address space of the processes are linked to either physical memory or a page file on disk. The running processes themselves are not aware that certain contents of its virtual space is saved to the hard drive, it is handled by the memory manager. When a page saved to the page file is accessed by the process, it is handled with a page fault. The contents that was written to the page file is then loaded back into physical memory.

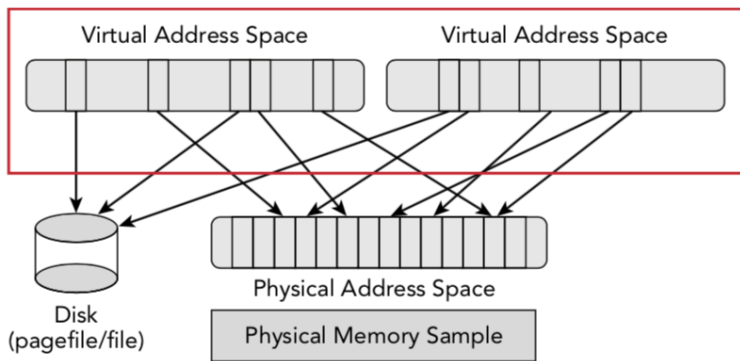


Figure 3.1: Illustration showing how virtual addresses can be mapped to physical from Case and Walters (2014e).

3.2 Demand Paging

Demand paging is a mechanism that is utilized by the memory manager to implement virtual memory (Case and Walters, 2014e). This mechanism is based on keeping the pages needed by running processes resident in physical memory, while the less prioritized pages can be moved to a secondary storage. Demand paging makes space usage in memory more efficient, leading to higher capacity and better performance. Windows has a system for managing these pages, so that the pages that are required to stay in memory do so. This is done to improve stability and performance. To store the pages that are removed from memory, Windows uses a page file that is stored on the file system. The file is hidden in the normal file explorer, and is called pagefile.sys (Case and Walters, 2014e). These can exist in different locations based on the configuration of the system. This configuration can be found in the Windows SYSTEM registry hive. It is most frequently found on the root directory of the volume Windows is installed on. This mechanism creates some issues in memory forensics, as the regions of memory that could be most interesting to the examiner might be in the page file. As shown in Figure 3.1, access to a memory sample is not necessarily sufficient.

3.3 Memory Compression

The problem using a hard drive as a secondary storage for memory data is that when some contents need to be read from the disk and written back in memory, it takes a lot of time. Systems insufficient physical memory needs to read and write data to the slow hard drive frequently. Windows 10 introduces a new mechanism that handles this in a different way. Instead of being written to the file system, the page is compressed and put in a virtual page file. When memory compression was first introduced, the paged out memory was stored in the System-process. In Windows 10 version 1607 this was changed to a new dedicated process called Memory Compression¹. One of the reasons for doing this was that users were troubled by the fact that the System process occupied so much memory, which was in-fact just compressed memory data. The Memory Compression process is a minimal process, meaning that no DLLs are loaded. It also lacks a lot the meta data that a normal process would have. This gives the kernel an empty address space to use as a store for compressed pages (Yosifovich, 2017a).

The primary goal of memory compression was to compress pages belonging to the new Windows 10 applications in the new Universal Windows Platform (UWP). According to Windows Internals 7, the Memory Compression process uses Microsoft's Xpress algorithm. Their experiments has shown that this algorithm compresses data to around 30-50% of their original size (Yosifovich, 2017a). The algorithm balances size with speed, which makes sense as memory compression is implemented for better performance.

¹The process appears as "MemCompression" in process listings.

3.4 Microsoft's Xpress Algorithm

Microsoft created the Xpress compression algorithm in 2011, which consists of three different variants. The three variants are plain LZ77, LZ77+Huffman and LZNT1 (Microsoft, 2018e). The Xpress algorithm used by Windows 10 memory compression is plain LZ77, as shown by the research presented in Section 5.2.

3.4.1 Plain LZ77

Tests in Section 5.3.7 have proven plain LZ77 to be the algorithm used in memory compression. This is the fastest variant of the Xpress-algorithm. "LZ" is short for Lempel-Ziv, which contains the last names of the two Professors, Abraham Lempel and Jacob Ziv, at Israel Institute of Technology who wrote the algorithm. The number "77" comes from the year the algorithm was written, 1977 (ETHW.org, 2015). LZ77 works by replacing strings of characters with a single identifier. These characters are called "tokens". The algorithm searches for new strings, and each time it finds a new one, it adds it to the string, and also to a table or dictionary. Each time the algorithm finds a string of characters that it recognizes from its table, it only outputs the token that represents the string.

An example of how the algorithm could work would be:

"It's the right right, right?"

could become:

"It's the right& &, &?"

In this example, the token is represented by the character "&".

It makes sense to use this algorithm for memory compression, as the main purpose of implementing it was to increase performance. Having a balance between how fast it compresses and how much space is saved seems like a logical solution for the mechanism.

3.4.2 RtlCompressBuffer

RtlCompressBuffer is used by the MemCompression process to compress pages and store them in its process space for later use. This is shown with the Windows Performance Toolkit in Section 5.2. Figure 3.2 shows the parameters that are passed to the function. The parameters are valuable when performing debugging, because it shows how the function is utilized by the process that executes it. As the goal of the thesis is to decompress data, the parameters will be presented in detail in Section 3.4.3, where RtlDecompressBufferEx is covered. As the two functions gives access to about the same information, it is not necessary to cover it more than once. The only parameter unique to RtlCompressBuffer is the format of CompressionFormatAndEngine.

Syntax

```
NT_RTL_COMPRESS_API NTSTATUS RtlCompressBuffer(  
    USHORT CompressionFormatAndEngine,  
    PCHAR UncompressedBuffer,  
    ULONG UncompressedBufferSize,  
    PCHAR CompressedBuffer,  
    ULONG CompressedBufferSize,  
    ULONG UncompressedChunkSize,  
    PULONG FinalCompressedSize,  
    PVOID WorkSpace  
);
```

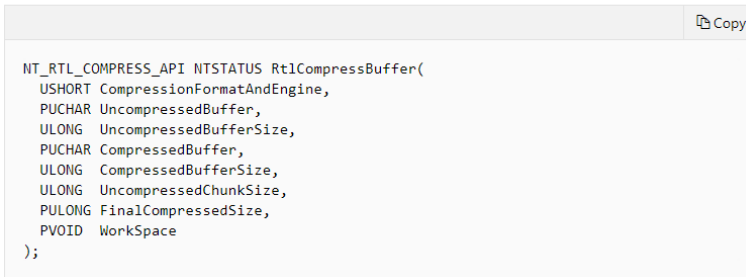


Figure 3.2: Parameters of RtlCompressBuffer from Microsoft (2018c).

CompressionFormatAndEngine

This parameter specifies the compression format and engine type as a bitmask. Figure 3.3 shows the alternatives of formats and engine types that is available. The parameter must be set in the form of a bitmask or a combination of one format type and one engine type. For example `COMPRESSION_FORMAT_XPRESS | COMPRESSION_ENGINE_STANDARD`.

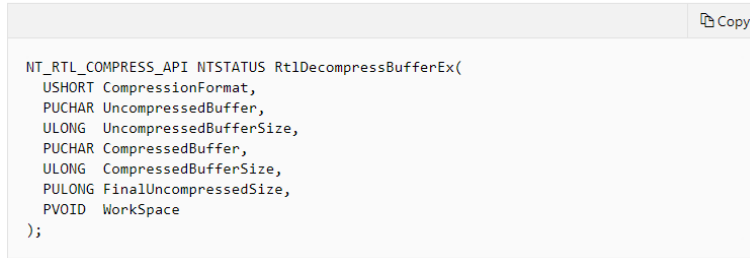
Value	Meaning
COMPRESSION_FORMAT_NONE	Not supported by this function.
COMPRESSION_FORMAT_DEFAULT	Not supported by this function.
COMPRESSION_FORMAT_LZNT1	The function will perform LZ compression.
COMPRESSION_FORMAT_XPRESS	The function will perform Xpress compression.
COMPRESSION_FORMAT_XPRESS_HUFF	The function will perform Xpress Huffman compression.
COMPRESSION_ENGINE_STANDARD	The <i>UncompressedBuffer</i> buffer is compressed using an algorithm that provides a balance between data compression and performance. This value cannot be used with COMPRESSION_ENGINE_MAXIMUM.
COMPRESSION_ENGINE_MAXIMUM	The <i>UncompressedBuffer</i> buffer is compressed using an algorithm that provides maximum data compression but with relatively slower performance. This value cannot be used with COMPRESSION_ENGINE_STANDARD.
COMPRESSION_ENGINE_HIBER	Not supported by this function.

Figure 3.3: Alternatives of the CompressionFormatAndEngine-parameter from Microsoft (2018c).

3.4.3 RtlDecompressBufferEx

RtlDecompressBufferEx is the function used for decompressing pages that has been compressed by the MemCompression process, shown in Section 5.3 through kernel debugging. This function decompresses an entire compressed buffer, which differs from the RtlDecompressChunks or RtlDecompressFragment that only decompresses parts of a buffer.

Syntax



```
NT_RTL_COMPRESS_API NTSTATUS RtlDecompressBufferEx(
    USHORT CompressionFormat,
    PCHAR UncompressedBuffer,
    ULONG UncompressedBufferSize,
    PCHAR CompressedBuffer,
    ULONG CompressedBufferSize,
    PULONG FinalUncompressedSize,
    PVOID WorkSpace
);
```

Figure 3.4: Parameters of RtlDecompressBufferEx. Figure is copied from Microsoft’s documentation (Microsoft, 2018d)

Parameters

The parameters used in RtlDecompressBuffer are shown in Figure 3.4.

1. **CompressionFormat**
The CompressionFormat is similar to the CompressionFormatAndEngine in RtlCompressBuffer, but only handles the Format. The alternatives are the same as shown in Figure 3.3, but only the COMPRESSION_FORMAT_* applies to this parameter.
2. **UncompressedBuffer**
An address in memory that points to the buffer allocated by the caller to receive the decompressed data from the CompressedBuffer.
3. **UncompressedBufferSize**
The size of the buffer that receives the uncompressed data, in bytes.
4. **CompressedBuffer**
An address in memory that points to the buffer containing the actual compressed data that will be decompressed with this function.
5. **CompressedBufferSize**
The size of the CompressedBuffer, in bytes.
6. **FinalUncompressedSize**
A memory address that contains the final uncompressed size of the decompressed data. As MemCompression always compresses a page, this value will always be 4096 on a normally configured system.

7. **FinalUncompressedSize**

A memory address that contains the final uncompressed size of the decompressed data. As MemCompression always compresses a page, this value will always be 4096 on a normally configured system.

8. **WorkSpace**

A memory address that points to a work space buffer used by the function during decompression.

3.4.4 Essential Arguments - 64-bit Parameter Handling

The 64-bit architecture enables the usage of 16 registers that is available for general-purpose. Looking at Windows 10 functions in a debugger requires some background knowledge on how registers store information. For example, when looking at the RtlDecompressBufferEx function, the interesting information lies within the parameters of the function calls. Table 3.1 shows that the registers RCX, RDX, R8 and R9 are used for the first four arguments.

Registers	Use
RCX	First Integer Argument
RDX	Second Integer Argument
R8	Third Integer Argument
R9	Fourth Integer Argument

Table 3.1: Table showing registers used for passing parameters in 64-bit architecture from Microsoft (2018b).

The remaining arguments are passed on the stack. This is because of the `__fastcall` calling convention used by the x64 architecture (Microsoft, 2018a). The function that makes the call to another function, is responsible for setting up the stack and passing the arguments correctly (Microsoft, 2017c).

With this information, it is possible to use a program like the Windows Debugger to inspect the information passed to the proper function calls. Figure 3.5 shows what arguments would be found on which locations.

```
NT_RTL_COMPRESS_API NTSTATUS RtlDecompressBufferEx(  
    USHORT CompressionFormat,  
    PCHAR UncompressedBuffer,  
    ULONG UncompressedBufferSize,  
    PCHAR CompressedBuffer,  
    ULONG CompressedBufferSize,  
    PULONG FinalUncompressedSize,  
    PVOID WorkSpace  
);
```

First four integer arguments is found in the registers.

Remaining arguments are passed on the stack

Figure 3.5: RtlDecompressBufferEx syntax with explanation of how arguments are handled, based on Microsoft (2017c).

Memory Forensics

This chapter and subsequent sections explains memory acquisition and analysis, and presents available tools and techniques. Memory forensics has become increasingly relevant as adversaries and criminals are making attempts to hide their activity. Some computer network attacks involves malware that only lives in memory, and never touches the hard drive. Without a memory sample, it would be impossible to fully analyze such incidents (Kaspersky, 2015).

4.1 Memory Acquisition

Memory acquisition is the act of extracting a copy of the physical memory from a running system. This is potentially one of the most important steps in data collection, as memory contains a wealth of valuable information. The incorrect acquisition of physical memory could result in a corrupt memory sample, or crash the running system. This could also happen if everything is done correctly, based on how the system is configured. However, it is still considered a risk worth taking (Case and Walters, 2014c).

4.1.1 How to capture memory

The typical way to capture memory is inserting a removable storage media that contains memory dumping software into the system. The memory dumping software is run, and the sample is saved to the removable device. This is to minimize the footprint of the memory dumping, as the the process could potentially overwrite valuable data both in memory and on the hard drive. Software that can acquire physical memory requires administrative privileges. Some known tools for memory acquisition are RamCapturer (Belkasoft, 2016), DumpIt (Suiche, 2016), AccessData FTK Imager (AccessData, 2018a), and the pmem-suite (Cohen, 2016).

If administrative privileges are unavailable, it used to be possible to use the firewire interface to get **physical access** to running memory. This approach is becoming obsolete, as the firewire is rarely seen anymore. Also it only gives access to 4GB of memory.

When dealing with **virtual machines**, the process is often a lot simpler. Virtualization software often has functionality for creating snapshots, which includes a copy of its "physical" memory. The file that contains the memory snapshot is simply copied out and can be analyzed on a separate system. Volatility parses these files directly without any need for conversion. Previous research has shown the effectiveness of using virtualized environment for recreating incidents to study computer attacks (Arnes, 2006).

To scale memory dumping, **remote acquisition** is also possible. An example of this could be to have software on each computer in an enterprise with memory dumping capabilities. A memory sample can be acquired from an administrator-computer, and transferred over the network to the computer of the forensics analyst (Case and Walters, 2014c). Figure 4.1 shows a flow chart of how to do memory acquisition based on the different situations.

