User Space Memory Analysis

Master's Thesis

University of Twente

Author:
Edwin Smulders

Committee:
dr. J.Y. Petit
prof. dr. P.H. Hartel
R.B. van Baar, MSc (NFI)

November 13, 2013

Contents

Chapter 1

Introduction

1.1 Digital Forensics

Forensics is the study of finding the facts of events in the past, given evidence recovered from a scene. Typically, this is used to prove or disprove actions in criminal or civil law. Digital forensics is similar, but because of its digital nature, it introduces some new challenges.

Digital forensics deals with all evidence that is digital. Of course, this includes your PC, mobile phone or even your (smart) TV. Therefore, digital forensics generates new types of evidence and analysis and the amount of data to be analyzed is enormous. For example, analysis of the internal memory of a TV is completely different from the analysis of the contents of a terabyte harddisk, because of both the type and the amount of information contained.

The scene of events in digital forensics is typically different from other forensics. Events may take place on a device, but these events may affect other devices and the network. As an example, filling in a form on a webpage may result in the webpage server executing a certain action. This makes digital forensics more complex, but this also means that actions and events can sometimes be confirmed by different sources.

1.2 Memory Forensics

Memory forensics is the part of digital forensics that deals with the main memory (i.e. Random-Access Memory) of computer systems. In incident response, analysis can be done on the live system, but it can also be done on a dump of the volatile memory. RAM is called volatile memory because it can possibly lose its data after a restart.

There are several reasons why this memory is examined. First, applications are typically loaded from a harddisk, but some applications, such as malware or just-in-time compiled code, may not exist on the harddisk of the machine. Second, the harddisk of the computer may be encrypted, in which case the encryption key may be in memory. There are many other situations where an encryption key can be in memory; every application that makes use of encryption will have the key in memory at some point.

Encryption of the main (i.e. bootable) harddisk is the only situation where the key is always in memory.

Third, a source of information that can be examined is the application state, which can show that certain actions have happened. Finally, the data in memory may represent personal information about the user, such as webpages visited. Another good example of this is text that has been typed, but not saved to disk; this is buffered in memory.

The analysis of a memory dump often starts with the analysis of the kernel memory, for example to derive a structure for the rest of the memory. Examples of this are the process list and virtual address translation. It is also used to find information intrinsic to the kernel, such as network connections.

There are a number of techniques for memory analysis, which includes scanning (using heuristics), probabilistic analysis and deriving semantics using a predefined structure (following pointers).

The acquisition of a memory dump is a whole topic of research in itself, since modern operating systems do not allow access to the full memory by a normal application, i.e. not a (kernel mode) driver. We will speak briefly of memory acquisition in our related work, as well as the method of acquisition we will use, but it is not our main topic.

Our research is focused on the application state and user data, a choice we have motivated in our problem statement.

1.3 Organisation of this document

This document is organised as follows. After this introduction, we present the problem statement and the research questions. This is followed by the related work. In chapter 4 we present an overview of the methods we have used, in combination with some background information, where needed. Chapter 5 goes into more detail about our method of volatile memory acquisition and is meant to clarify how to reproduce our work. In chapter 6 you can find the technical (implementation) details of the methods we chose, followed by a summary of items we have taken into consideration during our research in chapter 7. Chapter 8 shows example cases, meant to give an idea of the impact of our analysis, followed by the results and evaluation in chapter 9. Finally, we end with a conclusion in chapter 10 giving an answer to our research questions as well as presenting the future work.

Chapter 2

Problem Statement and Research Questions

2.1 Problem Statement

In this chapter, we will summarize some background information on the problem. This information is used to conclude a problem statement (the final section of this chapter). Later (in the related work) we will give extra context to this problem.

2.1.1 Problem Background

Size of the memory dump

The size of physical memory in average consumer computers is currently several gigabytes large. The operating system kernel uses only a small part of this memory (typically tens to a few hundred megabytes, as seen in the Windows task manager and Linux equivalents). Most structures in kernels have been analyzed. Research has shown that of the remaining space, around 25% could be identified as files mapped in memory [44]. This leaves a large part of the memory still unidentified or unanalyzed.

Usage of memory

In recent years, physical memory has been plentiful. This means that applications have been using more memory, as a trade-off for CPU usage, to increase the speed of the application. A good example of this are browsers, where loading a single webpage could use up to 200MB of memory. This shows that there is a lot of information to be found in application memory.

Similarities in usage

The way the physical memory is used is similar in many applications. Either they are compiled by the same compiler, they use the same libraries or the same application virtual machine/interpreter. The concepts used are also mostly the same for all applications and operating systems; a stack looks virtually the same in all applications.

Finding the facts

The same information can be found in multiple places. In forensics, the goal is to find facts about events in the past, to reconstruct a scenario with near absolute certainty. Even if no new information is found in application memory, this information can be used to confirm information found elsewhere.

Analysis of user space

Analysis of user space memory is not trivial, even though computers are deterministic, because the information in memory may have been created by using information which has already been deleted from memory. An example of this would be the symbols of an executable, which may be unloaded after use. Sometimes, this data can be reintroduced to do analysis.

2.1.2 Problem statement

We have summarized in the previous sections that more and more memory is used by applications. In fact, too much to manually process. We have also concluded that the structures used in memory are really common, and contain important information. Therefore, it is problematic that there is no way to do automated analysis of this memory. This means that there is data available which is currently not used in finding the facts.

First of all, we need to identify the technical challenges that exist in user space memory analysis. A clear overview of the currently existing challenges will help both our research and the progress of the field of memory forensics.

Furthermore, we can identify what data is located in the virtual address space of the applications that are currently running. Missing in current tooling and research are the identification of the building blocks, the common structures and libraries that exist in user space applications.

At the end of this research, if we can give a clear overview of the existing problems and challenges, and if we can identify structure in the identified data and possibly use this structure to extract forensic evidence, we have helped progress the research in memory forensics.

2.2 Research Questions

Our main research question is:

> Can we define a generalizable approach for analysis of user space applications in volatile memory?

2.2.1 Subquestions

Some subquestions which should be considered during our research:

- What can we learn from the stack of a process?
- How can we analyze the use of libraries?
- Is the access to source or header files mandatory to perform memory examination?
- How does the operating system or system architecture affect application analysis?
- What are the difficulties in analysis of applications written in VM-based or interpreted languages?

Chapter 3

Related Work

In this section we will summarize the current state of the art in memory forensics. We will first give a small summary on different techniques of memory acquisition. After that we will describe the analysis of specific platforms (operating system kernels) as well as research on multiple platforms and research on user space applications.

3.1 Acquisition

There are two very distinct methods of acquiring memory dumps: hardware and software. One example of a hardware method relies on physical devices which provide some form of Direct Memory Access (DMA). The software method uses a program to make a dump of the entire memory. Acquisition is not the main topic of our research, but it is discussed (briefly) to identify some problems in memory analysis.

3.1.1 Hardware

In [8] a PCI device is shown which captures the physical memory using DMA. A limitation of this approach is that the card needs to be installed beforehand. A testing method for the validation of constantly changing memory is not provided.

Another device capable of DMA is Firewire. [28] provides a good summary of current methods in acquisition using Firewire. The technique is not new, and has been used as a way to attack machines, before it was used as a forensics method. Using a toolset made by Adam Boileau (MetlStorm), a device can be configured for reading memory [16, 5].

The advantage of DMA acquisitions is that they are fast and does not need software on the host machine. But DMA/Firewire is not always available, is not atomic (processes continue to run) and is vulnerable to subversion. Rutkowska shows that using Memory Mapped I/O, not only can parts of the memory be hidden from the DMA device, but it can be tricked into reading arbitrary data [35].

An old but less used approach is the cold boot attack. Memory (DRAM) retains its contents for seconds to minutes after power is lost. This time can be extended by cooling the DRAM, long enough to place it in a computer used for acquisition. [23] shows the attack is still applicable on modern computers and can be executed without specialized hardware. They also suggest that the attack is effective in combination with methods to reconstruct encryption keys, with full disk encryption ever more prevalent in the real world.

3.1.2 Software

In the beginning of memory forensics, software acquisition was often done through device files; `/dev/kmem` on Linux and `\\.\PhysicalMemory\` on Windows. Because of abuse by attackers, these systems have been disabled on most modern operating systems [8]. One of the tools to access these devices and make an exact copy is `dd`, and George Garner's Windows port of it [18]. When these devices are not available, forensic investigators have to resort to alternate methods, such as using the NTSystemDebugControl API [45, 19].

These techniques run as user level, which comes with limitations. To acquire the memory, the acquisition program has to map the memory pages, but these pages may have a different memory type, which will give caching problems [46]. User mode techniques are also subject to rootkit subversion. [11] implements a rootkit which employs data misdirection: it tricks forensics tools into thinking they extracted real data by hooking into low level APIs (rootkit subversion).

According to Schatz, acquiring memory through the hibernation file is also vulnerable to rootkit subversion, because drivers are notified of the hibernation.

Because these existing techniques are unreliable, Schatz presented a new method to do software acquisition. This method, appropriately named BodySnatcher, takes away control of the host hardware with a second, acquisition specific OS (acquisition OS) [36].

Finally, most virtual machines allow taking a snapshot of the state, which is stored in a file. This can be used to examine the memory of the virtual machine (but not the memory of the host machine).

Common problems in software acquisition include the requirement of root access to the device, as well as the potential for smearing, the corruption of data because the acquisition is not atomic and the device is still running.

3.2 Platforms

In this section we will discuss the analysis of memory on multiple different platforms. Windows has seen the most research, but Linux, Mac OS X and Android will also be represented. It should be noted that any of the methods mentioned in this sections may be obsoleted in the future by the release of new versions of an operating system, but commonly, only the implementation changes.

3.2.1 Windows

Because of the large user base, Windows is the most important platform for memory forensics and is therefore the platform with the most tools and research. In 2005, the

Digital Forensics Research Workshop (DFRWS) organised the first memory analysis challenge, which kickstarted the memory analysis scene. In the process, three tools were released: Windows Memory Forensics Toolkit, MemParser and Kntlist.

Windows Memory Forensics Toolkit was developed by Mariusz Burdach, which enumerates processes and modules [7]. MemParser, developed by Chris Betz, enumerates active processes and dumps their memory [4]. Kntlist is a tool that detects multiple internal kernel lists to produce various lists of threads, processes, handles and objects. It was developed by George Garner and Robert-Jan Mora [19].

It should be noted that these tools rely on kernel structures and therefore do not detect objects that have been terminated, or hidden by Direct Kernel Object Manipulation (DKOM), a technique to modify these structures. In 2006, Andreas Schuster came up with a technique to search the full memory for structures that represent objects and processes, using structures such as the pool header, object header or the eprocess structure. For this, he developed a script named PTfinder. Using this tool, he showed that some information persisted in memory, even after reboots [38]. In a different research, Schuster also looked for kernel memory pool structures in a similar fashion. Finding this information was useful as an information source to find smaller structures, such as ARP and TCP records [37].

In the same year, Walters and Petroni released Forensics Analysis Toolkit (FATKit). One of the main features of this framework is the automated interpretation of C data structures. This helps in finding all possible structures in memory. It also adds address space reconstruction and support for extra analysis and visualization [33].

In 2007, Brendan Dolan-Gavitt describes how the Virtual Address Descriptor (VAD) tree, a Windows kernel structure, can be used to find memory ranges that are allocated to a process. These are the ranges as seen by the process, so they might be mapped files, loaded DLLs or privately allocated regions. In the paper, they show how to list DLLs loaded by a process, detect library injections and defeat DLL hiding (specific DKOM techniques) [13].

The next step in Windows memory forensics was to find memory-mapped files. In 2008, Ruud van Baar used the previously mentioned VAD tree to find these files (instead of previous methods, such as file carving). Experiments showed that this method identified about 25% of all pages in a dump as part of a mapped file, of which 40% could be linked to one or more processes [44].

Using PTFinder, Schuster showed in 2008 that freed kernel memory could persist for over 24 hours on a mostly idle system. He noted a contrast with userland memory, which could disappear in a matter of minutes after an application has been terminated [39, 41].

In the same year, Dolan-Gavitt showed that the Windows registry could be found in memory. They noted a slightly different address translation mechanism, but overall the same structure as the on-disk version, so the use of regular forensic tools could be used on the registry. The memory version of the registry was shown to be important because there were parts which were not on disk, as well as parts which could have been modified (by an attacker) [14].

A problem that arose in Windows memory forensics was the identification of the

Windows version. James Okolica found a way to find the necessary kernel objects in Windows NT-like kernels without knowing the Windows version. The approach relies finding specific symbols, such as debug headers, in the memory dump [30].

To detect kernel structures, signatures can be used, which are often hand-made and weak (depending on fields that can be changed). In an attempt to defeat DKOM and detect hidden structures, Dolan-Gavitt et al. present a way to generate robust signatures; signatures of kernel structures that do not rely on data that may be changed. Attempts to evade the signature by modifying the structure will cause the operating system to consider it invalid [15].

In 2012, Gu et al. created a way to fingerprint operating systems from memory only. Their method uses signatures to identify kernel code, including some mechanisms that account for randomization, such as kernel ASLR. Using this method they were able to correctly identify 46 different kernels, including 9 Windows kernels [22].

3.2.2 Linux

The internal structure for the Linux kernel is very similar to the Windows kernel in terms of design principles. Besides the smaller user base, this may explain the lack of published research on the Linux kernel in memory forensics; there is not a lot to be researched on the Linux kernel that has not already been done on the Windows kernel. There is some research on specific items that can be found in Linux memory dumps. One such thing is Pettersson's search for cryptographic keys for full disk encryption. In his research he shows how to find the keys for the relatively old `cryptoloop` encryption, as well as the newer `dm-crypt` encryption. He also states that the only defense against key recovery of full disk encryption is toughening the system against acquisition techniques. One such example is a password protected screensaver (lockscreen) [34]. If a computer is automatically locked, no software programs can be executed to acquire the memory.

In 2008, Sherri Davidoff published her research on cleartext passwords in Linux memory. The result showed that many applications retained their passwords plain text in memory. Even ssh and root passwords could be found, although not in memory belonging to any still existing process [12].

3.2.3 FreeBSD and Mac OS X

Many designs in the Mac OS X and FreeBSD kernels do not differ much from the Windows and Linux kernels, conceptually, so there have been few publications for these operating systems.

Matthieu Suiche developed the method to read Mac OS X virtual memory, including most relevant kernel structures, such as process lists [42]. Continuing on this work, Andrew Hay published a way to list open files in memory for Mac OS X, similar to Van Baar's research on the Windows platform [24]. FreeBSD was mentioned in [22] as one of the kernels that could be identified, but has otherwise been mentioned in few publications.

3.2.4 Android

Joe Sylve discusses a few issues with Android memory forensics. Android, being based on Linux, is similar to Linux, but there are a few differences. The main issue is that the architecture is of most mobile phones is based on ARM instead of Intel/x86, which requires a reimplementation of page address translation. Another important difference is the use of the Dalvik virtual machine. Dalvik controls all user space applications, opening the way for automated analysis of all Android applications by generic analysis of the virtual machine [43].

