Spring 2012

Forensic analysis of linux physical memory: Extraction and resumption of running processes.

Ernest D. Mougoue
James Madison University

Forensic Analysis of Linux Physical Memory:

Extraction and Resumption of Running Processes

Ernest Djomani Mougoue

A thesis submitted to the Graduate Faculty of

JAMES MADISION UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2012

DEDICATION

To Thee for every drop—

The bitter and the sweet.

To Thee for the desert road,

And for the riverside;

For all Thy goodness hath bestowed,

And all Thy grace denied.

To Thee for both smile and frown,

And for the gain and loss;

To Thee for the future crown

And for the present cross.

To Thee for both wings of love

Which stirred my worldly nest;

And for the stormy clouds which drove

Me, trembling, to Thy breast.

Adapted from "*I Thank Thee*" – Jane Crewdson (1850)

ACKNOWLEDGEMENTS

A number of people have contributed to the completion of this project and my journey at James Madison and to them I am very grateful. My first thanks go to my family and friends all over the world: We may not see each other often, but your continuous support and encouragements are worth more than you can imagine to me. Special thanks to my brother Ghislain (Gman) who has been and continues to be a center piece to my recent achievements.

I would like to thank my committee chair and advisor Dr. Florian Buchholz for his guidance and patience through the thesis process which was completely new to me. My gratitude also goes to the other members of my committee Dr. Bernstein and Dr. Wang for their time spent reviewing my work.

My time at JMU was particularly impacted by many individuals: the faculty of the department of Computer Science (especially Dr. Mata-Toledo, Dr. Fox, Dr. Elvis Tjaden and Taz Daughtrey), the faculty and staff of the College of Business (I have to mention by name my boss for two years Dave Jones, thanks for giving me a chance), the members of the African Student Organization, the staff of the International Student and Scholar Services, the members of the International Students Association, the members of the Student Technology Advisory Committee (STAC), the Information Technology staff at JMU and the Zeta Chapter of Virginia of the Upsilon Pi Epsilon honor society.

Finally, I would like to express my appreciation to all the three generations of graduate students with whom I shared the “CS Grad Lounge” and countless memories, especially Ally, Joel, Will, Sufi, Justin, Tyler, Deborah, Ben, Stephen (GLG), Jasen, Brian, Jona, Newton, Jake, Xuewen, Tom, Fred and Brandon.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Traditional digital forensics' procedures to recover and analyze digital data were focused on media-type storage devices like hard drives, hoping to acquire evidence or traces of malicious behavior in stored files. Usually, investigators would image the data and explore it in a somewhat "safe" environment; this is meant to reduce as much as possible the amount of loss and corruption that might occur when analysis tools are used. Unfortunately, techniques developed by intruders to attack machines without leaving files on the disks and the ever dramatically increasing size of hard drives make the discovery of evidence difficult. These increased interest in research on live forensics (attempting to obtain evidence while the system is running) and on volatile memory forensic analysis. Because of the important role they play in computing systems, volatile memory is a source of information about running processes, network connections, opened files and/or loaded kernel modules that might be valuable to forensic investigations.

In this thesis we show that when provided with an image of the physical memory of a Linux system, it is possible to extract data about a specific running process, enough to be able to resume its execution on a prepared environment. We also describe two proof-of-concept tools `gettsk` and `memexec` developed for this purpose. This would allow investigators to not only obtain information about a suspicious running task from a RAM dump, but also to perform further inquiry through techniques such as malware analysis.

1. INTRODUCTION

Digital forensic investigations aim at establishing the patterns, causes and consequences of attacks on a computer system. Investigators attempt to determine the nature and chronology of events that might have caused system failure or loss of valuable assets. They seek evidence of changes in configurations and any artifact that could alter the behavior of the system or put it in an undesirable state. This is done mainly by capturing and analyzing storage devices for threats and exploited vulnerabilities. In some cases, malicious software (or malware) are responsible for the misbehavior, in which situation suspected software programs are identified and studied through a set of activities called malware analysis. These techniques help determine what damages the malware can expose the system to, thus allowing investigators to take appropriate decisions and measures to mitigate and ultimately annihilate its effects. Because of the danger they represent, it is commonly recommended to perform forensic analyses of malicious programs in a non-production and heavily controlled environment.

Historically, only permanent storage media such as hard disk drives were examined by investigators who hoped to find evidence in files on disk. Since they do not lose their contents when the machine is powered down, researchers have been able to develop methods to analyze files and their associated metadata for knowledge valuable to forensic investigations. However, they do not constitute the only sources of information; a considerable amount of data about computer systems can be obtained by exploring the contents of volatile media amid the primordial role they play in performing operating systems tasks. It is possible, for example, to obtain information about the processes

running on a host at a given time, as well as associated objects such as opened/mapped files and network connections, by examining the physical memory. However, the volatile nature of the memory and the fact that their use is highly operating system-dependent make the acquisition and analysis of its contents challenging. Even when the RAM is successfully acquired and enough process-specific data retrieved, it is still difficult to perform an effective malware analysis.

In this document, we describe ways to collect information about a running process, when provided with an image of a Linux 2.6 physical memory, and resume its execution on another host. This will permit investigators not only to examine the data for evidence, but also to perform further malware analysis in a controlled environment.

1.1. Background, Overview and Problem Statement

Computer-related attacks against individuals and companies referred to as *cybercrime*, have been on the rise for the past decades. Various types of cybercrime exist depending on parameters such as the target and the techniques used to execute the attack; they are often classified as follows with respect to the crime perpetrated [16]:

- *Identity theft*: Access to and misuse of personal information for fun or profit.
- *Cyber harassment*: Use of computer systems to threaten or stalk another person.
- *Unauthorized access to computer systems or data*: Also known as computer crime.
- *Fraud*: Bank and other financial fraud, data piracy.

- *Non-access computer crimes*: causing damage without gaining control of the computer system. This includes for example denial-of-service and virus attacks.

The consequences of these attacks on personal and group welfare are considerable as shown by an Identity Theft Resource Centre (ITRC) report, which states that 662 major data breaches were identified in 2010, exposing more than 16 million personal records [29]. The U.S. department of Justice also reported from a survey that 67% of participating businesses have been victims of at least one cybercrime in 2005, with a total in losses estimated at $867 million [38]. In a similar survey, the Computer Security Institute (CSI) found out that the average loss per business in 2007 was around $350,000, nearly the double of the previous year results [39]. Cybercrime has escalated to introduce the concept of *cyber warfare*, which refers to actions that a country/nation or governmental organization takes to compromise the computing systems of other countries'/nations' in order to disrupt, create damage or obtain secret information. A recent example of such actions is the Stuxnet worm that spread around the world and targeted specific industrial control systems [45]. Therefore, the need for digital forensic investigations and malware analysis techniques, to better understand these attacks and reduce the effects of cybercrime, is growing very fast and so is research in these areas.

Early digital forensics efforts were focused on permanent media storage, since the filesystem can be preserved and studied to detect anomalies [8]. Their contents are imaged or copied into other devices and generally attached to different machines for further investigation. Unfortunately, intruders have developed techniques to hide tracks of their mischief such as deleting compromising evidence, and performing their attacks without leaving files on the disks. Also, it is difficult to obtain information about the

programs running on the system, the network connections opened, the loaded kernel modules or other process and operating system-related artifacts at the time of acquisition, as they are generally stored on main memory. This increased the interest in *live forensics* (attempt to obtain evidence while the system is running) and in the analysis of volatile media such as the Random Access Memory or RAM. The storage acquisition and analysis methodologies as well as their limitations are developed in Section 2.1.2.

The main memory plays an important role to the effective functioning of a computing system, as it is used to keep track of the interactions between the OS, the applications running on the machine, the filesystem and the hardware (CPU). Therefore, harvesting the contents of the RAM is of undeniable value to a forensic investigation, especially if the acquisition is done correctly, at the right time and can be parsed for evidence. For instance, some passwords and encryption keys can be found in memory [25, 30], as well as structures holding important information about running processes [43]. Fortunately, in the past decade, major progress in documenting what kind of data can be extracted from an image of RAM and how to obtain it have been made (discussed in Section 2.3). However the resulting tools and techniques are mostly operating system-dependent, because of the differences in the way each uses the memory. Furthermore, investigators do not have a complete understanding of what incidents might have happened on the system just from a memory analysis, since the results are mainly presented in the form of lists (such as list of running processes, open files and network connections) and represent the state of the machine at a specific moment (they do not include an evolution of the situation over time).

In light of all the above, two questions concerning the forensic analysis of physical memory come to mind. First, when provided with the RAM dump of a machine, what kind of data can be extracted? Second, how can we make the obtained information more useful to investigations?

Taking into consideration the fact that most of the incidents happening on computers are linked to applications, our approach is concentrated on acquiring knowledge about specific processes being run on the machine when the RAM was captured and restoring their execution state. Is all the data necessary to bring a process back to life available from the memory image? Assuming the answer is affirmative, how hard is it to resume a process? What kind of environment is required? These are some of the questions that will be considered in this thesis.

1.2. Purpose and Motivation

The purpose of our work is to determine to what extent it is possible to collect data about a specific process running on the machine and resume its execution on another "safe" environment, from an image of the memory of a Linux 2.6 kernel.

The immediate benefit that such a capability offers is bringing together the advantages of both "dead" and "live" forensic analysis techniques (described in Section 2.1.1). Indeed, obtaining an image of the RAM allows the investigation to be performed with minimum corruption of the state of the original machine; Meanwhile gathering the context and environment of a process and resuming its execution on a prepared host gives more confidence in the integrity of the live tools used [35]. It would also permit

investigators to have a repeatable analysis process (which is of great value when evidence is presented in a court of law) and give them the ability to perform additional inquiries on extracted artifacts without risk to production systems.

An important aspect of digital forensics and incident response is the study of malicious software or malware analysis in order to determine their purpose, stop their progression and/or protect against their propagation (See Section 2.2). One of the techniques used here is *dynamic malware analysis*, which involves running the malware to observe and monitor its interactions with the system rather than attempting to determine the malware execution path without actually running it. Our work could allow investigators to perform dynamic malware analysis on a suspicious program rebuilt from an image of the RAM.

A more detailed description of the potential benefits of our work can be found in Section 2.5.

1.3. Document Structure

This document is divided in six parts. Following this introduction, Chapter 2 gives a general overview of the field of digital forensics and presents prior work related to the analysis of physical memory; the contributions that our study bring are also discussed. Chapter 3 documents what relevant information about a running process is available in a Linux RAM and where to locate it in the 2.6 kernel. Chapter 4 and Chapter 5 describe in detail, the activities to perform in order to acquire and store process specific data, as well as ways to transport the execution of the process onto another host. They both include a proof-of-concept experiment showing results produced by applying the

described methods. We conclude with a summary of the thesis, a statement of the major problems encountered during research as well as future work.

2. DIGITAL FORENSICS BACKGROUND, RELATED WORK AND CONTRIBUTION

This chapter presents a general view of what digital forensics and malware analysis are about, followed by a review of major developments in the forensic analysis of physical memory. We close the chapter with an enumeration of the potential benefits that our research can bring to the forensics community.

2.1. Overview of Digital Forensics Process and Practices

According to Wiles [51], digital forensics can be defined as "*the preservation, identification, extraction, interpretation and documentation of computer evidence*". This definition suggests that to get the best results acceptable in a legal system, investigations require a variety of activities (preservation, extraction, interpretation and documentation). Over the years, researchers have proposed different processes and codes of conduct to perform these various tasks. The following sections describe general digital forensics investigation process models as well as the major activities they comprise.

2.1.1. Digital Forensics Methodologies

A main concept in defining digital forensics models, consists in following existing procedures used for physical crime scenes, and includes technology-specific guidelines. Carrier and Spafford proposed a model of digital investigation, admissible in a court of law, which easily integrates with refined law enforcement methodologies for a mutual

benefit [11]. The result of such collaboration is a link between human suspects and digital evidence. Carrier summarizes the process into three general stages [7]:

- *Preservation*: Collect and copy the digital data in its entirety and integrity.
- *Search and Analysis*: includes examination, interpretation and recovery of the acquired data.
- *Reconstruction*: report and documentation of events leading to the crime.

In practice however, applying these recommendations is not a simple issue; factors surrounding the investigation such as the state of the machine at different stages of the process have to be taken into account.

In the early years of digital forensics, *"dead"* acquisition was used. This method requires investigators to power down the machine(s) to be examined to avoid potential loss or modification of data on the system. Afterwards, media such as hard drives and other data-persistent devices are harvested for analysis (see Fig. 1).