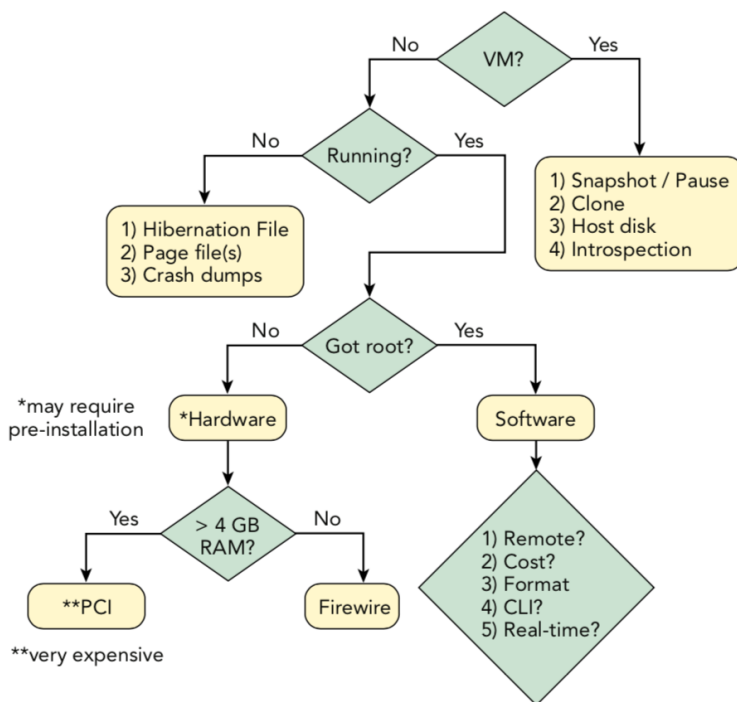


Figure 4.1: Memory acquisition flow chart from Case and Walters (2014c).

4.1.2 When to capture memory

The concept of smearing was mentioned in Section 4.2.3, and this must be taken into consideration when taking acquiring memory. When capturing a memory sample using software, the process will take some time. The amount of time required is mostly dependent upon the amount of physical memory to be acquired, and the transfer rate to the

secondary storage. Write speed to the secondary storage may also be a bottle-neck. The longer the process takes, the more potential for inconsistencies. To mitigate this fact, the memory acquisition should happen when the system is as idle as possible. For example, the acquisition should not happen during Windows updates or at system startup. The person responsible for acquisition should also keep in mind that running memory dumping software can tip off an attacker that they are being watched, and could lead to them wiping their tracks and destroying data.

4.2 Memory Analysis

Memory analysis is the process of analyzing computer memory to find digital evidence. A complete physical memory dump contains valuable data for a forensic examiner. The list of potential evidence is extensive and ever-growing as toolkits evolve, and includes the following:

- Running processes and system objects
- Open and closed network connections
- User credentials
- Loaded drivers
- Encryption keys
- Registry hives, Master File Table, and event logs
- Remnants of console commands
- Remnants of clear-text data that is otherwise encrypted on disk Luttgens (2014b)

There are several tools available for memory analysis. Volatility (Volatilityfoundation, 2016), Rekall (Cohen, 2018) and Redline (FireEye, 2018) are probably the most commonly used frameworks.

4.2.1 Processes

Regardless of what questions the examiner is required to answer, the process list is a good place to start an investigation. It tells the examiner what is currently running on the system. If the goal is to find out if malware was run, a process listing could show malware running on the system. If the user of the computer was suspected to have visited illegal web sites, the examiner could look for running browsers.

This shows the output of a process listing using Volatility on the memory dump WINDEV1710EVAL.dmp. The output is trimmed for visual purposes.

```

volatility -f WINDEV1710EVAL.dmp --profile=Win10x64_16299 pslist
Volatility Foundation Volatility Framework 2.6
Name          PID    PPID    Thds    Hnds    Start
-----
System         4      0      163     0    2018-03-11 13:04:21 UTC+0000
smss.exe      516     4       4       0    2018-03-11 13:04:21 UTC+0000
csrss.exe     612    604     11      0    2018-03-11 13:04:24 UTC+0000
smss.exe     684    516     0 ----- 2018-03-11 13:04:24 UTC+0000
wininit.exe   692    604     5       0    2018-03-11 13:04:24 UTC+0000
csrss.exe    704    684     11      0    2018-03-11 13:04:24 UTC+0000
winlogon.exe  752    684     6       0    2018-03-11 13:04:25 UTC+0000
services.exe  816    692     23      0    2018-03-11 13:04:25 UTC+0000
lsass.exe    824    692     9       0    2018-03-11 13:04:25 UTC+0000
fontdrvhost.exe 916    752     6       0    2018-03-11 13:04:25 UTC+0000
svchost.exe  924    816     30      0    2018-03-11 13:04:25 UTC+0000
fontdrvhost.exe 932    692     6       0    2018-03-11 13:04:25 UTC+0000
...

```

From the process listing, Volatility has many options to further inspect a process. One option is to dump the entire virtual memory space associated with a process by using its process ID. Another option could be to view the command line parameters of that process to determine what arguments were passed to it. It is also possible to extract the executable image from the process (Luttgens, 2014b).

4.2.2 Hibernation Files

A hibernation file potentially contains a full physical memory dump from the last time the system was hibernated. This is valuable in several ways. It could give access to a memory sample from a computer that was shut down. The incident that started the case could have occurred several weeks or months ago, and the hibernation file could potentially contain more relevant data than a current memory sample. Hibernation files are common on laptops, as they are commonly created when the user closes the lid. The hibernation file is compressed by default on most Windows systems, but there are tools that can convert them back to a working memory sample. Volatility can read hibernation files directly, but it takes a lot of time, as it must be decompressed.

4.2.3 Page Files

As page files contains data that has been in computer memory, it is potentially a very interesting source of data in digital investigations. The page file can contain console activity, domains, IP-addresses, credit card numbers, encryption keys, web page fragments, small files, passwords, etc. As it is very hard to get context of the data in the page file, the investigator must know exactly what to look for. For example, a unique domain used by malware could be an indicator of compromise (Richard, 2014b). However, as the Windows Defender process also gets its pages stored in the page file, it can contain some seemingly malicious activity that could be misinterpreted as an actual compromise. Another complication is that Windows does not sanitize blocks of data before allocating them the the page file. This means that old data deleted from the hard drive could appear in the page file. Page files is also discussed in Section 3.2.

In theory, it is possible to get a physical memory sample to provide context if the page file is provided. Because of memory smearing¹ the kernel structures that provides context into the page file are inconsistent as dumping memory and copying the page file takes time while the operating system is still running, and thus changing (Richard, 2014b).

¹Refers to the inconsistencies in a memory sample as a result of the operating system still running and changing memory while it is being captured (Carvey, 2005).

Methodology

This chapter and the subsequent sections contains the methodology used to create a proof-of-concept tool for decompressing memory in memory samples and page files. The topics include finding the algorithm, analyzing Windows kernel code, and the scenarios for testing the proof-of-concept tool.

5.1 Development of a Proof of Concept Tool

For the development of the MemoryDecompression tool, the student will test hypotheses and answer research questions. The main hypothesis that that affects the software development, is that memory regions compressed by the Windows operating system can in-fact be decompressed. The following questions needed to be answered to develop this tool:

- How does Windows compress and decompress pages from memory?
- What are specific parameters required to decompress memory?

5.2 Finding the Compression Algorithm

This section contains the process of finding the compression algorithm used by Windows 10 during memory compression. As mentioned in Section 3.3, the memory manager uses Microsoft's Xpress algorithm to perform compression. Section 3.4 explains the different algorithms used by Xpress. The alternatives are LZ77+Huffman, plain LZ77, and LZNT1. To find out which one is being used here, one approach is to find out what the MemCompression process actually does. The tool called Windows Performance Analyzer, hereby referred to as "Analyzer", mentioned in Section 1.9, will be used for this purpose. It has functionality for showing what Windows processes are doing, and what functions they are using. To create the data required for the Analyzer, the Windows Performance Recorder, hereby referred to as "Recorder", is used. Figure 5.1 shows how the recorder looks to the user. After running the recorder, the data is saved to a file. This is opened in the Analyzer.

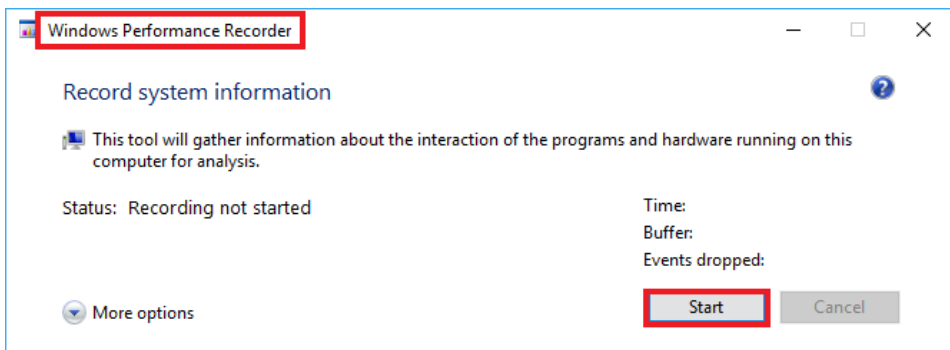


Figure 5.1: Screenshot from Windows Performance Recorder.

After opening the file, it is of interest to see how the MemCompression process behaves. Under "Computation", it is possible to look at "CPU Usage (Sampled)", and from there to "Flame by Process, stack". This is shown in Figure 5.2. This area is of interest to the research. When looking at the stack presented in the "Flame by Process" view, some

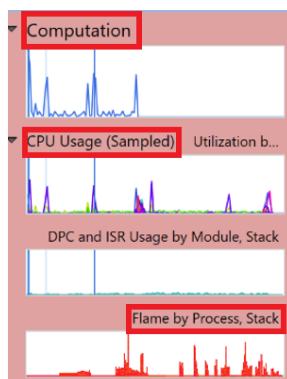


Figure 5.2: Screenshot from Windows Performance Analyzer, navigating to the Flame by Process view.

interesting Windows functions can be seen. Figure 5.3 shows the stack from the Mem-Compression process. The function from ntoskrnl.exe, RtlCompressBuffer seems to be a good pivot point for further analysis. Figure 5.3 also shows RtlCompressBufferXpress-LzStandard, which contains the "Xpress" string. This correlates with the literature from Section 3.3, saying that Microsoft's Xpress algorithm is used in memory compression. To further research the RtlCompressBuffer-function, another tool is required.

Line #	Process	Stack	Count	Sum
1	MemCompression (1492)		825	
2		n/a	427	
3		[Root]	398	
4		ntoskrnl.exe!KiStartSystemThread	398	
5		ntoskrnl.exe!PspSystemThreadStartup	398	
6		- ntoskrnl.exe!SMKM_STORE_MGR<SM_TRAITS>::SmCompressCtxWorke...	240	
7		- - ntoskrnl.exe!SMKM_STORE_MGR<SM_TRAITS>::SmCompressCtxPro...	236	
8		- - - ntoskrnl.exe!RtlCompressBuffer	197	
9		- - - ntoskrnl.exe!RtlCompressBufferXpressLz	197	
10		- - - - ntoskrnl.exe!RtlCompressBufferXpressLzStandard	196	
11		- - - - - ntoskrnl.exe!RtlCompressBufferXpressLzStandard<itself>	195	
12		- - - - - ntoskrnl.exe!KiInterruptDispatchNoLockNoEtw	1	
13		- - - - - - ntoskrnl.exe!RtlCompressBufferXpressLz<itself>	1	
14		- ntoskrnl.exe!SMKM_STORE_MGR<SM_TRAITS>::SmCompressCtx...	16	
15		- ntoskrnl.exe!SMKM_STORE_MGR<SM_TRAITS>::SmCompressCtx...	14	
16		- ntoskrnl.exe!MmBuildMdlForNonPagedPool	5	

Figure 5.3: Screenshot from Windows Performance Analyzer, stack of the MemCompression process in the Flame by Process view.

5.3 Kernel Debugging with WinDbg

The Windows Debugger has functionality for examining a running Windows 10 system by utilizing remote kernel debugging. This technique requires a remote system, hereby referred to as "guest" and a debugging system, hereby referred to as "host". All commands used in this section is documented by Microsoft (Microsoft, 2017b).

5.3.1 Lab Setup

For the guest, a virtual machine with Windows 10 will be used. For the host, a physical machine running Windows 10 with WinDbg will be used.

The following setup is used.

HOST

Figure 5.4 shows the OS-version. The virtualization software used was VMWare Workstation 12.1.1 build-3770994. The version of WinDbg was 10.0.16299.91. The host is also set up with Windows Subsystem for Linux, which gives access to a Linux-console in Windows (Microsoft, 2016). This console is used in combination with the Powershell console throughout the experiments.

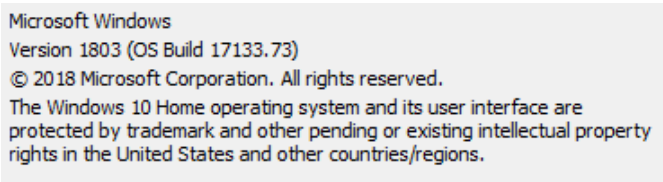


Figure 5.4: Windows 10 version on the host.

GUEST

Figure 5.5 shows the OS-version. The guest was configured with 2GB of memory to make the required memory pressure needed to force memory compression lower. To enable support for debugging over the network, it was necessary to activate it by running `bcdedit1` in a command prompt in Windows. The guest is also setup with Windows Subsystem for Linux, same as the host.

```
PS> bcdedit /dbgsettings net hostip:<ip-address> port:<desired-port>  
key:<key set by the user>
```

¹BCDEdit is a command-line tool for managing Boot Configuration Data (Microsoft, 2017a).

```
Microsoft Windows
Version 1709 (OS Build 16299.371)
© 2017 Microsoft Corporation. All rights reserved.
The Windows 10 Enterprise Evaluation operating system and its user
interface are protected by trademark and other pending or existing
intellectual property rights in the United States and other
countries/regions.
```

Figure 5.5: Windows 10 version on the guest.

5.3.2 Inspecting RtlCompressBuffer

In WinDbg, there is an option to set breakpoints on function calls. If the breakpoint hits on the correct function call, the arguments can be inspected by looking at the locations mentioned in Section ???. After attaching WinDbg to the kernel debugging interface, the debugger now controls the virtual machine. The virtual machine is "broken into" by the debugger, enabling a large number debugging operations. The interesting information should be found before the operating system makes a function call to RtlCompressBuffer with the MemCompression process. The "bp"-command can be run to see if the function is found by the debugger.

```
1: kd> x nt!RtlCompressBuffer*
fffff801'38841000 nt!RtlCompressBufferLZNT1 (void)
fffff801'38422530 nt!RtlCompressBufferXpressLz (void)
fffff801'384225a0 nt!RtlCompressBufferXpressLzStandard (void)
fffff801'38676aa4 nt!RtlCompressBufferProgress (<no parameter info>)
fffff801'38676d14 nt!RtlCompressBufferXpressHuffMax (<no parameter info>)
fffff801'38677b5c nt!RtlCompressBufferXpressLzMax (<no parameter info>)
fffff801'38676c60 nt!RtlCompressBufferXpressHuff (<no parameter info>)
fffff801'386d7978 nt!RtlCompressBufferProcs = <no type information>
fffff801'3867739c nt!RtlCompressBufferXpressHuffStandard (<no parameter
info>)
fffff801'38521290 nt!RtlCompressBuffer (<no parameter info>)
```

The output shows that the interesting function is present at offset fffff801'38521290, and some sub-functions that has names correlating with the background theory from Section 3.4. These functions are likely called depending on what arguments RtlCompressBuffer is given by the callee.