This is exactly what Holger Macht did in his thesis on live memory forensics on Android. In his research, Volatility was used to examine the memory of an Android system. In addition, a new plugin was developed to automatically examine the Dalvik VM [27].

3.3 Multi-platform

Some methods have been applied (or are applicable) to multiple platforms. Mohammed Al-Saleh and Ziad Al-Sharif published a paper on the lifetime of data in TCP buffers. In their study, they compared Ubuntu 10.04 and Windows 7. They identified several use cases: Live processes with open sockets, live processes with closed sockets, terminated processes, a program that just receives, a program that does not even call `recv()` and a real world example, Apache and IIS. Their results showed that on both operating systems, the data may still be cached in the TCP buffers, even after the program has closed. The lifetime of the data was more than 24 hours. Only the case where `recv()` was not called showed a faster disappearance of the data [1].

Volatility is a multi-platform memory analysis framework. Originally called Volatools, it was developed in 2007 by Walters and Petroni for the Windows platform. Since then, it has grown to be a multi-platform tool, gaining support for Linux and soon Mac OS X. The way it is set up, it has support for multiple memory dump formats and many different address spaces (which may differ depending on the platform). Plugins operate on top of this abstraction layer, needing no knowledge of the address space, only of (virtual) memory addresses [48, 47].

3.4 Userland

There has been some research into the analysis of user space applications. One example is the 2012 research by Olajide et al., where they investigate the amount of user input that can be retrieved from the Windows applications Outlook and Internet Explorer. However, they use a previously known user input and do not detail the methods they use to retrieve the input. In the paper, they divide the results in two ways, quantitative (how many times the input can be found) and qualitative (completeness) [31].

An example of analysis of one specific application can be found in research by Matthew Simon. Using pieces of data known beforehand, he found out how Skype stores data in memory. While Skype was running, he could find passwords and user contacts stored in memory. User contacts also remained in memory after the Skype process had terminated [40]. This analysis technique relies on being able to get a memory dump with known data. This is often the case with user applications, but with malware for example, this is not the case.

In [2] a formal methods approach is taken to the (library) calls on the application stack, to recreate a model of the execution history. This model is also verified against the program assembly code. Using this model, they know the state of the program at the time of the memory dump. In the paper, nothing is mentioned about the user data that might be on the stack [2].

Ernest Mougoue takes a completely different approach in application memory forensics. In his master thesis, he describes a program called Memexec, which can extract and resume a user space application. In a special vm, he can restore a process extracted from a memory dump by using a kernel module to insert the correct kernel structures, as well as restore register settings for the process. A restored process can give the forensic analyst some insight into a user's recent activity [29].

Cozzie et al. took a different approach to data in user space. Using normalized memory data, namely (Address, String, Data and Zero), a (Bayesian) probabilistic approach has been defined to detect data structures in memory. The implementing program is called Laika. This program has been used to detect data structures from known viruses, Agobot, Kraken and Storm. These viruses could be detected with 99% accuracy [10]. It should be noted that these detected structures had no semantics attached to them.

A paper by Hejazi defines sensitive information in memory. One of the methods to find this sensitive information is by analysis of the call stack. In the call stack they look for specific functions in the libraries that are used (in this case, DLLs - Windows). Parameters for these functions are extracted manually [25].

3.5 Virtual Machines

A subject we have not touched yet is the discovery of virtual machines in memory dumps of host machines. This is exactly what Graziano et al. have done in their tool Actaeon. In their approach, they are able to detect the existence of hypervisors using Intel VT-x technology, and are able to analyze the virtual machine using a Volatility plugin [21].

3.6 Other

Memory forensics deals with large binary files. Conti et al. devised a method to identify or classify parts of the data. On pieces of 1kB, they used statistical analysis to compare each part to precomputed statistical scores for several data types. For these scores, they used Shannon Entropy, Arithmetic mean, Chi square and Hamming Weight, combined to one average score. A fragments score was compared to these precomputed scores using a nearest neighbour algorithm. From their results, they learned that they could identify (with reasonable probability) the following data types: Random/Compressed/Encrypted, Base64 encoded, UUencoded, Machine Code (ELF and PE), Plain text, Bitmap [9].

Gareth Owen showed a way to extract encryption keys using behavioral analysis on the program code. With this method he was able to automatically extract RC4 keys from three different implementations. This is relevant to memory forensics, because the program code can be found in memory. The automated extraction of keys from a memory dump is mentioned as future work [32].

3.7 Conlusion

In the context of the related work, it is still unclear what the (technical) challenges in user space memory forensics are. If we look at the work by Arasteh ([2]) and Hejazi ([25]), we see that problems and limitations of the actual recovery of functions from the stack are only considered briefly or are not discussed at all. Furthermore it is not always clear what information from memory is used and how this is recovered or accessed from a memory dump.

If we revisit our problem statement, it is clear that the identification of general problems in user space memory analysis is important. It is also important that our research is repeatable and clearly identifies what data can be used to recover information.

Chapter 4

Methodology

4.1 Introduction

In this chapter, we present the methods we use to analyze memory dumps. Tools already exist for kernel space memory analysis, which we use as the base of our methodology. The tool we use in our research is Volatility, a memory analysis framework featuring a plugin system which enables us to develop plugins which work on user space memory. First, we explain how Volatility enables us to work on memory. Second, we give an overview of the methods we developed or applied ourselves.

Linux is the target platform of our research, with memory dumps made using Virtualbox. The open nature of the Linux platform makes research easier. The source of the Linux kernel is public, which may be required in new memory forensics research. The architecture of Linux processes is not substantially different from other operating systems, making it a suitable platform.

4.2 Methods

In this section we describe the methods we use and the reason why we chose these methods. It includes some background information where required.

Most of the memory analysis methods described here are based on following the original structure of the memory as used by the operating system. A search based on structure uses pointers to follow the data in the same way the application or the operating system would. The advantage of this approach is that we can extract semantics from the program, but it is sometimes hard to follow the data because certain information may be deleted from memory, or worse, modified by the operating system or the application. Following the structure builds a tree of context information about the data; if data is modified, the tree contains incorrect information or it is impossible to continue processing. Other methods are affected less by this problem, but don't provide as much context.

4.2.1 Volatility

Volatility is the framework we use to implement our methods. It already implements a number of features for memory analysis, allowing us to focus on implementing the methods we need for user space memory analysis. These features include the use of kernel objects, virtual address translation and memory maps.

A global overview of the memory layout as Volatility provides it is shown in Figure 4.1. On the highest level, it shows a separation between kernel and user space memory. Our methods focus on user space, where the application and the libraries are located. Some of the kernel space items in the figure are explained in the following subsections.

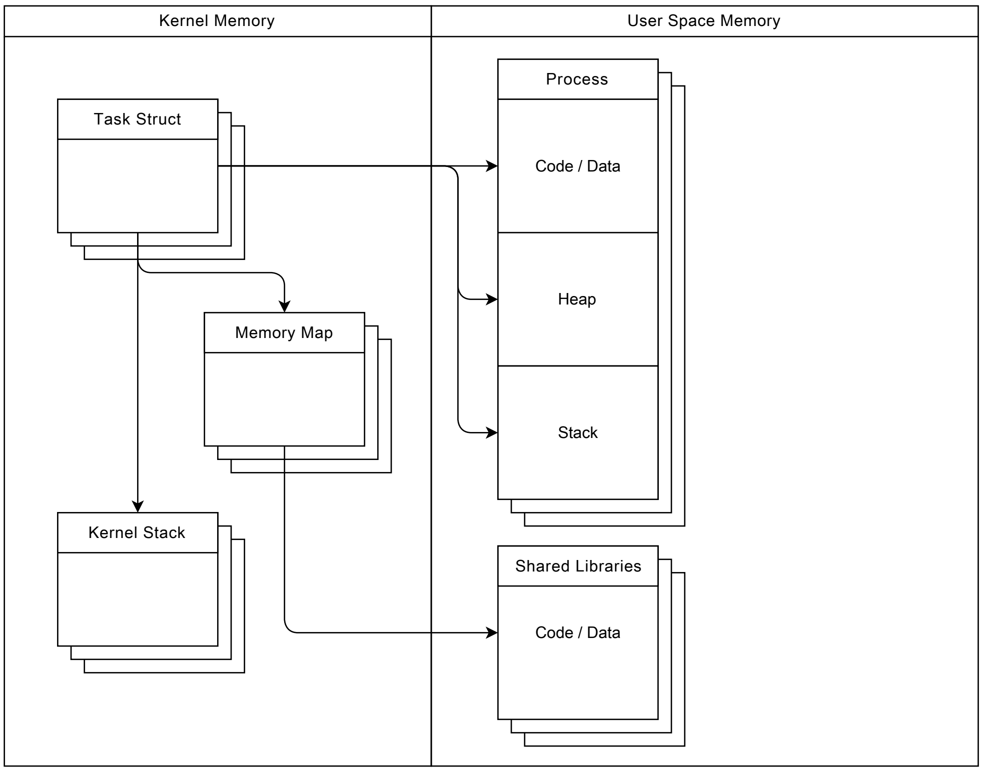

Figure 4.1: A schematic overview of memory as provided by current tools.

Virtual address translation

In modern operating systems, virtual addressing is used to give applications all the memory they need without any regard for the hardware, which is managed by the operating system. This is the first essential part in user space memory forensics; without it, analysis of user space is substantially more difficult.

This functionality is integrated in Volatility in the form of layered *address spaces*. In Figure 4.1, this is shown as the separation between kernel and user space. Volatility provides an address space for kernel space memory, as well as a separate address space for each application.

Kernel Objects

Each process is registered with the operating system in such a way that there is an object (i.e. a struct) which has information on this process. This object enables us to identify information about the process, such as the running state, and provides access to relevant structs that identify memory maps. In Linux terminology, the process is defined by the task struct, shown in the kernel space section in Figure 4.1.

Memory maps

A typical application has mapped several areas in memory, such as libraries, the stack and the heap. Access and identification is essential to get an understanding of the memory layout. This is managed by the memory map object belonging to a specific process (task struct). Along with this, each mapped area may have some properties, such as an executable flag. This enables us to separate executable code from data. These memory maps are also shown in Figure 4.1, connected to the task structs.

4.2.2 Registers

To understand the process memory, we need to be able to find and use the CPU registers. Registers are a feature of the CPU (a type of CPU cache) to have fast access to a limited set of data, for use in instructions. These registers are needed for user space memory analysis, because due to conventions they contain valuable information, such as the stack pointer and the instruction pointer. Registers are available in memory because of the technique called context switching.

The identification of the CPU registers is a building block for some of the other methods described in this chapter. The technique is not completely new (it was done before for the x86 architecture in [29]), but it is not supported in current tooling and has not been done for x86_64.

Context Switching

Because the operating system usually runs more processes than there are physical CPUs in the system, time needs to be divided between processes. This is called scheduling. The switch from one process to another is called a context switch. In these switches, the CPU state (as far as required) is stored in memory. An important part of this state are the registers [26].

CPUs (such as the Intel 80386 and its successors) also support hardware context switching, but these features are unlikely to be used in modern operating systems, because as it turns out, software context switching is faster [6].

The combination of the necessity for context switching and unused hardware support explains why we expect that register contents can be found in memory.

4.2.3 System call analysis

One part of the program state is the currently executing system call, if any. By recovering the name and arguments of this system call, we can learn what the process was doing at the time of the memory dump.

The system call mechanism provides a way for a user process to interact with the kernel. Using a special processor instruction (e.g. `int 0x80` or `syscall`) a kernel function is called. From a programmer's perspective, this is much like a function call (and are often called by wrappers in system libraries), but there are some differences.

Using the CPU registers, we expect to be able to analyze these differences and be able to recover system call details. This is a new method and builds on both register recovery and existing functionality in Volatility.

4.2.4 Stack analysis

The user space stack describes how the application reached its current state, as well as stores its local data. Our method of stack analysis involves four parts:

- Recovery of function addresses from return addresses. For this we need to make use of the Procedure Linkage Table (PLT).
- Matching function addresses to function symbols.
- Recovery of stack traces using registers and return addresses.
- Recovery of stack frames without a frame pointer.

In the following sections we introduce some background information for these methods. Some of these methods are not new, but common techniques in debugging applications. There has been limited research so far in applying these techniques to memory dump analysis. Some methods in this section are affected by our choice to use Linux, because concepts such as the PLT do not exist or work differently on other operating systems.

Procedure Linkage Table

To understand how we can use return addresses, we must take a look at the the Procedure Linkage Table (PLT). The PLT is a section in ELF/Linux executable files. It is used to dynamically resolve functions in shared libraries, also known as *dynamic loading*. This dynamic loading is done at execution time.

Dynamically resolving functions based on function symbols is necessary because this allows us to recompile a shared library without updating all applications that depend on it.

To enable this dynamic resolving, the PLT adds a level of indirection to our function calls. For each function, a special section is added to the PLT. The function call in the code calls this section, rather than calling the shared library directly.

This special section consists of two parts. The first part uses a pointer to the Global Offset Table (GOT) section. It jumps to the address listed in the GOT. If the function has already been resolved, this address is the address of the function in the shared library. This process can be seen in Figure 4.2. If it has not been resolved yet, it points to the second part in this function's PLT section.

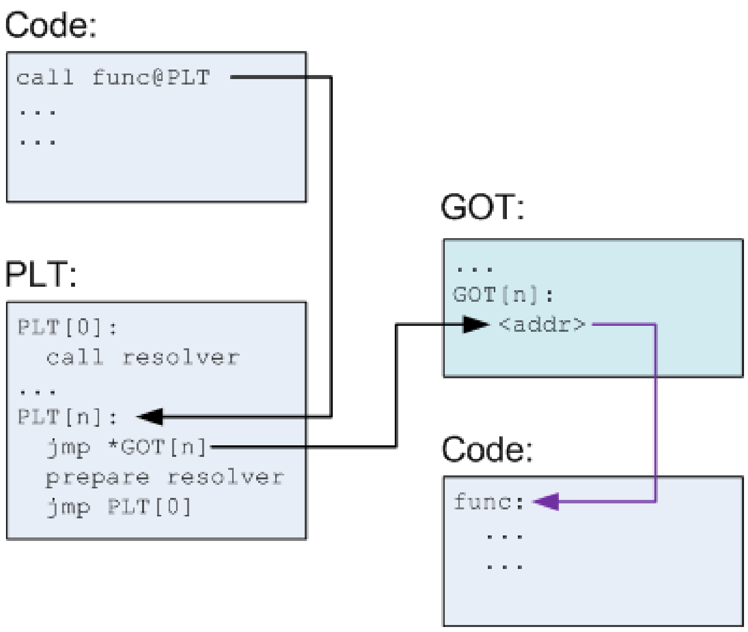

Figure 4.2: The calling process using the PLT, after it has been resolved. The top code block is the calling program, the bottom code block is the shared library [3].

The second part prepares some values on the stack, after which it calls the dynamic loader. For this it uses a special section in the PLT. This process can be seen in Figure 4.3.

For more detail on the PLT and the dynamic loader, we forward the reader to [3].

In our method, we use this information to calculate the correct function address from a return address.