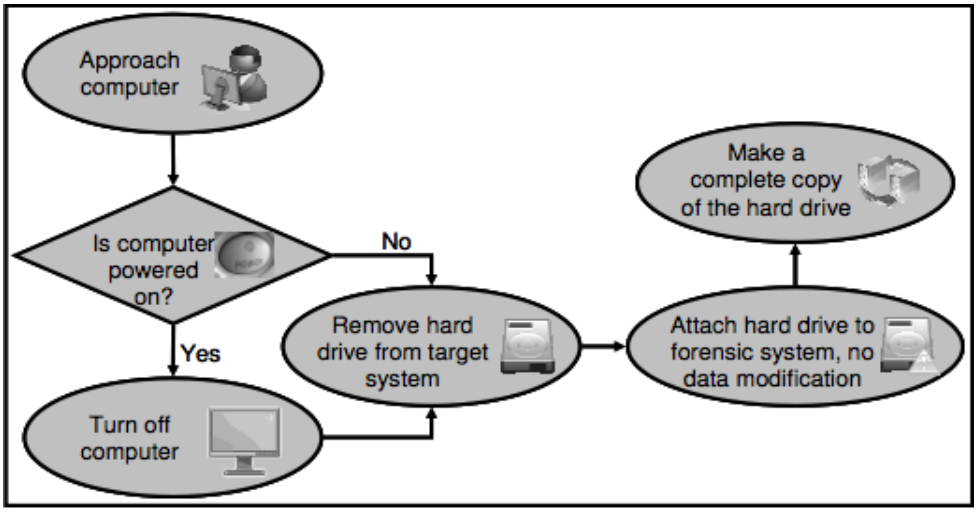

Figure 1. "Dead" Forensic Acquisition [31]

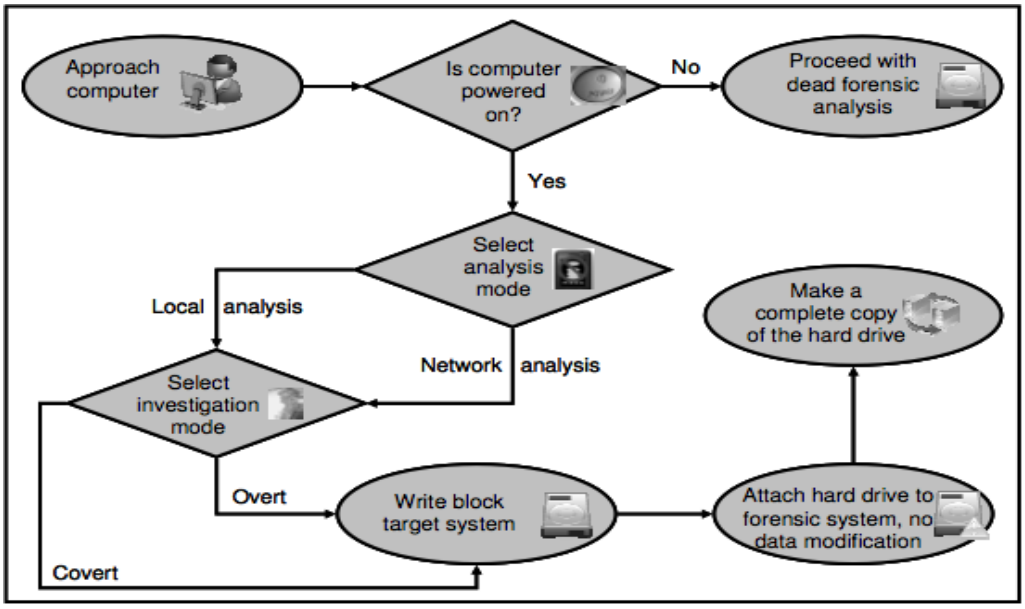

Figure 2. "Live" Forensic Acquisition [31]

More recently, new techniques emerged to acquire and analyze data while the system is still running. They form what is called "*live*" forensic analysis (see Fig. 2). It is worth noting that both Figure 1 and Figure 2 do not address acquiring volatile data, illustrating the fact that it was not appropriately considered in early forensic investigations.

Lessing and Von Solms published a comparative study of both frameworks ("dead" and "live" forensic analysis) [31]. According to their observations, "dead" analysis procedures are straightforward and more or less platform independent. Also, chances that evidence are corrupted are very small since the computer is not running. However, dead forensic analysis presents some limitations, which include:

- Volatile data (data that exist when the machine is running and is destroyed after a shutdown) such as network information and physical memory are difficult to acquire, even impossible when the system is already shut down.

- With the increasing size of hard drives, investigations become time-consuming and valuable evidence could be missed as a consequence of the large amount of information.
- It is now common practice to use encryption on disks for protection against attackers who have physical access to the system. When the machine is powered off, the encryption keys (usually residing in memory) necessary to access the contents of the storage device are not available [25, 26]. In such situations, the investigator may not be able to analyze the collected data.

Live forensic analysis on the other hand allows analysis of volatile data through techniques such as "trusted" command shells (local analysis) or remote connections (network analysis). The investigators interact directly with the underlying operating system and therefore have access to the memory; with the appropriate credentials, disk encryption is not an issue anymore [26]. Live analysis however can be associated with shortcomings that can diminish the value of forensic evidence in court [31, 9, 26]:

- Evidence can be altered. Because of the interaction with the operating system, tools or mistakes from the investigator might modify files and/or memory locations on the system and potentially affect evidence.
- The tests performed during live forensics are generally not repeatable, since the state of the system changes.
- The tools are platform dependent. Every system is different, so the tools must be tailored to fit specific platforms: what works for one might not be usable in the other.

- The presence of a kernel rootkit (malicious software running on a system, at the kernel level and with privileged permissions while at the same time hiding its existence to users by changing the normal behavior of the operating system) could provide investigators with erroneous, incomplete or no information at all.

An alternative to both "dead" and live analysis would be to complementarily combine them during a unique investigation process. Mrdovic et al. [35] claim that such a combination is feasible with the use of memory imaging/dumping and virtualization. Indeed, both persistent and volatile data could be imaged/copied and then the studied machine powered down. Finally, a virtual environment could be created to mimic the behavior of the investigated computer. At this point, some "live" interaction could happen without significant impact on the integrity of the collected evidence. Although most of the shortcomings of dead analysis do not apply when this technique is used and the process is repeatable, the live interaction in the virtual machine is still operating-system based and thus subject to evidence manipulation.

2.1.2. Digital Forensics Activities

2.1.2.1. Acquisition and Preservation

It is crucial for investigators to correctly collect and preserve data from machines; any mistake could lead to loss of evidence or rejection from court. Carrier indicates that a great deal of information pertaining to a digital forensic investigation can be obtained from files in external storage media like hard drives [8]. He also describes general techniques used to acquire data from these devices, such as the *dd* command in UNIX

systems, which performs a byte-to-byte copy from one medium to the other, and the concept of hashing to verify the integrity of copied data.

Unfortunately, in addition to the disk encryption discussed earlier, hard drives wear out over time or can be damaged; also, attackers have developed ways to hide their tracks through *data destruction* (deleting files potentially containing evidence of intrusion) [23] and *data contraception* (performing attacks without writing to any file on disk, also referred to as *memory resident malware*) [24]. Therefore, the interest in obtaining other sources of data such as volatile storage media grew significantly.

Because of the transient nature of volatile data, its acquisition is challenging and time sensitive. Methods and tools exist to efficiently dump the RAM, classified by Burdach in two major groups [6]: Software and hardware-based methods.

- *Software-based memory acquisition* consists of utilities loaded on the live system to dump the RAM on an external device. Generally, the effects of running the toolkit on the system are known to the investigator and are considered during analysis. However, if the machine is infected, there is a chance that the tool used to image the memory gets compromised by the malicious program and that misleading evidence is produced. Moreover, it is difficult to verify that the imaging operation was performed correctly, since the memory is constantly changing. These integrity and verifiability issues are not a concern in a virtual environment as most of the virtualization toolkits (such as VMware and Xen) provide mechanisms to acquire a snapshot of the RAM (pause the execution of the host and collect the memory) [47]. The

contents of memory are constantly changing and thus data collected by the software tool might be inconsistent.

- *Hardware-based methods* implant devices on the computer to bypass the Operating System. They use DMA (Direct Memory Access) to access the RAM and copy its contents to external storage media. Unfortunately, attackers might be able to detect the devices and as a result, provide erroneous information. An example of such hardware components includes a hidden PCI card that would be visible to the machine only when enabled by an investigator; however, this method is impractical as it requires the PCI card to be installed and configured on the machine before the incident occurs [10]. Also, a firewire device (IEEE 1394) that accesses the physical memory independently of the CPU can be used [34]. This firewire acquisition technique can slightly modify the state of the machine (in case the operating system has to activate the port) and, if not properly performed, crash the system.

Contrary to popular belief, some of the RAM hardware on the market does not immediately lose their contents when the system is powered down; there is a small time window of data persistence (several seconds), which could be significantly extended by freezing the RAM chips [25]. This retention capability could be exploited by investigators to copy the contents of the memory by rapidly restarting the machine after a shutdown; the full operating system is not loaded, rather, a specially crafted program installed on a USB drive or attached to a network/firmware boot is used to start the

machine and dump the contents of the RAM [25]. This method however presents a timing challenge, and can be tried only once: in case of failure, everything will be lost.

2.1.2.2. Data Analysis

Much information can be obtained by examining the data acquired from a hard drive for a forensic investigation: deleted files can be restored; log files, browsing history and more can help support or refute theories about the incidents [8]. The investigators typically have to rebuild the volumes and partitions from the image, based on the specific type of file systems in use, and then search through the files for tangible evidence. The rapid increase in hard-drive size, the proliferation of file types/formats and the rise of storage management technologies such as RAID and LVM have complicated the investigators' ability to discover evidence. In such situations, information gathered by analyzing the physical memory can be particularly useful since they might provide preliminary answers and help the investigator prioritize on the type of information to look for in the filesystem. Also, as researchers are developing more ways to extract, parse and study the contents of memory, some investigators have started to give a lot more value to the data from the RAM.

Despite the importance of the information that could be found in the RAM, research in forensic analysis of memory is relatively young (no more than a decade old). Initially, investigators derived clues by searching for strings on the binary dump of the memory, hoping to uncover sequences of characters that might have a meaning such as the name of a process or a loaded kernel module. Unfortunately, this method is not efficient because of the important number of strings that can be found in a memory dump. It was not until the release of the 2005 *Memory Analysis Challenge* for the 5th edition of

the *Digital Forensics Research Workshop* (DFRWS) [2, 17], that an active community began making progress on answering the question of what can be discovered given a snapshot of memory (more details in Section 2.3).

2.2. Overview of Malware Analysis

In the aftermath of a computer attack, or after a security incident has been detected, a set of activities, referred to as *incident response*, is generally performed to identify the threats on the system, reduce potential damages and ultimately prevent the attack to happen again in the future. The SANS (SysAdmin, Audit, Networking and Security) Institute proposed a standard approach to incident response composed of six consecutive steps [15]: Preparation, Identification, Containment, Eradication, Recovery and Lessons Learned.

Malware Analysis, the study of an unwanted and potentially malicious piece of software (such as virus, Trojan, spyware, etc…), is an inherent part of the incident response process and is usually associated with the identification phase. Indeed, malware analysis activities attempt to determine exactly what actions the malware performs and to some extent, how it was introduced into the system. This permits the responders to efficiently recover from incidents and strengthen their system against similar infections.

The traditional methodology in performing malware analysis involves collecting all the available artifacts (such as code and/or executable files) of the malicious software and transporting them into a contained (i.e. cannot infect production machines) host specifically prepared for observation and analysis. Depending on whether the malware is

actually run or not, there exist two different types of malware analysis techniques: *static* and *dynamic* analysis [15, 1].

2.2.1. Static Malware Analysis

Static analysis refers to the set of techniques used to determine how a piece of malware works without executing it on a machine. This usually involves reverse engineering the software and/or examining its code/binaries to understand what it is supposed to do. The most common tools range from the advanced decompilers and disassemblers to the simple source code viewers and string matching utilities [40]. Static analysis permits investigators to walk through the malicious code and immediately identify the weaknesses exploited by the malware and therefore plan for effective defense mechanisms to be implemented on the system. Also, most or all of the execution paths of the malicious software can be covered through this analysis method and it is usually faster than dynamic analysis [15, 1, 17]. However, static analysis has a few limitations:

- For large and complex code, it may be difficult to completely predict what the malware does; Investigators sometimes have to approximate the overall behavior.
- *Obfuscation* techniques have been developed to make the disassembly and reverse engineering process more difficult. Obfuscation is concerned with modifying the structure and syntax of a program so that it would be hard to dissect through code analysis, while at the same time conserving its

functionality and efficiency. These techniques make static analysis harder, not impossible.

- Sometimes the code examined during static analysis is not necessarily the code executed on the machine; *metamorphic code* (code that reprogram itself) and *polymorphic code* (different code at each execution, though having the same functionality) are examples of such programs and thus static analysis may not be accurate in these cases.
- It is also possible that the malware will respond differently depending on an input provided. In some cases for example, that input represent additional code that the attacker sends over the network and thus is not necessarily available to the investigator.

2.2.2. Dynamic Malware Analysis

Dynamic malware analysis techniques, contrary to static analysis, attempt to determine the interactions between the malware and the system, by actually running the program and analyzing its behavior. Safety restrictions are more important in these situations, since running the unknown piece of software could be disastrous for the host: The malware could destroy key functionalities, break loose and potentially harm other machines and networks. It is therefore imperative for the investigator to prepare a contained and asset-free environment in which to launch the execution of the malware for analysis. The common activities in dynamic analysis include monitoring the system and

library calls made by the program or examining each machine instruction with the help of debuggers or machine emulators [1, 17].

The major benefit of dynamic analysis over static analysis is the assurance that the machine instructions being analyzed are the ones that the program actually executes; i.e. it is not affected by techniques like obfuscation and polymorphic code. On the other hand, dynamic analysis has some drawbacks:

- For complex and large programs, it is difficult to run through all their execution paths and thus predictions on their behavior in a random setup are difficult to make.
- In dynamic analysis, there is a limited view of the internals of the program. Unless using a virtual machine (VM) coupled with introspection techniques (monitoring and inspecting guest operating systems in a virtual environment from the outside to observe behavior [36]), one can only observe input/output and library/system calls.
- The analysis environment is not invisible to the malware. As mentioned in the previous paragraph, the safety measures taken in building the analysis environment are of crucial importance. Malicious software can be tailored to behave differently depending on the underlying configurations of the system. For example, to ensure absolute containment, investigators can build a VM similar the one infected, without network connections; however, it is possible to determine that the program is running on a VM by checking the hardware configurations that virtualization software usually utilize, through techniques such as timing attacks on virtual and emulated environments. An attacker

could then program the software not to execute properly when such properties are detected.

2.3. Advances in Forensic Analysis of Physical Memory

Numerous utilities have been developed to help investigators perform memory analysis. Most of these tools are however platform dependent, due to the unique way each operating system (and even specific versions) handles the RAM. We now present some of these tools and related developments, organized with respect to the underlying family of operating systems.