The next step is to set the breakpoint on the function. This is done with the "bp"-command.

```
1: kd> bp nt!RtlCompressBuffer
```

To list all breakpoints and look at their status, the "bl"-command is used

```
1: kd> bl
0 e Disable Clear fffff801'38521290 0001 (0001) nt!RtlCompressBuffer
```

The next step is to run the virtual machine until the breakpoint hits. This is done with the "g" command. Immediately following is information about what breakpoint hit, and the first instruction on within the function call.

```
1: kd> g
Breakpoint 0 hit
nt!RtlCompressBuffer:
fffff801`38521290 4883ec58 sub rsp,58h
```

As the RtlCompressBuffer can be used by other processes in Windows 10, it is necessary to find out if the breakpoint has occurred on the correct one. To see the process context of the current thread, the "!process"-command is used.

```
1: kd> !process
PROCESS fffffd9099ad3a040
SessionId: none Cid: 0640 Peb: 00000000 ParentCid: 0004
DirBase: 0c980002 ObjectTable: fffffab8c1ba6c540 HandleCount: 0.
Image: MemCompression
```

This is the correct function call to inspect. To look at the interesting information regarding the arguments, it is necessary to see what data the registers and the stack contains. The registers can be viewed with the "r"-command.

```
1: kd> r
rax=ffffd9099ad35000 rbx=ffffd90998d1e028 rcx=0000000000000003
rdx=ffffd9099d021f4f rsi=ffffd90997ca8b30 rdi=ffffd90998d1e070
rip=fffff80138521290 rsp=ffffeb80678b5b18 rbp=ffff82012790d000
r8=0000000000001000 r9=ffffd90998d1e070 r10=0000000000001000
r11=0000000000000000 r12=fffff801388379c0 r13=fffff80138837eb0
r14=ffffd90998d1e000 r15=ffffd9099d021f4f
```

There are two arguments that are of particular interest in this list, and that is RCX and R8. These represents the CompressionFormatAndEngine and UncompressedBufferSize, which are important parameters for decompressing data.

- **R8** is simple to decode. It displays the hex-value 0x1000, which translates to 4096 decimal. This means that the UncompressedBufferSize equals 4096. This is valuable information for creating the brute-force tool, as it reveals what the original size of the compressed data is.
- **RCX** has the value 0x3. This number needs to be decoded to find out what compression format and engine is used. One way of finding it is to look within the winnt.h header file which ships with Microsoft Visual Studio 2017, an integrated development environment for Windows.

```

19482 #define COMPRESSION_FORMAT_NONE (0x0000)
19483 #define COMPRESSION_FORMAT_DEFAULT (0x0001)
19484 #define COMPRESSION_FORMAT_LZNT1 (0x0002)
19485 #define COMPRESSION_FORMAT_XPRESS (0x0003)
19486 #define COMPRESSION_FORMAT_XPRESS_HUFF (0x0004)
19487 #define COMPRESSION_ENGINE_STANDARD (0x0000)
19488 #define COMPRESSION_ENGINE_MAXIMUM (0x0100)
19489 #define COMPRESSION_ENGINE_HIBER (0x0200)

```

Figure 5.6: Screen shot from Visual Studio showing contents from winnt.h.

As covered in Section 3.2, this parameter is represented by a bitmask. Figure 5.6 shows what values the bitmask consists of. Had the compression format been LZNT1 and the engine HIBER, the value of the bitmask would be 0x0202. The number 0x0003 translates to the combination of format type `COMPRESSION_FORMAT_XPRESS` and engine type `COMPRESSION_ENGINE_STANDARD`. These are essential parameters that needs to be passed correctly to the function when creating the decompression tool.

5.3.3 Finding the Decompression Function

As there is a function called "RtlDecompressBuffer", it is natural to assume that this function is involved in the process of decompressing the paged out data. When setting a breakpoint on this function in WinDbg, nothing happens. The operating system does not seem to be using this function, atleast not too frequently. It is possible to set breakpoints on multiple functions at the time with the "bm"-command. To see if what functions are available, the "x"-command is used.

```

3: kd> x nt!RtlDecompressBuffer*
fffff801'3842ed00 nt!RtlDecompressBufferXpressHuff (void)
fffff801'38925cc0 nt!RtlDecompressBufferLZNT1 (void)
fffff801'38678ba0 nt!RtlDecompressBufferXpressLzProgress (<no parameter info>)
fffff801'38678144 nt!RtlDecompressBufferProgress (<no parameter info>)
fffff801'38502ce0 nt!RtlDecompressBufferEx2 (<no parameter info>)
fffff801'386d7858 nt!RtlDecompressBufferProcs = <no type information>
fffff801'3842ec80 nt!RtlDecompressBufferEx (<no parameter info>)
fffff801'3866f330 nt!RtlDecompressBuffer (<no parameter info>)
fffff801'38678204 nt!RtlDecompressBufferXpressHuffProgress (<no parameter info>)
fffff801'386787e0 nt!RtlDecompressBufferXpressLz (<no parameter info>)

```

To see if any of the other decompression functions are used by the operating system, breakpoints are placed on all of them.

```

3: kd> bm nt!RtlDecompressBuffer*
1: fffff801'3842ed00 @"nt!RtlDecompressBufferXpressHuff"

```

```
2: fffff801`38925cc0 @!"nt!RtlDecompressBufferLZNT1"
3: fffff801`38678ba0 @!"nt!RtlDecompressBufferXpressLzProgress"
4: fffff801`38678144 @!"nt!RtlDecompressBufferProgress"
5: fffff801`38502ce0 @!"nt!RtlDecompressBufferEx2"
6: fffff801`3842ec80 @!"nt!RtlDecompressBufferEx"
7: fffff801`3866f330 @!"nt!RtlDecompressBuffer"
8: fffff801`38678204 @!"nt!RtlDecompressBufferXpressHuffProgress"
9: fffff801`386787e0 @!"nt!RtlDecompressBufferXpressLz"
```

When running the virtual machine, the breakpoint immediately hits.

```
3: kd> g
Breakpoint 6 hit
nt!RtlDecompressBufferEx:
fffff801`3842ec80 4053 push rbx
```

It is also necessary to make sure that this function is used by the MemCompression process.

```
3: kd> !process
PROCESS fffffd9099ad3a040
SessionId: none Cid: 0640 Peb: 00000000 ParentCid: 0004
DirBase: 0c980002 ObjectTable: fffffab8c1ba6c540 HandleCount: 0.
Image: MemCompression
```

The MemCompression process is using RtlDecompressBufferEx to perform decompression. The next step is to extract the compressed data and the decompressed data, then try to decompress the compressed data and get the exact same result as the MemCompression process.

5.3.4 Inspecting RtlDecompressBufferEx

To verify that the correct parameters were found in Section 5.3.2, the same inspection will be performed on RtlDecompressBufferEx. The debugger is set on the breakpoint before RtlDecompressBufferEx, so the "r"-command is used to view the registers.

```
1: kd> r
rax=000000000000001ca rbx=0000000000000002 rcx=0000000000000003
rdx=ffffe7811d1bd000 rsi=00000000029c9e60 rdi=ffffe7811d1bd000
rip=fffff801f0a91990 rsp=ffffbf0968c5a278 rbp=ffffc18980496050
r8=0000000000001000 r9=00000000029c9e60 r10=7fffffffefefefefc
r11=ffffbf0968c5a3a8 r12=00000000029c9e60 r13=ffffc1897e5a9000
r14=ffffe7811d1bd000 r15=ffffbf0968c5a4a8
```

Section 3.4.3 shows which arguments are passed to the function.

- **RCX** is 0x3, which is decoded with Figure 5.6, and gives the same compression format as RtlCompressBuffer, COMPRESSION_FORMAT_XPRESS.
- **RDX** is the UncompressBuffer. This register contains the kernel address a location in memory that will receive the data that is decompressed by the function.
- **R8** is the UncompressedBufferSize. The value of this register is 0x1000. This means that the compressed data will decompress to 4096 bytes. This value also correlates with RtlCompressBuffer.
- **R9** is the virtual address to the compressed data in the MemCompression process.

The remaining arguments are passed on the stack before entering the function call. These are found by looking at the instructions before the function call to RtlDecompressBufferEx. In fast call, as mentioned in Section 3.4.4, the function that calls RtlDecompressBufferEx is responsible for passing the arguments on the stack.

The registers show that the same arguments passed to RtlDecompressBufferEx were also passed to RtlCompressBuffer. This finding makes it highly likely that these are the respective functions used to compress and decompress data to the MemCompression process.

5.3.5 Extracting Data for Testing

To create a tool that decompresses data from a memory dump, it is necessary to start in the most basic form. Using the RtlDecompressBufferEx-function to get a "before-and-after-shot" of the data it decompresses. The goal is to extract this data, and replicate the process of decompression with a program. To get this data, three parameters are required. The UncompressBuffer, the CompressedBuffer, and the CompressedBufferSize. The first two have been shown in previous sections, and is found in registers **RDX** and **R9**. The last one is found on the stack. As mentioned in Section 5.3.4, this can be found by looking at the instructions just before the function call to RtlDecompressBufferEx. In WinDbg this is done by looking at the instructions before the return address of the RtlDecompressBufferEx. The "k"-command can be used to look at the call-stack, and find the return address to the caller of the decompression-function.

```
1: kd> k
Child-SP          RetAddr          Call Site
00 fffffbf09`68c5a278 ffffff801`f0cec3e2 nt!RtlDecompressBufferEx
01 fffffbf09`68c5a280 ffffff801`f0cec6ae nt!ST_STORE<SM_TRAITS>::
StDmSinglePageCopy+0x246
```

The RetAddr is then used to inspect the preceding assembly instructions. This is done using the "ub"-command with the return address and

```
1: kd> ub ffffff801`f0cec3e2 L10
nt!ST_STORE<SM_TRAITS>::StDmSinglePageCopy+0x1fd:
```

```

fffff801f0cec3af 0f83a4000000    jae     nt!ST_STORE<SM_TRAITS>::
StDmSinglePageCopy+0x2bd (fffff801`f0cec459)
fffff801f0cec3b5 488b4c2468      mov     rcx,qword ptr [rsp+68h]
fffff801f0cec3ba 448bc2          mov     r8d,edx
fffff801f0cec3bd 48894c2430      mov     qword ptr [rsp+30h],rcx
fffff801f0cec3c2 4c8bce          mov     r9,rsi
fffff801f0cec3c5 488d4c2458      lea    rcx,[rsp+58h]
fffff801f0cec3ca 488bd7          mov     rdx,rdi
fffff801f0cec3cd 48894c2428      mov     qword ptr [rsp+28h],rcx
fffff801f0cec3d2 0fb78de0030000 movzx   ecx,word ptr [rbp+3E0h]
fffff801f0cec3d9 89442420        mov     dword ptr [rsp+20h],eax
fffff801f0cec3dd e8ae55daff      call   nt!RtlDecompressBufferEx
(fffff801f0a91990)

```

Since the stack follows the "Last In First Out" (Knuth, 1997) principal and CompressedBufferSize-argument is the fifth, and thus the first stack-argument to be used, it is passed last on the stack. The last argument to be passed to the stack is the instruction at offset fffff801f0cec3d9, mov dword ptr [rsp+20h], eax. This happens just before the call to RtlDecompressBufferEx. Tests have shown that this argument mostly can be found in the **RAX**-register, as the value comes from the **EAX**-register, which represents the lower 32-bits of RAX.

To see the value of the CompressedBufferSize-argument, the "dq"-command is used on the offset of the stack pointer + 0x28. The stack pointer register, RSP (Microsoft, 2018b), contains the address of the stack at the current instruction. The instruction says that the value of eax is moved to rsp+20h, but when the call to RtlDecompressBufferEx happens, which is where the breakpoint is set, the stack is incremented with the 64-bit address of the return pointer. This means that the value placed on the stack at rsp+20h, is now at rsp+28h, since a 64-bit address takes 8 bytes. The proper "dq"-command will then be:

```

1: kd> dq @rsp+28 L1
ffffbf09`68c5a2a0 00000000`000002bb

```

The compressed data can be extracted to a file on disk. This can be achieved through the .writemem-command. The length is 2bb, represented by L2bb in the command.

```

1: kd> .writemem C:\Users\User\MASTER-THESIS-TESTING\test-dbg-compressed.bin
@r9 L2bb
Writing 2bb bytes.

```

To extract the decompressed data, RtlDecompressBufferEx must first finish, as its job is to decompress the data. The Windows debugger has a feature called "Step Out". This runs the current function until it returns to the function that called it. When this is done when the breakpoint is at the RtlDecompressBufferEx-function, the pointer to the address of UncompressedBuffer in the **RDX**-register now contains the uncompressed data. The .writemem-command is used to extract the data, and as seen by inspecting the function, the size of the UncompressedBuffer is 0x1000 or 4096 bytes. The output of the debugger says "Writing 1000 bytes.", but this value is misleading and should be "1000h" or "0x1000" as it is hexadecimal.

```
1: kd> .writemem C:\Users\User\MASTER-THESIS-TESTING\test-dbg-uncompressed.bin
ffffb5802c67c000 L1000
Writing 1000 bytes.
```

Powershell `dir`-command is used to look at the folder and verify that the correct sizes has been written to the disk.

```
PS C:\Users\User\MASTER-THESIS-TESTING> dir

Mode                Length Name
----                -
-a-----          699 test-dbg-compressed.bin
-a-----         4096 test-dbg-uncompressed.bin
```

5.3.6 Decompressing Testing Data

The test-files from the previous section will be tested in two ways. The compressed test-file will be attempted decompressed by a program using the Windows function `RtlDecompressBuffer`. The uncompressed test-file will be attempted compressed by the same program, but by using the `RtlCompressBuffer`-function. The results will show if this is a valid method that can be developed further to work on page files and memory dumps. If the output-files from the program are identical to the result of the `MemCompression`-process, the test is a success. This will be verified using `md5-hash`.

The program used for this testing is a slightly configured version of C++ code found online (AceFinity, 2014). The actual code used is found in listing 8.2, in Appendix B. The program was appropriately named "CompressDecompress" as it can do both.

For testing decompression, the file "test-dbg-compressed.bin" from the previous section was used as input. The output file was named "test-tool-decompressed.bin".

```
PS C:\Users\User\MASTER-THESIS-TESTING> .\CompressDecompress.exe
PS C:\Users\User\MASTER-THESIS-TESTING> dir

Mode                Length Name
----                -
-a-----          699 test-dbg-compressed.bin
-a-----         4096 test-dbg-uncompressed.bin
-a-----         4096 test-tool-decompressed.bin
```

The size 4096 matches, so to test that the files are identical `md5sum` is used.

```
PS C:\Users\User\MASTER-THESIS-TESTING> bash -c "md5sum *"
15448c64a3d3c522fbfc312326beebab test-dbg-compressed.bin
11a112a873db64cb2fa61b498782f278 test-dbg-uncompressed.bin
11a112a873db64cb2fa61b498782f278 test-tool-decompressed.bin
```

The `CompressDecompress`-tool has successfully decompressed data in the same way as the `MemCompression`-process.

Next is comparing compressed data from RtlCompressBuffer to compressed data from the MemCompression process to ensure that they operate identically. The same procedure as with decompression will be performed. To compress instead of decompress, a slight change is necessary in the code of the CompressDecompress-tool. At line 143 in the code, the line "#define COMPRESS" must be de-commented by removing the two forward slashes at the beginning. Also, the file names must be changed to match the compression operation. The input file to be compressed in this experiment is "test-dbg-uncompressed.bin", and the output file is named "test-tool-compressed.bin".

```
PS C:\Users\User\MASTER-THESIS-TESTING> .\CompressDecompress.exe
PS C:\Users\User\MASTER-THESIS-TESTING> dir

Mode                Length      Name
-----
-a----             699  test-dbg-compressed.bin
-a----             699  test-tool-compressed.bin
-a----            4096  test-dbg-uncompressed.bin
-a----            4096  test-tool-decompressed.bin
```

The size 699 matches, so to test that the files are identical md5sum is used.

```
PS C:\Users\User\MASTER-THESIS-TESTING> bash -c "md5sum *"
15448c64a3d3c522fbfc312326beebab test-dbg-compressed.bin
15448c64a3d3c522fbfc312326beebab test-tool-compressed.bin
11a112a873db64cb2fa61b498782f278 test-dbg-uncompressed.bin
11a112a873db64cb2fa61b498782f278 test-tool-decompressed.bin
```

The testing of RtlDecompressBuffer and RtlCompressBuffer is successful, and can be used to decompress data from the MemCompression-process.