Match function symbols

Function symbols (not to be confused with debug symbols) are not in memory anymore for the Linux operating system, as opposed to the related work (e.g. [25]) which works on Windows. In this method we show that we can match function addresses to function symbols, if we have access to the original binary executables or libraries. This is a new technique in memory forensics.

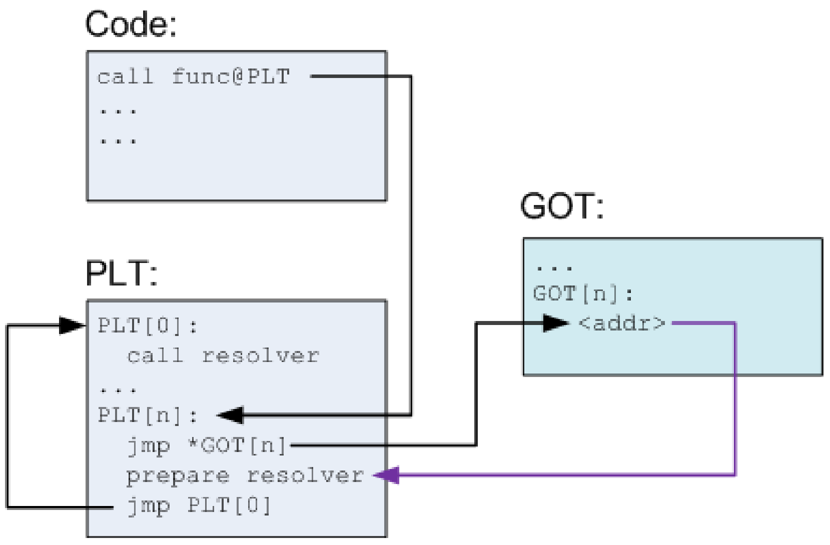

Figure 4.3: The calling of the dynamic linker using the PLT [3]. Note that the arrow originating from the GOT now points to the PLT, based on the value of `<addr>`.

Stack Frames

Stack frames are a structure on the process stack which provide data for the currently executing function. A call stack, besides storing local data, is used for storing the return address. This is the address of the next instruction, after the current function has finished. Each section on the stack, starting with the parameters of the called function and the return address, is called a frame. Figure 4.4 shows a schematic example of stack, with multiple stack frames.

The basics of the stack frame are defined by the following code:

```
caller:
    ...
    call  callee

callee:
    push  %ebp
    mov   %esp, %ebp
    ...
    leave
    ret
```

The `call` instruction places a return address on the stack. The callee has to execute its own instructions to make sure the base pointer (EBP/RBP) and stack pointer (ESP/RSP) are returned to the state they were in for the caller[2].

[1]`http://commons.wikimedia.org/wiki/File:Call_stack_layout.png`
[2]The E-variant registers are for x86, the R-variant for x86_64.

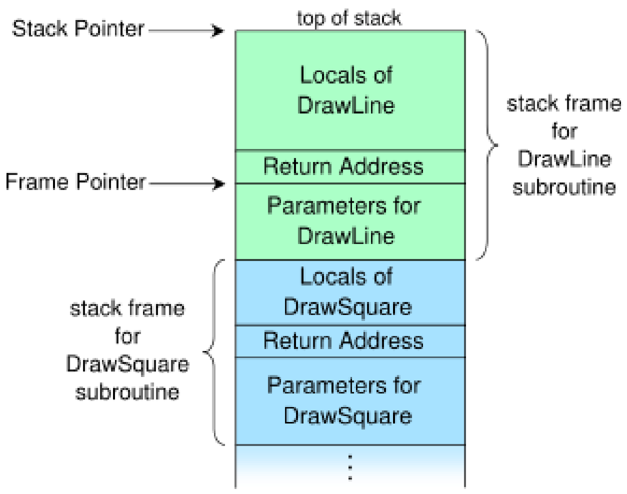

Figure 4.4: Example of stack frames[1].

Frame Pointer Optimizations

Modern compiler optimizations include an optimization which removes the use of a frame pointer. Current research in memory forensics deals with normal stack frames, not taking into account any optimizations. We expect that the compiler, as a part of the system architecture, affects memory analysis. In our approach we analyze the effects of optimizations on memory forensics.

Loader data

Not all data on the stack is useful in a forensics context. We try to distinguish between the environment, the program arguments, the loader data, the current stack frames and the old stack frames. To find where the loader data ends and the stack frames begin, we have developed a method to find basic markers, such as the start of the main function.

4.3 Summary

In figure 4.1 we showed the basic layout we started with. In figure 4.5 we have updated this diagram to show the parts we add to the current tooling (by using both existing research and new research). All new items are highlighted in grey: registers, system calls and stack analysis (frames and data). The analysis of the system call and the frames depends on both the registers and existing Volatility functionality. The other parts (including the function addresses and symbols) do not rely on the registers to work.

The impact of the components added in the figure is the improved understanding and automated analysis of user space applications and their state. The analysis and use of the registers are a basis for all of the other parts.

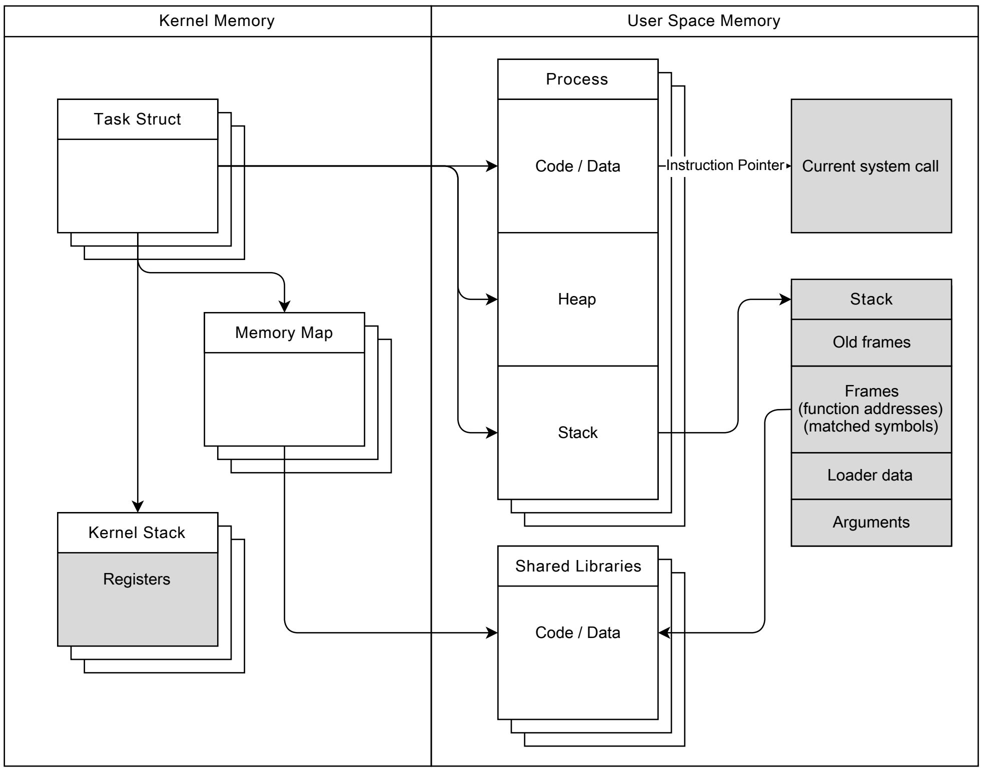

Figure 4.5: A schematic overview of our research into user space memory analysis.

Chapter 5

Data Acquisition

In this chapter, we explain the methods we use for the acquisition of memory dumps and the programs within these dumps. This chapter is meant as a partial description of our research method; every time we refer to our own test data, this is the method we have used to acquire the data. It is not a core topic of our research. We create and use our own memory dumps to have control over the system, for testing purposes.

5.1 Virtualbox

Virtualbox is the Virtual Machine system we use to acquire memory dumps. It allows us to dump the memory of the whole virtual machine while the system is running. The memory dump format is supported by Volatility.

This command lets us dump the memory of a virtual machine:

```
VboxManage debugvm "Ubuntu 13.04" dumpguestcore --filename /path/to/filename.dump
```

During the acquisition, the virtual machine is paused (by the Virtualbox system, not by us). This has the effect that there is no *smearing* (the corruption of memory by acquiring it from a running system).

5.2 Operating Systems

For our operating systems we use different distributions of Linux. Volatility already takes care of different kernel versions for us. We still have to take into account if an operating system is 32 bit of 64 bit.

List of operating systems and kernel versions used in our research:

- Ubuntu 13.04 x64
- Debian 7 x64

5.3 Applications

For our methods, we create snapshots of several applications. All applications can be found in Appendix A together with a description. The reason for their use will be described in the sections that refer to these programs.

Especially for our own test applications, it is important that we pause these applications at the correct locations in the code. One of the options to do this, is to run them in a debugger, e.g. `gdb`. This is also useful for the comparison of our results for a running application.

5.3.1 GDB

In the GNU Debugger, there are several methods to make an application pause at a specific location. This is done using breakpoints.

```
break <function name>
break <address>
break <line number>
```

This also works for interpreted languages. If the interpreter for a script calls an underlying function (shared library call), we can use this to halt the interpreter at that point.

It is important that the use of the debugger does not affect the running application. For example, by default it turns off ASLR. Because our goal is a reasonably realistic pause, we can turn this on again. To properly enable Address Space Layout Randomization in gdb, we can use:

```
    set disable-randomization off
```

It is also possible to attach the debugger to an already running program:

```
    gdb <program> <pid>
```

This lets us start the program without interference of the debugger.

Chapter 6

Core Implementation

In this chapter we try to explain the core parts of our implementation. Each topic has its own section, but because they are often related, sometimes you may find some cross-references.

6.1 Registers

In our methodology, we have explained why we are able to find CPU register contents in memory. In this section, we will explain where these registers are located (for Linux x64) and how to recover these values.

6.1.1 Finding the registers

To find and identify the registers, we first need to know their contents at any given point during process execution. We can do this by simply starting the debugger and pausing the application. In `GDB` we can show the registers with `info r`.

Then we need to find these values in memory. We can do this by making a memory dump of the system where we have the debugger running. This guarantees the same contents as we can see in the debugger. Typically, the registers are stored on the kernel stack, so that is where we can look for them.

Below we show these registers for 64 bit linux. We have found this by looking at the first items on the kernel stack, for which we can find the address by following the pointer in `task_struct.thread.sp0` (a pointer in a kernel structure).

64 bit (x86_64) Linux

These are an example of the registers we found for 64 bit Linux. The memory contents column is shorter because we found only 21 values in one place.

Debugger contents		Memory contents	
rax	0x0	0x0	r15
rbx	0x0	0x0	r14
rcx	0x19	0x7fffffffe620	r13
rdx	0xd	0x400650	r12
rsi	0x4	0x7fffffffe510	rbp
rdi	0x7ffff7bd8758	0x0	rbx
rbp	0x7fffffffe510	0x7ffff7864010	r11
rsp	0x7fffffffe4f8	0x7fffffffe2a0	r10
r8	0x7ffff7bd0660	0x7ffff785e1e4	r9
r9	0x7ffff785e1e4	0x7ffff7bd0660	r8
r10	0x7fffffffe2a0	0x0	rax
r11	0x7fffff864010	0x19	rcx
r12	0x400650	0xd	rdx
r13	0x7fffffffe620	0x4	rsi
r14	0x0	0x7ffff7bd8758	rdi
r15	0x0	0xffffffffffff	orig_rax
rip	0x7fffff864010	0x7ffff7864010	rip
eflags	0x202	0x33	cs
cs	0x33	0x202	eflags
ss	0x2b	0x7fffffffe4f8	rsp
ds	0x0	0x2b	ss
es	0x0		
fs	0x0		
gs	0x0		

The left columns show the registers as presented by the debugger. The right columns show the registers as found in order on the kernel stack. The order is identified by simply comparing values; the algorithm to restore the registers is deterministic and the order is always the same, so we manually compared and mapped the values. We identified identical values by running some extra tests until we found a difference (not shown here), and confirmed it with the source code of the linux kernel[1].

There are a few differences between the registers shown in the debugger and the values found in memory. First, it shows the segment registers ds, es, fs and gs, but these are not (at this position) in memory. Second, the value for orig_rax is both in memory and in the linux kernel source code, but does not appear in debugger output.

6.2 System Calls

In the previous section we explained how to access the registers. In this section we will explain one use of the registers, the detection of the currently executing system call. In

[1]https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/ptrace.h

this new method, we are able to recover both the name and the arguments of the system call (called by a specific process).

6.2.1 Detection

When a system call is invoked, execution is transferred to the system kernel. This has the effect that the instruction pointer register (IP) for the user space process is no longer increased until the system call has finished[2] (simplified: the process is effectively idle until the system call has finished). Only when the system call has finished, the IP is increased, pointing to the next instruction. We can use this effect to determine the last instruction; while in this 'idle' state, the last instruction will be the `syscall` instruction (for x86_64, i.e. `0x0f05`). If it is not this instruction, no system call is currently executing.

6.2.2 Matching system call and arguments

The arguments, including the selection of the system call, are passed through registers. For Linux x86_64, the convention tells us that the system call number is placed in `%rax`, with the arguments placed in `%rdi, %rsi, %rdx, %r10, %r8, %r9`.

For Linux, Volatility already offers a plugin (`linux_check_syscalls`) to read the system call table, a structure in kernel memory. Using this table, we can find the name of the current system call. The plugin can also tell us if the system call was hooked (which can be useful in analysis of malware).

In our implementation, we use this existing Volatility plugin to match our detected system call to the human readable name.

The arguments can be retrieved from the registers automatically, but no typing is done on these arguments yet in the current implementation. A researcher doing analysis still has to consult the documentation to interpret the arguments[3].

6.3 Reverse engineering the PLT

On the stack we encountered the so called *return addresses.* This chapter explains how we can use this address to determine the function to which the stack frame belongs.

6.3.1 Using the return address, example

In the methodology section we explained how the PLT works. Knowledge of how the PLT works is necessary to understand how we can use the return address the called function. In this section we will show the calculations that have to be made.

Consider the return address `0x40076e`. It is taken from an actual memory dump, but that is not really relevant here. The return address points to an instruction, typically

[2]The actual IP CPU register now points to kernel code, it's only the saved process IP that stays constant.

[3]e.g. `http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64`

the instruction directly after a CALL instruction. To find this call, we disassemble the 5 bytes before, i.e. the bytes at address 0x400769.

```
0x400769: E8 B2 FE FF FF    # CALL 0x400620
```

This is a call to a function section in the PLT. If we disassemble the instruction at that address, we get the following:

```
0x400620: FF 25 02 0A 20 00 # JMP QWORD [RIP+0x200a02]
```

For both of these, we have to disassemble using the correct offset, i.e. the correct instruction pointer. This enables us to use this approach for both the binary and for shared libraries, compiled using PIC (Position Independent Code).

This last JMP instruction is the first part of the PLT, as explained in the previous section. It lets us jump to the address that is listed at the address in the instruction (i.e. it's a pointer pointer). If we were to look in the binary, we would see the address of the next part of this function's PLT, but if we were to look in memory, we would see the correctly resolved address of the function we're calling.