2.3.1. RAM Analysis on Microsoft Windows

The DFRWS challenge, mentioned in Section 2.1.2.2 above, provided contestants with a memory dump from a Windows 2000 machine to be analyzed. From this emanated two research results:

First, Betz studied the way the Windows 2000 kernel uses the memory, and designed a command line tool called *Memparser* that produces a list of running processes on the system, even the ones that were hidden to the operating system [2].

Garner and Mora developed *KnTList* (part of the now commercial toolset *KnTTools*), which could list the running and hidden processes on the machine, as well as access some other system's information such as time parameters and ARP cache [19].

In 2005, Burdach published a guide to discovering Windows 2000 and Windows XP kernel data structures. He developed a follow-up tool *WMFT* (Windows Memory

Forensics Toolkit), which can enumerate active processes and loaded modules as well as other kernel structures such as driver objects when provided with their addresses in memory.

A year later, Schuster created the tool *PTfinder* which is capable of discovering, from an image of the RAM, all processes and threads running on Windows NT systems (including ones that were potentially deleted by kernel attacks or persisted through a reboot). This utility also provided users with a graphical and hierarchical view of the processes discovered, so investigators could easily determine parental relationships between the threads [42].

In 2007, Walters and Petroni introduced a set of tools to analyze memory dumps from Windows XP SP2 systems: *volatools*. The suite evolved to become *The Volatility Framework*, one of the most used open source toolkits for Windows memory forensic analysis. It provides important functionalities, such as [50]:

- Information on running processes and threads (list of open files, addressable memory, loaded DLLs)
- Information on network connections
- Support for different types of memory dumps (Windows crash dumps, hibernation)
- Information about the registry and loaded kernel modules

Moreover the framework can be extended through plug-ins and a beta version to support Linux operating systems is currently in development [14].

Another well-known tool is *Memoryze*, developed by the company MANDIANT in 2008, which has support for a variety of versions of the Microsoft Windows Operating System.

Memoryze can allow its users to acquire an image of the RAM as well as perform the analysis on both on live systems and RAM images. It provides the following, among other features [33]:

- List the virtual address space of a given process including:
- List all network sockets that the process has open, including any hidden by rootkits.
- Specify the functions imported and exported by the EXE and DLLs.
- Verify the digital signatures of the EXE and DLLs. (This is disk based.)
- Output all strings in memory on a per process basis.

2.3.2. RAM Analysis on OSX

Research on memory analysis on OSX systems is still rudimentary: we were not able to find any tool to perform analysis on OSX RAM. However, Suiche published a paper describing the kernel internals of this operating system [43]. He laid out details on memory management of processes, files and other kernel structures, necessary to retrieve valuable information from MAC OSX memory dumps. Because OSX is a Unix-like system and is largely based on BSD (Berkeley Software Distribution), it has similarities with Linux kernel structures, such as the existence of a doubly-linked list object to keep track of processes running on the machine (see Chapter 3).

2.3.3. RAM Analysis on Linux

Various utilities for the forensic analysis of Linux RAM have been developed. One of the earliest is Burdach's *iDetect*, a tool that can provide information about user-mode processes running on the system, as well as files mapped into memory. It was designed for Linux 2.4-x and uses the System.map file (symbol table used by the kernel) to obtain the address of important kernel structures.

Urrea developed a proof-of-concept for what he called "the basis of later research in RAM forensics" [46]. He focused his efforts on kernel 2.6 and described means to obtain information on running processes and recover files loaded in physical memory. He also discussed ways to explore potential swap space, to complete the memory analysis.

In 2008, Case et al. proposed a framework, called *FACE*, for digital forensic investigation that finds and correlates evidence from multiple sources (e.g. disk images, memory images and network capture) and multiple targets [12]. Their framework included *Ramparser,* a tool that analyzes Linux 2.6-x memory dumps and provides information on running processes, network connections and loaded kernel modules. In an attempt to reduce the platform dependency of Linux memory analysis tools, *Ramparser* was later improved to support a variety of kernel versions [21]. This was achieved by dynamically reverse engineering core kernel functions for each version, to obtain data structures' offsets necessary for a deep memory analysis and save them in an extensible database.

SecondLook, a commercial tool from *Pikewerks Corporation* was recently introduced [37]. It provides users with a GUI interface for ease of use as well as a command line interface for custom analysis. Running processes, network connections and loaded kernel modules are some of the information the tool provides to investigators.

Kollar developed a tool called *Foriana* that, when provided with a memory dump, attempts to determine the corresponding operating system by using pattern matching and performing a string search on the image (hoping to identify the name of a known standard process such as "init" for example for Linux systems) [30]. The tool uses heuristics to find kernel structures in memory, and then generates a list of loaded modules and running processes.

A more recent project started by Girault resulted in *volatilitux* [20]. *volatilitux* can extract memory-mapped files from the RAM, output the list of running processes as well as produce process-specific information such as the list of open files and memory mappings.

2.4. Work Related to Process Resumption

Resuming the execution of a process loaded in memory is closely related to the way it has been extracted. Most of the efforts realized in this prospect, were directed towards dumping the binaries of the process. The *procmemdump* and *procexedump* plug-ins of the Volatility tool are prime examples [50]: they permit investigators the obtain executables from the memory of Windows systems. We could not find any similar work for the analysis of an image of the Linux RAM. However, for live Linux systems, Ilo presented a proof-of-concept tool to extract the binaries of a running process [28]. Although his primary goal consisted in building a single binary file from the artifacts collected from the memory about a running process, he also proposed a sequence of steps that one could take in order to recover the state of the process on another host or at another time. His proposal included the following:

- Get the files used by the process, place them in the appropriate location and open them.
- Create a new process
- Copy process segments and registers in the appropriate locations
- Launch the execution of the process

An area of research related to our project is *process migration* or process *checkpoint/restart* techniques on Linux systems, which is concerned with saving the state of a running process from a live machine in order to resume its execution on another host or for rollback purposes. Linking checkpointing and digital forensics is not a completely new initiative, it was proposed by Foster and Wilson in their introduction to process forensics [18]. They argue that checkpointing can be a powerful tool in the arms of the digital forensic investigator or to the incidence response team, since crucial evidence can be obtained from processes running on a host.

One approach to checkpointing consists in loading a kernel module into the system and using it to save the state of a chosen process onto a file and restart its execution through the same module. CRAK is an example of a tool that uses the Linux Kernel Module (LKM) option to restart a previously saved Linux process or group of processes [52].

Currently, we did not find any work that extracts a running process from an image of memory and resume its execution.

2.5. Contributions of our Work

Many advantages can be associated with the recovery of the context of a running process from a memory image as well the resumption of its execution in another environment. We now present and discuss a non-exhaustive list of them in detail.

Benefits to General Forensic Investigations:

- We cannot stress enough the importance to a case, of the information that is available when analyzing non-volatile storage. It is possible, for example to determine what kind of machine it was in the first place (version of operating system and underlying hardware characteristics in case this was not known), what modules were loaded on the system and even find some password and encryption keys that could help break a case. With the list of running processes that our work provides, investigators might be able to identify unknown, unexpected and/or potentially dangerous programs that were on the machine at the time of acquisition of the RAM image. They could then take a closer look at the type of resources that the process was accessing such as the user running the process, the opened file descriptors (includes files, network connections and ports) and the memory mappings.
- The fact that we are provided with an image of the memory would normally imply the use of "dead" analysis techniques, meaning that there would be no live interaction and the observation of the evolution of the system over time is not an option. As indicated in Section 2.1.1, it is possible to combine both "live" and "dead" analysis on one case, with the use of memory dumping and virtualization. This is one of the main ideas behind our study, as the resumption component

gives “life” to the data obtained from the RAM and thus allows for “live” analysis to occur without significant changes to the integrity of the collected evidence. Moreover, the problems of live techniques are minimized as the process is repeatable and the impact of tools is limited.

Benefits to Malware Analysis:

After an investigator has detected a potentially harmful process running on a machine, the next step is usually to perform malware analysis.

- The CPU state (register set), at the moment of acquisition of the memory, of the process of interest is available and can be taken into account when reverse engineering is performed. Moreover, the code section of the memory mappings associated with the application can be dumped into binaries that can use for static malware analysis.
- When provided with an image of RAM, being able to resume the execution of a process so that dynamic analysis can be performed is already an important contribution. More so, if the resumption phase can be done in a safe environment built for the purpose of observation, as production systems will be compromised. Our approach suggests building a virtual machine, similar to the original system (which is actually a common malware analysis technique), in which the execution of the program will continue. Most virtualization software provide additional monitoring capabilities through VM introspection that could be useful to an investigator.

- A legitimate question to ask is why not dump the binaries of the process from the memory and launch its execution from the beginning? This method has been used before (see Section 2.4) and would certainly allow for malware analysis to take place. The problem with this approach is that most attackers are aware of the latest forensic techniques such as studying the binary in a virtual environment. To work around such techniques, they tend to perform VM detection and/or check for other configurations and change the behavior of the program accordingly. These checks generally occur when the program starts its execution. Thus, if the process is resumed from where it stopped when the memory image was collected, there is a decent chance that this detection phase is over.

3. The Linux Kernel and Process Information in Memory

Before we describe how to obtain relevant information about a running process from a RAM dump and resuming its execution, it is primordial to understand how the Linux kernel manages the memory, especially process-specific data. Indeed, to maintain control over the execution flow on the machine, the kernel must keep track of process activity by storing various attributes such as the process' state, children and CPU registers. A look into the C source code of the kernel reveals that this information is organized in memory through a number of data structures—There are many ways to access the source code of the Linux kernel such as looking at location `/usr/src/linux-headers-x/` ('x' represents the kernel version) on the local machine (if the source files are installed) or browsing online repositories [32]—The following subsections will describe some of these data structures and their relationships, for the kernel version 2.6.35 on x86 systems, as well as present the memory management mechanisms used in Linux.

3.1. Linux Memory Management

3.1.1. The Virtual Memory

To ensure that CPU multithreading and time-sharing (i.e. the possibility to have more than one process use the same system resources over a certain period of time) are efficient, it is crucial to share the physical memory between processes. Because of the constraints and differences that may exist between the actual size of the physical memory

and the amount of space needed by the all the programs running on the system, the Linux operating system implements *virtual memory*, which is a layer of abstraction of the RAM to processes and users.

On a 32-bit x86 system for example, the whole 2^{32} = 4 GB addressable space, ranging from `0x00000000` to `0xFFFFFFFF`, is considered available to each user process and each address is referred to as *linear address* or *virtual address*. The kernel, also uses a virtual address space but splits it into two parts at the `PAGE_OFFSET` (macro defined as `0xC0000000`) mark: The first part (of size 3GB) is mapped to the currently running process and changes at every context switch, while the second part is fixed and used for other kernel operations [22]. As a result of this split, every user process reserves 1GB in its linear address space mapped to the kernel's first virtual memory zone and used when the process in running in kernel mode (See Fig. 3). The rest of the process' virtual space is directly addressable and is divided into *pages* of size `PAGE_SIZE` (can be 4KB, 2MB or 4MB on x86. In practice, it is typically 4KB).

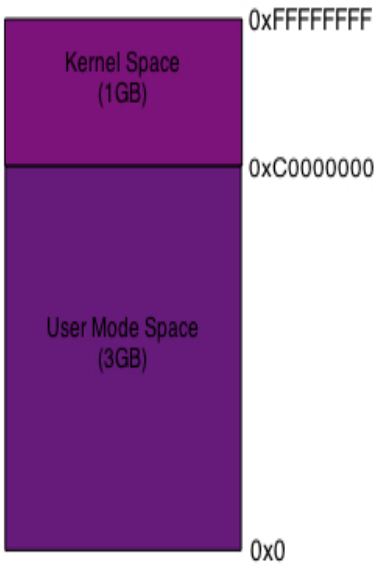

Figure 3. Virtual Memory Layout for User Mode Process

3.1.2. From Virtual to Physical Memory: Translation and Paging

The actual physical memory does not follow the same scheme as its virtual counterpart. Rather, *physical addresses* are used to map the memory cells in memory chips and operating systems use a number of mechanisms to translate between the physical and linear addresses.

Similar to the virtual address space of a user process, the RAM is divided into sets of contiguous addresses called *page frames*. These frames are grouped into zones with specific functions. In Linux for instance, there are three physical memory zones [22]:

- *ZONE_DMA*: From 0 to 16 MB, used for Direct Access Memory (DMA) page frames.
- *ZONE_NORMAL*: Ranges from 16MB to 896MB, contains non-DMA pages with virtual memory mapping.
- *ZONE_HIGHMEM* (if applicable): 896MB and onwards and is not directly mapped to the kernel.

At boot time, Linux directly maps physical address 0 with the kernel virtual address `PAGE_OFFSET` (`0xC0000000`). Therefore, when faced with a kernel virtual address, its physical equivalent is obtained simply by subtracting `PAGE_OFFSET`. For example, linear kernel address `0xc0342FF4` corresponds to actual physical memory location `0x00342FF4`. The conversion process is completely different for user mode virtual addresses. In this case, the Linux operating system uses *virtual memory paging* to map the virtual pages to physical page frames in memory. The paging mechanism consists in dividing the linear address in parts, used as offsets in mapping structures called *page tables*. In its version

2.6.35, the kernel implements four levels (splitting up the linear address in up to five parts) of page tables:

- Page Global Directory (PGD)
- Page Upper Directory (PUD)
- Page Middle Directory (PMD)
- Page Table (PTE)

These levels are linked to each other (in the order cited) and each table contains addresses to a set of tables on the next level (see Fig. 4). This scheme was built to fit the 32-bit and 64-bit architectures, as well as the Physical Address Extension or PAE (support for some Intel processors with 36 address pins, i.e. can address up to 64 GB of memory with three page levels) and extended paging (translation for systems with 4MB page sizes with just one page level) [3].