5.3.7 Testing LZ77

To make sure what algorithm is actually used by RtlDecompressBuffer and RtlCompressBuffer, some testing was necessary. Microsoft has documented how the algorithms are used (Microsoft, 2018e), so to test for LZ77 the same example is used. Figure 5.7 shows an example where LZ77 is used on some data. The CompressDecompress.exe tool will be used to reproduce the example. If CompressDecompress.exe produces the same result with RtlCompressBuffer given the parameters gathered through kernel debugging, it concludes that LZ77 is used.

The file "abc.txt" was created, with consecutive abc's.

```
PS> Get-Content .\abc.txt
abcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcabcbcb
```

The code used in CompressDecompress.cpp to make this happen, was:

as the practical example of "Plain LZ77" in Figure 5.7. This experiment concludes that plain LZ77 is the correct algorithm.

5.4 Proof of Concept Tool: MemoryDecompression

MemoryDecompression uses a brute-force approach to decompress memory pages that has been compressed by Windows 10. For now, the tool can only be run on a Windows 8 or Windows 10 client, as the `ntdll.dll` function used to decompress, `RtlDecompressBuffer`, is only available on those versions. It is a console application, which means it runs in either a command prompt or Powershell console in Windows. By making it a console application, it is easier to integrate in scripts or other programs used to automate the forensics preprocessing. Had the tool been written as a Graphical User Interface application, automation through scripting would be challenging.

Usage:

```
MemoryDecompression.exe <input file or folder> <output-file>
```

Input file or folder

The input parameter can be either a file or a folder. If the input parameter is a folder, the tool will attempt to decompress ALL the files it contains. Investigators can use a tool like Volatility to extract the compressed memory from the MemCompression process. Output files from both `vaddump` and `memdump` (Volatility plugins) works. The intended purpose of using a folder as input is for the `vaddump`-plugin output, which dumps VAD-segments in 128kb files. The investigator can also extract page files (`pagefile.sys`) from a disk image and run the decompressor on the file. Tests have shown that this can take over 4 hours on a 6GB page file with the current version of MemoryDecompression.

Output file

The output parameter specifies the file which the decompressed output data is written to.

5.4.1 Choice of Programming Language

Computer operations that involves compressing or decompressing data is resource demanding. It is desirable to utilize a low-level programming language to operate as effective as possible. As Volatility is written in Python, and one of the goals of the decompression-tool was integration with the existing solutions, Python was considered. The issue of performance was addressed in "In Lieu of Swap" (Richard, 2014a), which makes Python a poor candidate. It was also convenient to use a programming language that could easily utilize Windows API functions.

Taking the arguments into consideration, the programming language chosen was C++.

5.4.2 Sample Output

The MemoryDecompression tool is meant for decompressing compressed contents in memory dumps, page files, and hibernation files.

Example with page file as input:

```
PS> MemoryDecompression.exe pagefile.sys pagefile-decompressed.bin
```

```
Decompressing .\pagefile.sys
```

```
Total decompressed pages: 803827
Total compressed data: 1111505498 bytes
Total decompressed data: 3292475392 bytes
```

```
Decompression completed in:
```

```
Total Microseconds: 16731799969
Total Seconds: 16731
Total Minutes: 278
Total Hours: 4
```

This is an example from a real page file, and it shows how long it takes to decompress contents in a large file.

Example with folder of VAD-segments as input:

The tool works directly on a memory dump, but it will take a lot of time, so we want to extract the compressed data from the MemCompression process within the memory dump using Volatility. Outputs are minimized for readability. Firstly, Volatility is used on the memory dump to find the process ID of the MemCompression process.

```
volatility -f memory.dmp --profile=Win10x64.16299 pslist
```

```
Name PID Start
```

```
-----
```

```
MemCompression 1708 2018-08-24 07:38:02 UTC+0000
```

The PID is used to dump the VAD-segments of the process, containing the compressed data.

```
volatility -f memory.dmp --profile=Win10x64.16299 vaddump -p 1708
-D vaddump-folder/
```

The folder containing the VAD-segments can now be used as input for the MemoryDecompression-tool.

```
PS> MemoryDecompression.exe vaddump-folder all-vads-decompressed.bin
Decompressing MemCompression.4afa580.0x000000000000b0000-0x000000000000cffff.dmp
Decompressing MemCompression.4afa580.0x000000000000d0000-0x000000000000effff.dmp
Decompressing MemCompression.4afa580.0x000000000000f0000-0x0000000000010ffff.dmp
Decompressing MemCompression.4afa580.0x00000000000a10000-0x00000000000a2ffff.dmp
```

```
Decompressing MemCompression.4afa580.0x0000000000a30000-0x0000000000a4ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000a50000-0x0000000000a6ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000a70000-0x0000000000a8ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000a90000-0x0000000000aaffff.dmp
Decompressing MemCompression.4afa580.0x0000000000ab0000-0x0000000000acffff.dmp
Decompressing MemCompression.4afa580.0x0000000000ad0000-0x0000000000aeffff.dmp
Decompressing MemCompression.4afa580.0x0000000000af0000-0x0000000000b0ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000b10000-0x0000000000b2ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000b30000-0x0000000000b4ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000b50000-0x0000000000b6ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000b70000-0x0000000000b8ffff.dmp
```

```
Total decompressed pages: 1239
Total compressed data: 1789658 bytes
Total decompressed data: 5074944 bytes
```

```
Decompression completed in:
Total Microseconds: 12657726
Total Seconds: 12
Total Minutes: 0
Total Hours: 0
```

5.4.3 How MemoryDecompression Works

To decompress contiguous compressed data from a file, MemoryDecompression uses a brute-force approach. When a page is compressed on a normal system, the size of the page being compressed is always 4096. This means that the decompressed size that is returned by the RtlDecompressBuffer function should always be 4096. When the tool successfully decompresses a page, it must jump to the next compressed data within the input file. As mentioned in Section 3.3, the compressed data is stored in 16 byte chunks. This means that when decompression is successful, the next compressed data starts at the next 16 byte chunk. The offset of the next compressed data will therefore be the next number divisible by 16. **For example** if MemoryDecompression successfully decompressed 900 bytes from the beginning of the input file, the offset of the next compressed data will not start at offset 901, but at the next chunk at offset 912.

Simplified pseudo code:

```
Check that the parameters are correct, quit if they aren't.
Read the input file or files.
If the input is a folder, do decompression on all files in the folder.
Read first 4096 bytes of the file and checks if the buffer is all zeros
    If the buffer is all zeros, jumps 4096 bytes - To avoid wasting time
    ↪ decompressing zeros.
Start reading an incrementing amount of bytes from the beginning of the
    ↪ file into a buffer
    Use RtlDecompressBuffer with Xpress format and standard engine, with
    ↪ the buffer as the contents of the argument CompressedBuffer
```

If the decompression succeeds AND the Uncompressed size equals 4096
Write the decompressed data to the output file, and jump to the
↳ next chunk offset
Else, increment the buffer by 1 byte and try again.
If the buffer reaches 4096 bytes without successful decompression,
↳ moves the start offset 16 bytes and tries again.
When the end of the file is reached, print statistics - Total duration of
↳ running the tool, and total compressed and decompressed data.

5.5 Testing MemoryDecompression

This section contains the parameters for testing how console and web browser activity can be hidden by the MemCompression-process. As Section 4.2 states, console and web browser activity are important pieces of evidence that can reveal crucial information in a digital investigation. Console activity is tested by typing some commands that can easily be found in a memory dump. Browser activity is tested by visiting a website with with a web browser that is easily found in a memory dump. After creating the evidence, the goal is to simulate other activity with a tool called "TestLimit.exe" (Microsoft, 2012). This tool has the option to create a high memory demand, which should force the Windows 10 memory manager to compress pages from idle processes to make space. A memory snapshot is taken after running TestLimit.exe, and Volatility is used to extract the VAD-segments. The MemoryDecompression tool is then used on the VAD-segments to decompress the obfuscated data. After the data is decompressed, string searches for the activity on the compressed and decompressed VAD-segments is performed. In the browser experiment, the page file is also included.

The same virtual machine used in Section 5.3 was used when testing MemoryDecompression.

5.5.1 Scenario: Console Activity

The following console activities were performed on the virtual machine.

- Opened Bash.exe
- Typed `'echo "Hello Olashamann!"'`
- Opened Powershell.exe
- Typed `'echo "Hello Duriell!"'`
- Ran `TestLimit.exe -m 100` in the Powershell console
- Memory snapshot taken

5.5.2 Scenario: Browser Activity

The following web browser activities were performed on the virtual machine.

- Opened web browser Microsoft Edge
- Went to nrk.no, and clicked an article containing the word, "farenivået"
- Memory snapshot 1 taken
- Opened Powershell.exe
- Ran `TestLimit.exe -m 100`
- Memory snapshot 2 taken

Figure 5.8 shows a screenshot taken from the virtual machine, showing how the web site looked for the user.



Figure 5.8: Screenshot from the web browser activity.

Chapter 6

Results

This chapter contains the results produced by testing the MemoryDecompression tool on Windows 10 memory data. The findings are briefly discussed in both experiments.

6.1 Results: Console Activity

This section contains the results of the scenario described in Section 5.5.1. The first step of decompressing memory is extracting the VAD-segments from the MemCompression process. This can be done using the `vaddump`-plugin available for Volatility. To use this plugin, the process ID of the MemCompression-process is needed. This can be found by running Volatility with the `pslist`-plugin. The output is trimmed to only show the relevant information.

```
# volatility -f Windev1802Eval-214c0030.vmem --profile=Win10x64.16299
pslist
Name          PID  Start
-----
MemCompression 1536 2018-10-27 13:37:02 UTC+0000
```

The `vaddump`-plugin can now be run on the process ID 1536. The content is extracted to a specific directory with the `-D`-parameter.

```
# volatility -f Windev1802Eval-214c0030.vmem --profile=Win10x64.16299
vaddump -p 1536 -D Duriell-Olashamann-vaddump
```

To get the size of the compressed data, the command `du -h` command is used. This shows Disk Usage in "M", which means Mega Bytes. The amount of VAD-segments are found by listing the directory contents with `ls -l`, using `grep` to make sure only relevant lines are counted, and `wc -l` to count the amount of lines printed. # `du -h`

```
Duriell-Olashamann-vaddump/
494M Duriell-Olashamann-vaddump/
```

```
# ls -l Duriell-Olashamann-vaddump | grep MemComp | wc -l
3947
```

MemoryDecompression.exe is then run in a PowerShell-console on the 3947 VAD-files. The tool prints the file it is currently decompressing to the console output, so the output is trimmed.

```
PS C:\Users\User\MASTER-THESIS-TESTING> .\MemoryDecompression.exe
.\Duriell-Olashamann-vaddump\.\Duriell-Olashamann-all-vads-decompressed
Decompressing MemCompression.4afa580.0x0000000000010000-0x000000000002ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000030000-0x000000000004ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000050000-0x000000000006ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000070000-0x000000000008ffff.dmp
Decompressing MemCompression.4afa580.0x0000000000090000-0x00000000000affff.dmp
Decompressing MemCompression.4afa580.0x00000000000b0000-0x00000000000cffff.dmp
...
Decompressing MemCompression.4afa580.0x000000001f0f0000-0x000000001f10ffff.dmp
Decompressing MemCompression.4afa580.0x000000001f110000-0x000000001f12ffff.dmp
Decompressing MemCompression.4afa580.0x000000001f130000-0x000000001f14ffff.dmp
Decompressing MemCompression.4afa580.0x000000001f150000-0x000000001f16ffff.dmp
Decompressing MemCompression.4afa580.0x000000001f170000-0x000000001f18ffff.dmp
Decompressing MemCompression.4afa580.0x000000001f190000-0x000000001f1affff.dmp
Decompressing MemCompression.4afa580.0x000000001f1b0000-0x000000001f1cffff.dmp

Total decompressed pages: 340227
Total compressed data: 462025832 bytes
Total decompressed data: 1393569792 bytes
```

```
Decompression completed in:
Total Microseconds: 3864607933
Total Seconds: 3864
Total Minutes: 64
Total Hours: 1
```

The output shows that it took 1 hour and 4 minutes to complete the decompression. It also shows that 462025832 bytes of compressed data was found, and it decompressed to 1393569792 bytes of decompressed data.

$$\frac{\text{Total compressed bytes}}{\text{Total decompressed bytes}} = \frac{462025832 B}{1393569792 B} = 0.3315 \approx 33\%$$

This means that the data in this experiment is compressed to approximately 33% of the original size by the memory manager, which corresponds to the numbers 30-50% from Windows Internals 7 mentioned in Section 3.3.

To determine if potentially important artifacts is obfuscated as a result of memory compression, a string search for the relevant words will be conducted. The results from

the compressed data will be compared with the results from the decompressed data.

6.1.1 String search on compressed data

The string search is conducted using both unicode and ascii characters. Unicode is accounted for with the `-el` parameter passed to `strings`. The keywords that is expected to be found are searched for with `grep -E 'Duriell|Olashamann'`. This tells `grep` to use regular expression, and return results that contains "Duriell" OR "Olashamann".

```
# strings -el Duriell-Olashamann-vaddump/* | grep -E 'Duriell|Olashamann'
Duriell

# strings Duriell-Olashamann-vaddump/* | grep -E 'Duriell|Olashamann'
"Hello Olashamann!"
echo "Hello Olashamann!"
$ echo "Hello Olashamann!"
/cho "Hello Olashamann!"b
```

The string search returned one hit on "Duriell" with Unicode-characters, and four hits on "Olashamann" with ASCII-characters.

6.1.2 String search on the decompressed data

The same process is conducted on the decompressed data as in Section 6.1.1.

```
# strings -el Duriell-Olashamann-all-vads-decompressed | grep -E
'Duriell|Olashamann'

Hello Olashamann!
Duriell
Duriell
Hello Duriell!
"Hello Duriell!@"
"Hello Duriell
"Hello Duriell!:
Hello Duriell!
Hello Duriell!
"Hello Duriell!@"
Hello Duriell!
promptDuriell!
echo "Hello Duriell!"
echo "Hello Duriell
Hello Duriell
Hello Duriell
Hello Duriell
```

```
Hello Duriell
echo "Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
echo "Hello Duriell!@
Hello Duriell!@
Hello Duriell!@
Hello Duriell!@
Hello Duriell!@
echo "Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
echo "Hello Duriell!:
Hello Duriell!:
Hello Duriell!:
Hello Duriell!:
Hello Duriell!:
echo "Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
echo "Hello Duriell!"
Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
echo "Hello Duriell!"
Hello Duriell!
Hello Duriell!
Hello Duriell!
Hello Duriell!
echo "Hello Duriell!"
echo "Hello Duriell!"
echo "Hello Duriell!"
promptDuriell!
Hello Duriell!
Hello Duriell!
Windows PowerShell Copyright (C) Microsoft Corporation. All rights
reserved.
```

```

PS C:\Users\User> echo "Hello Duriell!"
Hello Duriell!
PS C:\Users\User>
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo\root" fo
user@WinDev1802Eval:/mnt/c/Windows/System32$
user@WinDev1802Eval:/mnt/c/Windows/System32$
user@WinDev1802Eval:/mnt/c/Windows/System32$ echo "Hello Olashamann!"
Hello Olashamann!
user@WinDev1802Eval:/mnt/c/Windows/System32$

# strings Duriell-Olashamann-all-vads-decompressed | grep -E
'Duriell|Olashamann'

cho "Hello Olashamann"
"Hello Olashamann!"
echo "Hello Olashamann!"
[00m$ echo "Hello Olashamann!"
echo "Hello Duriell!"
echo "Hello Duriell!"
[DHello Olashamann!
Hello Olashamann!
echo "Hello Olashamann!"
echo "Hello Olashamann!"
echo "Hello Olashamann!"