The address is: RIP[4] + 0x200a02 = 0x400620 + 6 + 0x200a02 = 0x601028.

```
# In the binary
0x601028: 00 00 00 00 00 04 06 26
# In memory
0x601028: 00 00 7F FF F7 BD 86 F0
```

In memory, the value is 0x7ffff7bd86f0. If we know that our shared library is loaded at 0x7ffff7bd8000, we know that we are calling a function at offset 6F0 in our library.

Matching a function name

One of the issues in memory forensics is matching a number to a name (symbol). For Linux, symbols are unloaded when they are not needed, so they are not in memory.

One of the solutions is to use external symbols, gathered from the binary. We can use the linux tool nm. This works the same way for both the shared library and the executable.

```
# optionally with -D for dynamic symbols only
$ nm libtestlibrary.so
...
00000000000006f0 T function_one
000000000000071e T function_two
...
```

Here we can see that offset 6F0 is a function called function_one.

[4]Instruction Pointer in the x86_64 architecture

6.3.2 Limitations

There are some limitations to this approach. For address resolving, the method only works if the code contains the function. If it instead uses a register (e.g. `call %rdx` or `jmp %rdx`), as is often the case with the use of function pointers, the approach fails. Without doing advanced interpretation of the assembly code, we are unable to determine the contents of the register.

6.4 Stacks and Calling Conventions

In our methodology we introduced background information on the stack layout. In this section we show how calling conventions define this layout and how we can use this in memory forensics. Please note that the information described in this chapter is not always valid when compiler optimizations are used. For more information on that topic, see Chapter 6.5.

6.4.1 Calling Conventions

What defines the layout of the application stack? In short, nothing. An application is free to use whatever layout in memory they want. But chances are, if a specific platform or compiler is used, there are calling conventions which define the stack frames. These conventions differ per language, operating system and platform, and can be used in memory forensics to make assumptions about the stack layout.

As an example we will be using C/C++ calling conventions in this research. For brevity, we will leave floating point registers out of this document. [17] explains everything in full detail.

cdecl

The cdecl (C declaration) is a convention which originated in the C language and is used commonly on the x86 architecture. It does several things related to the stack and registers:

- Place arguments to the called function on the stack.
- The EAX register is used for the return value.
- EAX, ECX and EDX are caller-saved, the rest are callee-saved.

In this convention, the caller is responsible for cleaning the arguments from the stack.

stdcall

This convention is mostly the same as cdecl, but the callee is responsible for cleaning up the stack.

fastcall

In this convention, instead of passing the arguments on the stack, a number of registers are used, depending on the platform. This reduces the number of memory accesses required for making function calls, speeding up the call process.

thiscall

The thiscall convention is used for non-static member functions in C++. It is comparable to the cdecl convention, but in addition to the parameters, a *this* pointer to the object is added last on the stack.

x86_64

64 bit systems commonly use a system comparable to the fastcall convention, because of the increased amount of registers. 6 registers are used for the parameters, but the Windows platform uses a different set than the Linux, BSD and Mac platforms.

6.4.2 Stack Alignment

There is another convention which can be useful in reading values from the stack, which is the stack alignment. If you know something about the compiler that was used to compile a program, you can determine the likely alignment. In [17], chapter 5 shows an overview of different compilers and platforms.

6.4.3 Special stack sections

The kernel object of a process may include references to specific areas on the stack. If we look at our implementation (for x86_64 Linux), we can see that the `task_struct`[5] has references to both the environment and the program arguments. From this we can assume that these areas are common for each application and initialised by the kernel. It should be noted, however, that nothing prevents an application from modifying these sections.

6.4.4 Implementation

In our implementation we implemented the detection of stack frames by using the frame pointer backwards. For this we needed to use both the frame pointer and the stack pointer in the registers, as recovered in section 6.1. Because we chose x64 as our platform (due to time constraints), and the conventions dictate that arguments are generally not on the stack but in registers, we have not implemented any argument detection after finding the stack frames.

[5]`https://github.com/torvalds/linux/blob/master/include/linux/sched.h`

6.4.5 Summary

In short, we have to take into account the following parameters that affect the stack layout.

- Platform.
- Operating system.
- Compiler.
- Calling conventions.
- Stack alignment.
- Kernel initialised stack sections.
- Applications are free to use the stack in any way they want.

6.5 Optimizations

In the previous section (6.4) we discuss the recovery of stack frames with the use of the frame pointer. This section explains the effects of optimization (and related compile options) on memory forensics and the recovery of the stack frames.

6.5.1 Optimization

Modern compilers offer a large collection of optimizations that result in a reduction of code, faster code and decreased memory usage. In the coming sections we will discuss the effects of the `-O2` GCC option on the usage of the stack. This optimization setting includes omission of the frame pointer.

6.5.2 Layout before optimization

In Listing 1 we show the stack of a program (see Appendix A.1) that was compiled without any optimization.

```
Address           Value            Annotation
00007fffffffe480: 00007fffffffe4c0
00007fffffffe488: 00007ffff7bd8884 return address for 00007ffff7bd8891
                                                          ( nested_one )
00007fffffffe490: 00007fff00001343
00007fffffffe498: 0000000000001344
00007fffffffe4a0: 000000000000133c
00007fffffffe4a8: 0000133cf7bd0660
00007fffffffe4b0: 00007ffff785e1e4
00007fffffffe4b8: 00007ffff7ffe268
00007fffffffe4c0: 00007fffffffe4e0
00007fffffffe4c8: 0000000000400895 return address for 00007ffff7bd882c
                                                     ( nested_function )
00007fffffffe4d0: 0000267200000000
00007fffffffe4d8: 00007fff00002676
00007fffffffe4e0: 00007fffffffe510
00007fffffffe4e8: 0000000000400817 return address for 000000000040082d
                                                       ( library_calls )
00007fffffffe4f0: 00007fffffffe5f8
00007fffffffe4f8: 00000001004006e0
00007fffffffe500: 00007fffffffe5f0
00007fffffffe508: 000013380000000c
00007fffffffe510: 0000000000000000
00007fffffffe518: 00007ffff7831ea5 return address
```

Listing 1: Annotated version of a stack dump, unoptimized code

After the values that denote the return addresses, we've added the name and address of the functions that were called. We can see that each address above a return address points to the previous frame.

6.5.3 Layout after optimization

In Listing 2 we show the stack of the same program (Appendix A.1), but this time compiled with optimizations enabled.

```
Address           Value            Annotation
00007fffffffe4f0: 0000000000000000
00007fffffffe4f8: 0000555555554a59 return address for 00007ffff7bd87d0
                                                     ( nested_function )
00007fffffffe500: 00005555555548cc
00007fffffffe508: 00005555555548c3 return address for 00005555555549f0
                                                       ( library_calls )
00007fffffffe510: 0000000000000000
00007fffffffe518: 00007ffff7831ea5 return address for 0000555555554890
```

Listing 2: Annotated version of a stack dump, optimized code

It has been paused at roughly the same time, but because of the optimizations we had to take some liberties.

Observations

- The function `nested_one` is missing. This is because the code for this function has been inlined.
- The stack frames no longer have a base pointer (the value right above the return address, in the unoptimized case).
- Less values are saved to the stack.

Especially the lack of a frame pointer has some impact; we now rely on scanning the stack to determine the return addresses, instead of following the structure the frame pointer provides.

There have been several techniques that somewhat mitigate this problem. In debugging, for example GDB, a technique is applied called Prologue Analysis. It scans the prologue (the first few instructions) of the function and uses these to determine the stack layout[6].

In memory forensics, the method in [2] may account for the missing frame pointer by combining information about the stack trace from the program code.

6.5.4 Limitations

Because this approach is a scan for addresses, it may find addresses of functions which do not belong in the current execution trace. This happens when a function reserves space for local data without actually writing to this space.

6.5.5 Old stack

The same approach can also be applied to the old parts of the stack (everything below the stack pointer). As long as a return address has not been overwritten, it can still be found on the stack.

This information can be forensically relevant, but due to a lack of context, the accuracy of this data may be low.

6.6 Finding Main

The main function is a special case. It has a return address just like any other function, but it is of a different nature. The issue here is that the call of main, the start of the program, happens from the main Linux library, libc. The call is not quite comparable to 'normal' calls in that it happens through a `CALL %REG` construction (it could be any

[6] `http://sourceware.org/gdb/onlinedocs/gdbint/Algorithms.html`

register). The reason for this is that the address of the main function is not known to the library beforehand. In this section, we will explain how we can find this address, as well as use it to find the address of the main function itself. The goal is to determine the section of the stack that is more likely to have application data. In itself, it is also an extra validation of the start of the stack.

6.6.1 Without optimization

For finding the return address in a scenario without optimization, we will use the stack example from the previous section, Listing 1.

Without optimization, we still have the frame pointer to use as a reference. As soon as the frame pointer becomes zero (address `0x7fffffffe510`) in Listing 1, the following return address is the return address for `main`. To identify the address of the main function, we can use the same approach as explained in the next section.

6.6.2 With optimization

With optimization on, we no longer have access to the frame pointer, so we have to try something else.

In the examples (Listing 1, 2), the return address of `main` was `0x7ffff7831ea5`[7], which is offset `0x21ea5` into `libc-2.17.so` (the version of libc we are currently using). Right before this offset, we find the code displayed in Listing 3.

```
21e9b:       48 8b 4c 24 18          mov    0x18(%rsp),%rcx
21ea0:       48 8b 10                mov    (%rax),%rdx
21ea3:       ff d1                   callq *%rcx
```

Listing 3: Code in libc

The example in use here was `libc-2.17.so` from Ubuntu Linux 12.04. To show that there is similar code in other versions, we have shown the relevant code from versions in both Arch Linux and Debian Linux in Listing 4.

```
# Arch Linux
21a0b:       48 8b 10                mov     (%rax),%rdx
21a0e:       48 8b 44 24 18          mov     0x18(%rsp),%rax
21a13:       ff d0                   callq  *%rax

# Debian Linux
1ec89:       e8 bc 79 01 00          callq *0x18(%rsp)
```

Listing 4: Code in libc from other Linux versions

[7]It is equal because ASLR was not enabled in these experiments.

Each of these pieces of code does the same thing: call an address at the offset `0x18` of the register rsp, otherwise known as the value located three words below the return address we are currently looking at. What is this value? It is the start of the main function in the executable.

In our implementation, we've taken the following approach:

- Scan for pointers into `libc` code.
- Check the code right before the offset for the offset from the register rsp.
- Check for the address of main at this offset.

If these points all match, we have found the address of the main function and its return address.

6.6.3 Implementation boundaries

Because we state that we scan for specific pointers into `libc` code, it would be nice to have some validation of this scan.

In our experiments we noticed three things:

- A part of the ELF header remained in memory, including the *Entry Point*.
- The start code (`_start`) generated by GCC is almost the same for multiple versions.
- The start code calls `__libc_start_main`, which calls the program's `main`.

These three points enable us to calculate the return address for the function that calls the main code, `__libc_start_main`, by simply adding a fixed offset to the entry point. This value should be somewhere on the stack. Scanning for this address is easier than scanning for a return address into `libc` and provides a bit of extra validation for the whole process.

6.6.4 Limitations

Both finding the main return address and the __libc_start_main return address rely on very specific pieces of code. While these pieces of code may commonly be the same on multiple systems, there is no guarantee that it is. Research could be done into reverse emulating the code to calculate the contents of the register. The alternative is to store signatures for specific libc versions.

6.7 Limitations

In this chapter we discuss some general limitations which we have not addressed earlier (because they did not fit the rest of the topics).

6.7.1 Swap

An issue which we encountered frequently during our research was the unavailability of pages due to swap. In our implementation, we frequently disassemble code from shared libraries, e.g. `libc`. Due to the linux swap implementation, it is possible that a certain page in a shared library is swapped out for one process, but not for another [20]. This has the effect that our methods may work only for a part of the processes in the memory dump.

6.7.2 Platform dependence

If you look at our description of the implementation, each part depends on either platform conventions or platform specific instructions. While the approach may be generalized to work for all platforms, for the implementation this is not the case.

Chapter 7

Considerations

In this chapter we will describe some items we have taken into consideration during our research, but on which we have not done any in depth analysis. The main item is the effect of different compile options. Others items we have investigated are static compilation and stripped binaries.

7.1 Compile Options

In this section we will explain a number of (security) compile options and determine if the affect the analysis of memory. For the list of options we have used Debian's Hardening documentation[1] in combination with the GCC compiler. Another item we have investigated is static compilation and stripped binaries.

7.1.1 Hardening

We have used this list of common compile options for GCC:

- Format string security.
- Buffer overflow protection (fortify source).
- Stack protector.
- Position Independent Executable (for ASLR).
- Relocation Read-Only (RELRO).
- Bind Now.

In the following sections we will explain why these options do or do not affect memory analysis. It should be noted that changes in the code do not necessarily change the layout of the stack or other items in memory.

[1] `http://wiki.debian.org/Hardening`

Format string security

Compilation option: -Wformat -Wformat-security -Werror=format-security.

Effect: Warns about insecure printf/scanf functions.
Affects memory analysis: No, because the changes are compile-time only.

Fortify

Compilation option: -O2 -D_FORTIFY_SOURCE=2
Note: the option only works with an optimization level (-O) of at least -O1.

Effect: Replace insecure unlimited length buffer function calls with length-limited calls.
Affects memory analysis: No, only changes in function calls.

Stack Protector

Compilation option: -fstack-protector –param ssp-buffer-size=4.

Effect: This option adds safety checks against stack overwrites. Effectively, this places a so-called 'stack cookie'.
Affects memory analysis: Potentially.

This option will add a value on the stack. This is not interesting when following values such as the EBP and ESP, but when you need to look at local values or arguments, you need to keep in mind that you can ignore this value.

Position Independent Executable

Compilation option: -fPIE -pie.

Effect: Generate code which only uses relative offsets, so it can be used in combination with ASLR.
Affects memory analysis: Potentially.

This option generates somewhat different instructions for function calls. However, these instructions are the same as the function calls in shared libraries (PIC), which we have already shown can be used in stack frame identification. ASLR in itself is no issue for memory analysis, because we can identify the correct mappings.

RELRO and Bind Now

These last two options affect the dynamic loader.

Compilation option: -Wl,-z,relro,-z,now.

Effect: RELRO makes sure several sections in the binary are turned Read-Only before handing control over to the program. Bind Now makes sure to resolve all dynamic functions before handig control of the GOT over to the program, so it too can be made Read-Only.
Affects memory analysis: Only Bind Now.

Read-Only sections do not affect memory analysis in itself. However, dynamic resolving of function symbols in the Global Offset Table means that you can determine which functions have not been called yet. If the symbols have been resolved beforehand, this is no longer possible. We refer back to section 6.3 for an explanation on dynamic resolving of symbols.

7.2 Static compilation

In static compilation, no shared libraries are used in the linking phase of the compilation. This means that it does not use shared libraries in the final executable and therefore no function symbols need to be included. If the executable is stripped (meaning unneeded symbols are removed), no function symbols can be recovered, only function addresses.