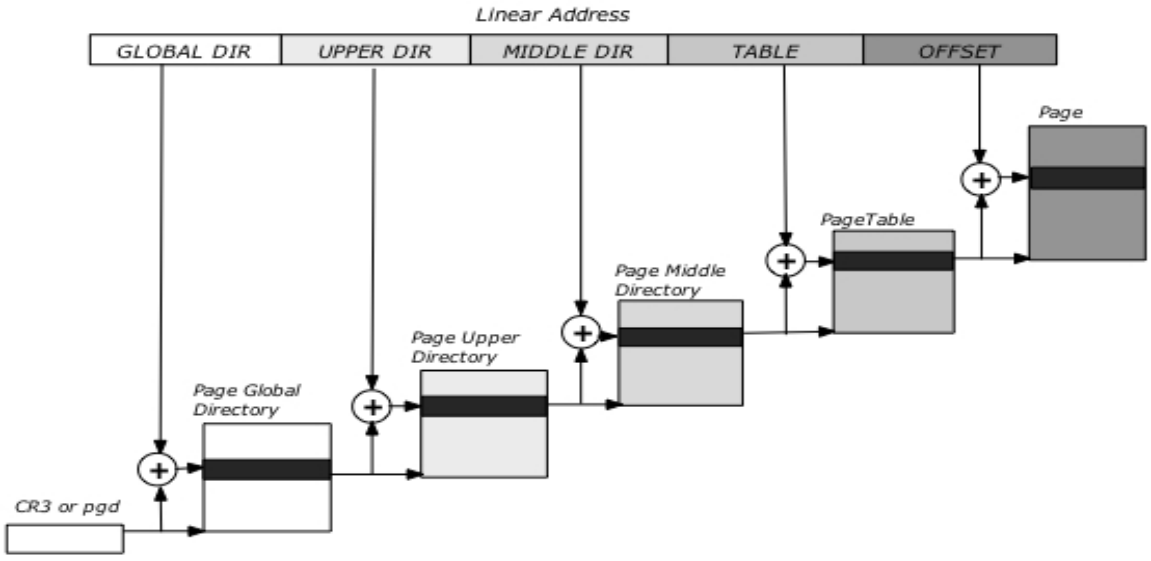

Figure 4. The Paging Mechanism [3]

For Intel x86 32-bit systems, the virtual address is divided in two parts which serve as offsets in the only two levels of page tables implemented (here, PUD and PMD do not exist).

3.2. Process-Related Kernel Data Structures

3.2.1. The Process Descriptor

Probably the most important process bookkeeping kernel artifact, the *process descriptor* is the entry point in memory to all the available information about a specific process or *task* (as commonly referred to in the kernel jargon). It is represented by the `task_struct` (cf. Fig. 5) data structure defined in `include/linux/sched.h` (from the top level directory of the kernel source code), whose fields are used to identify, store and track the various aspects of a task including but not limited to [3, 40]:

- *Process State*: Through the `state` field in the `task_struct`, the execution status (running, stopped, interruptible or not, etc…) of the task is saved.
- *Process Id and Name*: Mostly used for identification, the id and name of the process are respectively kept in the `pid` and `comm` attributes in the `task_struct`.
- *Credentials*: These represent the permissions that are associated with the process and are used to determine what operations the task is (or is not) allowed to perform. It is represented in the process descriptor by an attribute of type `cred`, a kernel structure defined in `include/linux/cred.h`. This

structure contains information such as the user (`uid`) and group (`gid`) ids associated with process.

- *Scheduling Properties*: For a more efficient use of the CPU execution cycles and control over the process switching and scheduling activities, the kernel stores via the process descriptor a number of values such as the process priority and scheduling queue.
- *Process Address Space*: User space tasks require memory areas to store segments (code, data, stack, etc…) or to map files. The set of virtual addresses reserved for a process form its address space. The kernel keeps track of these memory locations through the `mm` field of type `mm_struct` (more details in Section 3.2.2).
- *Process Relationships*: The process descriptor provides a link to the parent, children and siblings processes (using respectively the `parent`, `children` and `sibling` fields) of a task.
- *Filesystem Properties*: With the `files` and `fs` variables in `task_struct`, the kernel can keep track of the files opened by a process as well as the current working directory (more details in 3.2.3).
- *Process CPU-related information*: The process descriptor, through a variable of type `thread_struct`, also gives access to information about relations between the process and the CPU, such as the address of the Kernel Mode Stack of the process (location where the CPU registers of the process are stored in memory).

include/linux/sched.h

```
1168 struct task_struct {
1169        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped*/
...
1170        void *stack;
1171        atomic_t usage;
1172        unsigned int flags;     /* per process flags, defined below */
...
1182
1183        int prio, static_prio, normal_prio;
1184        unsigned int rt_priority;
1185        const struct sched_class *sched_class;
1186        struct sched_entity se;
1187        struct sched_rt_entity rt;
1188
...
1221        struct list_head tasks;
...
1224        struct mm_struct *mm, *active_mm;
...
1243        pid_t pid;
1244        pid_t tgid;
...         struct task_struct *real_parent; /* real parent process */
1257        struct task_struct *parent;
...
1261        struct list_head children;      /* list of my children */
1262        struct list_head sibling;
...
1298        const struct cred *cred;
...
1305        char comm[TASK_COMM_LEN]; /* executable name excluding path*/
...
1319/* CPU-specific state of this task */
1320        struct thread_struct thread;
1321/* filesystem information */
1322        struct fs_struct *fs;
1323/* open file information */
1324        struct files_struct *files;
1325/* namespaces */
1326        struct nsproxy *nsproxy;
1327/* signal handlers */
1328        struct signal_struct *signal;
1329        struct sighand_struct *sighand;
1330
1331        sigset_t blocked, real_blocked;
1332        sigset_t saved_sigmask;
1333        struct sigpending pending;
...
1502};
```

Figure 5. Excerpt of `task_struct`

The kernel also keeps track of all the processes currently running on the machine by maintaining a *circular doubly linked list* of process descriptors, through the `tasks` field

(of type `list_head`, the kernel's implementation of a circular doubly linked list) in each `task_struct`. Therefore, it is possible to obtain a list of all running tasks by traversing the linked list (See Appendix B), potentially starting from the kernel address of the `init_task` process (named *swapper* and first item in the doubly linked list). This address can be obtained in various ways such as looking into the kernel symbol table file *`/boot/System.map-x`* ('x' represents the kernel version) [46] and through heuristics-based search performed on the memory image [20].

3.2.2. Process Address Space

The Linux kernel uses a set of related data structures to manage the virtual memory space needed by a process. As stated in the previous subsection, each process descriptor has a field of type `mm_struct` dedicated to that purpose. The `mm_struct` data structure sometimes referred to as the *memory descriptor*, is defined in *`include/linux/mm_types.h`* and retains among others, the following information (See Fig. 6):

- *Page Global Directory*: The `pgd` field in `mm_struct` stores this value, which represents the entry for this process' memory areas in the Page Global Directory (see Section 3.1.2).
- *Memory Segments*: The kernel also uses the `mm_struct` to monitor process' segments in memory. Thus the `start_code` and `end_code` attributes refer to the limits of the text/code segment of a task, `start_brk` and `brk` to the growing heap, `start_data` and `end_data` to the data section containing initialized static variables, `start_stack` to the start of the stack region,

`arg_end` and `arg_start` to the command line arguments, `env_start` and `env_end` to the environment variables.

- *Memory Regions*: These are non-overlapping sets of adjacent virtual addresses that map into the various memory segments of a process. The `mmap` field (of type `vm_area_struct`) of `mm_struct` points to the first of these regions. A memory segment is actually a group of memory regions serving the same purpose.

include/linux/mm_types.h

```
222 struct mm_struct {
223         struct vm_area_struct * mmap;           /* list of VMAs */
224         struct rb_root mm_rb;
...
236         pgd_t * pgd;
237         atomic_t mm_users;          /* How many users with user space? */
238         atomic_t mm_count;   /* How many references to "struct mm_struct"
                                        (users count as 1) */
239         int map_count;                      /* number of VMAs */
...
254         unsigned long start_code, end_code, start_data, end_data;
255         unsigned long start_brk, brk, start_stack;
256         unsigned long arg_start, arg_end, env_start, env_end;
257
...
313 };
```

Figure 6. Excerpt of `mm_struct`

For every task, the kernel maintains a linked-list of its associated memory regions. Therefore, each `vm_area_struct` (defined in *`include/linux/mm_types.h`*) includes a `vm_next` field pointing to the next region on the list, as well as a link (through the `vm_mm` attribute) back to the memory descriptor describing the address space they belong to. The structure also has self-describing elements such as `vm_start`, `vm_end` and `vm_page_prot`, respectively representing the beginning address, end address and access

permissions of the region. In case the `vm_area_struct` describes a non-anonymous memory-mapped disk file or shared library, the `vm_file` field is non-null and points to a `file` structure (see Section 3.2.3), while `vm_pgoff` represents the offset of the region in the mapped file in question. Fig. 7 gives a sample overview of the kernel structures related to a task's address space and their relationships.

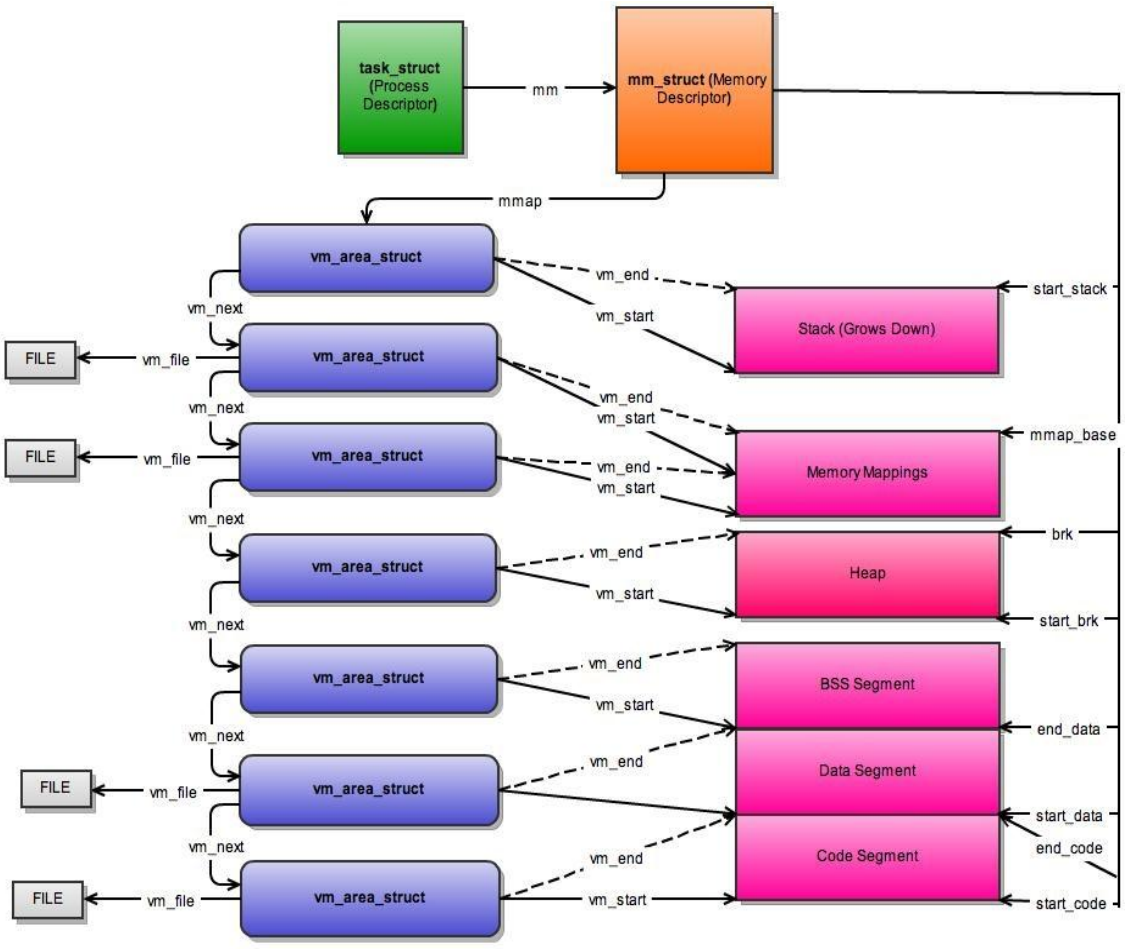

Figure 7. Process Address Space Structures

3.2.3. Processes and Filesystem Kernel Structures

Processes in Linux constantly interact with files on disks; they are either mapped in memory for easier access or opened for basic input/output operations. However, regular named disk files are not the only type of files in the Linux system: *directories, symbolic links, device files, sockets and pipes* are all considered files [3, 40]. The Linux kernel uses a number of structures to store and maintain data on the relationships between process and files:

- *`fs_struct`*: Defined in *`include/linux/fs_struct.h`*, it is accessed from the process descriptor of a task through the `fs` field and keeps information on the current working directory (`pwd` attribute), the root directory (`root` attribute) and their associated mounted filesystem. The same `fs_struct` can be shared between different processes that are executing from the same location; the field `count` is used to store the number of process descriptors linked to this structure.
- *`files_struct`*: also accessible from a process' `task_struct` through the `files` attribute, it is defined in `include/linux/fdtable.h` and is sometimes referred to as the *open file table structure* to describe the type of information it holds. The `fdt` field of `files_struct` is of type `fdtable` (also called *file descriptor table*) that keeps, among other data, the maximum number of files that a process can have opened (`max_fds` field) and the array

of file descriptors currently opened by the task. In Linux, the first three entries are `stdin`, `stdout` and `stderr`.