```

The string searches returned 60 hits for Unicode strings and 11 hits for ASCII on the uncompressed data. *Note that the last eleven lines of the Unicode output are two lines, but the output is slightly prettified for visual purposes.*

6.1.3 Discussion

The results clearly shows that data is being obfuscated by the memory manager. An interesting finding is that Unicode strings seems to be more "vulnerable" to obfuscation, as the compressed content only contains one result for Unicode and the decompressed content contains 60. This could be related to how the compression algorithm works. When ASCII words are used by the compression algorithm, they remain in cleartext. This may work differently with Unicode characters.

The Unicode output from the uncompressed data also gives some concrete indications that a console was used type the commands. The decompressed strings shows the message displayed when opening a Powershell console. The string `PS C:\Users\User> echo "Hello Duriell!"` shows the Powershell command being executed. This data was not available in compressed form, and shows that decompressing memory gives valuable information. Ubuntu console usage can also be found. The decompressed string `user@WinDev1802Eval:/mnt/c/Windows/System32$ echo`

"Hello Olashamann!" shows indications of a Linux console being used. It also gives some valuable information about what user was logged on and executing commands.

There are a lot of uncertainty factors with this experiment that will be discussed in Section 7.

6.2 Results: Browser Activity

This section contains the results of the scenario described in Section 5.5.2.

In this experiment, two memory dumps were extracted to see how much data was obfuscated by simulating user activity. In addition, the VAD-segments from the Mem-Compression process were extracted and decompressed. The page file was also extracted and decompressed after the simulation. One before the TestLimit.exe tool was run, and one after. To find the relevant string, Strings.exe from SysInternals (Russinovich, 2018) was used. In the previous experiment, Ubuntu in Windows was used to present the results. In this experiment, Powershell is used. Testing with slightly different tools that supposedly does the same job is desirable, as the tools potentially could be a source of misinformation.

The three sources of evidence used in this experiment are:

- **Before-compress-Windev1802Eval-c165348a.vmem**
- **After-compress-Windev1802Eval-c165348a.vmem**
- **pagefile.sys** after TestLimit.exe was run

6.2.1 Before TestLimit.exe

To find out how many strings related to the browser activity that can be found, a few different commands were used. First, the strings.exe tool was run and the output written to the file **"before-strings.txt"**.

```
PS> strings.exe Before-compress-Windev1802Eval-c165348a.vmem >
↪ before-strings.txt
```

All strings containing "fareniv" is then extracted from **"before-strings.txt"**, and written to the file **"before-strings-fareniv.txt"**

```
PS> Get-Content .\before-strings.txt | Select-String fareniv >
↪ .\before-strings-fareniv.txt
```

To find the amount of lines in the file **"before-strings-fareniv.txt"**, Powershell command Measure-Object -Line is used. The output shows that 706 lines contained the relevant string. The results of all the string searches is presented in table 6.1 in Section 6.2.4.

```
PS> Get-Content .\before-strings-fareniv.txt | Measure-Object -Line
```

```
Lines Words Characters Property
-----
706
```

In addition to strings from the memory snapshot, the VAD-segments from the MemCompression process were also extracted and decompressed using the MemoryDecompression tool. This was executed in the same way as described in Section 6.1.

```
# volatility -f Before-compress-Windev1802Eval-c165348a.vmem
```

```

--profile=Win10x64.16299 pslist
Name          PID    Start
-----
MemCompression 1708 2018-10-28 13:30:49 UTC+0000

# volatility -f After-compress-Windev1802Eval-c165348a.vmem
--profile=Win10x64.16299 vaddump -p 1708 -D vaddump-before/

```

MemoryDecompression was run on the vaddump-before folder. Output file named **vaddump-before-decompressed.bin**.

```

PS> .\MemoryDecompression.exe .\vaddump-before
↳ vaddump-before-decompressed.bin

```

The relevant strings were then extracted.

```

PS> strings.exe vaddump-before-decompressed.bin >
↳ vaddump-before-decompressed-strings.txt
PS> Get-Content vaddump-before-decompressed-strings.txt | Select-String
↳ fareniv > .\vaddump-before-decompressed-strings-fareniv.txt
PS> Get-Content .\vaddump-before-decompressed-strings-fareniv.txt |
↳ Measure-Object -Line

```

```

Lines Words Characters Property
-----
42

```

Strings was also run on the VAD-segments in compressed form.

```

PS> strings.exe vaddump-before/* > vaddump-before-strings.txt
PS> Get-Content vaddump-before-strings.txt | Select-String fareniv >
↳ .\vaddump-before-strings-fareniv.txt
PS> Get-Content .\vaddump-before-strings-fareniv.txt | Measure-Object
↳ -Line

```

```

Lines Words Characters Property
-----
3

```

6.2.2 After TestLimit.exe

The exact same procedure is executed on the memory snapshot after TestLimit.exe is run. The files containing the strings were named **"after-strings.txt"** and **"after-strings-fareniv.txt"**

```

PS> strings.exe After-compress-Windev1802Eval-c165348a.vmem >
↳ after-strings.txt
PS> Get-Content after-strings.txt | Select-String fareniv >
↳ .\after-strings-fareniv.txt
PS> Get-Content .\after-strings-fareniv.txt | Measure-Object -Line

```

```

Lines Words Characters Property
-----
172

```

The amount of lines containing "fareniv" are significantly reduced after TestLimit.exe is run.

In addition to strings from the memory snapshot, the VAD-segments from the MemCompression process were also extracted and decompressed using the MemoryDecompression tool. This was executed in the same way as described in Section 6.1.

```
# volatility -f After-compress-Windev1802Eval-cl65348a.vmem
--profile=Win10x64-16299 vaddump -p 1708 -D vaddump-after/
```

MemoryDecompression was run on the vaddump-after folder. Output file named **vad-segments-decompressed.bin**.

```
PS> .\MemoryDecompression.exe .\vaddump-after
↪ vaddump-after-decompressed.bin
```

```
PS> Get-Content .\decompressed-strings-fareniv.txt
| Measure-Object -Line
```

```
Lines Words Characters Property
-----
372
```

6.2.3 Pagefile.sys

To analyze the page file, it must first be extracted from the hard drive of the virtual machine. To do this, tools from The Sleuthkit will be used. Mmls is used for listing the partitions.

```
PS> mmls.exe Windev1802Eval-disk1.vmdk
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-----	0000000000	0000002047	0000002048	Unallocated
002:	000:000	0000002048	0001126399	0001124352	NTFS / exFAT (0x07)
003:	000:001	0001126400	0266336255	0265209856	NTFS / exFAT (0x07)
004:	-----	0266336256	0266338303	0000002048	Unallocated

The partition containing the data is the second NTFS-partition. To list the root directory of the hard disk file, fls.exe is used with the -o parameter specifying what sector offset to start reading from.

```
PS> fls.exe -o 0001126400 Windev1802Eval-disk1.vmdk
r/r 4-128-1: $AttrDef
r/r 8-128-2: $BadClus
r/r 8-128-1: $BadClus:$Bad
r/r 6-128-4: $Bitmap
r/r 7-128-1: $Boot
d/d 11-144-4: $Extend
r/r 2-128-1: $LogFile
r/r 0-128-6: $MFT
r/r 1-128-1: $MFTMirr
```

```

d/d 58-144-1:  $Recycle.Bin
r/r 9-144-17:  $Secure:$SDH
r/r 9-144-16:  $Secure:$SII
r/r 9-128-18:  $Secure:$SDS
r/r 10-128-1:  $UpCase
r/r 10-128-4:  $UpCase:$Info
r/r 3-128-3:   $Volume
d/d 85484-144-1: Documents and Settings
r/r 336645-128-1: pagefile.sys
d/d 59-144-1:  PerfLogs
d/d 60-144-6:  Program Files
d/d 1151-144-6: Program Files (x86)
d/d 1251-144-6: ProgramData
d/d 85665-144-1: Recovery
d/d 117052-144-1: samples
d/d 59919-144-8: scripts
r/r 39847-128-1: swapfile.sys
d/d 52343-144-6: System Volume Information
d/d 1407-144-5: Users
d/d 1464-144-6: Windows
V/V 341504:    $OrphanFiles

```

The file is extracted using the `icat-tool` with the inode-number of the `pagefile.sys`, **336645-128-1**.

```

PS> icat.exe -o 0001126400 Windev1802Eval-disk1.vmdk 336645-128-1 >
↪ pagefile.sys

```

To view the size of the file, the `dir`-command is used. The `-hidden` parameter is necessary as the `pagefile.sys` is hidden by default in Windows.

```

PS> dir -hidden pagefile.sys

```

Mode	LastWriteTime	Length	Name
-a-hs-	29.10.2018 12.33	6442450944	pagefile.sys

The page file is approximately 6GB. `MemoryDecompression` is then run on the page file.

```

PS > .\MemoryDecompression.exe .\pagefile.sys
↪ .\page-file-decompressed.bin
Decompressing .\pagefile.sys

```

```

Total decompressed pages:      803827
Total compressed data:         1111505498 bytes
Total decompressed data:       3292475392 bytes

```

```

Decompression completed in:
Microseconds: 16731799969
Seconds:      16731
Minutes:      278
Hours:        4

```

MemoryDecompression spent 4 hours and 38 minutes to decompress approximately 1GB of compressed data to 3GB of decompressed data.

$$\frac{\text{Total compressed bytes}}{\text{Total decompressed bytes}} = \frac{1111505498 \text{ B}}{3292475392 \text{ B}} = 0.3376 \approx 34\%$$

This experiment also corresponds to the tested compression rate of 30-50% from Windows Internals 7 mentioned in Section 3.3.

Next, strings was run on the decompressed data from the page file to see how many strings that matched "fareniv" would be found.

```
PS> strings.exe page-file-decompressed.bin >
  ↪ pagefile-decompressed-strings.txt

PS> Get-Content .\pagefile-decompressed-strings.txt | Select-String
  ↪ fareniv > .\pagefile-decompressed-strings-fareniv.txt

PS C:\Users\Aleks-Forensics\Documents\Master-thesis\Browser-artifact-test>
  ↪ Get-Content .\pagefile-decompressed-strings-fareniv.txt |
  ↪ Measure-Object -Line

Lines Words Characters Property
-----
      897
```

The exact same procedure was executed on the original page file to see the amount of relevant strings that could be found with no decompression. The text file containing the results from the "fareniv"-strings was named "pagefile-strings-fareniv.txt".