7.3 Stripped executables

In a stripped executable, only dynamic symbols (used for dynamic loading) remain; all symbols which are not strictly needed are removed from the binary. This has the effect that function addresses from the binary itself can no longer be matched to the function symbols and thus can not be used in memory analysis.

From a preliminary review of the files on some Linux distributions (Debian and Ubuntu), we note that all ELF executables in the `/bin/` directory are stripped, so on a typical setup, only shared libraries can be analyzed.

Chapter 8

Case Examples

In this chapter we will describe two cases where we try to recover some specific information from a memory dump. We will do this using both existing tools, our own tools and some manual analysis.

8.1 Python networking test case

In Appendix A.2 we show a very small python test program which sets up a network connection. For the sake of creating a somewhat realistic scenario, we will block outgoing connections (on this port)[1], so that we have a moment to capture the memory. The goal of this scenario is to recover information about this attempted network connection.

8.1.1 Analysis with existing plugins

Volatility offers some existing plugins to do analysis on network connections in the memory dump.

linux_lsof

The linux_lsof plugin lists open files for all processes. For our process (10871), the python networking client, the output looks like this:

```
Pid      FD   Path
-------- ---- ----
10871    0    /dev/tty5
10871    1    /dev/tty5
10871    2    /dev/tty5
10871    3    socket:/TC:[201966]
```

This shows that the process has 3 references to its terminal (e.g. stdin, stdout, stderr), and one TCP connection. But it does not offer much detail yet.

[1]e.g. `iptables -A OUTPUT -p tcp --dport 1234 -j DROP`

linux_netstat

The linux_netstat plugin shows the open network connections. This should give us some more information about the TCP connection of our program. This is the relevant output:

```
TCP      127.0.0.1:35785 127.0.0.1:0     SYN_SENT            python/10871
```

Alright, so this shows us where the program is connecting, but why is the destination port still shown as 0?[2]

8.1.2 Our analysis

A part of our implementation could be used to do analysis on this program. How can we find the port that the program was trying to connect on?

Method: Analysis of syscalls

It is possible to determine the currently executing system call (using the method from section 6.2). Our program is 'waiting' for this system call:

```
Syscall: sys_connect
Address: ffffffff815b3360
Parameters:
    0000000000000003
    00007fffd755f3b0
    0000000000000010
    00007fffd755f0e0
    0000000000000001
    00007fffd755eba1
```

This shows that our program is currently waiting for the `sys_connect` system call. As per the documentation, this system call has 3 parameters:

- `int fd` - The file descriptor used for the connection.
- `struct sockaddr * uservaddr` - The virtual address for the `sockaddr` to be used.
- `int addrlen` - The length of the address.

The file descriptor is (as shown in the parameters) 3; as per the output of `linux_lsof`, this is the 'file' we are interested in.

The second parameter is what we are interested in. In the appendix, Listing 5 we show the approximate shape of the `sockaddr` struct.

By booting up the `linux_volshell` plugin, we can check what is on this virtual address.

[2]It is possible that this is a bug in Volatility, but it still serves as a good example.

```
>>> dd(0x00007fffd755f3b0, space = self.proc.get_process_address_space())
7fffd755f3b0  d2040002 0100007f 00000000 00000000
7fffd755f3c0  00000000 00000000 004af6d2 00000000
7fffd755f3d0  00000005 00000000 004e09c8 00000000
```

According to the struct, this translates to the following:

```
Family:         0x2 = AF_INET
Port:         0x4d2 = 1234
Address: 0x7f000001 = 127.0.0.1
```

This shows that we can retrieve information about the network connection which was not previously available (the port). With this we have reached the goal of this scenario.

```
struct sockaddr_in {
  short int          sin_family; // one of AF_INET, AF_UNIX, AF_NS, AF_IMPLINK
  unsigned short int sin_port;   // port, i.e. 0-65535
  struct in_addr     sin_addr;   // struct to IP address
  unsigned char      sin_zero[8];// remaining bytes 0
};

// used in sockaddr_in
struct in_addr {
  unsigned long s_addr;           // 32 bit IP address, network byte order
};
```

Listing 5: The definition of the `sockaddr` struct.

8.1.3 Impact

This analysis shows a method (possibly automated) which recovers new information or validates existing information. This is directly useful to a forensic analyst.

8.2 Analysis of the steam locomotive

In this section we will analyze a simple terminal program. We will do this for the joke program `sl`[3]. The program shows a steam locomotive riding over the screen as shown in Listing 8.2. As a goal for this scenario, we will try to determine just how far the steam locomotive has driven over the screen. The intent of this section is to show both the possibilities in stack analysis as well as the difficulties when modern optimizations are used.

We will do this analysis for the program compiled both with and without compiler optimizations, so that we can later use it in our evaluation to discuss the effects.

[3]`https://github.com/mtoyoda/sl`

```
                  (  ) (@@) ( )  (@)  ()    @@    O     @     O     @      O
             (@@@)
         (    )
      (@@@@)

    (   )
   ====        ________                ___________
D _|  |_______/        \__I_I_____===__|_________|
|(_)---  |   H\________/ |   |        =|___ ___|      _________________
/     |  |   H  |  |     |   |         ||_| |_||     _|                \_____A
      |  |   H  |__--------------------| [___] |   =|                        |
 ________|___H__/__|_____/[][]~\_______|       |   -|                        |
/ |   |-----------I_____I [][] []  D   |=======|____|________________________|
 =| o |=-O=====O=====O=====O \ ____Y___________|__|__________________________|
-=|___|=    ||    ||    ||    |_____/~\___/          |_D__D__D_|  |_D__D__D_|
_/      \__/  \__/  \__/  \__/      \_/               \_/   \_/    \_/   \_/
```

Listing 6: Example of the ASCII-art steam locomotive

We chose this program because it is simple to compile and analyze. In our chapter about optimization, section 6.5, we have already discussed the effects of optimization on our own test program.

Listings are included in Appendix B because of their size. It should again be noted that the memory dumps used in this example are not exactly the same.

8.2.1 Analysis

In Listing B.1 we present a stack dump of the program `sl` extracted from a memory dump using our tools. It has been compiled without optimizations, so no `-O` option.

To recover the state, we first look at the called functions. The function at the top of the stack is `usleep`. If we look at either the source or the binary, we see that the only occurrence is in the main loop of the program. This is the first indicator of the state of the program. The function `usleep` itself is not very interesting.

The second indicator in this case is the counter variable for the main loop. Because we have not used optimizations, we can identify the value `0xfffffffa` (-6) on the stack as this counter.

Because we already know something about how this program works (by checking either the assembly code or the C code), this is enough to tell that the steam locomotive has just driven over the left edge of the screen.

In Listing B.2 we have a similar stack dump, but this time for the optimized version of the program (`-O2`).

We can tell from the stack dump that the program is not exactly in the same state; it is currently writing something to the screen using the `__write` function. A write to the terminal also shows that we are still in the main loop. As a side note, because our analyzer now scans for return addresses, we also find the `usleep` function from our previous example.

Manual analysis of the optimized binary shows that the main loop counter is in a register this time, so it is not on the stack. However, we can also find that this register is saved to the stack in some of the functions that are called. Using this, we find the value `0xfffffff4` (-12) as the main loop counter.

Again, we can conclude that the steam locomotive has just driven over the left edge of the screen. However, this is not a very reliable way of recovering information from a program.

Listings of analyzer execution including debug info are included in Appendix B.2.

8.2.2 Impact

Using our tools, we can do a very quick analysis of programs that have been compiled without optimizations. It adds to the speed at which we can do manual examination of a memory dump. However, when programs are compiled with optimizations, we still have difficulties recovering information, even when we know what stack frames there are.

Chapter 9

Results and evaluation

In this chapter we will show our results and evaluate our research. Our results are displayed as statistics on the parts of a memory dump we were able to analyze. Our evaluation consists of a number of experiments; we will test systems with and without swap and with and without optimization. For the parts of our research which we can not properly test, we will present a confidence value. Finally, we will compare our work to others in the field.

9.1 Test data - Ubuntu 13.04

In this section we will show the possibilities in our implementation by giving statistical information on one of our test memory dumps. It includes some basic information, such as the number of tasks and threads, which we need to calculate averages, but are otherwise not part of our work. The rest of the values are explained as part of this example. We will only show the results for one memory dump; each memory dump is different and an average for multiple memory dumps does not add any value.

Total tasks:	70	
Tasks ignored:	42	
Actual tasks:	28	
Threads:	76	
Found __libc_start_main:	9	9/28 = 32%
Found main:	6	6/28 = 21%
Tasks (0 frames)	5	5/28 = 18%
Threads (0 frames)	62	62/76 = 81%
Total frames	649	
Found function address	571	571/649 = 88%
Symbols matched	131	131/571 = 23%
Syscall:	98	98/104 = 94%

Table 9.1: Results testdata Ubuntu 13.04

In table 9.1 we show these basic results of our implementation. The results can be interpreted as follows.

Of the 70 tasks in our memory dump, 28 could be analyzed. The remaining 42 tasks are things like kernel threads, which are different from normal tasks and are not part of this research. This matches the output of the volatility `linux_pstree` as well as the output of `ps` on the system. The remaining 28 tasks also had a combined 76 threads, which gives a total of 104 threads to analyze.

Of these 104 threads, 94% could be shown as waiting for a syscall.

To show the results for sections 6.3, 6.4 and 6.5, we look at the number of frames we were able to find and analyze. In total, we found 649 possible frames. Of these frames, we were able to match 571 (88%) to a function. Of these addresses, 131 could be matched to a function symbol if we used all `.so` files from `/lib` and `/usr/lib`. This can be explained if you take into account that an application may first call a few internal functions before calling shared library functions. We did not include symbols for internal functions of standard Ubuntu applications because we found that 100% of the executables were stripped.

In the analysis of the main tasks, we could validate 32% by finding the return address of `__libc_start_main` and 21% by positively finding `main` itself. This is the result of our work in section 6.6. During our experiments, we have concluded that some of these could not be validated because parts of `libc` was swapped out. Threads were not included in this statistic, because we do not expect threads to have these function frames.

In some of the analyzed tasks and threads we failed to find any frames (i.e. return addresses) at all. This happened in 18% of tasks and 81% of threads. This was also the effect of swap.

9.2 Tests and evaluation of swap use

In sections 6.7 and 9 we state that the lack of access to the swap file or partition is a problem for analysis on user space memory. In this section we will evaluate the impact of swap on memory dump analysis.

We will do this by creating memory dumps with and without swap. These will be analyzed and we will compare the statistics to determine the effect. It should be noted that these images are not exactly the same (this is hard, if not impossible).

First, we present data for a freshly booted system. Table 9.2 represents an Ubuntu system which has just booted, with a swap partition enabled. As we can see, we are able to recover frames in all threads and processes. If we look at table 9.3, we get fairly similar results.

Now we will look at a system which has been in use for a while. What we expect to see is that we will have more difficulty doing analysis on a system which has swap enabled.

To fill the memory, we simply loaded some large files. In table 9.4 we can see the results of this for a system with swap disabled. We see here that it is very similar to a

Total tasks:	73	
Tasks ignored:	46	
Actual tasks:	27	
Threads:	76	
Found __libc_start_main:	22	22/27 = 81%
Found main:	14	14/27 = 51%
Tasks (0 frames):	0	0/27 = 0%
Threads (0 frames):	0	0/76 = 0%
Total frames:	1487	
Found function address:	1220	1220/1487 = 82%
Symbols matched:	510	510/1220 = 41%

Table 9.2: A freshly booted Ubuntu system, with swap enabled.

Total tasks:	75	
Tasks ignored:	46	
Actual tasks:	29	
Threads:	76	
Found __libc_start_main:	22	22/29 = 75%
Found main:	15	15/29 = 51%
Tasks (0 frames):	0	0/29 = 0%
Threads (0 frames):	0	0/76 = 0%
Total frames:	1551	
Found function address:	1285	1285/1551 = 82%
Symbols matched:	543	543/1285 = 42%

Table 9.3: A freshly booted Ubuntu system, with swap disabled.

freshly booted system.

In table 9.5 we have a system with swap enabled and memory filled. To give a clear example of properly swapped data, we created some extra processes to get a good amount of swap; a total of 89M (on a 256M memory system). This table gives us a different result; we are unable to find frames in about 84% of all threads and 61% of all processes. It also affects our ability to find the return addresses for main and __libc_start_main, reducing it to just 16% and 8%, as well as the total number of frames (445), showing that parts of the stack get swapped out.

These results show the importance of the analysis of swap space when doing analysis on user space memory. However, it should be noted that modern amounts of RAM (e.g. 8+ GB) are not filled as easily and the impact may be smaller in realistic settings.

9.3 Registers

In section 6.1 we explain how we use kernel structure to recover the contents of CPU registers for processes and threads. There is no way to determine if the values of these

Total tasks:	75	
Tasks ignored:	42	
Actual tasks:	33	
Threads:	76	
Found __libc_start_main:	26	26/33 = 78%
Found main:	19	19/33 = 57%
Tasks (0 frames):	0	0/33 = 0%
Threads (0 frames):	0	0/76 = 0%
Total frames:	1460	
Found function address:	1197	1197/1460 = 81%
Symbols matched:	513	513/1197 = 42%

Table 9.4: An Ubuntu system with filled memory, swap disabled.

Total tasks:	91	
Tasks ignored:	42	
Actual tasks:	49	
Threads:	76	
Found __libc_start_main:	4	4/49 = 8%
Found main:	8	8/49 = 16%
Tasks (0 frames):	30	30/49 = 61%
Threads (0 frames):	64	64/76 = 84%
Total frames:	445	
Found function address:	379	379/445 = 85%
Symbols matched:	65	65/379 = 17%

Table 9.5: An Ubuntu system with filled memory, swap enabled.

registers are correct, except for manually checking a single process. We can however provide a confidence value. For this we used the following properties:

- The register RSP, the stack pointer, should point to the stack of the thread. We can determine which memory section is the stack for this process by using different `task_struct` values. RSP should point to this section.
- The register RIP, the instruction pointer, should point to an executable memory section.

We have calculated this for all memory dumps used above (i.e. the swap tests and the optimization tests) and found that these properties hold for 100% of all processes and threads.

The extraction of the registers was the base of our research; without it, none of the other parts were possible.

9.4 System call analysis

We were able to detect system calls, as shown in section 6.2 and the example in section 8.1. In table 9.6 we show which syscalls were found in a typical memory dump.

In the dump in question, this showed 94% of processes waiting for a system call, with the other 6% as not waiting for a system call. Results in other memory dumps may vary.

For this detection method, we have used the premise that the instruction pointer for an idle process points to a system call. We can combine this with the assumption that most processes on a system are in an idle state. Our results show this to be true.

In the example (section 8.1) we show how this system call can be used by a forensic analyst to learn additional information about a running process. In itself, most system calls are not very interesting, wait or sleep give no information. However, `sys_connect` (as in the example) or `sys_read` (amongst others) can show locations of buffers or sockets as used in the program. For example, the editor nano will frequently hang in the `sys_read` system call, waiting for input.