- *`file`* (see Fig. 8): defined in *`include/linux/fs.h`* and referred to as *file descriptor*, it is the main structure used by the kernel to get information about the relationship between a process and the file, such as the process access mode (attribute `f_mode`), the current offset (`f_pos`) and a pointer to the file operations table (`f_op`; the table stores pointers to functions that are associated with the file). The structure also holds general file data including the user credentials (`f_cred`) as well as the directory entry (from which the pathname of the file can be constructed), and its associated mounted filesystem metadata, both accessible through the `f_path` field.

include/linux/fs.h

```
917 struct file {
918         /*
919          * fu_list becomes invalid after file_free is called and queued via
920          * fu_rcuhead for RCU freeing
921          */
922         union {
923                 struct list_head        fu_list;
924                 struct rcu_head         fu_rcuhead;
925         } f_u;
926         struct path             f_path;
927 #define f_dentry        f_path.dentry
928 #define f_vfsmnt        f_path.mnt
929         const struct file_operations    *f_op;
930         spinlock_t              f_lock;  /* f_ep_links, f_flags, no IRQ */
931         atomic_long_t           f_count;
932         unsigned int            f_flags;
933         fmode_t                 f_mode;
934         loff_t                  f_pos;
935         struct fown_struct      f_owner;
936         const struct cred       *f_cred;
...
950         struct address_space    *f_mapping;
...
954 };
```

Figure 8. Excerpt from `file`

Although the concepts and design remain the same for subversions of Linux 2.6, there might exist some slight differences in the code of the described structures. The disparities become greater when compared with older versions such as Linux 2.4. For example, rather than having an entire structure to handle process credentials (i.e. the field `struct cred` in the process descriptor), Linux 2.4 and even early Linux 2.6 versions just had extra fields added to the `task_struct` (`uid`, `gid`, `suid`, etc…). This is one of reasons why most the tools developed to analyze the RAM are operating system-dependent. To this effect, our experiment was entirely performed on machines running the same Linux distribution, on top of the same kernel version 2.6.35, even though it might work on different versions.

4. EXTRACTING PROCESS INFORMATION FROM THE RAM (GETTSK)

Chapter 3 gives an overview of how the Linux operating system manages the memory and keeps track of the tasks running on the machine. Now, we discuss the mechanisms necessary to retrieve process-related information, when provided with an image of the RAM of a Linux system. We start by describing the environment setup to carry out our experiment; then we document what specific data is extracted as well as how do to it. We conclude the chapter by presenting the results and discussing the limitations of our approach on the proof-of-concept.

4.1. Initial Setup and Image Acquisition

In order to perform the procedures described in the next sections, we need the image of the physical memory acquired from a Linux machine. To that end, a virtual environment was built with the following specifications:

- *Virtualization Software*: VMware Workstation 6.5.7
- *Operating System*: 32-bit Linux Ubuntu 10.10 – Kernel 2.6.35.22
- *Amount of RAM*: 512 MB
- *Hard Drive*: 20 GB, with no swap space.

These configurations were not selected randomly. Indeed, the virtual setup would allow us to easily obtain the contents of the RAM, as virtualization software such as VMware use a file on the disk of the host computer (physical machine on which the environment is

built) to simulate the memory. When a snapshot of the virtual machine is taken, or when it is pause at some point in time, that file (with *.vmem* extension if VMware is used) can be copied and will represent an image of the Ram at that exact moment [47]. On the other hand, obtaining the image of the memory from a running full blown system can be tedious as discussed in Section 2.1.2.1.

Ubuntu is one of the most used distributions of Linux; version 10.10 (also called *Maverick Meerkat*) was the latest release at the beginning of our project and is based on the 2.6.35 kernel.

The size of the RAM was chosen to avoid the complexities that a `ZONE_HIGHMEM` would introduce to the virtual to physical address translation (see Section 3.1.2 and [5]), thus it was kept below 896MB. The size of the virtual hard drive was chosen arbitrarily, but we opted for no swap space. The swapping mechanism uses some preconfigured space on disk to extend the available physical memory by moving some pages from the RAM to the allocated area on hard drive. The complexity associated with this practice will be ignored.

After building the virtual environment, the next activity involves creating a process and running it, at which point the VM can be paused and a snapshot taken so that an image of the memory can be copied. Because the created process serves as basis to the proof-of-concept implementation and is central to the rest of the experiment, it should be specially crafted to be simple and flexible. The following source code is the C program (`test.c`) that we used to create such a process:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
      float i, j;

      for (i=0; i<500000; i++)
            for(j=0; j<50000; j++);

      int fd;

      fd = open("sample.txt", O_RDWR|O_CREAT);

      write(fd, "Done.\n", 6);
      close(fd);

      return 0;
}
```

This program executes a nested for loop, then opens a file called `sample.txt` (creates it if it does not exist) in which it writes the string `"Done."`. The file is then closed and the program exited. The purpose of the nested for loop is to give us enough time to take a snapshot of the VM and even get some information about the process for verification purposes in future stages. The constant values were chosen to ensure that the wait time was sufficient. A simple call to the `sleep()` function would have been more predictable, unfortunately returning from system calls during the resumption phase is problematic (see Chapter 5).

Following is the source code to a second C program (*`test2.c`*) on which the experiment was also performed. This program was designed to verify if data (variables in this case) are preserved through the procedure:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
```

```
        float i, j;
        int fd, value;
        char buf[5];
        int k=4;
        static char dec[] = "0123456789";

        value = 12 + 32;

        for (i=0; i<500000; i++)
              for (j=0; j<50000; j++);

        value += j;

        do {                    /* Implementation of itoa()* /
              buf[k--] = dec[value%10];
              value /= 10;
        } while(value);


        fd = open("sample.txt", O_RDWR|O_CREAT);
        write(fd, buf, sizeof(buf));
        write(fd, "\nDone.\n", 8);

        close(fd);

        return 0;
}
```

The variable `value` is modified before and after the nested loop is executed, and the final value (`50044`) is to be printed into the "`sample.txt`" file. The `do..while` statement is used to convert `value` into a string buffer to be written in the output file.

Both `test.c` and `test2.c` were compiled using the following `gcc` commands (which imply the default level of optimization - level 0):

```
gcc –Wall –o test test.c

gcc –Wall –o test2 test2.c
```

The binaries were then run, a snapshot of the VM was taken while the nested loop was being executed and the memory images collected.

4.2. Extracting Process Information

When an investigator is provided with a snapshot of memory for analysis, after ensuring the integrity and preservation of the image, there are a number of activities that she or he can perform, using the tools and methods described in Section 2.3., to obtain information about the state of the machine at acquisition time. The following is a general sequence of steps that can be followed in case the examiner wishes to retrieve process-specific data from an image of RAM in order to resume its execution. For each of these steps, we will also present and explain how they were applied to the memory image and the `test` and `test2` processes of Section 4.1 by the tool `gettsk`. Written in Python, `gettsk` takes as input the image file and the task's process ID and produces the artifacts related to the specified process (The source code for `Gettsk` is included in Appendix C).

4.2.1. Identifying the Process to be Saved

The first activity is concerned with determining what processes were running on the machine at the moment of acquisition of the RAM, and choosing among them which one might be of interest. As mentioned in Section 3.2.1, the Linux kernel maintains a circular doubly linked list of all the processes loaded in memory. Therefore, by traversing the structure, we can obtain the list of running tasks. One of the difficulties here lies on finding a starting point. There are two approaches to solve this issue:

- If the filesystem of the machine is available, the address of the process descriptor (Section 3.2.1) for the `swapper` process can be retrieved from the

kernel symbol table file `/boot/System.map-x` ('x' being the kernel version) [46].

- If the investigator does not have access to files, it is still possible to find the address of the `task_struct` of the `swapper` process, by combining search and known facts about the way Linux manages the memory. For example, one can start with searching the image for strings such as "`swapper`" or "`init`" (first two kernel tasks, that are always running on a well-functioning Linux system), then use the fact that their PID is respectively 0 and 1 and that every kernel address is greater than `PAGE_OFFSET` (equivalent of `0xC0000000`) to build a heuristic algorithm that would find the address to a process' `task_struct` as well as offsets to some of its fields.

For the purpose of our study, we used the filesystem of the virtual machine, specifically the file `/boot/System.map-2.6.35-22-generic` to read the linear kernel address of the `init_task`: **`0xc07c76e0`**. The actual address in memory is obtained by subtracting `PAGE_OFFSET` (see Section 3.1.2).

Having the starting point is not enough for the traversal; we need to be able to move from one task to the next. The kernel uses the `tasks` field of the process descriptor for each process to link it to its neighbors in the list.

The offsets to fields in kernel data structures are derived manually by navigating through the source code for the structure, accounting for the size of each field (taking into consideration the storage size of basic types, for example one byte for long and pointers, etc…). Automating the computation of these offsets is problematic because of C preprocessor conditional groups defined in the kernel source code. For example, `#ifdef`

statements are common and add or remove fields to kernel structures depending on certain macros. However, the value of these macros cannot be obtained from the RAM image alone.

Using this procedure, the offsets for the `tasks, comm` and `pid` fields of the `task_struct` can be found and used to walkthrough and list the processes running on the machine from the memory image (Appendix A contains all the offsets used during our experiment; these values are very specific to our environment and apply only to the kernel version 2.6.35.22). This is exactly what the program `ps.py` (whose Python source code is in Appendix B) does on our proof-of-concept image.

4.2.2. Retrieving the Process' Artifacts and Metadata

The previous subsection explains how to obtain the list of running processes from the memory image. The investigator can now identify a specific task that could be of interest to the case and proceed to extract the data related to that process from the RAM.

General Information:

These refer to the data that is used for identification or to determine the basic status of the process and include:

- *The Process ID*: Integer value uniquely assigned to tasks running on a machine. It is mainly used for identification purposes. It is the `pid` field of the `task_struct` for the process. It is passed as input to the `gettsk` tool.

- *The Process' name*: String also used for identification, excluding the execution path. It is the `comm` field of the process descriptor. In our illustrative examples, "`test`" and "`test2`" would represent the name.
- *The Execution State*: The field `state` of the `task_struct` describes the running status of the program. If the value is negative, the process is unrunnable, runnable/running if it's 0, or can take the values showed in Fig. 9.

include/linux/sched.h

```
182 #define TASK_RUNNING            0
183 #define TASK_INTERRUPTIBLE      1
184 #define TASK_UNINTERRUPTIBLE    2
185 #define __TASK_STOPPED          4
186 #define __TASK_TRACED           8
...
191 #define TASK_DEAD               64
192 #define TASK_WAKEKILL           128
193 #define TASK_WAKING             256
194 #define TASK_STATE_MAX          512
```

Figure 9. Possible values of the execution state

- *Process Flags*: Provide more details on the status of the task. They can take a wide range of values such as `PF_STARTING` when the process is being created, `PF_KTHREAD` for kernel threads, `PF_VCPU` if run on a virtual CPU or `PF_SUPERPRIV` if used super-user privileges. The list of all the possible options can be found in `include/linux/sched.h`.
- *Process Credentials*: As mentioned in Section 3.2.1, they refer to permissions associated with the process. A pointer to a `cred` structure can be found in the `task_struct`. We can thus obtain the pairs `(UID, GID)`, `(SUID, SGID)`,

(EUID, EGID) and (FSUID, FSGID) which denote the real, saved, effective and filesystem-related user and group ID.

The following table summarizes what general information does the gettsk tool extract, as well as actual values for our test and test2 processes when statically compiled (More in Section 5.2.3).

Metadata	Data Structures	Fields	Values for test.c	Values for test2.c
ID	task_struct	pid	21526	14475
Name	task_struct	comm	test	test2
State	task_struct	state	0	0
Flags	task_struct	flags	0x00000078	0x00000078
Credentials	cred	uid, gid, euid, egid, suid, sgid, fsuid, fsgid	1000 (same value for all of them)	1000 (same value for all of them)

Table 1. General Information extracted by gettsk

Process Address Space:

The memory segments and mappings are core components to Linux processes. We first collect structural information from the memory descriptor, that is, start and end virtual addresses to memory segments (code, data, stack, heap, arguments and environment variables). They can be found in the mm_struct of the process.

To ease the collection of data from memory, `Gettsk` implements a generic data structure (called `Structure`) to mimic the different structures used by the Linux kernel to maintain process information:

```
class Structure:
      def __init__(self, **kwds):
            self.__dict__.update(kwds)

      def __str__(self):
            state = ["%s = 0x%08x" % (att, value)
                  for (att, value)
                  in self.__dict__.items()]

            return '\n'.join(state)
```

For example, the following `vmstruct` is used to store information from the memory descriptor of a process in the kernel.

```
vmstruct = Structure(start_code, end_code,
                        start_data, end_data, start_brk, brk,
                        arg_start, arg_end, start_stack,
                        env_start, env_end)
```

Using the starting address of the `task_struct` and the configured offsets, `gettsk` can access the data in the RAM image and copy them into the built structures .

Next, we extract the memory regions' metadata and their contents. The Linux kernel maintains a linked list, sorted by starting virtual address, to keep track of memory areas. The memory descriptor of each process has a field `mmap` of type `vm_area_struct` that points to the first element of the list of regions. Each `vm_area_struct` or region descriptor has a field `vm_next` pointing the current area to the next (and a field `vm_prev` pointing to the previous). These can be used to navigate the list of region and extract the self-descriptive data for each: `vm_start`, `vm_end`, `vm_prot`, `vm_flags`, `vm_pgoff` and `vm_file` defined in Section 3.2.2. In case the value of `vm_file` is not null, it points

to a `file` structure from which the pathname of the file can be obtained (refer to Section 3.2.3).