```
PS > Get-Content pagefile-strings-fareniv.txt | Measure-Object -Line

Lines Words Characters Property
-----
      24
```

6.2.4 Discussion

The results clearly shows that data is being obfuscated by the memory manager. This experiment also shows that the page file contains lots of potentially interesting data that is obfuscated as a result of compression. 1GB of compressed data was decompressed to 3GB, which means that the 6GB page file contained 8GB of potentially interesting data after decompression. This is just an assumption, what the remaining 5GB of page file data contains has not been examined in this experiment.

Table 6.1 shows that the amount of strings related to "fareniv" found in the different locations. The amount of relevant strings in the memory dump was reduced to approximately 20% after running TestLimit.exe. A few of the relevant strings could be found in the MemCompression address space. 6 relevant strings before decompressing the MemCompression process contents, and 372 after. As the page file was only tested after TestLimit was run, there it is possible that it coincidentally contained strings related to "fareniv".

	Memory Snapshot	VAD-segments PID 1708	VAD-segments Decompressed PID 1708	Page File	Page File Decom- pressed
Before TestLimit	706	3	42	-	-
After TestLimit	172	6	372	24	897

Table 6.1: Shows the amount of strings related to "fareniv" that were found in the different locations.

Discussion

In this chapter, the results are discussed more in depth. The results from the applied experiments are discussed, followed by two initial strategies that were considered for solving the problem. The MemoryDecompression tool is evaluated, and potential improvements is presented. The submission and result of the Volatility Plugin contest is mentioned as an indication of quality in regards to MemoryDecompression. Future work is discussed with some possible alternatives, and finally the two hypotheses are evaluated in regards to the applied experiments and existing literature on the topic.

7.1 Applied Experiments

This section contains discussion about the practical experiments performed in this thesis. The impact of the findings, the impact of the tool and general weaknesses of the applied experiments are discussed.

7.1.1 Findings

The results from the practical experiments shows that potentially important data is being obfuscated by the Windows memory manager. Both experiments show that memory is being compressed and obfuscated when applying high memory pressure. The first experiment shows that Unicode strings could be more at risk for being compressed than ASCII characters. The second experiment confirms that specific strings can be found in the page file by decompressing its compressed contents.

7.1.2 Forensic Soundness

Forensic soundness was discussed in Section 2.3, and speaks to the repeatability of the results that a tool or technique produces. The experiments in this thesis should be repeatable by following the same methodology. It is expected that the results will differ to a certain

degree, but the results should show that interesting strings are being obfuscated. The results will differ due to countless unreliable factors caused by the operating system running normally.

As an attempt of validating the results with dual tool verification, mentioned in Section 2.3, the experiments were performed with slightly different tools. Using both Linux and Windows tools to extract strings from the output file reduces the chance of incorrect results. There is no other tools that has similar functionality to MemoryDecompression, so any tool validation in that regard is not possible. The experiments could have been performed with both Volatility and Rekall to see if the results differed.

The forensic soundness of MemoryDecompression is quite poor at this time. The tool requires a lot of testing before it can be trusted in real-life cases.

7.1.3 Impact

The results demonstrate that the MemoryDecompression tool could be useful in cyber crime investigations and incident response. As discussed in Section 2.4, Windows is by far the most used operating system. As the awareness of the importance of analyzing computer memory in computer forensics becomes better, the more relevant memory forensics will become. As this tool also works on page files, it will also be useful on disk images. As MemoryDecompression is only applicable to Windows 10 memory samples and disk images, the value of the tool will increase over time as Windows 10 replaces more and more Windows 7 systems. Though the transition is going faster than previous upgrades of Windows versions, likely according to the free upgrade Microsoft offered, it is still expected to increase further (Knight, 2018). The Windows Server series could also be getting the memory compression feature as a default or optional setting, which also expands the usefulness of MemoryDecompression.

The practical experiments show that more data will be available for analysis than with previous versions of Windows. As compressed memory data is resident both in the page files and memory dumps, there is potential for a lot more evidence data available to the analyst. As the compression algorithm compresses data to approximately 30-50% of its original size, decompressing will lead to approximately two to three times more data for the analyst. In cases where the malicious actor has made successful efforts in hiding their activity, MemoryDecompression could become the extra factor that makes it possible to solve the case.

7.1.4 Weaknesses

There are a number of weaknesses with the practical experiments that need to be addressed.

Number of experiments

The results are based on two experiments, which is a small number in the sense of statistical significance. There could be coincidences in these two experiments that create faulty results. As the experiments take significant time to plan and perform, it was limited to two experiments. The time was prioritized to make a working decompression program.

The consequence of this is that the results should be subject to experimental and statistical validations as part of future work.

Faulty Program

As the MemoryDecompression tool was used to create the results, it must be taken into consideration that the tool could be a source of error. There could be mistakes made by the student while programming the tool that makes the result faulty. It could potentially make the result more extreme than if a normal user had used it.

TestLimit to demonstrate memory pressure

The TestLimit tool was used to simulate user activity over time by overloading physical memory, making the memory manager compress pages to make space. This technique could be an unrealistic way of simulating the average user, as it uses significant memory space over a very short time.

Virtual Machine as lab environment

Virtual environments does not always act the same way as physical machines would. They are often more stable and reliable, and could therefore be a source of faulty results. Testing on a variety of environments is desirable to take more variables into account when doing research.

2GB of Physical Memory

As the experiments are performed on a virtual machine with almost the minimum amount of memory required to run Windows 10 in a satisfying way, it could also give unrealistic results. A computer with 4GB of memory or above would probably require a significantly high memory pressure to compress the amounts of data as seen in the experiments.

7.2 Evaluation of MemoryDecompression

The MemoryDecompression tool performs as intended, but it is still a proof-of-concept tool. A lot of time and effort went into making it fast enough so that speed would not be an eliminating factor when considering this tool. Visual Studio was used to benchmark performance try to improve. Early tests showed that MemoryDecompression would use 108 hours on a 2.5GB page file, which would be unacceptable. Some memory leaks were also dealt with, leading to filling memory after running for a while. This was mostly due to memory allocations that were not properly de-allocated.

7.2.1 Integrity

A few experiments have been performed testing the integrity of MemoryDecompression. The integrity is tested by running the tool multiple times with the same parameters to see

if the exact same results are produced. The tests included comparing the SHA256-hashes from the output files. The size of the files in bytes was also compared.

7.2.2 Improvements

This section contains the potential for improvement of the MemoryDecompression tool.

Multi Processing

To increase performance, adding support for multi processing for MemoryDecompression should be implemented. Computer forensics workstations often need to have more power than the average user to handle processing disk images and memory dumps. As they often have powerful CPUs with several cores, this makes them great for utilizing multi processing applications.

Better Input Handling

The tool currently reads the whole input file into memory before the decompression is performed. This means that it will fail if the file is larger than the available physical memory. As both page files and memory samples can be quite large, this is an improvement that is highly prioritized. The 32-bit version is unable to handle files bigger than 2GB, as this is the limit of its virtual memory size. This need for improvement was brought to the student's attention through the participation in the Volatility Plugin Contest, which will be further discussed in section 7.2.3.

Variable Page Size

The memory manager compresses one page at a time. As the page size is configurable, the tool might not work on all configurations of Windows 10. This could be improved by making an input parameter that could set the page size, if not 4096 bytes.

More Statistical Output

Especially in digital forensics related to law-enforcement cases, rigid documentation is necessary. MemoryDecompression could output more data in a report format, that would avoid manual documentation. Timestamps of files, hashes of the input and the output files, and data sizes in more human readable formats are some potential candidates.

Some improvements requires further research into how the memory manager handles memory compression, and will be covered in Section 7.4.

7.2.3 Volatility Plugin Contest

The Volatility Plugin Contest is a contest by the Volatility Foundation hosted for the 6th time as of this year (Walters, 2018). The purpose is to encourage research and development in the field of memory analysis.

The MemoryDecompression tool was submitted to the Volatility Plugin Contest 2018. The main reason for the submission was to get some of the best people in the world to try

the tool, and thus getting some quality feedback. The submission ended up as 2nd place (Hale Ligh, 2018a). This is a good indication of quality, and a positive result regarding the potential usefulness of the tool.

7.3 Strategies for Decompressing Memory

The main goal of this thesis was to solve the issue of memory compression in Windows 10 in digital forensics, and creating a tool that could be utilized by examiners in investigations. To do so, it was necessary to have a concrete strategy. The student considered two approaches as strategies for achieving the main objective.

7.3.1 Strategy 1: Decompression using a Brute Force approach

By gaining knowledge about how Windows compresses and decompresses memory, it could be possible to utilize this knowledge to create a tool that can decompress data from any file containing compressed data. The tool could be used to decompress data from memory dumps, page files, or hibernation files. This strategy requires finding where the compressed memory is located, and finding out how the Windows memory compression works in detail. The end goal would be a tool that finds compressed data in a file or folder and decompresses it.

Strengths(+):

- More likely to result in a positive outcome as the level of complexity is lower

Weaknesses(-):

- Only gives the examiner raw paged data with no process context

7.3.2 Strategy 2: Reverse Engineering Windows kernel code

The second strategy was an extension of first one. As the memory compression process is only an empty process space used for storing compressed pages outswapped from other processes, it does not contain any meta data. To correctly swap compressed memory back from the memory compression process into the process space of the owning process, it would be necessary for the operating system to have stored this information about the compressed data somewhere in memory. Experiments in Section 5.2 have shown that this data resides in the memory space belonging to the kernel. As the kernel space is largely undocumented by Microsoft, it would require reverse engineering of the kernel code to get an understanding of the meta data that relates to the MemCompression process. This strategy requires finding data structures in kernel memory containing this meta data, and finding the correct way of decompressing the data. The end goal would be a tool that locates these meta data structures, finds the locations of compressed data in memory and decompresses data. This approach would also potentially display process information about the originating process of the compressed pages.

Strengths(+):

-
- For memory dumps, this strategy could result in a more sophisticated tool.
 - More valuable information, such as process context for the decompressed content.

Weaknesses(-):

- As reverse engineering undocumented parts of the Windows kernel is a very complex task, this strategy has a higher chance of failure.
- Analyzing page files would still require a brute force approach, so this strategy would possibly only work on memory dumps.

7.3.3 Evaluation

The two strategies were both valid methods of reaching the end goal of the thesis. The best outcome would be achieving the goals of both strategies, and create a tool that covers all required aspects of memory decompression in computer forensics. However, the student's skill and knowledge in reverse engineering makes achieving this approach unrealistic within the time frame. Attempts were made by the student to analyze the kernel structures necessary, but the conclusion after tens of hours of work was that this approach could potentially jeopardize the goal of the thesis by spending too much time without actionable findings.

7.3.4 Conclusion

Considering the strengths and weaknesses of the two strategies, in addition to the limitations of the student, Strategy 1 was chosen. The risk of spending too much time on reverse engineering complex code was regarded as substantial. Taking that risk could potentially jeopardized the creation of the tool, as this was also considered a time consuming task.

7.4 Future Work

Future work is required to improve the understanding of how memory compression affects digital forensics, and to improve capabilities of the forensics community. This section presents potential areas where further research is required.

7.4.1 Volatility Extension

One aspect of the need for future work in relation to memory compression has been covered in Section 7.3.2. The MemoryDecompression tool works as intended by giving investigators access to more data otherwise hidden. However, it only gives access to raw data with no context. Future work could focus on creating a more sophisticated tool that sorts the decompressed data into process context. A possible end-result could be an extension to a tool like Volatility that automatically locates and decompresses pages from the MemCompression process. When parsing a memory sample, this extension should locate and decompress pages from the MemCompression process and sort them into their original process address space. The process should be transparent to the investigator using Volatility.

7.4.2 Pattern Searching for compressed data

A potential for future work is to research ways to locate the compressed data in a big file, like a page file or memory dump. As the MemoryDecompression tool parses all data it is presented with, a more sophisticated way of approaching the input file or folder could be to only target compressed data within them. Some way of finding higher entropy than usual, or finding a signature of the data compressed with Microsoft's Xpress Algorithm.

7.4.3 MemoryDecompression platform-agnostic

Future work could also include the following. As the MemoryDecompression tool uses the Windows API to perform decompression, it could be useful for the community to get it re-written to a platform-agnostic version working on all platforms.

For example, msuhanov wrote a blog post showing that registry keys can be found in compressed form in Windows 10 memory and page files (MSUHANOV, 2018). More research in the same category is needed to fully understand the impact and potential of memory compression.

7.4.4 Pooltag Analysis

During this thesis, the student attempted to find consistent meta data sources of the memory compression process. By using WinDbg, the pooltags related to the Store Manager were analyzed. The pooltags are sorted into categories, and by listing the ones starting with "sm", it was possible to find all of them. As it is possible to scan for pooltags in a memory sample, this could be a valid method for parsing the meta data (Case and Walters, 2014d).