The limitation of this approach, as with all other memory analysis, is that it is a snapshot of the system. If a process is not currently executing a system call, there is nothing to analyse.

sys_ioctl	63
sys_poll	8
sys_select	7
sys_futex	6
sys_epoll_wait	4
sys_nanosleep	3
sys_read	3
sys_wait4	3
sys_rt_sigtimedwait	1

Table 9.6: Number of syscalls detected

9.5 Stack analysis

There are a few things we need to evaluate regarding the detection of stack frames, the calculation of function addresses and the matching of function symbols. The main way we do this is the manual inspection of sample processes taken from memory dumps. We also use the results on dumps from systems with swap disabled, because section 9.2 shows that this significantly changes the results.

First, we need to validate the total amount of stack frames detected in a memory dump. The results in 9 show totals ranging from 445 to 1551 stack frames. In section 9.2 we have already explained the difference in these numbers. However, to check if these numbers are correct totals, we need to compare them to the actual number in a running system. Because of the difficulty of doing so (it is practically infeasible because of our

__libc_free	111
pthread_mutex_unlock	74
ioctl	64
posix_memalign	59
memalign	59

Table 9.7: Top 5 detected symbols

inability to pause the system yet still work with it) we have compared a number of samples.

From these samples we have concluded the following:

- In all cases, the current frames were detected.
- In some cases, extra (old) frames were detected (example: Listing B.2).
- In some cases, the frames detected matched the program exactly (example: Listing 2).

This shows that the detected number of total frames is always equal or higher than the actual number of frames. Because each process is different, we decided it was not useful to calculate any averages from these samples. This is also because a forensic analist is usually interested in specific process.

Next, we look at the number of return addresses for which we could not calculate a function address. In section 6.3.2 we already state that we have problems detecting the correct function if the call uses a register instead of an address. From the results (section 9) we see that this happens in 12% to 19% of all cases.

In section 6.6 we describe a method for the case where the call to a register is the call to the main function. This is a specific instance of the register call problem, evaluated in section 9.6. If we look at the results for the memory dumps without swap (to not skew the results), we see that this is only effective in about half of the cases (51-57%). This shows that extra research is needed to apply proper reverse engineering techniques to this forensic problem. In table 9.7 we show the top 5 detected symbols in a memory dump. The numbers were created by doing analysis given only the symbols of shared libraries, not executables, and may change depending on running programs and library versions used on the operating system of which the dump was made. The symbols shown are typical low level memory and application management functions on a linux system. The detection of these symbols and frames can potentially be used by a forensic analyst to gain insight into the data layout.

9.6 Register call analysis

Because our results show that a significant amount of functions for stack frames could not be recovered due to register calls, we have further analyzed these frames. For this

File	Offset (bytes)	Count
/lib/x86_64-linux-gnu/libc-2.17.so	1023515	76
/lib/x86_64-linux-gnu/libglib-2.0.so.0.3600.0	446131	66
/lib/x86_64-linux-gnu/libglib-2.0.so.0.3600.0	299482	7
/opt/VBoxGuestAdditions-4.2.8/sbin/VBoxService	174153	7
/lib/x86_64-linux-gnu/libc-2.17.so	1109675	5

Table 9.8: Table showing the top 5 most frequent locations where a register call occurs. The first column lists the executable or shared library where the call originates. Offset means the offset in bytes within this file. The final columns shows how often this register call occurs within a full memory dump. The full version of the table can be found in the appendix, table C.1.

analysis, we have used a memory dump without swap, earlier used in section 9.2, table 9.3.

Out of all frames (1485) we found 199 return addresses defined by a call to a register, an average of 13.4%. First, we look at how these frames are divided over all processes. The results of this are plotted in Figure 9.1. The number of register calls per process or thread ranges from zero to 50% (3 out of 6). Most threads have at least a few register calls, showing the impact of this problem.

Next, we look at the origin of these register calls. Table 9.8 shows the five most common origins, defined by library and offset of the register call. The table shows an interesting result, namely that two thirds of all register calls originate from just two call instructions. Upon further analysis, the functions that contain these calls are `libc:__clone` and `glib:g_test_init`.

The first function, `__clone`, is used to create threads. The number of occurrences of this function (76) corresponds exactly to the amount of threads we have analyzed, validating our detection of this call.

The second batch of calls appears to originate from the function `g_test_init`, a part of the testing framework for GLib.

By implementing analysis of these functions, a large part of the register calls can be detected. We have not implemented this dus to time constraints.

9.7 Method weaknesses

The methods in our research are designed to help a forensic researcher start an analysis of a memory dump. In this context, we need to look at how our detection and analysis methods can be avoided by an 'attacker' (e.g. a piece of malware).

Since our methods work on a low level, it is able to detect any basic program functionality. These methods of subterfuge come to mind:

- No use of function calls. This may be feasible for small or handcrafted malicious applications. Return addresses, regardless of the source, are always detectable (inherently to the architecture).

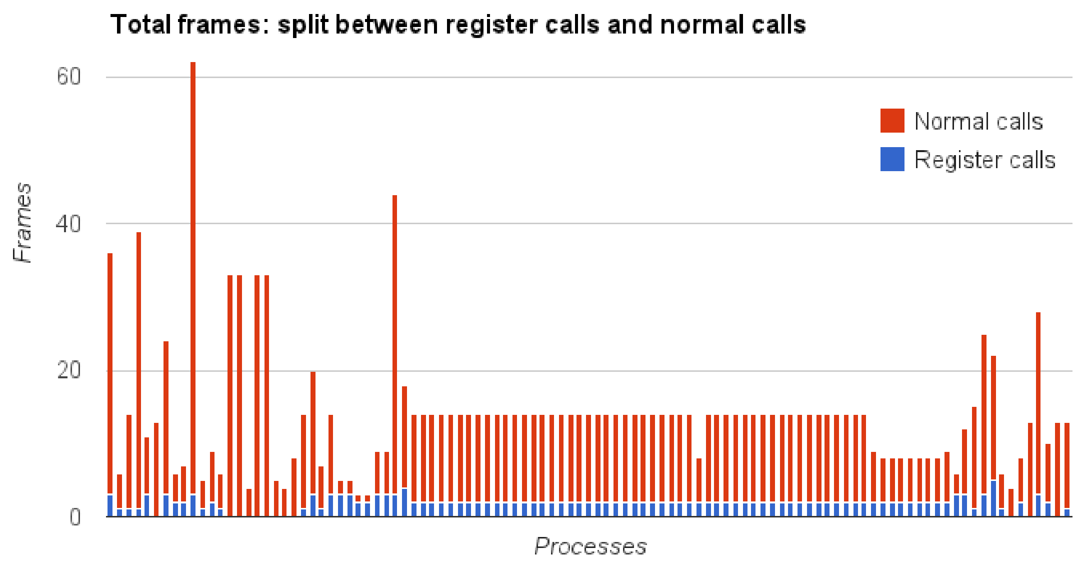

Figure 9.1: This chart shows the total number of frames, one bar per process/thread. The bars are divided between frames resulting from normal calls (red) and register calls (blue).

- Proper stack cleanup. This prevents the detection of old stack frames and data.
- No use of system calls. This is infeasible if the program wants to actually achieve anything.
- Static compilation. This prevents the use of function symbols.
- Use register calls to call functions. This makes automatic analysis harder, however it is possible to do manual analysis on these calls.

None of our methods were directly intended to fix any of these problems, however it is good to keep these points in mind when applying this and future research.

9.8 Literature Comparison

In this section we will evaluate our work by comparing it to the related work, to see what we add to the field of memory forensics.

9.8.1 Registers

To compare the recovery of registers, we look at [29]. In this research, the registers were recovered as part of a bigger project and were not specifically evaluated. In our work, we have shown we could recover the registers by using the same approach, for x64. In addition to that, we have provided a confidence value for the correctness of the contents of these registers.

The analysis of system calls using the registers is new in the context of memory forensics, so we are not able to compare this to existing literature.

9.8.2 Frames and function calls

For the evaluation of frames and function calls, we look at Hejazi et al. [25] and Arasteh et al. [2], who investigate the same topic.

In [25] there is a focus on forensically sensitive data. They are able to find stack frames in a Windows process and continue to analyze the specific functions that are forensically sensitive. In this analysis they make no mention of frame pointer omission, which we do take into account. [2], which describes te recovery of the stack trace by comparing it to the program model, does take into account frame pointer omission and succeeds in recreating a stack trace. This is something we have not been able to do in our research.

In both works, they briefly describe the methods to recover function addresses and function symbols, which is apparently easier for Windows systems. In our work, we have introduced methods for both of these things for the Linux platform.

In neither of these papers, the possibility of function calls in the `call %reg` form have been taken into account. We have defined this problem and determined the impact of this problem.

Other problems we have defined which have not been described earlier are the swap problem and the problem of finding the `main` function (the last one may be specific to Linux).

9.9 Summary

In this section follows a summary of our contribution to the field of memory forensics. As per our methodology, we have implemented a number of items for Linux user space memory analysis, including:

- Recovery of CPU registers.
- Analysis of system calls.
- Analysis of return addresses/frames.
- Matching function symbols to return addresses.

In the context of this implementation, we have evaluated our work and some problems that impact the analysis of user space memory. Our evaluation includes:

- A confidence value on the correctness of the recovered register values.
- An explanation on the use of system calls.

- Manual examination on the recovery of frames and matching function symbols.

Of the problems we have encountered, we have done an extra evaluation of these problems:

- The use of swap, which currently has no implementation in Volatility.
- The impact of return addresses from register calls, which are harder to analyze.

9.10 Conclusion

Our methods performed with reasonable accuracy, being able to extract up to 88% of function addresses from stack frames, as well as being able to match function symbols to (a part of) these addresses. Using the extracted registers (100% confidence value) we were also able to recover the current system call.

The used methods were also able to recover frames when compiler optimizations were in use, at the cost of false positives (older frames, the number varying with each program).

Finally, the methods helped identify and explain problems in memory analysis, such as (unanalyzed) swap use and function register calls.

Chapter 10

Conclusion

In this study, we tried to define the state of the art of user space memory analysis, to identify the problems that are encountered and to implement possible new methods to further the field. In this chapter we will conclude with answers to the questions we asked at the start, a conclusion with an answer to the main research question and a summary of possible future work.

10.1 Answers to research subquestions

In this section we will try to answer the subquestions to our research question.

10.1.1 What can we learn from the stack of a process?

For this question, we will look back at sections 6.4 and 6.5. We have seen that we can recover frames with and without the frame pointers (the latter with lesser accuracy). We have also shown that the structure greatly depends on the architecture and the relevant calling conventions.

On architectures with more registers, such as x86_64 in our proof of concept implementation, we have seen that less information is available on the stack because arguments to procedures will be passed in registers.

Furthermore, we have identified and analyzed the problem of register calls, which increase the difficulty of stack analysis.

Except for basic data such as arguments, environment and the current stack, we also have access to old data of the stack (section 6.5.5). Common calling conventions specify nothing about the stack cleanup, which means that old data is left here, however possibly corrupted (overwritten).

These techniques are relevant for a forensic analyst because they can give an initial view of the memory and can help in deciding where to focus their attention.

10.1.2 How can we analyze the use of libraries?

We can analyze the use of libraries by looking at references to these libraries, such as return addresses on the stack. We have shown in our proof of concept (section 6.3) that we can calculate the address of the function that was called, and the results (chapter 9) show we can do so with a reasonable accuracy.

Given that we have access to basic dynamic function symbols, we have also shown that we can match these to the functions. However, these commonly cannot be found in memory.

10.1.3 Is the access to source or header files mandatory to perform memory examination?

In our research as well as our proof of concept, we have not made use of source or header files.

Header files give no additional info on the existence of certain dynamic functions/symbols in a library, however they may give an insight into the parameters of functions and the shape of data structures. This could be used in the step after the basic analysis of the stack.

Source files may give insight for a forensic analyst, but due to compile options they may also give a slightly distorted image, compared to the actual binary. For example, optimizations may inline a function where you would not expect it (see also sections 7,6.5).

10.1.4 How does the operating system or system architecture affect application analysis?

To answer this question we will have a look at our proof of concept implementation.

First, calling conventions (section 6.4) are specific to a combination of the architecture and the compiler. Second, register contents are architecture specific, and the recovery of these contents is operating system specific. Third, common application (ABI) structures (such as the PLT, section 6.3), while common concepts, depends on the operating system and are hard to abstract in an implementation.

On a higher level, we have also put a number to the impact of swap on memory analysis. This shows how the operating system configuration affects analysis.

In summary, during our implementation we have used various bits of the operation system and the architecture which could have a large effect on application analysis.

10.1.5 What are the difficulties in analysis of applications written in VM-based or interpreted languages?

We have not spent a lot of time investigating this question, but we did do analysis of a Python process in section 8.1. From this we would like to conclude that it is possible to examine an interpreter by analyzing its low level (operating system) functionality.

10.2 Conclusion

We started this thesis with an explanation of where memory forensics fits in the field of digital forensics. With this field growing, so grows the importance of memory forensics. When we started this study we asked the question how we could improve memory forensics, and decided to tackle the analysis of user space applications. The question we asked was:

> Can we define a generalizable approach for analysis of user space applications in volatile memory?

In the answers to the subquestions above we have covered a broad array of items that are useful or even required to analyze user space memory. Something we have not yet answered is what defines the state of an application in memory. This we have also tried to cover in sections 6.1 and 6.2, where we investigate the saved CPU registers.

With all these parts of the analysis, we believe we have an approach that includes all the essentials in user space memory analysis. This does not mean the approach is complete; there is always more to investigate, either on an abstract level or platform or operating dependent research. However, we can call this approach a solid basis for further investigation.

We hope that the reader now has a clear view on the use of memory forensics. Furthermore, we end this thesis with the hope that the reader learned something new and is able to use this knowledge to solve future problems.

10.3 Future Work

This section will describe some possible topics in future work.

10.3.1 Construct core dumps

A common concept in (Linux) debugging is the creation of the *core dump*. This is a dump of the memory of a process, including any relevant extra information, such as registers. This dump can be used to examine the process without keeping the process running.

One of the ideas we started this research with was the extraction of such a core dump from a memory dump, but because we were not sure we would be able to get all the information, we did not continue this idea. Now that we have a more clear view about how to extract things like registers, the next step would include research into this extraction.

Steps to take:

- Examine the core dump process (in Linux kernel code).
- Identify all necessary parts in the ELF core dump format.

- Recover the available parts from memory.
- Re-introduce data that is not in memory.

The goal is to extract enough data from memory to be able to analyze it in common debugging tools. This approach would skip custom tools needed to analyze process memory.

10.3.2 Implement swap

Since we have shown that swap can be important, the next step would be to implement it. This would require the following:

- Identification of kernel structs for swap access.
- Knowledge of the workings of page tables in combination with swap.
- Acquisition of the swap device/file at the same time as the memory dump.

Especially the acquisition could be a challenge if the swap area is actively used.

Acknowledgements

I must thank my supervisors Jonathan and Ruud for their constant feedback and guidance during this project, as well as their patience reading too many drafts during the past months.

Furthermore, I need to thank the Volatility team for answering my questions on some of the technical parts of this research, and for awarding me the 4th place in the 2013 plugin contest.