For anonymous memory areas, that is, regions that are not mapped to a file on disk (`vm_file` is null), `gettsk` dumps their contents, page by page, from the memory image. This is done by following the following pseudo-algorithm, where `PAGE_SIZE` represents the size of a page frame (4KB on x86 Linux systems):

```
start
vaddress =  vm_start
while (vaddress <= vm_end )
      Convert vaddress to physical address
      Dump page at physical address
      vaddress = vaddress + PAGE_SIZE
end
```

Machine-dependent data:

The CPU registers associated with the running task also have to be saved, as they are part of the process state when it is paused. When performing context switching, the Linux kernel saves the values of the general registers at the bottom of the Kernel Mode Stack of the process [3]. In older versions of Linux, the location of `task_struct` was always right below the Kernel Mode Stack of the process, so the register set could be obtained using the following formula (in the case of a 8KB kernel stack, with p representing the address of the process descriptor) [52]:

```
struct pt_regs *regs = ((struct pt_regs *)(2*PAGE_SIZE + (unsigned long)p)) - 1.
```

However, this setup posed a security risk, so, since version 2.6, the Kernel Mode Stack is randomly allocated. Luckily, the kernel stores its address in memory. Indeed, the field `thread.sp0`, where `thread` is a field of `task_struct` (of type `thread_struct` - structure storing CPU specific data for the process), contains a pointer to the bottom of the Kernel Mode stack right where the register set is saved.

For 32-bit Linux Kernel 2.6 systems, the register set structure (`pt_regs`) is defined in the source code as follows (location *`arch/x86/include/asm/ptrace.h`*):

```
struct pt_regs {
         unsigned long bx;
         unsigned long cx;
         unsigned long dx;
         unsigned long si;
         unsigned long di;
         unsigned long bp;
         unsigned long ax;
         unsigned long ds;
         unsigned long es;
         unsigned long fs;
         unsigned long gs;
         unsigned long orig_ax;
         unsigned long ip;
         unsigned long cs;
         unsigned long flags;
         unsigned long sp;
         unsigned long ss;
 };
```

The size of this structure is 68 bytes (8 * 17 fields) and thus, the following formula can be used to obtain the registers set (See Fig. 10):

```
struct pt_regs *regs = ((struct pt_regs *)((unsigned long)(p->thread.sp0) - 68)
```

With the address of the top of the Kernel Mode Stack, `gettsk` copies the process registers in a byte stream.

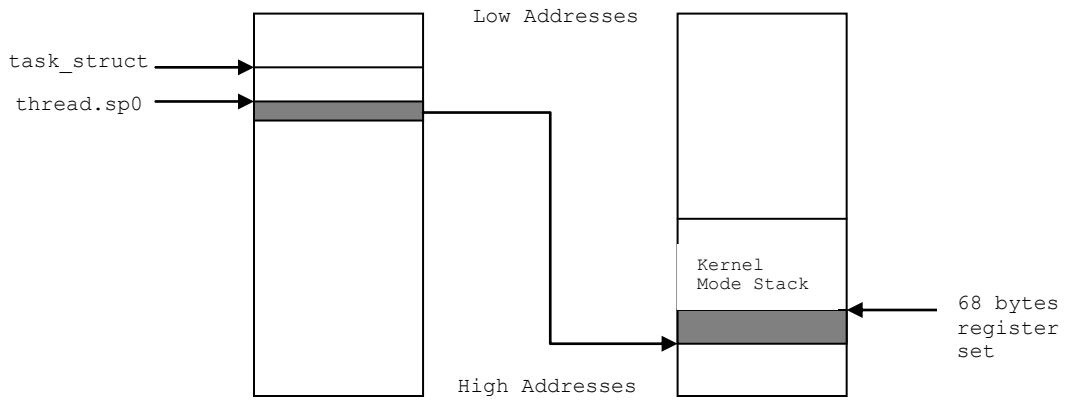

Figure 10. Finding the Process Register Set

Filesystem information:

The interaction between tasks and the filesystem is fundamental to the functioning of a computing system. In Linux, files can be of different types: Regular named files, directories, symbolic links, device files, sockets or pipes.

As mentioned in Section 3.2.3, the kernel manages an array of file descriptor numbers (integers uniquely associated with files used by a task), through the file descriptor table, that keeps track of all the files currently opened by the process. Each file descriptor number refers to a structure of type `file` (see Fig. 8), which holds data about the file in question and its relationship with the process.

The tool `gettsk` does not fully support extracting filesystem information from memory. It only collects the pathname of regular files on disk and does not deal with either pipes or sockets (See Section 4.3).

Child Processes Information:

It is possible that the process to be extracted had child processes also running on the machine. The kernel keeps track of all the children of a process through a circular doubly linked list accessed through the `children` field of the `task_struct`. Their information should also be retrieved and their hierarchy (child processes might also have children) preserved. It would allow investigators to have a more accurate depiction of their effects on the system resources, as well as the communication between them.

Because each of the threads in the hierarchy also has their execution state saved in memory, the procedure described so far to retrieve information about a task can be repeated for all the threads and saved in a tree-like structure.

`Gettsk` does not support retrieving child processes data. This is not problematic for our proof-of-concept since out `test` and `test2` processes do not have child processes.

4.3. Saving the Obtained Results

All the artifacts extracted in the previous steps are to be saved in a simple, efficient and portable format. Indeed, the size of the information collected can be considerable, especially if the contents of memory regions are dumped. Also, it should be simple enough for the investigators to be able to decipher the data and find evidence. One of the goals of our exercise is to determine if it is possible to resume the execution of the saved process, potentially on a different host. Therefore, the requirement that the output should be portable plays a special role in these circumstances.

The `Gettsk` tool organizes the retrieved data in an XML (eXtensible Markup Language) file. The XML language was chosen because it is standardized widely supported and

defines rules for building files that are both human and machine readable. Moreover, it is customizable, as users can construct and implement specific formats that valid files have to follow: they are called schemas. Appendix D contains the XML schema (defined using XML Schema Definition or XSD language) used by `gettsk` to store the recovered data. The following is an excerpt of the resulting file for out `test` process, displaying how the name, pid, credentials and register set are stored:

```
<?xml version="1.0" ?>
<process arch="x86" kernel_version="2.6.35" magic_number="FFC0">
	<pid>
		21526
	</pid>
	<name>
		test
	</name>
	<state>
		0
	</state>
	<flags>
		0x00000078
	<flags/>
	<credentials egid="1000" euid="1000" fsgid="1000" fsuid="1000" gid="1000"
sgid="1000" suid="1000" uid="1000"/>
	<reg_set encoding="binary-base64">
	AAAAAMQhnL9QIZy/AGmeR8CJBAgoIZy/AQAAAHsAAAB7AAAAAAAADMAAAQ////74IECHMAA
AACAgAAACGcv3sAAAAAAAAA
	</reg_set>
```

For binary streams that are dumped in the output, such as CPU registers or the contents of memory areas, `base64` encoding is used to transform the data in textual form. This was used first to accommodate XML standards that require the use of Unicode encoding. `Base64` is also a very common encoding/decoding scheme widely used to transfer data on the Internet.

4.4. Limitations of `Gettsk`

The `gettsk` implementation described in the previous section is developed as a proof-of-context for extracting process information from an image of RAM. There are still a lot of unexplored or unsupported features that could be of interest to an investigator. They include:

- *Determining offsets to fields of kernel structures in memory*: This is probably the most important issue the tool still faces. The offsets and configurations displayed in Appendix A were obtained by hand and are specific to a version of the Linux Kernel. Therefore, for an investigator to use this tool on another Linux platform, these values have to be reexamined and reevaluated. This shortcoming is tied to the fact that the tool does not have much room for extension to different versions of the kernel.
- *Files*: We mentioned in Section 4.2.2 that `gettsk` does not support extracting information about pipes and sockets (although they might contain knowledge valuable to investigators such as external IP addresses and/or inter-process communication data). They require information about other processes or external entities that we could not obtain (although they may exist) from the memory image. For files on disk, only their pathnames is saved to allow the investigator to at least have a list, which can be examined further for potential evidence. Our proof-of-concept processes do not have any open file, therefore the outcome of our experiment is not compromised, but in case a task of interest has open files, more information about them should be extracted

including the file descriptor number, the current position in the file, its permissions and open mode (read, write, etc…). To prepare for the resumption phase, these files should be packaged together with the XML output file and transported to the host machine.

- *Process Hierarchy*: `Gettsk` also does not store any information about parent and child processes. In an ideal situation, every task that is linked to the process passed as input should be extracted and saved following the same hierarchy (i.e. with details about their relationships with each other).
- *Processes executing a system call*: It is possible that at the moment the RAM was collected, a process of interest was waiting on a system call (this would have been the case for our `test` process if the `sleep()` system call was used, refer to Section 4.1). In such a case, in addition to the process information, the state of the kernel should be captured. `Gettsk` does not explore this option.
- *Inter-Process Communication (IPC)*: Tasks often cooperate with each other through techniques such as message passing, synchronization or Remote Procedure Calls (RPC). In this Chapter, apart from sockets for networking and pipes, we did not discuss what other kernel objects are required to perform IPC. They include signals, message queues, held semaphores, shared memory metadata and of course the other processes involved in the transactions.

5. Resuming the Execution of an Extracted Process (memexec)

We have successfully extracted process information from an image of the RAM of a Linux system. We now focus on the question of whether it is possible to resume the execution of the task using the data collected. It turns out that under certain circumstances the answer is affirmative. This chapter will first describe what type of configuration is required from the machine in which the resumption will occur, and then will follow a discussion on how this can be actually be performed. A tool, `memexec` built to implement these recommendations on our proof-of-concept experiment will also be presented and explained, as well as its results and limitations.

5.1. Requirements and Design Options

Earlier in this thesis, we mentioned the need for the environment in which malware analysis is performed to be contained. This is one of the main pillars of our approach to resume a process obtained from a memory image. However, the process cannot just be transported onto any machine and revived; there is a very good chance that it will not work at all. To ensure that the resources and configurations are the ones expected by the task, the host should be as close as possible to the initial system. Ideally, the machine from which the RAM was acquired would be the primary choice for resumption. Unfortunately, it is usually a production system which might be compromised in an unknown manner, and thus is susceptible to produce manipulated

evidence, misguide the investigator and/or propagate malware to other computers. The following are some of the most important configurations that the host machine should have:

- The underlying hardware of the host must be the same as the original. Processes are built and compiled for a specific machine and are highly dependent on the CPU architecture.
- Similarly, it is preferable for the operating system of the receiving machine to match with the initial computer. However, this requirement is weaker than the restriction on the hardware: It is possible that the resumption process will work on different versions of the same operating system, especially for Linux if they are from the same patch level (version 2.6.34 and 2.6.35 for example) or if the new kernel is a recompile of the original.
- Unless it was (or will be) used by the saved process, a file on disk does not need to be replicated in the host machine. However, because the pathname of opened and mapped files depend on the filesystem of the original machine (root directory and current working directory), it might be necessary for some files to exist in specific locations. It is possible to work around this requirement, by manually renaming the files.
- Similarly to files on disk, the users and groups existing in the original system are not necessary to the resumption of the task. But, in case the owner of the initial process needs to be restored, the permissions of the user/group in question should be the same in the two machines, especially on resources accessed by the saved task.

After the data has been extracted from the memory image and the receiving machine is adequately configured, the resumption can begin. The general idea behind resuming the execution of a process involves moving the saved information into the appropriate locations and launching its execution. For the Linux kernel, this activity can be performed either from User Space or from Kernel Space.

From User Space, this can be achieved by using the tracing capabilities of Linux (with the `ptrace()` call for example) and other system calls such as `mmap()` to recreate memory mappings or file operations (`open()`, `write()`, etc…) for opened files. This would allow for a cross-version solution, since most system calls are kept across kernel changes. This method might work for very simple processes, but issues will quickly arise for more complex tasks, as system calls do not permit users to change specific values in kernel data structures, which might be desirable or required in some cases. Furthermore, it would be difficult to modify some process metadata such as name, pid or execution state from user space without hacking the kernel.

The alternative solution involves restoring the saved artifacts while running in kernel mode. This gives direct access to kernel data structures and thus all the changes necessary for process resumption can be made. However, this technique is very kernel-dependent, thus the solution might not work in different kernel versions. Here, two options are available: Either patch the kernel or use a Loadable Kernel Module (LKM). Patching the kernel implies creation of new system calls and/or modification of existing kernel code that would allow the user to carry out their task. The kernel has to be recompiled and the

machine restarted every time a new functionality has to be added or modified, which can become problematic and cumbersome for large project.

Using a loadable kernel module offers more flexibility and is much easier on the developer. Indeed, a kernel module is a set of functionalities that can be added and removed from the kernel (using the `insmod` and `rmmod` commands respectively) without rebuilding it. They are particularly useful when creating drivers for devices and adding them to the kernel.

The kernel module approach is the one used for our experiment and the toolset `memexec` includes the `memlkm` program implementing a pseudo device driver (`/dev/memexec`), which will permit us to execute customized code in kernel mode.

5.2. Restoring Process Artifacts

5.2.1. General Methodology

In Section 4.2, we described ways to retrieve information about a running process from memory and prepare a package that would help reconstruct the extracted process and resume its execution on a similar environment. We now discuss how to achieve such reconstruction. Two basic methods can be considered here:

- Create a new kernel task from scratch containing only information from the saved process and launch its execution.
- Use a process already running on the host and modify its behavior and artifacts to mimic the saved process.

The first approach is similar to the proposal made by Ilo [28] (see Section 2.4), which involves building a process from the ground up (implies creating a new `task_struct` and a new address space), insert data saved during the extraction stage and schedule its execution. This would be ideal from a forensic perspective, given that all the information about the revived process will be known to the investigator. But, this turns out to be a daunting task without the help of the operating system because it requires accounting for every single kernel structure and performing the appropriate initialization for each of them. If even one variable or structure is not correctly instantiated, a system crash or an unwanted behavior can happen.