```
kd> !poolused 2 sm*
```

```
Sorting by NonPaged Pool Consumed
```

Tag	NonPaged		Paged		
	Allocs	Used	Allocs	Used	
smNp	794	3252224	0	0	store node pool allocations
smCB	611	2502656	0	0	allocations
smWw	512	2097152	0	0	allocations
smBt	390	1597440	0	0	various B+Tree allocations
smSt	117	1407856	0	0	various store allocations
smRg	15	524224	0	0	in-memory store region array
...					
smPS	13	832	0	0	allocations
smSw	1	560	0	0	allocations
smFp	12	448	0	0	virtual forward progress entry
smKG	1	160	0	0	encryption key registry path buffer
smHT	1	80	0	0	allocations
TOTAL	3703	11604624	0	0	

No consistencies were found that could make this a valid search method. However, it is possible that further research on the topic can make some useful contributions the analysis of compressed memory.

7.4.5 Reverse Engineering WinDbg libraries

The file DbgEng.dll used by WinDbg was attempted reverse engineered by the student to find out where the meta data structures of the compressed data were present in memory. Some interesting functions were found during these attempts, for example "ReadCompressedPageDataAndDecompress" which also seems to use an instance of RtlDecompress-BufferEx. Fully reverse engineering this code and rewriting it to a forensics tool could potentially lead to a more sophisticated solution to decompressing memory in computer forensics. The reason for attempting to analyze DbgEng.dll was because of a tips from Alex Ionescu¹ by e-mail. "The deepest way to create a tool for decompressing memory would be to reverse engineer the Windows debugger. It has code to deal with compressed pages. This code is either in kdexts.dll or DbgEng.dll. Look for structures related to the "store" and nt!.SM_*".

7.4.6 Statistical Verification

One of the weaknesses regarding the practical testing was the amount of experiments. Ideally, the experiments should have been performed with various techniques, various environments and with a broader spectre of artifacts than simply strings. Future work should include a larger scale of experiments with more diverse artifacts. Having a large statistical data basis of how the tool performs in various scenarios would increase the statistical significance. The goal is to increase the forensic soundness, covered in Section 2.3, of MemoryDecompression. Insufficient testing increases the chance that the results was affected by factors outside the students control.

7.4.7 Further Development of MemoryDecompression

Future work includes further development on the proof-of-concept tool. Section 7.2.2 covers potential improvements, but this list is likely to expand as more experiments are performed.

Further development also implies keeping the tool updated. Microsoft applies changes to the operating system frequently, and forensics tools must be maintained. Minor changes to how memory compression works can make the tool useless.

7.5 Hypothesis Discussion

The two hypothesis presented in Section 1.3 will be discussed in this section.

- **Hypothesis 1:** *By analyzing operating system behaviour, it is possible to create a method for decompressing, and thus de-obfuscating, data from memory samples or page files that has been compressed by the Windows 10 memory manager.*
- **Hypothesis 2:** *De-obfuscating the data compressed by Windows 10 memory manager can uncover digital evidence for computer forensics investigators.*

¹Co-author of Windows Internals and Chief Architect at CrowdStrike. World-class security architect and expert in low-level system software, kernel development, security training and reverse engineering (CROWD-STRIKE, 2018)

7.5.1 Hypothesis 1

Hypothesis 1 could get a definitive answer in this thesis. If the kernel debugging and reverse engineering returned no actionable results of finding the decompression mechanism, the hypothesis would not have been confirmed. By finding the correct algorithm and using it successfully on memory data, the hypothesis has been proven.

Taking the weaknesses of the practical work into consideration, it is still safe to say that the thesis has proven that it is possible to de-obfuscate the compressed data. Even without the practical experiments from chapter 6, the experiments performed with the CompressDecompress tool on the extracted memory pages in Section 5.3.6 would be a good indicator that it would also work on memory samples and page files.

7.5.2 Hypothesis 2

The practical experiment in Section 6.1 showed that digital evidence was obfuscated by memory compression. When MemoryDecompression was run on the memory data, digital evidence was de-obfuscated. In a digital investigation, this type of information could have been used as actionable digital evidence of console activity.

All literature that relates to computer forensics on page files is relevant in this discussion. When extracting and de-obfuscating data from memory samples or page files, the examiner is presented with the same data they would see in cleartext in a page file on earlier versions of Windows. The existing literature on page files confirms that they can be used to find digital evidence, which speaks in favor of Hypothesis 2. As covered in Section 1.4 and Section 4.2.3, the page file is traditionally analyzed using unstructured methods. Artifacts can be found using strings-, anti-virus- and Yara-searches. Also running carving tools like Foremost and Bulk-extractor can uncover a wealth of important data. These sections also presents various artifacts and scenarios where this analyzing the page file can uncover digital evidence.

The practical experiments from Section 6.2 shows that the disk images also yields valuable information, as the number of strings relating to the scenario was significantly higher after decompression.

Through the work in this thesis and the existing literature, Hypothesis 2 is supported. However, further research and testing is required to find out if digital evidence will be found in real-life cases.

Conclusion

This thesis was based upon two hypotheses. This section will provide the conclusions of the thesis.

- **Hypothesis 1:** *By analyzing operating system behaviour, it is possible to create a method for decompressing, and thus de-obfuscating, data from memory samples or page files that has been compressed by the Windows 10 memory manager.*

Hypothesis 1 has been confirmed, and the conclusion is that decompressing and de-obfuscating memory from memory dumps and page files is in fact possible. This was proven with the development of the decompression tool, and the practical experiments on Windows 10. The experiments performed through kernel debugging revealed the actual function used in decompressing memory, which made it possible to create a tool that successfully decompressed data from memory dumps and page files. The practical experiments showed that MemoryDecompression de-obfuscated strings in memory, making them available for analysis.

- **Hypothesis 2:** *De-obfuscating the data compressed by Windows 10 memory manager can uncover digital evidence for computer forensics investigators.*

Previous research, existing literature and the practical work in this thesis supports Hypothesis 2 in that decompressing and de-obfuscating data from memory can uncover digital evidence. Taking the weaknesses of the practical experiments into consideration, the results indicates that MemoryDecompression can be a valuable tool in memory and disk forensics. The tool should be considered as proof-of-concept software, and still requires extensive testing and improvements before it can be relied upon in an investigation.

The practical experiments has shown that potentially interesting strings can be found in compressed parts of memory both in memory samples and in page files. It also shows some indications that Unicode-characters are more vulnerable to obfuscation through memory compression.

Finally, the thesis concludes that submitting the MemoryDecompression tool to Volatility Plugin Contest and achieving a 2nd place is a good indicator that the results from this thesis will bring value to the digital forensics community. Although, it is not to be misinterpreted as scientific proof in any regard.

Bibliography

- AccessData, 2018a. Evidence acquisition using accessdata ftk imager. <https://articles.forensicfocus.com/2018/03/02/evidence-acquisition-using-accessdata-ftk-imager/>, accessed: 2018-12-06.
- AccessData, 2018b. Ftk imager 4.2.0. <https://marketing.accessdata.com/ftkimager4.2.0>, accessed: 2018-12-12.
- AceFinity, 2014. Rtlcompressbuffer rtldecompressbuffer. <https://www.sysnative.com/forums/programming/8605-rtlcompressbuffer-and-rtldecompressbuffer-print.html>), accessed: 2018-08-08.
- Arnes, A., 2018a. The digital forensics process. In: Digital Forensics. Wiley, pp. 27–66.
- Arnes, A., 2018b. Forensic soundness. In: Digital Forensics. Wiley, p. 29.
- Arnes, A., H. P. V. G. K. R., 2006. Digital forensic reconstruction and the virtual security testbed wise. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA, pp. 144–163.
- Belkasoft, 2016. Belkasoft ram capturer: Volatile memory acquisition tool. <https://belkasoft.com/ram-capturer>, accessed: 2018-12-06.
- Carrier, B., 2005. Digital investigation foundations. In: File System Forensic Analysis. Addison Wesley, pp. 3–16.
- Carvey, H., 2005. Digital forensics of the physical memory. <https://seclists.org/incidents/2005/Jun/22>, accessed: 2018-12-06.
- Case, A., H. L. M. L. J., Walters, A., 2014a. Analyzing the page file(s). In: The Art of Memory Forensics. Wiley, pp. 110–113.
- Case, A., H. L. M. L. J., Walters, A., 2014b. The Art of Memory Forensics. Wiley.
- Case, A., H. L. M. L. J., Walters, A., 2014c. Memory acquisition. In: The Art of Memory Forensics. Wiley, pp. 69–114.

-
- Case, A., H. L. M. L. J., Walters, A., 2014d. Pool-tag scanning. In: *The Art of Memory Forensics*. Wiley, pp. 129–140.
- Case, A., H. L. M. L. J., Walters, A., 2014e. Systems overview. In: *The Art of Memory Forensics*. Wiley, pp. 3–26.
- Cohen, M., 2016. The pmem suite of memory acquisition tools. <http://blog.rekall-forensic.com/2016/05/the-pmem-suite-of-memory-acquisition.html>, accessed: 2018-12-06.
- Cohen, M., 2018. Rekall forensics. <http://www.rekall-forensic.com/>, accessed: 2018-12-06.
- Cowen, D., 2015. Forensic lunch 11/13/2015 26:25 - 29:00.
URL <https://youtu.be/5qUZqj1tHmU?t=1585>
- CROWDSTRIKE, 2018. Alex ionescu. <https://www.crowdstrike.com/blog/author/alex-ionescu/>, accessed: 2018-12-08.
- Eriberto, 2014. The sleuth kit commands. http://wiki.sleuthkit.org/index.php?title=The_Sleuth_Kit_commands, accessed: 2018-12-05.
- ETHW.org, 2015. Lempel-ziv data compression algorithm, 1977. https://ethw.org/Milestones:Lempel-Ziv_Data_Compression_Algorithm,_1977, accessed: 2018-11-25.
- FireEye, 2018. Redline. <https://www.fireeye.com/services/freeware/redline.html>, accessed: 2018-12-06.
- GuidanceSoftware, 2018. Encase forensic. <https://www.guidancesoftware.com/encase-forensic>, accessed: 2018-12-12.
- Hale Ligh, M., 2018a. Results from the 2018 volatility contests are in! <https://volatility-labs.blogspot.com/2018/11/results-from-annual-2018-volatility-contests.html?m=1>, accessed: 2018-11-25.
- Hale Ligh, M., 2018b. Volatilityfoundation - community. <https://github.com/volatilityfoundation/community/tree/master/AleksanderOsterud>, accessed: 2018-12-06.
- Hex-Rays, 2017. About ida. <https://www.hex-rays.com/products/ida/>, accessed: 2018-12-12.
- Kaspersky, 2015. Kaspersky lab uncovers duqu 2.0. https://www.kaspersky.com/about/press-releases/2015_duqu-is-back-kaspersky-lab-reveals-cyberattack-on-its-corporate-network-that-also-hit-high-profile-victims-in-western-countries-the-middle-east-and-asia, accessed: 2018-12-06.
- Knight, S., 2018. Windows 10 surpasses windows 7 in global market share according to statcounter. <https://www.techspot.com/news/73068-windows-10-surpasses-windows-7-global-market-share.html>, accessed: 2018-12-08.

-
- Knuth, D., 1997. Stacks, queues, and deques. In: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, p. 238243.
- Kraus, A., 2016. Windows 10 memory compression and more. <https://aloiskraus.wordpress.com/2016/10/03/windows-10-memory-compression-and-more/>, accessed: 2018-09-25.
- Luttgens, J., 2014a. Investigating windows systems. In: *Incident Response*. McGraw Hill Education, pp. 271–380.
- Luttgens, J., 2014b. Memory forensics. In: *Incident Response*. McGraw Hill Education, pp. 356–374.
- Maartman-Moe, C., T. S. A. A., 2009. The persistence of memory: forensic identification and extraction of cryptographic keys. DFRWS.
- Microsoft, 2012. Tools to simulate cpu / memory / disk load. <https://blogs.msdn.microsoft.com/vijaysk/2012/10/26/tools-to-simulate-cpu-memory-disk-load/>, accessed: 2018-12-03.
- Microsoft, 2016. Run bash on ubuntu on windows. <https://blogs.windows.com/buildingapps/2016/03/30/run-bash-on-ubuntu-on-windows/yxIRIVUSLbi8ualJ.97>, accessed: 2018-12-04.
- Microsoft, 2017a. Bcdedit command-line options. <https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/bcdedit-command-line-options>, accessed: 2018-11-25.
- Microsoft, 2017b. Commands. <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/commands>, accessed: 2018-11-25.
- Microsoft, 2017c. fastcall. <https://docs.microsoft.com/en-gb/cpp/cpp/fastcall?view=vs-2017>, accessed: 2018-11-25.
- Microsoft, 2018a. Overview of x64 calling conventions. <https://msdn.microsoft.com/en-us/library/ms235286.aspx>, accessed: 2018-11-25.
- Microsoft, 2018b. Register usage. <https://msdn.microsoft.com/en-us/library/9z1stfyw.aspx>, accessed: 2018-11-25.
- Microsoft, 2018c. Rtlcompressbuffer function. <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-rtlcompressbuffer>, accessed: 2018-11-25.
- Microsoft, 2018d. Rtldecompressbufferex function. <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-rtldecompressbufferex>, accessed: 2018-11-25.
- Microsoft, 2018e. Xpress compression algorithm. <https://msdn.microsoft.com/en-us/library/hh554002.aspx>, accessed: 2018-11-25.

-
- MSUHANOV, 2018. Memory compression and forensics. <https://dfir.ru/2018/09/08/memory-compression-and-forensics/>, accessed: 2018-12-08.
- NetApplications.com, 2018. Operating system market share - desktop/laptops. <https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpsp=2017&qpnp=1&qptimeframe=Y&qpcustomd=0>, accessed: 2018-12-06.
- Richard, G., C. A., 2014a. In lieu of swap: Analyzing compressed ram in mac os x and linux. DFRWS 9 (4).
- Richard, G., C. A., 2014b. Swap files as a source of evidence. In: In Lieu of Swap: Analyzing Compressed RAM in Mac OS X and Linux. DFRWS.
- Richard, G., C. A., 2016a. Compressed in-memory swap. In: Memory forensics: The path forward. Elsevier.
- Richard, G., C. A., 2016b. Memory forensics: The path forward. DFRWS.
- Richard, G., C. A., 2016c. Windows hibernation file analysis. In: Memory forensics: The path forward. Elsevier.
- Russinovich, M., 2018. Sysinternals suite. <https://docs.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite>, accessed: 2018-12-06.
- Suiche, M., 2016. Your favorite memory toolkit is back for free! <https://blog.comae.io/your-favorite-memory-toolkit-is-back-f97072d33d5c>, accessed: 2018-12-06.
- TSS, 2018. What algorithm does memory press in win10 with memory compression? (translated from russian). <http://kitrap08.blogspot.com/>, accessed: 2018-12-11.
- VMWare, 2018. Workstation pro. <https://www.vmware.com/products/workstation-pro.html>, accessed: 2018-12-12.
- Volatilityfoundation, 2016. Volatility 2.6 (windows 10 / server 2016). <https://www.volatilityfoundation.org/26>, accessed: 2018-12-06.
- Walters, A., 2018. The 6th annual volatility plugin contest and the inaugural volatility analysis contest! <https://volatility-labs.blogspot.com/2018/05/the-6th-annual-volatility-plugin.html>, accessed: 2018-11-25.
- Yosifovich, P., I. A. R. M. S. D., 2017a. Memory compression. In: Windows Internals 7 - Part 1. Microsoft Press, pp. 449–456.
- Yosifovich, P., I. A. R. M. S. D., 2017b. Memory management. In: Windows Internals 7 - Part 1. Microsoft Press, pp. 301–482.
- Yosifovich, P., I. A. R. M. S. D., 2017c. System processes. In: Windows Internals 7 - Part 1. Microsoft Press, pp. 88–99.
- Yosifovich, P., I. A. R. M. S. D., 2017d. Virtual memory. In: Windows Internals 7 - Part 1. Microsoft Press, pp. 21–23.

Appendix A

Source Code

Listing 8.1: MemoryDecompression.cpp

```
1  #include "pch.h"
2  #include <iostream>
3  #include <Windows.h>
4  #include <fstream>
5  #include <string>
6  #include <strsafe.h>
7  #include <chrono>
8
9
10 #define STATUS_SUCCESS                ((NTSTATUS)0x00000000UL)
11 #define STATUS_BUFFER_ALL_ZEROS      ((NTSTATUS)0x00000117UL)
12 #define STATUS_INVALID_PARAMETER     ((NTSTATUS)0xC000000DUL)
13 #define STATUS_UNSUPPORTED_COMPRESSION ((NTSTATUS)0xC000025FUL)
14 #define STATUS_NOT_SUPPORTED_ON_SBS  ((NTSTATUS)0xC0000300UL)
15 #define STATUS_BUFFER_TOO_SMALL      ((NTSTATUS)0xC0000023UL)
16 #define STATUS_BAD_COMPRESSION_BUFFER ((NTSTATUS)0xC0000242UL)
17
18 // ntdll.dll is loaded to utilize the RtlDecompressBuffer function
19 HMODULE ntdll = GetModuleHandle(TEXT("ntdll.dll"));
20
21 typedef NTSTATUS (__stdcall *_RtlDecompressBuffer) (
22     USHORT CompressionFormat,
23     PCHAR UncompressedBuffer,
24     ULONG UncompressedBufferSize,
25     PCHAR CompressedBuffer,
26     ULONG CompressedBufferSize,
27     PULONG FinalUncompressedSize
28 );
29
30 // roundUp is used for finding the next sensible offset for a new chunk of
31 // data to decompress
32 int roundUp(int numToRound, int multiple)
33 {
34     int remainder = abs(numToRound) % multiple;
35     return numToRound + multiple - remainder;
36 }
37 // -----
38
```

```

39 // decompress_buffer uses RtlDecompressBuffer with the proper input
40 // parameters to decompress the buffer
41 BOOL decompress_buffer(UCHAR *buffer, const int bufferLen, const int
42 // uncompBufferLen, ULONG *uncompBufferSize, UINT *nextOffset, UCHAR *
43 // uncompBuffer)
44 {
45     BOOL success = false;
46     _RtlDecompressBuffer RtlDecompressBuffer = (_RtlDecompressBuffer)
47         GetProcAddress(ntdll, "RtlDecompressBuffer");
48
49     //UCHAR *uncompBuffer = new UCHAR[uncompBufferLen];
50     NTSTATUS result = RtlDecompressBuffer(
51         COMPRESSION_FORMAT_XPRESS | COMPRESSION_ENGINE_STANDARD, //
52         CompressionFormat, found through kernel debugging
53         uncompBuffer, // UncompressedBuffer
54         uncompBufferLen, // UncompressedBufferSize
55         buffer, // CompressedBuffer
56         bufferLen, // CompressedBufferSize
57         uncompBufferSize // FinalUncompressedSize
58     );
59
60     // Checks if the decompression succeeds, and that the uncompressed
61     // size is correctly 4096. If the conditions are correct, the roundUp
62     // function is used to find the next offset to compressed data.
63     if (result == STATUS_SUCCESS) {
64         if (*uncompBufferSize == (ULONG)4096) {
65             *nextOffset = roundUp(bufferLen, 16);
66             success = true;
67         }
68     }
69     return success;
70 }
71
72 // -----
73 // decompress_file performs the decompression and writes the decompressed
74 // contents to the output file
75 BOOL decompress_file(const WCHAR *input_filename, HANDLE hFileInput, UCHAR
76 // &zero_buffer, HANDLE hFileOutput, UINT *compressed_total, UINT *
77 // decompressed_pages)
78 {
79     // NextOffset keeps track of the offset within the current buffer
80     UINT nextOffset = 0;
81
82     // NextOffsetTotal keeps track of the offset within the file
83     UINT nextOffsetTotal = 0;
84
85     DWORD fileSizeHi = 0;
86     DWORD fileSize = GetFileSize(hFileInput, &fileSizeHi);
87     BYTE *fileBuffer = new BYTE[fileSize];
88     DWORD bytesRead;
89     BOOL readSuccess = ReadFile(hFileInput, fileBuffer, fileSize, &
90         bytesRead, NULL);
91     if (!readSuccess) return 0;
92
93     // Loops through the contents of the input file until it reaches the
94     // end

```

```

84 LOOP: while (nextOffsetTotal < fileSize) {
85
86     UCHAR *buffer = new UCHAR[4096];
87     UCHAR *uncompBuffer = new UCHAR[4096];
88
89     // Buffer needs to start at j = 0, since "i" starts at
90     // nextOffsetTotal which is only 0 at the first page
91     UINT j = 0;
92     for (int i = nextOffsetTotal; i < 4096 + nextOffsetTotal; i++) {
93         buffer[j] = (UCHAR)fileBuffer[i];
94         j++;
95     }
96     // Returns an error message if the buffer somehow fails to be
97     // allocated
98     if (!buffer) {
99         std::cout << "An error occurred while reading the bytes of the
100         // input file into the buffer." << std::endl;
101         return 1;
102     }
103
104     // Checks if the current buffer is all zeroes, if that is the case
105     // it jumps 4096 bytes in the input file.
106     int ifNull = memcmp(buffer, &zero_buffer, 4096);
107     if (ifNull == 0) {
108         nextOffsetTotal += 4096;
109         goto LOOP;
110     }
111
112     // Sets the initial buffer size to 15, as 16 bytes is the smallest
113     // size a 4096 page can be compressed to by RtlDecompressBuffer
114     // with 16-byte chunk size.
115     int bufferLen = 15;
116
117     // Loops while the length of the buffer is below 4096. The buffer
118     // increments with one after each failed attempt, and tries again
119     // until it successfully decompresses or the buffer reaches 4095
120     // bytes.
121     while (bufferLen < 4096) {
122         bufferLen++;
123         ULONG realDecompSize = 0;
124
125         // Calls decompress_buffer with the given parameters
126         BOOL decompressed_buffer = decompress_buffer(buffer, bufferLen
127             , 4096, &realDecompSize, &nextOffset, uncompBuffer);
128
129         // Writes to file if a 4096 bytes page is successfully
130         // decompressed
131         if (decompressed_buffer) {
132             BOOL WriteFile_result = WriteFile(hFileOutput,
133                 uncompBuffer, realDecompSize, &realDecompSize, NULL);
134
135             // Checks if the output file was successfully written to,
136             // and returns an error message if it was not
137             if (!WriteFile_result) {
138                 std::cout << "An error occurred while writing to the
139                 // output file." << std::endl;

```