Finally, I would like to thank all the other people that were able to answer questions on the internals of Linux and various other topics.

Bibliography

[1] M. I. Al-Saleh and Z. a. Al-Sharif. Utilizing data lifetime of TCP buffers in digital forensics: Empirical study. *Digital Investigation*, 9(2):119–124, Nov. 2012.

[2] A. R. Arasteh and M. Debbabi. Forensic memory analysis: From stack and code to execution history. *Digital Investigation*, 4:114–125, Sept. 2007.

[3] E. Bendersky. Position Independent Code (PIC) in shared libraries, 2011.

[4] C. Betz. DFRWS 2005 Forensics Challenge: Chris Betz Results, 2005.

[5] A. Boileau. Hit by a bus: Physical access attacks with Firewire. *Presentation, Ruxcon*, 2006.

[6] D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd Edition.* O'Reilly, 3 edition, 2005.

[7] M. Burdach. An Introduction to Windows memory forensic. pages 1–8, 2005.

[8] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, Feb. 2004.

[9] G. Conti, S. Bratus, A. Shubina, B. Sangster, R. Ragsdale, M. Supan, A. Lichtenberg, and R. Perez-Alemany. Automated mapping of large binary objects using primitive fragment type classification. *Digital Investigation*, 7:S3–S12, Aug. 2010.

[10] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for data structures. *Symposium on Operating Systems Design and Implementation*, pages 255–266, 2008.

[11] D. Bilby. Low down and dirty: anti-forensic rootkits. 2006.

[12] S. Davidoff. Cleartext passwords in linux memory. *Massachusetts Institute of Technology*, pages 1–13, 2008.

[13] B. Dolan-Gavitt. The VAD tree: A process-eye view of physical memory. *Digital Investigation*, 4:62–64, Sept. 2007.

[14] B. Dolan-Gavitt. Forensic analysis of the Windows registry in memory. *Digital Investigation*, 5:S26–S32, Sept. 2008.

[15] B. Dolan-Gavitt and A. Srivastava. Robust signatures for kernel data structures. *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577, 2009.

[16] M. Dornseif. 0wned by an iPod. *Presentation, PacSec*, 2004.

[17] A. Fog. Calling conventions for different C++ compilers and operating systems. *Copenhagen University College of Engineering*, 2012.

[18] G. Garner. Forensic Acquisition Utilities, 2006.

[19] G. M. Garner. DFRWS 2005 Forensics Challenge: Kntlist, 2005.

[20] M. Gorman. Understanding The Linux Virtual Memory Manager. (February), 2004.

[21] M. Graziano, A. Lanzi, and D. Balzarotti. Hypervisor Memory Forensics. *s3.eurecom.fr*.

[22] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Memory-Only Operating System Fingerprinting in the Cloud. 2012.

[23] J. Halderman and S. Schoen. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM - Security in the Browser*, 52(5):91–98, 2009.

[24] A. Hay. Forensic Memory Analysis for Apple OS X. 2012.

[25] S. Hejazi, C. Talhi, and M. Debbabi. Extraction of forensically sensitive information from windows physical memory. *Digital Investigation*, 6:S121–S131, Sept. 2009.

[26] R. Love. *Linux Kernel Development 3rd Ed.* Novell Press, 2010.

[27] H. Macht. Live Memory Forensics on Android with Volatility. *informatik.uni-erlangen.de*, (January), 2013.

[28] A. Martin. Firewire memory dump of a Windows XP computer: a forensic approach. *Black Hat DC*, pages 1–13, 2007.

[29] E. Mougoue. Forensic Analysis of Linux Physical Memory: Extraction and Resumption of Running Processes. (May), 2012.

[30] J. Okolica and G. L. Peterson. Windows operating systems agnostic memory analysis. *Digital Investigation*, 7:S48–S56, Aug. 2010.

[31] F. Olajide, N. Savage, and C. Shoniregun. Digital Forensic Research - The Analysis of User Input on Volatile Memory of Windows Application. pages 231–238, 2012.

[32] G. Owen. Automated forensic extraction of encryption keys using behavioral analysis. *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, pages 171–175, June 2012.

[33] N. L. Petroni, A. Walters, T. Fraser, and W. a. Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, Dec. 2006.

[34] T. Pettersson. Cryptographic key recovery from linux memory dumps. *Chaos Communication Camp*, 2007.

[35] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. *Black Hat DC*, 2007.

[36] B. Schatz. BodySnatcher: Towards reliable volatile memory acquisition by software. *Digital Investigation*, 4:126–134, Sept. 2007.

[37] A. Schuster. Pool allocations as an information source in Windows memory forensics. *Conference Proceedings on IT-incident management and IT-forensics*, pages 104–115, 2006.

[38] A. Schuster. Searching for processes and threads in Microsoft Windows memory dumps. *Digital Investigation*, 3:10–16, Sept. 2006.

[39] A. Schuster. The impact of Microsoft Windows pool allocation strategies on memory forensics. *digital investigation*, 5:58–64, 2008.

[40] M. Simon and J. Slay. Recovery of Skype Application Activity Data from Physical Memory. *2010 International Conference on Availability, Reliability and Security*, pages 283–288, Feb. 2010.

[41] J. Solomon, E. Huebner, D. Bem, and M. Szeynska. User data persistence in physical memory. *Digital Investigation*, 4(2):68–72, June 2007.

[42] M. Suiche. Mac OS X physical memory analysis. *Blackhat DC*, (February), 2010.

[43] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3-4):175–184, Feb. 2012.

[44] R. van Baar, W. Alink, and a.R. van Ballegooij. Forensic memory analysis: Files mapped in memory. *Digital Investigation*, 5:S52–S57, Sept. 2008.

[45] A. Vidstrom. Forensic memory dumping intricacies - ntsecurity.nu, 2006.

[46] A. Vidstrom. Memory dumping with NtSystemDebugControl - ntsecurity.nu, 2006.

[47] A. Walters. Volatility — Memory Forensics — Volatile Systems, 2007.

[48] A. Walters and N. Petroni. Volatools: Integrating Volatile Memory into the Digital Investigation Process. *Black Hat DC 2007*, 2007.

Appendix A

Test Programs

A.1 Shared Library Calls

This C program calls both a custom shared library as well as the standard `libc`. Comments describe optional positions in the code to make memory dumps. It consists of two .c files and two .h files.

testprogram2.h

```
int internal_only(int a);
void library_calls();
```

testprogram2.c

```
#include <stdio.h>
#include "testprogram2.h"
#include "testlibrary2.h"

int main(int argc, char *argv[])
{
    // dump here (1) right after main
    int test = 5 + 7;
    // dump here (2) right after a silly instruction
    printf("Start of main, argc was %d\n", argc);
    int one = internal_only(0x1337);
    // dump here (4) check the return value
    printf("We are now going to do library calls\n");
    library_calls();
    // dump here (8) void return value
    return 0;
}

int internal_only(int a){
    // dump here (3) after a function call
    return a + 1;
}

void library_calls()
{
    printf("Start of library_calls() function\n");
    int one = function_one(0x1338, 0x1339);
    printf("result of function_one was %d\n", one);
    int two = function_two(0x133a, 0x133b);
    printf("result of function_two was %d\n", two);
    int nested = nested_function(0x133c);
    printf("Result of nested_function was %d\n", nested);
}
```

testlibrary2.h

```
int function_one(int a, int b);
int function_two(int c, int d);
int nested_function(int e);
int nested_one(int f, int g, int h, int i, int j, int k, int l, int m);
int nested_two(int n, int o);
```

testlibrary2.c

```
#include "testlibrary2.h"
#include <stdio.h>

int function_one(int a, int b)
{
    printf("Called function_one with combined parameters a+b = %d\n", a + b);
    return a + b + 1;
}

int function_two(int c, int d)
{
    printf("Called function_two with combined parameters c+d = %d\n", c + d);
    return c + d + 1;
}

int nested_function(int e)
{
    printf("Called nested_function, parameter e + 1 = %d\n", e+1);
    int bla = nested_one(0x133d, 0x133e, 0x133f, 0x1340, 0x1341, 0x1342,
                                                            0x1343, 0x1344);
    return bla+e;
}

int nested_one(int f, int g, int h, int i, int j, int k, int l, int m)
{
    // dump here (5) after a function with many arguments
    printf("Sum of arguments of nested_one was %d\n", f+g+h+i+j+k+l+m);
    // dump here (6) what happened to the registers?
    int two = nested_two(0x1345, 0x1346);
    return f+g+h+i+j+k+l+m+two;
}

int nested_two(int n, int o)
{
    // dump here (7) deepest stack
    return n + o;
}
```

A.2 Python TCP Client

This program is a python script that uses `socket.create_connection` to set up a connection. This high level function calls the C functions `getaddrinfo` and `connect`.

```
import socket
connection = socket.create_connection(('localhost', 1234))
```

Appendix B

Optimization Evaluation

B.1 Stack dumps

Examples of application stacks, using both optimized and unoptimized code.

Listing B.1: Annotated stack dump of `sl`, unoptimized code

```
Address           Value            Annotation
00007fffd51d22b0: 00007fffd51d22e0
00007fffd51d22b8: 0000000000400c68 return address for 00007f42500c6900
                                                               ( usleep )
00007fffd51d22c0: 00007fffd51d23c8
00007fffd51d22c8: 0000000100400950
00007fffd51d22d0: 00007fffd51d23c0
00007fffd51d22d8: 00000001fffffffa
00007fffd51d22e0: 0000000000000000
00007fffd51d22e8: 00007f424fff5ea5 return address for 0000000000400b47
                                                                 ( main )
00007fffd51d22f0: 0000000000000000
00007fffd51d22f8: 00007fffd51d23c8
00007fffd51d2300: 0000000100000000
00007fffd51d2308: 0000000000400b47
 ...
  part removed
 ...
00007fffd51d2390: 0000000000400950
00007fffd51d2398: 00007fffd51d23c0
00007fffd51d23a0: 0000000000000000
00007fffd51d23a8: 0000000000400979 return address for 00007f424fff5db0
                                                     ( __libc_start_main )
```

Listing B.2: Annotated stack dump of `sl`, optimized code

```
Address           Value            Annotation
00007fff292a9430: 00007f2b7c5fca90
00007fff292a9438: 00007f2b7c0872f3 return address for 00007f2b7c0f8b40
                                                               ( __write )
```

```
00007fff292a9440: 0000000000000013
00007fff292a9448: 00007f2b7c3cf280
00007fff292a9450: 0000000000000040
00007fff292a9458: 0000000001397fe0
00007fff292a9460: 000000000000001d
00007fff292a9468: 00007f2b7c0871d2 return address
00007fff292a9470: 0000000000000040
00007fff292a9478: 000000000000001e
00007fff292a9480: 000000000000001e
00007fff292a9488: 00007f2b7c088905 return address for 00007f2b7c087190
00007fff292a9490: 00007f2b7c3cf280
00007fff292a9498: 00007f2b7c087890 return address for 00007f2b7c0888f0
                                                         ( _IO_do_write )
00007fff292a94a0: 00007f2b7c3cf280
00007fff292a94a8: 00007f2b7c610f74 return address for 00007f2b7c0442c0
                                                          ( __sigaction )
00007fff292a94b0: 00007f2b7c5fca90
00007fff292a94b8: 00007f2b7c616654 return address for 00007f2b7c610f40
00007fff292a94c0: 00000000000001b8
00007fff292a94c8: 000000000000000b
00007fff292a94d0: 00000000fffffff4
00007fff292a94d8: 00000000013a27b0
00007fff292a94e0: 00007f2b7c5fca90
00007fff292a94e8: 0000000000400a5c
00007fff292a94f0: 00007fff292a9620
00007fff292a94f8: 0000000000000000
00007fff292a9500: 0000000000000000
00007fff292a9508: 00007f2b7c0ff934 return address for 00007f2b7c0cdd10
                                                          ( __nanosleep )
00007fff292a9510: 0000000000000000
00007fff292a9518: 0000000002625a00
00007fff292a9520: 0000000000000000
00007fff292a9528: 0000000000400a03 return address for 00007f2b7c0ff900
                                                               ( usleep )
00007fff292a9530: 00007fff292a9620
00007fff292a9538: 0000000000000000
00007fff292a9540: 0000000000000000
00007fff292a9548: 00007f2b7c02eea5 return address for 0000000000400950
                                                                 ( main )
00007fff292a9550: 0000000000000000
00007fff292a9558: 00007fff292a9628
00007fff292a9560: 0000000100000000
00007fff292a9568: 0000000000400950
 ...
  part removed
 ...
00007fff292a95f0: 0000000000400a5c
00007fff292a95f8: 00007fff292a9620
00007fff292a9600: 0000000000000000
00007fff292a9608: 0000000000400a85 return address for 00007f2b7c02edb0
                                                    ( __libc_start_main )
```

B.2 Execution (debug)

These listings are examples of output by our plugins. The first part consists of debug information from our implementation (e.g. the calculations as explained in this thesis), the second part are the resulting stack frames.