The second method on the other hand, is closer to the restart component of the CRAK process migration tool [48], where the program that parses the "checkpoint" file (file containing information about a saved process) is the one being modified to resemble the original task. This approach is simple since the operating system already built the foundation for the preexisting process and so, only the insertion of saved information (such as memory regions and the register set) is to be performed. However, artifacts associated with the modified task remain in memory and might have effects on the resumed task. But, since the process is chosen, its impact is known to the investigator. This method is the one used during our experiment.

The following steps give a general overview of what activities should be done to resume the execution context of a saved process:

1. Create a new process: This is optional depending on the method used.
2. Modify the process to insert information gathered from memory.
3. Launch the execution.

The next subsections will illustrate how these activities were achieved during our proof-of-concept experiment, giving details on the inner-workings of the `memexec` toolset.

5.2.2. Implementation (Memexec)

`Memexec`, written in C (see Appendix E), is divided in two components: a loadable kernel module (`memlkm`) that will be used to restore the artifacts of the saved process and a User Space program that parses the input files and launches the kernel mode operations. After the kernel module is loaded into the system, `memexec` takes as input an XML file following the format described in Chapter 4, and performs the following activities:

1. *Parsing the XML file into data variables*

Using the *`libxml2`* library (see Makefile in Appendix E), the input file is parsed and the information is saved in a `memproc` data structure, storing all the process artifacts contained in the file. The structure is defined as follows:

```
struct memproc {

      unsigned long old_pid, state; /* PID and state */
      int flags;            /* Process Flags */
      int seg_num;         /* Number of Segments */
      int file_num;       /* Number of Open Files */

      char * regs;            /* Bytes holding the register set */
      char name[16];          /* Name */
      struct credentials memcred;   /* process user credentials */
      struct vmstruct vm_map;       /* Memory descriptor info */
      struct segments * seglist;    /* Memory regions */
      struct open_file * filelist;  /* List of open files */
      struct signals sig;

};
```

It is also at this stage that the base64 encoded data in the XML input file is transformed back into binary streams.

2. *Switching to Kernel Mode and restoring extracted artifacts*

After parsing the data from the XML file, the next step involves rebuilding the saved process. `Memexec` then opens the device file `/dev/memexec` and through the device's Input/Output Control (ioctl), it can execute kernel mode instructions. The variable of type `memproc` containing the saved information is passed to the ioctl call. As we mentioned earlier, the tool does not create a new process, rather it modifies the program used to open the device file (also called `memexec` in our case). Next the task's metadata is restored as follows:

General Information:

When in Kernel Mode, the address of the `task_struct` of the currently running process can be obtained using the macro `current`. Therefore it is simple to access and replace fields of the process descriptor. For example, the name of the process (which is the `comm` field of `task_struct`) is accessed with: `current->comm`.

Therefore, the name, process flags and execution state can all be restored by replacing the value appropriate fields in the process descriptor by the corresponding data from the saved process.

Concerning the process ID, the saved value must not be in use in the current system and must not be larger than a maximum (can be found in `/proc/sys/kernel/pid_max`). Thus, before the PID is replaced, some tests for availability must be made. `Memexec` does

not support restoring the PID of the process. This does not affect the resumption of our `test` and `test2` processes, but it might be problematic to a forensic investigation if the task of interest uses its PID during its execution.

We mentioned in Section 5.1 that if the credentials associated with the task extracted from memory should be restored, the corresponding users and groups (at least their IDs) must exist on the host machine and have the same privileges on the resources used by the resumed process. If this is satisfied, a new credentials data structure can be created with the values obtained from the extracted process. `Memexec` does not support restoring the credentials.

Process Address Space:

First we have to restore the virtual memory structure. Each segment attribute has to be replaced with its saved counterpart. The memory descriptor of the process can be directly accessed through the `mm` field of the process descriptor. Therefore, the following line of code will successfully change the address of the beginning of the code section for the process (`tmp` holds `vm_struct` data for the current region as defined in Section 4.2.2). This can be repeated with the other segments to reconstitute the virtual memory structure of the saved task:

```
current->mm->start_code  = tmp.start_code;
```

Each memory area is either mapped to a file or anonymously. In the first case, the region can be restored by remapping the file in question. Of course, the requirement that files on disk to be restored must exist on the host machine (see below and Section 5.1) applies in this case. The following code carries out the task, using the kernel function `do_mmap_pgoff()`:

```
do_mmap_pgoff(f, vm_start, size, mmap_prot, mmap_flags, pgoff);
```

The argument `f` is of type `file *`, and represents the file to be mapped; `vmstart` is the saved stating virtual address for the region; `size` is the amount of space to be mapped (obtained by subtracting the saved start virtual address from the end address of the memory area); `mmap_prot` designates the protection flags for the frames of the region; `mmap_flags` specify what kind of region is mapped (for example, because stack regions grow downwards, the macro `VM_GROWSDOWN` is used); `pgoff` is the offset at which the file should be mapped.

For anonymously mapped regions, the `do_mmap()` kernel function first allocates the needed amount of space, then the contents of the original memory area can be copied into the reserved space using the function `copy_to_user()` (This function permits to copy data from kernel space into a virtual memory location of the current process).

`Memexec` creates these new mappings without getting rid of the existing ones on the task being modified. Every time this was attempted, the process was killed by the Operating System. Thus the memory regions of the modified process are still present in memory after the address space is restored, except those that occupied virtual addresses used by the extracted process (they were replaced by `memexec`).

CPU Registers:

The Linux kernel provides a convenient function to locate the register set (from the kernel mode stack) of the currently running process: `task_pt_regs(current)`, defined in the source file `arch/x86/include/asm/processor.h`. Thus, the following lines of C code will be enough to restore the CPU registers:

```
struct pt_regs * regs;
regs = task_pt_regs(current);
*regs = *tmp;           // tmp is a pointer to the saved registers
```

Filesystem and Child Process Information:

To recover the file table of the process extracted from a RAM image, all the regular files must be reopened on the host machine and their position pointer moved to the saved location. More importantly, because it is used by the task to refer to the file, their file descriptor number must be restored as well. This can be done either by using both the `open()` and `dup()` system call from User Space or by directly manipulating the file table data structure in kernel mode [52].

As we mentioned in Section 5.1, the file in question must exist on the new system and have the same pathname and permissions as in the initial machine. If for some reason the pathname requirement is not an option (for example, subdirectories might not exist or security concerns might exist), the investigator can manually change the location of the file in the input data to suit his configurations.

Our experimental tool `memexec` does not support restoring the file table. Also, pipes and sockets were not explored: they imply communications with other processes or other machines, while our study was focused on the extraction and resumption of a single process.

Similarly, reviving a group of potentially related tasks is not included in our proof-of-concept. Therefore child processes were not considered. The general idea behind resuming the execution of a group of processes involves resuming each thread separately and reconstructing the links between them through kernel objects (`parents, siblings`, and `children` fields of the `task_struct` for example).

5.2.3. Results

After the process artifacts are restored, the kernel module can exit and yield the execution back to User Space. At this point, the next instruction will be fetched from the Instruction Pointer register and executed in the new context, which in theory will resume the execution of the original process in the new host.

During our experiment, the process was successfully revived. Indeed, when `memlkm` returned with no errors, rather than the next instruction of `memexec` (closing the device file), the nested loop in our `test` and `test2` processes was executed.

The rest of the program behaved differently depending on the initial compilation method:

5.2.3.1 Results for Dynamic Compilation:

In our first attempt at the experiment, the `test` program was compiled dynamically, the RAM acquired, the metadata of the `test` extracted with `gettsk` and the process resumed.

In the original machine, while the nested loop of `test` process was being executed, we were able to print its memory mappings using the Linux's *`/proc`* pseudo-filesystem as seen in Fig. 11.

The top 4 lines on the picture represent the last lines of a *`ps`* command run on the system to get the PID of `test`; then the *`/proc/PID/maps`* command was run to display the mappings. The first column in the output represents the start and end virtual addresses used by the region. The second shows the set of permissions associated with frames of the area (read, write, execute and private). The third column is the offset in the file (if

any). The last three are respectively the device major and minor, the inode and the pathname of the file mapped, if any.

```
airness  20672  0.0  0.6   6548  3260 pts/0    Ss   14:33   0:00 bash
airness  21344 96.2  0.0   1684   248 pts/0    R+   14:56   0:23 ./test
airness  21351  2.5  0.6   6504  3096 pts/1    Ss   14:56   0:00 bash
airness  21369  0.0  0.2   4592  1092 pts/1    R+   14:57   0:00 ps aux
airness@ubuntu:~/Desktop$ cat /proc/21344/maps
00112000-00269000 r-xp 00000000 08:01 921748     /lib/libc-2.12.1.so
00269000-0026a000 ---p 00157000 08:01 921748     /lib/libc-2.12.1.so
0026a000-0026c000 r--p 00157000 08:01 921748     /lib/libc-2.12.1.so
0026c000-0026d000 rw-p 00159000 08:01 921748     /lib/libc-2.12.1.so
0026d000-00270000 rw-p 00000000 00:00 0
00f01000-00f1d000 r-xp 00000000 08:01 921741     /lib/ld-2.12.1.so
00f1d000-00f1e000 r--p 0001b000 08:01 921741     /lib/ld-2.12.1.so
00f1e000-00f1f000 rw-p 0001c000 08:01 921741     /lib/ld-2.12.1.so
00fae000-00faf000 r-xp 00000000 00:00 0          [vdso]
08048000-08049000 r-xp 00000000 08:01 917600     /home/airness/Desktop/test
08049000-0804a000 r--p 00000000 08:01 917600     /home/airness/Desktop/test
0804a000-0804b000 rw-p 00001000 08:01 917600     /home/airness/Desktop/test
b7713000-b7714000 rw-p 00000000 00:00 0
b7721000-b7723000 rw-p 00000000 00:00 0
bfc4a000-bfc6b000 rw-p 00000000 00:00 0          [stack]
airness@ubuntu:~/Desktop$ clear
```

Figure 11. Original Mappings for the Dynamically Compiled `test` Program

The same procedure was repeated when the process was resumed and it resulted in Figure 12. The bold lines show the mappings restored from the original `test` process. The other mappings are associated with the process launched by `memexec` that were not deleted. This picture also demonstrates that the name of the process was successfully modified.

Unfortunately, after the nested loop, the process threw a segmentation fault while executing the `open()` function. We later discovered that every instruction that did not involve a system call can be successfully run, suggesting that the problem was associated with the dynamic linking of the shared libraries. We therefore opted to do the same experiment with a statically compiled binary.

```
airness@ubuntu:~/Desktop/Code/Memexec$ ps aux | grep test
airness  17908  0.0  0.2   5192  1280 pts/0    D+   07:14   0:00 ./test
airness  17912  0.0  0.1   4012   760 pts/1    S+   07:14   0:00 grep test

airness@ubuntu:~/Desktop/Code/Memexec$ sudo cat /proc/17908/maps
00110000-00112000 r-xp 00000000 08:01 921753     /lib/libdl-2.12.1.so
00112000-00269000 r-xp 00000000 08:01 921748     /lib/libc-2.12.1.so
0026a000-0026c000 r--p 00157000 08:01 921748     /lib/libc-2.12.1.so
0026c000-0026d000 rw-p 00159000 08:01 921748     /lib/libc-2.12.1.so
0026d000-00270000 rw-p 00000000 00:00 0          [heap]
00610000-00767000 r-xp 00000000 08:01 921748     /lib/libc-2.12.1.so
00767000-00768000 ---p 00157000 08:01 921748     /lib/libc-2.12.1.so
00768000-0076a000 r--p 00157000 08:01 921748     /lib/libc-2.12.1.so
0076a000-0076b000 rw-p 00159000 08:01 921748     /lib/libc-2.12.1.so
0076b000-0076e000 rw-p 00000000 00:00 0          [heap]
00823000-00847000 r-xp 00000000 08:01 921756     /lib/libm-2.12.1.so
00847000-00848000 r--p 00023000 08:01 921756     /lib/libm-2.12.1.so
00848000-00849000 rw-p 00024000 08:01 921756     /lib/libm-2.12.1.so
00974000-00990000 r-xp 00000000 08:01 921741     /lib/ld-2.12.1.so
00990000-00991000 r--p 0001b000 08:01 921741     /lib/ld-2.12.1.so
00991000-00992000 rw-p 0001c000 08:01 921741     /lib/ld-2.12.1.so
00b58000-00b59000 r-xp 00000000 00:00 0          [vdso]
00d2f000-00d42000 r-xp 00000000 08:01 917700     /lib/libz.so.1.2.3.4
00d42000-00d43000 r--p 00012000 08:01 917700     /lib/libz.so.1.2.3.4
00d43000-00d44000 rw-p 00013000 08:01 917700     /lib/libz.so.1.2.3.4
00d82000-00ea4000 r-xp 00000000 08:01 1050096    /usr/lib/libxml2.so.2.7.7
00ea4000-00ea8000 r--p 00121000 08:01 1050096    /usr/lib/libxml2.so.2.7.7
00ea8000-00ea9000 rw-p 00125000 08:01 1050096    /usr/lib/libxml2.so.2.7.7
00ea9000-00eaa000 rw-p 00000000 00:00 0          [heap]
00f01000-00f1d000 r-xp 00000000 08:01 921741     /lib/ld-2.12.1.so
00f1d000-00f1e000 r--p 0001b000 08:01 921741     /lib/ld-2.12.1.so
00f1e000-00f1f000 rw-p 0001c000 08:01 921741     /lib/ld-2.12.1.so
00fae000-00faf000 rwxp 00000000 00:00 0          [heap]
08048000-08049000 r-xp 00000000 08:01 919388     /home/airness/Desktop/test
08049000-0804a000 r--p 00000000 08:01 919388     /home/airness/Desktop/test
0804a000-0804b000 rw-p 00001000 08:01 919388     /home/airness/Desktop/test
0804b000-0804c000 r--p 00002000 08:01 541910     /home/airness/Desktop/Code/Memexec/memexec
0804c000-0804d000 rw-p 00003000 08:01 541910     /home/airness/Desktop/Code/Memexec/memexec
09b00000-09b63000 rw-p 00000000 00:00 0          [heap]
b7713000-b7714000 rw-p 00000000 00:00 0
b7721000-b7723000 rw-p 00000000 00:00 0
b77c4000-b77c6000 rw-p 00000000 00:00 0
b77d2000-b77d5000 rw-p 00000000 00:00 0
bfc4a000-bfc6b000 rw-p 00000000 00:00 0          [stack]
bfdf2000-bfe13000 rw-p 00000000 00:00 0
```