```

127         return 0;
128     }
129
130     // Increments the total amount of compressed bytes that
        has been decompressed for use in the console output
        statistics
131     *compressed_total += bufferLen;
132     nextOffsetTotal += nextOffset;
133
134     // Increments the total amount of compressed pages
135     *decompressed_pages += 1;
136     break;
137 }
138
139     // Checks if the buffer has reached 4096 bytes, and skips 16
        bytes ahead
140     if (bufferLen == 4096) {
141         nextOffsetTotal += 16;
142         break;
143     }
144
145     }
146     // Clears any allocated buffers to avoid a memory leak
147     delete[] uncompBuffer;
148     delete[] buffer;
149 }
150 // Clears the allocated buffer to avoid a memory leak, and closes the
        file handle
151 delete[] fileBuffer;
152 CloseHandle(hFileInput);
153 }
154
155 // -----
156
157 // Main function.
158 int main(int argc, wchar_t* argv[])
159 {
160     // std::chrono is used for measuring the duration of the program from
        start to finish
161     std::chrono::high_resolution_clock::time_point t1 = std::chrono::
        high_resolution_clock::now();
162
163     // Error check for correct number of arguments
164     if (argc != 3)
165     {
166         printf("Incorrect parameters\n\nUsage: MemoryDecompression.exe <
            input file or directory> <output-file>");
167         return 0;
168     }
169
170     // Defines argument one and two as the input and output file
171     const WCHAR *input_filename = argv[1];
172     const WCHAR *output_filename = argv[2];
173
174     // Creates handle to output file
175     HANDLE hFileOutput = CreateFile((LPCSTR)output_filename,
        FILE_APPEND_DATA, FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,

```

```

        FILE_ATTRIBUTE_NORMAL, NULL);
176 if (hFileOutput == INVALID_HANDLE_VALUE) {
177     printf("Failed to create output file");
178 }
179
180 UINT decompressed_pages = 0;
181 UINT compressed_total = 0;
182
183 // Creates a zero buffer later used to skip empty parts of input files
    , mostly relevant on page files.
184 UCHAR zero_buffer[4096] = { 0 };
185 DWORD input_file_attribute = GetFileAttributesA((LPCSTR)input_filename
    );
186
187 // If the input file is a directory
188 if (input_file_attribute & FILE_ATTRIBUTE_DIRECTORY)
189 {
190     WIN32_FIND_DATA ffd;
191     TCHAR szDir[MAX_PATH];
192     HANDLE hFind = INVALID_HANDLE_VALUE;
193
194     // Converts the directory name to a string that can be read by
        FindFirstFile and FindNextFile
195     StringCchCopy(szDir, MAX_PATH, (STRSAFE_LPCSTR)input_filename);
196     StringCchCat(szDir, MAX_PATH, TEXT("\\*"));
197
198     // Creates a handle to the first file in the input directory
199     hFind = FindFirstFile(szDir, &ffd);
200
201     // To skip the "." and ".." that is present in all directories,
        FindNextFile is used once here
202     FindNextFile(hFind, &ffd);
203
204     SetCurrentDirectory((LPCSTR)input_filename);
205
206     // Loops through all files in the input directory
207     while (FindNextFile(hFind, &ffd) != 0) {
208
209         // Creates handle to input file with read-only privileges
210         printf("Decompressing\t %s\n", ffd.cFileName);
211         HANDLE hFileInput = CreateFile((LPCSTR)&ffd.cFileName,
            GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, NULL);
212
213         // Checks if the file handle is valid, else it prints an error
            message
214         if (hFileInput == INVALID_HANDLE_VALUE) {
215             printf("Failed to open file input file\n");
216         }
217
218         // Calls function decompress_file_result with the given
            parameters
219         BOOL decompress_file_result = decompress_file(input_filename,
            hFileInput, *zero_buffer, hFileOutput, &compressed_total,
            &decompressed_pages);
220     }
221 }

```

```

222
223 // If the input file is not a directory
224 else
225 {
226     // Creates a handle to the input file
227     HANDLE hFileInput = CreateFile((LPCSTR)input_filename,
        GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL);
228
229     // Checks if the file handle is valid, else it prints an error
        message
230     if (hFileInput == INVALID_HANDLE_VALUE) {
231         printf("Failed to open input file");
232     }
233     printf("Decompressing \t %s\n", input_filename);
234
235     // Calls function decompress_file_result with the given parameters
236     BOOL decompress_file_result = decompress_file(input_filename,
        hFileInput, *zero_buffer, hFileOutput, &compressed_total, &
        decompressed_pages);
237 }
238
239 // Printing the statistics to console output
240 std::cout << "\n" << "Total decompressed pages: \t" <<
        decompressed_pages << std::endl;
241 std::cout << "Total compressed data: \t\t" << compressed_total << "
        bytes" << std::endl;
242 std::cout << "Total decompressed data: \t" << decompressed_pages *
        4096 << " bytes" << std::endl;
243 CloseHandle(hFileOutput);
244
245 // Creating duration statistics, and prints it to console output
246 std::chrono::high_resolution_clock::time_point t2 = std::chrono::
        high_resolution_clock::now();
247 auto duration_ms = std::chrono::duration_cast<std::chrono::
        microseconds>(t2 - t1).count();
248 auto duration_sec = std::chrono::duration_cast<std::chrono::seconds>(
        t2 - t1).count();
249 auto duration_min = std::chrono::duration_cast<std::chrono::minutes>(
        t2 - t1).count();
250 auto duration_hrs = std::chrono::duration_cast<std::chrono::hours>(t2
        - t1).count();
251
252 std::cout << "\n\nDecompression completed in:\nTotal Microseconds:\t"
        << duration_ms;
253 std::cout << "\nTotal Seconds:\t" << duration_sec;
254 std::cout << "\nTotal Minutes:\t" << duration_min;
255 std::cout << "\nTotal Hours:\t" << duration_hrs;
256 }

```

Appendix B

Source Code

Listing 8.2: CompressDecompress.cpp

```
1 #include "stdafx.h"
2 #include <iostream>
3 #include <Windows.h>
4 #include <fstream>
5
6 #define STATUS_SUCCESS ((NTSTATUS) 0x00000000UL)
7 #define STATUS_BUFFER_ALL_ZEROS ((NTSTATUS) 0x00000117UL)
8 #define STATUS_INVALID_PARAMETER ((NTSTATUS) 0xC000000DUL)
9 #define STATUS_UNSUPPORTED_COMPRESSION ((NTSTATUS) 0xC000025FUL)
10 #define STATUS_NOT_SUPPORTED_ON_SBS ((NTSTATUS) 0xC0000300UL)
11 #define STATUS_BUFFER_TOO_SMALL ((NTSTATUS) 0xC000023UL)
12 #define STATUS_BAD_COMPRESSION_BUFFER ((NTSTATUS) 0xC0000242UL)
13
14 HMODULE ntdll = GetModuleHandle(TEXT("ntdll.dll"));
15
16 typedef NTSTATUS(__stdcall *_RtlCompressBuffer)(
17     USHORT CompressionFormatAndEngine,
18     PCHAR UncompressedBuffer,
19     ULONG UncompressedBufferSize,
20     PCHAR CompressedBuffer,
21     ULONG CompressedBufferSize,
22     ULONG UncompressedChunkSize,
23     PULONG FinalCompressedSize,
24     PVOID Workspace
25 );
26
27 typedef NTSTATUS(__stdcall *_RtlDecompressBuffer)(
28     USHORT CompressionFormat,
29     PCHAR UncompressedBuffer,
30     ULONG UncompressedBufferSize,
31     PCHAR CompressedBuffer,
32     ULONG CompressedBufferSize,
33     PULONG FinalUncompressedSize
34 );
35
36 typedef NTSTATUS(__stdcall *_RtlGetCompressionWorkSpaceSize)(
37     USHORT CompressionFormatAndEngine,
38     PULONG CompressBufferWorkSpaceSize,
39     PULONG CompressFragmentWorkSpaceSize
40 );
```

```

41
42 //
-----

43 char *get_file_buffer(const char *filepath, int &buffer_len)
44 {
45     std::ifstream ifs(filepath, std::ofstream::binary);
46     if (!ifs.is_open()) return 0;
47     ifs.seekg(0, ifs.end);
48     buffer_len = ifs.tellg();
49     ifs.seekg(0, ifs.beg);
50     char *buffer = new char[buffer_len];
51     ifs.read(buffer, buffer_len);
52     return buffer;
53 }
54
55 //
-----

56 int write_file_buffer(const char *filepath, char *buffer, int buffer_len)
57 {
58     std::ofstream ofs(filepath, std::ofstream::binary);
59     if (!ofs.is_open()) return 1;
60     ofs.write(buffer, buffer_len);
61     ofs.close();
62     return 0;
63 }
64 //
-----

65 UCHAR *compress_buffer(const char *buffer, const ULONG bufferLen, ULONG
66     compBufferLen, ULONG *compBufferSize)
67 {
68     _RtlCompressBuffer RtlCompressBuffer = (_RtlCompressBuffer)
69         GetProcAddress(ntdll, "RtlCompressBuffer");
70     _RtlGetCompressionWorkSpaceSize RtlGetCompressionWorkSpaceSize = (
71         _RtlGetCompressionWorkSpaceSize) GetProcAddress(ntdll, "
72         RtlGetCompressionWorkSpaceSize");
73
74     ULONG bufWorkspaceSize; // Workspace Size
75     ULONG fragWorkspaceSize; // Fragmented Workspace Size (Unused)
76     NTSTATUS ret = RtlGetCompressionWorkSpaceSize(
77         COMPRESSION_FORMAT_XPRESS | COMPRESSION_ENGINE_STANDARD, //
78         CompressionFormatAndEngine
79         &bufWorkspaceSize, // CompressBufferWorkSpaceSize
80         &fragWorkspaceSize // CompressFragmentWorkSpaceSize
81     );
82
83     if (ret != STATUS_SUCCESS) {
84         std::cout << "An error ocurred while trying to calculate the
85             compression workspace size." << std::endl;
86         get_ntstatus_err(ret);
87         return 0;
88     }
89
90     std::cout << "Compression Workspace Size: 0x" << std::hex <<
91         bufWorkspaceSize

```

```

85         << std::dec << " (" << bufWorkspaceSize << ")" << std::endl;
86
87     VOID *workspace = (VOID *)LocalAlloc(LMEM_FIXED, bufWorkspaceSize);
88     if (workspace == NULL) {
89         std::cout << "Failed to allocate space for workspace" << std::endl;
90         return 0;
91     }
92
93     UCHAR *compBuffer = new UCHAR[compBufferLen];
94     NTSTATUS result = RtlCompressBuffer(
95         COMPRESSION_FORMAT_XPRESS | COMPRESSION_ENGINE_STANDARD, //
96         CompressionFormatAndEngine
97         (UCHAR *)buffer, // UncompressedBuffer
98         bufferLen, // UncompressedBufferSize
99         compBuffer, // CompressedBuffer
100        compBufferLen, // CompressedBufferSize
101        16, // UncompressedChunkSize
102        compBufferSize, // FinalCompressedSize
103        workspace // WorkSpace
104    );
105
106    LocalFree(workspace);
107    if (result != STATUS_SUCCESS) {
108        get_ntstatus_err(result);
109        return 0;
110    }
111
112    // std::cout << "Compressed DATA: " << compBuffer << std::endl;
113    std::cout << "Compressed Length: " << compBufferSize << std::endl;
114    return compBuffer;
115 }
116 //

```

```

117 UCHAR *decompress_buffer(const char *buffer, const int bufferLen, const
118     int uncompBufferLen, ULONG *uncompBufferSize)
119 {
120     _RtlDecompressBuffer RtlDecompressBuffer = (_RtlDecompressBuffer)
121         GetProcAddress(ntdll, "RtlDecompressBuffer");
122
123     UCHAR *uncompBuffer = new UCHAR[uncompBufferLen];
124     NTSTATUS result = RtlDecompressBuffer(
125         COMPRESSION_FORMAT_XPRESS | COMPRESSION_ENGINE_STANDARD, //
126         CompressionFormat
127         uncompBuffer, // UncompressedBuffer
128         uncompBufferLen, // UncompressedBufferSize
129         (UCHAR *)buffer, // CompressedBuffer
130         bufferLen, // CompressedBufferSize
131         uncompBufferSize // FinalUncompressedSize
132     );
133
134     if (result != STATUS_SUCCESS) {
135         get_ntstatus_err(result);
136         return 0;
137     }
138 }

```

```

136 // std::cout << "Uncompressed DATA: " << uncompBuffer << std::endl;
137 std::cout << "Uncompressed Length: " << uncompBufferSize << std::endl;
138 return uncompBuffer;
139 }
140
141 //
-----
142
143 // #define COMPRESS
144
145 int main()
146 {
147 #ifdef COMPRESS
148     const char originalFile[] = "C:\\Users\\User\\MASTER-THESIS-TESTING\\
        test-dbg-uncompressed.bin";
149 #endif
150     const char compressedFile[] = "C:\\Users\\User\\MASTER-THESIS-TESTING
        \\test-dbg-compressed.bin";
151 #ifndef COMPRESS
152     const char decompressedFile[] = "C:\\Users\\User\\MASTER-THESIS-TESTING
        \\test-tool-decompressed.bin";
153 #endif
154
155 #ifdef COMPRESS
156     int bufferLen;
157     char *buffer = get_file_buffer(originalFile, bufferLen);
158     if (buffer == 0) {
159         std::cout << "An error occurred while trying to read the bytes of the
            specified file into the buffer." << std::endl;
160         return 1;
161     }
162     ULONG realCompSize;
163     UCHAR *compressed_buffer = compress_buffer(buffer, bufferLen, bufferLen
        + 512, &realCompSize);
164     if (compressed_buffer == 0) return 1;
165
166     if (write_file_buffer(compressedFile, (char *)compressed_buffer, (int)
        realCompSize) != 0) {
167         std::cout << "An error occurred while trying to write the bytes of
            the compressed buffer to a file." << std::endl;
168         return 1;
169     }
170     std::cout << compressed_buffer << std::endl;
171 #else
172     int bufferLen;
173     char *buffer = get_file_buffer(compressedFile, bufferLen);
174     if (buffer == 0) {
175         std::cout << "An error occurred while trying to read the bytes of the
            specified file into the buffer." << std::endl;
176         return 1;
177     }
178
179     ULONG realDecompSize;
180     UCHAR *decompressed_buffer = decompress_buffer(buffer, bufferLen,
        bufferLen * 100, &realDecompSize);
181     if (decompressed_buffer == 0) return 1;

```

```
182
183     if (write_file_buffer(decompressedFile, (char *)decompressed_buffer, (
184         int)realDecompSize) != 0) {
185         std::cout << "An error ocurred while trying to write the bytes of
186             the compressed buffer to a file." << std::endl;
187         return 1;
188     }
189 #endif
190     return 0;
191 }
```