Listing B.3: Analysis of memory dump containing `sl`, unoptimized code

```
  ~  python2 volatility-svn/vol.py --profile LinuxUbuntu1304desktopx64 -f
    ~/evaluationdumps/ubuntu1304sl-optimized.dump linux_process_stack -p 1427
     -s ~/documents/ubuntusymbols -o evaluationdumps/ubuntu1304sl-optimized.
    stack.txt
Volatile Systems Volatility Framework 2.3_beta
*** Failed to import volatility.plugins.addrspaces.legacyintel (
    AttributeError: 'module' object has no attribute '
    AbstractWritablePagedMemory')
INFO    : volatility.plugins.linux.process_stack: Loading function symbols
    from directory: /home/dutchy/documents/ubuntusymbols
INFO    : volatility.plugins.linux.process_stack: Opened evaluationdumps/
    ubuntu1304sl-optimized.stack.txt for writing
INFO    : volatility.plugins.linux.process_stack: Starting analysis of task:
    pid 1427, thread name sl-optimized
INFO    : volatility.plugins.linux.process_stack:
    =================================================
INFO    : volatility.plugins.linux.process_stack: Found the stack at 0
    x00007fff29289000-0x00007fff292ab000
INFO    : volatility.plugins.linux.process_stack: Found libc (/lib/x86_64-
    linux-gnu/libc-2.17.so) at range: 0x00007f2b7c00d000-0x00007f2b7c1cb000
INFO    : volatility.plugins.linux.process_stack: Program code located at 0
    x0000000000400000-0x0000000000402a64
INFO    : volatility.plugins.linux.process_stack: Executable entry point ('
    _start' function): 0x0000000000400a5c
INFO    : volatility.plugins.linux.process_stack: Scanning for return address
     of __libc_start_main function, starting at program arguments (0
    x00007fff292aa898) downwards
INFO    : volatility.plugins.linux.process_stack: Scanned 594 stack addresses
     before finding the __libc_start_main return address
INFO    : volatility.plugins.linux.process_stack: Found the __libc_start_main
     return address (0x0000000000400a85) at address 0x00007fff292a9608
INFO    : volatility.plugins.linux.process_stack: Scanning for return address
     of main function, starting at __libc_start_main return address (0
    x00007fff292a9608) downwards
INFO    : volatility.plugins.linux.process_stack: Scanned 1 libc addresses on
     the stack before finding the main return address
INFO    : volatility.plugins.linux.process_stack: Found main stackframe at 0
    x00007fff292a9548
INFO    : volatility.plugins.linux.process_stack: The address of the main
    function is 0x0000000000400950
INFO    : volatility.plugins.linux.process_stack: No old-school stack frames
    detected, scanning for return addresses
INFO    : volatility.plugins.linux.process_stack: Scan range (%rsp to end) =
    (0x00007fff292a9418 to 0x00007fff292a9540)
INFO    : volatility.plugins.linux.process_stack: Found function address from
```

```
     instruction ['CALL', '0x7f2b7c0f8b40'] at offset 0x00007f2b7c0f8b40
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f2b7c0f8b40 = __write@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['CALL', '0x7f2b7c087190'] at offset 0x00007f2b7c087190
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['CALL', '0x7f2b7c0888f0'] at offset 0x00007f2b7c0888f0
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f2b7c0888f0 = _IO_do_write@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['JMP', 'QWORD', '[RIP+0x21ab52]'] at offset 0
    x00007f2b7c602520
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f2b7c0442c0 = __sigaction@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['CALL', '0x7f2b7c610f40'] at offset 0x00007f2b7c610f40
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['CALL', '0x7f2b7c0cdd10'] at offset 0x00007f2b7c0cdd10
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f2b7c0cdd10 = __nanosleep@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['JMP', 'QWORD', '[RIP+0x202732]'] at offset 0
    x0000000000400940
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f2b7c0ff900 = usleep@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x0000000000400950 = main@sl-optimized
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['JMP', 'QWORD', '[RIP+0x202772]'] at offset 0
    x00000000004008c0
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f2b7c02edb0 = __libc_start_main@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Found 10 frames!
INFO    : volatility.plugins.linux.process_stack:
[
Frame 0
========
Stack frame at 0x00007fff292a9440
Saved registers:
    ebp at 0x00007fff292a9430: 0x00007f2b7c5fca90
    eip at 0x00007fff292a9438: 0x00007f2b7c0872f3 (Return Address)
Frame function address: 00007f2b7c0f8b40
Frame function symbol: __write
,
Frame 1
========
Stack frame at 0x00007fff292a9470
Saved registers:
    ebp at 0x00007fff292a9460: 0x000000000000001d
    eip at 0x00007fff292a9468: 0x00007f2b7c0871d2 (Return Address)
,
Frame 2
========
Stack frame at 0x00007fff292a9490
Saved registers:
    ebp at 0x00007fff292a9480: 0x000000000000001e
    eip at 0x00007fff292a9488: 0x00007f2b7c088905 (Return Address)
Frame function address: 00007f2b7c087190
,
Frame 3
========
Stack frame at 0x00007fff292a94a0
Saved registers:
    ebp at 0x00007fff292a9490: 0x00007f2b7c3cf280
    eip at 0x00007fff292a9498: 0x00007f2b7c087890 (Return Address)
Frame function address: 00007f2b7c0888f0
Frame function symbol: _IO_do_write
,
Frame 4
========
Stack frame at 0x00007fff292a94b0
Saved registers:
    ebp at 0x00007fff292a94a0: 0x00007f2b7c3cf280
    eip at 0x00007fff292a94a8: 0x00007f2b7c610f74 (Return Address)
Frame function address: 00007f2b7c0442c0
Frame function symbol: __sigaction
,
Frame 5
========
Stack frame at 0x00007fff292a94c0
Saved registers:
    ebp at 0x00007fff292a94b0: 0x00007f2b7c5fca90
    eip at 0x00007fff292a94b8: 0x00007f2b7c616654 (Return Address)
Frame function address: 00007f2b7c610f40
,
Frame 6
========
Stack frame at 0x00007fff292a9510
Saved registers:
    ebp at 0x00007fff292a9500: 0x0000000000000000
    eip at 0x00007fff292a9508: 0x00007f2b7c0ff934 (Return Address)
Frame function address: 00007f2b7c0cdd10
Frame function symbol: __nanosleep
,
Frame 7
========
Stack frame at 0x00007fff292a9530
Saved registers:
    ebp at 0x00007fff292a9520: 0x0000000000000000
    eip at 0x00007fff292a9528: 0x0000000000400a03 (Return Address)
Frame function address: 00007f2b7c0ff900
Frame function symbol: usleep
,
Frame 8
========
Stack frame at 0x00007fff292a9550
```

```
Saved registers:
    ebp at 0x00007fff292a9540: 0x0000000000000000
    eip at 0x00007fff292a9548: 0x00007f2b7c02eea5 (Return Address)
Frame function address: 0000000000400950
Frame function symbol: main
,
Frame 0
========
Stack frame at 0x00007fff292a9610
Saved registers:
    ebp at 0x00007fff292a9600: 0x0000000000000000
    eip at 0x00007fff292a9608: 0x0000000000400a85 (Return Address)
Frame function address: 00007f2b7c02edb0
Frame function symbol: __libc_start_main
]
```

Listing B.4: Analysis of memory dump containing `sl`, optimized code

```
     ~  python2 volatility-svn/vol.py --profile LinuxUbuntu1304desktopx64 -f
    ~/evaluationdumps/ubuntu1304sl-unoptimized.dump linux_process_stack -p
    1428 -s ~/documents/ubuntusymbols -o evaluationdumps/ubuntu1304sl-
    unoptimized.stack.txt
Volatile Systems Volatility Framework 2.3_beta
*** Failed to import volatility.plugins.addrspaces.legacyintel (
    AttributeError: 'module' object has no attribute '
    AbstractWritablePagedMemory')
INFO    : volatility.plugins.linux.process_stack: Loading function symbols
    from directory: /home/dutchy/documents/ubuntusymbols
INFO    : volatility.plugins.linux.process_stack: Opened evaluationdumps/
    ubuntu1304sl-unoptimized.stack.txt for writing
INFO    : volatility.plugins.linux.process_stack: Starting analysis of task:
    pid 1428, thread name sl-unoptimized
INFO    : volatility.plugins.linux.process_stack:
    =================================================
INFO    : volatility.plugins.linux.process_stack: Found the stack at 0
    x00007fffd51b1000-0x00007fffd51d3000
INFO    : volatility.plugins.linux.process_stack: Found libc (/lib/x86_64-
    linux-gnu/libc-2.17.so) at range: 0x00007f424ffd4000-0x00007f4250192000
INFO    : volatility.plugins.linux.process_stack: Program code located at 0
    x0000000000400000-0x000000000040236c
INFO    : volatility.plugins.linux.process_stack: Executable entry point ('
    _start' function): 0x0000000000400950
INFO    : volatility.plugins.linux.process_stack: Scanning for return address
     of __libc_start_main function, starting at program arguments (0
    x00007fffd51d2890) downwards
INFO    : volatility.plugins.linux.process_stack: Scanned 157 stack addresses
     before finding the __libc_start_main return address
INFO    : volatility.plugins.linux.process_stack: Found the __libc_start_main
     return address (0x0000000000400979) at address 0x00007fffd51d23a8
INFO    : volatility.plugins.linux.process_stack: Scanning for return address
     of main function, starting at __libc_start_main return address (0
    x00007fffd51d23a8) downwards
```

```
INFO    : volatility.plugins.linux.process_stack: Scanned 1 libc addresses on
     the stack before finding the main return address
INFO    : volatility.plugins.linux.process_stack: Found main stackframe at 0
    x00007fffd51d22e8
INFO    : volatility.plugins.linux.process_stack: The address of the main
    function is 0x0000000000400b47
INFO    : volatility.plugins.linux.process_stack: Register %rbp was not 0,
    trying old-school stack frames
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['JMP', 'QWORD', '[RIP+0x202732]'] at offset 0
    x0000000000400940
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f42500c6900 = usleep@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x0000000000400b47 = main@sl-unoptimized
INFO    : volatility.plugins.linux.process_stack: Found function address from
     instruction ['JMP', 'QWORD', '[RIP+0x202772]'] at offset 0
    x00000000004008c0
INFO    : volatility.plugins.linux.process_stack: Instruction was a call to 0
    x00007f424fff5db0 = __libc_start_main@libc-2.17.so
INFO    : volatility.plugins.linux.process_stack: Found 3 frames!
INFO    : volatility.plugins.linux.process_stack:
[
Frame 0
========
Stack frame at 0x00007fffd51d22c0
Saved registers:
    ebp at 0x00007fffd51d22b0: 0x00007fffd51d22e0
    eip at 0x00007fffd51d22b8: 0x0000000000400c68 (Return Address)
Frame function address: 00007f42500c6900
Frame function symbol: usleep
,
Frame 1
========
Stack frame at 0x00007fffd51d22f0
Saved registers:
    ebp at 0x00007fffd51d22e0: 0x0000000000000000
    eip at 0x00007fffd51d22e8: 0x00007f424fff5ea5 (Return Address)
Frame function address: 0000000000400b47
Frame function symbol: main
,
Frame 0
========
Stack frame at 0x00007fffd51d23b0
Saved registers:
    ebp at 0x00007fffd51d23a0: 0x0000000000000000
    eip at 0x00007fffd51d23a8: 0x0000000000400979 (Return Address)
Frame function address: 00007f424fff5db0
Frame function symbol: __libc_start_main
]
```

Appendix C

Frames statistics

Table C.2: Register calls and normal calls, per process.

PID	Register calls	Normal calls	Total frames	Percentage
1	3	33	36	8.33%
240	1	5	6	16.67%
306	1	13	14	7.14%
313	1	38	39	2.56%
376	3	8	11	27.27%
378	0	13	13	0.00%
383	3	21	24	12.50%
384	2	4	6	33.33%
385	2	5	7	28.57%
464	3	59	62	4.84%
491	1	4	5	20.00%
511	2	7	9	22.22%
613	1	5	6	16.67%
808	0	33	33	0.00%
824	0	33	33	0.00%
836	0	4	4	0.00%
837	0	33	33	0.00%
842	0	33	33	0.00%
846	0	5	5	0.00%
848	0	4	4	0.00%
852	0	8	8	0.00%
871	1	13	14	7.14%
873	3	17	20	15.00%
909	1	6	7	14.29%
910	3	11	14	21.43%

Continued on next page

Table C.2 – continued from previous page

PID	Register calls	Normal calls	Total frames	Percentage
911	3	2	5	60.00%
912	3	2	5	60.00%
913	2	1	3	66.67%
914	2	1	3	66.67%
915	3	6	9	33.33%
916	3	6	9	33.33%
918	3	41	44	6.82%
927	4	14	18	22.22%
930	2	12	14	14.29%
931	2	12	14	14.29%
932	2	12	14	14.29%
933	2	12	14	14.29%
934	2	12	14	14.29%
935	2	12	14	14.29%
936	2	12	14	14.29%
937	2	12	14	14.29%
938	2	12	14	14.29%
939	2	12	14	14.29%
940	2	12	14	14.29%
941	2	12	14	14.29%
942	2	12	14	14.29%
943	2	12	14	14.29%
944	2	12	14	14.29%
945	2	12	14	14.29%
946	2	12	14	14.29%
947	2	12	14	14.29%
948	2	12	14	14.29%
949	2	12	14	14.29%
950	2	12	14	14.29%
951	2	12	14	14.29%
952	2	12	14	14.29%
953	2	12	14	14.29%
954	2	12	14	14.29%
955	2	12	14	14.29%
956	2	12	14	14.29%
957	2	12	14	14.29%
958	2	12	14	14.29%
959	2	12	14	14.29%
960	2	12	14	14.29%

Continued on next page

Table C.2 – continued from previous page

PID	Register calls	Normal calls	Total frames	Percentage
962	2	6	8	25.00%
963	2	12	14	14.29%
964	2	12	14	14.29%
965	2	12	14	14.29%
966	2	12	14	14.29%
967	2	12	14	14.29%
968	2	12	14	14.29%
969	2	12	14	14.29%
970	2	12	14	14.29%
971	2	12	14	14.29%
972	2	12	14	14.29%
973	2	12	14	14.29%
974	2	12	14	14.29%
975	2	12	14	14.29%
976	2	12	14	14.29%
977	2	12	14	14.29%
978	2	12	14	14.29%
979	2	12	14	14.29%
980	2	12	14	14.29%
981	2	7	9	22.22%
982	2	6	8	25.00%
983	2	6	8	25.00%
984	2	6	8	25.00%
985	2	6	8	25.00%
986	2	6	8	25.00%
987	2	6	8	25.00%
988	2	6	8	25.00%
989	2	7	9	22.22%
994	3	3	6	50.00%
995	3	9	12	25.00%
998	1	14	15	6.67%
1006	3	22	25	12.00%
1012	5	17	22	22.73%
1014	1	5	6	16.67%
1018	0	4	4	0.00%
1020	2	6	8	25.00%
1143	0	13	13	0.00%
1229	3	25	28	10.71%
1230	2	8	10	20.00%

Continued on next page

Table C.2 – continued from previous page

PID	Register calls	Normal calls	Total frames	Percentage
1349	0	13	13	0.00%
1435	1	12	13	7.69%

Table C.1: Number of registry calls per executable file or library.

File	**Offset (bytes)**	**Count**
/lib/x86_64-linux-gnu/libc-2.17.so	1023515	76
/lib/x86_64-linux-gnu/libglib-2.0.so.0.3600.0	446131	66
/lib/x86_64-linux-gnu/libglib-2.0.so.0.3600.0	299482	7
/opt/VBoxGuestAdditions-4.2.8/sbin/VBoxService	174153	7
/lib/x86_64-linux-gnu/libc-2.17.so	1109675	5
/opt/VBoxGuestAdditions-4.2.8/sbin/VBoxService	18988	5
/lib/x86_64-linux-gnu/libglib-2.0.so.0.3600.0	230980	3
/lib/x86_64-linux-gnu/libc-2.17.so	454472	3
/lib/x86_64-linux-gnu/libc-2.17.so	500175	3
/usr/sbin/rsyslogd	236627	2
/lib/x86_64-linux-gnu/libc-2.17.so	506437	2
/bin/dash	15245	2
/lib/x86_64-linux-gnu/ld-2.17.so	63236	2
/usr/sbin/rsyslogd	210528	1
/usr/sbin/rsyslogd	197994	1
/sbin/init	182266	1
/bin/dbus-daemon	284090	1
/bin/dbus-daemon	234171	1
/usr/sbin/console-kit-daemon	95498	1
/lib/x86_64-linux-gnu/libc-2.17.so	789952	1
/lib/x86_64-linux-gnu/libc-2.17.so	138915	1
/lib/x86_64-linux-gnu/libc-2.17.so	502638	1
/lib/x86_64-linux-gnu/libc-2.17.so	239862	1
/lib/x86_64-linux-gnu/libc-2.17.so	301711	1
/lib/x86_64-linux-gnu/libc-2.17.so	790461	1
/opt/VBoxGuestAdditions-4.2.8/sbin/VBoxService	299364	1
/bin/sleep	17850	1
/bin/dash	73962	1
/sbin/dhclient	1284682	1