Figure 12. Restored Mappings for the Dynamically Compiled `test` Process

5.2.3.2 Results for Static Compilation:

When a program is built statically, the needed libraries are merged into the binary which thus becomes a standalone executable. The resulting binary is obviously larger than its dynamic counterpart, but the dynamic loading mechanism is removed. Thus, the

memory mappings do not include shared libraries anymore (see Fig. 13 for the mappings of a `test` process run after a static compilation).

```
airness  21526 93.8  0.0    824     4 pts/0    R+   16:37   0:06 ./test
airness  21528  0.0  0.2   4592  1092 pts/1    R+   16:38   0:00 ps aux
airness@ubuntu:~/Desktop$ cat /proc/21526/maps
009ea000-009eb000 r-xp 00000000 00:00 0          [vdso]
08048000-080cd000 r-xp 00000000 08:01 917600     /home/airness/Desktop/test
080cd000-080cf000 rw-p 00084000 08:01 917600     /home/airness/Desktop/test
080cf000-080d1000 rw-p 00000000 00:00 0
099b9000-099db000 rw-p 00000000 00:00 0          [heap]
bf9a2000-bf9c3000 rw-p 00000000 00:00 0          [stack]
```

Figure 13. Original Mappings for the Statically Compiled `test` Process

With a statically compiled `test` process, the resumption procedure worked correctly. Indeed, all the instructions following the nested loop were successfully executed: the file "*`sample.txt`*" was created and the string "`Done.`" written into the file.

Similarly, for the `test2` example, the resumption worked and the correct value (`50044`) stored in the `value` variable was also printed into the output file, demonstrating that program data were successfully preserved the procedure.

The only hiatus happened when the task was being terminated. The following errors were displayed for the `test` process (a similar message was output for the `test2` task):

```
*** glibc detected *** ./test: free(): invalid pointer: 0x080cee20 ***
Segmentation fault
```

This error message is usually thrown when an unallocated resource is being deleted, suggesting that this might be a consequence to the data of the `memexec` process still remaining in memory.

5.2.4. Limitations of `Memexec`

Although `memexec` illustrates that it is possible to revive a process extracted from an image of memory, there are restrictions, unsupported features and unexplored possibilities still associated with this work. Some of them are listed below:

- *Dealing with dynamic loading*: The fact that `memexec` fails to execute system calls at the resumption phase when the program is dynamically compiled is an important shortcoming. Investigators will be faced, more often than not, with dynamically-linked malware which are bound to make system calls to access resources.
- *Deleting unused data from the `memexec` process*: As mentioned in Section 5.2, the memory mappings of the process being modified by `memexec` are not all expunged and their effects on the final result are unknown. This may be problematic to a forensic investigation, as the evidence discovered using our method might be rejected in court.
- *Unsupported process artifacts*: A number of process information is not restored the tool such as open files, pipes and sockets, process ID, user credentials and child processes data.
- *Restrictions on the host system*: The requirements that the host machine should have, as described in Section 5.1, do not give much flexibility to the investigator. For example, the `memexec` tool does not provide an alternative to restoring the memory mappings when the Filesystem is not available, even though their contents are saved in RAM.

6. CONCLUSION

In this thesis, we demonstrated that given an image of the RAM of a Linux machine, it is possible to extract enough information about a specific running process to be able to resume its execution on another environment. We also presented two proof-of-concept tools `gettsk` and `memexec` that respectively retrieve process artifacts from the memory image and use the saved data to revive the program.

Obtaining the memory image of a running machine in the first place is not easy. Although researchers developed both hardware and software methods to acquire the RAM, the constantly-changing nature of non-volatile memory pose a challenge to investigators. However, most virtualization software allows users to take a snapshot of a virtual machine and collect the contents of the memory from a file on disk. We took advantage of this feature to create memory images of Linux systems that we used to perform our experiments.

When the memory image is available, the first step involves identifying which of the tasks running on the machine, at the moment of capture, is of interest. This is done by parsing the RAM for the circular doubly-linked list of process descriptors maintained by the Linux kernel (from which all the other process-related kernel structures can be obtained as described in Chapter 3), in order to get the list of all the tasks loaded in memory. At this point, accessing the data is a question of determining the offsets of specific fields in the kernel data structures. The `gettsk` tool proceeds to store the obtained information in a file using the flexible and widely used XML format.

To be able to resume the process extracted from the RAM, the receiving system should be prepared adequately (see Section 5.1): the operating system, the underlying hardware as well as other resources such as the filesystem (especially those used by the task) should be similar to their equivalent in the original machine, to ensure a proper continuation of the execution of the task. It is also an imperative that the host be separate from production systems, so that potential infection is contained.

The actual resumption procedure can be performed in Linux either form User Space or Kernel Space, but making changes while in kernel mode allows direct access to the data structures that need to be restored for the execution to continue. That's the reason why the `memexec` toolset includes a loadable kernel module (`memlkm`) which does the actual work of placing the saved process' artifacts in their appropriate locations.

Although our experiment worked for our proof-of-concept configurations, both tools `gettsk` and `memexec` fail to consider some process-related data that could come in handy for an investigation such as open files and the ability to complete the resumption when the initial process is dynamically compiled (see Sections 4.4 and 5.2.4). This work is designed for a single process at the time and thus Inter Process Communications as well as network connections are not supported. Moreover, the implementation is highly dependent on the version of the kernel used during the experiment (Linux kernel 2.6.35) and will not function correctly on different platforms.

Future Work:

Our work described how a number of process-specific information can be retrieved from a memory image and how they can be used to continue the execution of

the task on another host. This will increase the chances for an investigator to discover and/or consolidate evidence, from a forensic analysis of the contents of the RAM. But as the previous paragraph suggests, there is still room for improvement, including the following:

- *Dynamically compiled binaries and system calls*: As discussed in Sections 5.2.3.1 and 5.2.4, our proposed solution does not work for dynamically compiled programs, probably due to the remnants of the process being modified during the resumption phase. This issue might be solved by restoring the information of the extracted process using a task created from scratch. It is worth noting that failures specifically occur when system calls are being executed, which suggest discrepancies with the process' libraries and/or the state of the kernel (which is not extracted by `gettsk`).
- *Extract and restore the files opened by the process*: Regular files, network sockets and pipes (see Section 4.4). This category can also include providing alternatives to recovering memory mappings. Our proposal requires that the files mapped in memory be present at the same location in the host system, but this might be difficult to fulfill, especially for included libraries. Since their contents are actually stored in RAM, they could be harnessed at the extraction phase to alleviate the requirement.
- *Extend support to groups of tasks, their relationships and transactions between them*: Neither child, sibling, nor parent processes are taken into account in our project, although they might be of use to a forensic investigation. Extracting from memory and resuming the execution of a group

of running processes and their relationships is thus a desirable feature that could be pursued in the future. Similarly, if the tasks of interest are involved in inter-process communications, they should be captured and revived with their transactions preserved.

- *Extend support to different kernel versions and different platforms*: Both tools introduced in this document are very kernel-dependent, primarily due to differences in data structures between versions of the Linux Kernel. A first step towards solving this issue could be automating the determination of fields' offsets in kernel structures, as discussed in Section 4.4. A more long term solution could involve a method used by Volatility [50], which allows the users to create and add profiles for specific kernel versions, thus providing more flexibility and control to investigators.

APPENDIX A: OFFSETS AND OTHER CONFIGURATIONS

The following Python declarations (*config.py*) represent the offsets of elements in the kernel structures used during the experimental phase of this project. It also includes hard coded configurations such as PAGE_OFFSET and the address of `init_task` found in the System.map symbol table.

```
******************** CONFIG.PY  ******************************

init_task     = 0xc07c76e0   # "Swapper" address obtained from System.map
kernel_start  = 0xc0000000     # PAGE_OFFSET
page_size     = 0x1000          # Size of a Page: 4KB

page_mask     = 0x00000fff

regs_size     = 68           # size of register set
sig_action_size     = 1280         # size of signal handler structure


"""
***** Task_struct Offsets: see include/linux/sched.h ********
"""

state_offset  = 0
stack_offset  = 8
flags_offset  = 24

task_offset   = 432
comm_offset   = 752
pid_offset    = 508
mm_offset     = 460
parent_offset       = 524

cred_offset   = 724
sp0_offset    = 808
fs_offset     = 908
files_offset  = 912
sighand_offset    = 924
sigpending_offset = 952
sigblocked_offset = 928


"""
******  mm_struct Offsets: see include/linux/mm_types.h     *********
"""

mmap_offset   = 0
pgd_offset    = 40

vmstruct_offset   = 124          # Offset for memory mappings of vm structures
                           # such as start_code, start_data, start_stack, etc...

"""
*****  vm_area_struct Offsets : see include/linux/mm_types.h  *****
```

```
"""

vm_mm_offset  = 0
vm_start_offset     = 4
vm_end_offset       = 8
vm_next_offset      = 12
vm_prev_offset      = 16
vm_pgprot_offset    = 20
vm_pgoff_offset     = 72
vm_flags_offset     = 20
vm_file_offset      = 76

"""
  Files_struct, fdtable, file, dentry and inode Offsets:
    see: include/linux/fdtable.h
            include/linux/fs.h
            include/linux/dcache.h
"""

fdt_offset    = 4           # File Table offset
fd_offset     = 4           # Array of file descritpor (fd) offset

dentry_offset       = 12

d_parent_offset     = 28          # Dentry's parent offset
name_len_offset     = 36
d_name_offset       = 40          # Dentry's Name offset

********************** END CONFIG.PY  **************************
```

APPENDIX B: LISTING RUNNING PROCESSES

Source code in Python of *ps.py* that lists the running processes from a Linux kernel 2.6.35 memory dump.

```
#!/usr/bin/python
import os
import sys
import struct
from config import *             # See Appendix A: config.py

def usage(arg):
        print ""
        print " Usage: %s <memory_file>" % (os.path.basename(arg))
        print ""

def printline(addr, f):                      # Print process info
        current = addr + pid_offset
        f.seek(current)
        ret, = struct.unpack('<L', f.read(4))
        print '%4d'.ljust(10) % ret ,

        current = addr + parent_offset
        f.seek(current)
        ret, = struct.unpack('<L', f.read(4))
        current = (ret - kernel_start) + pid_offset
        f.seek(current)
        ret, = struct.unpack('<L', f.read(4))
        print '%4d'.ljust(10) % ret ,

        if (debug):
                current = addr + cred_offset
                f.seek(current)
                current, = (struct.unpack('<L', f.read(4)))
                current = current - kernel_start + 4
                f.seek(current)
                print '%4d'.ljust(10) % (struct.unpack('<L', f.read(4))),
                print '0x%8x'.ljust(15) % (addr + kernel_start),

        current = addr + comm_offset
        f.seek(current)
        print '%16s'.ljust(25) % (f.read(16))

################################ MAIN ################################
debug = 0
line = ""

if (len(sys.argv) != 2):
        usage(sys.argv[0])
        sys.exit(0)

fields = "PID ".ljust(15) + "PPID".ljust(10)        # print headings
if (debug):
        fields += "PID".ljust(10)
        fields += "ADDRESS(Hex)".ljust(20)

fields += "NAME".ljust(25)

print fields

init = init_task - kernel_start
dumpfile = open(sys.argv[1], "rb")

current_task = init
printline(current_task, dumpfile)
dumpfile.seek(current_task + task_offset)
val, = struct.unpack('<L', dumpfile.read(4))
current_task = (val - kernel_start) - task_offset

while (current_task != init):
        printline(current_task, dumpfile)
        dumpfile.seek(current_task + task_offset)
        val, = struct.unpack('<L', dumpfile.read(4))
        current_task = (val - kernel_start) - task_offset

dumpfile.close()
```

APPENDIX C: GETTSK SOURCE CODE

This is the source code for the `gettsk` tool, made up of three files (the two files `structures.py` and `gettsk.py` printed here, plus `config.py` in Appendix A).

```
******************** STRUCTURES.PY ******************************

import os
import sys
import struct

from config import *
# from gettsk import *

class Structure:          # Collection of items to simulate kernel structures
        def __init__(self, **kwds):
                self.__dict__.update(kwds)

        def __str__(self):
                state = ["%s = 0x%08x" % (att, value)
                        for (att, value)
                        in self.__dict__.items()]

                return '\n'.join(state)


class AddresSpace:                        # mm_struct in Unix

        def __init__(self, address):

                self.base_addr = address - kernel_start
                self.mmap_addr = self.base_addr + mmap_offset
                self.pgd_addr  = self.base_addr + pgd_offset

        def getVmStruct(self, f): # Get vm structure for each Process Address Space

                f.seek(self.base_addr + vmstruct_offset)

                start_code, = struct.unpack('<L', f.read(4))
                end_code, = struct.unpack('<L', f.read(4))
                start_data, = struct.unpack('<L', f.read(4))
                end_data, = struct.unpack('<L', f.read(4))
                start_brk, = struct.unpack('<L', f.read(4))
                brk, = struct.unpack('<L', f.read(4))
                start_stack, = struct.unpack('<L', f.read(4))
                arg_start, = struct.unpack('<L', f.read(4))
                arg_end, = struct.unpack('<L', f.read(4))
                env_start, = struct.unpack('<L', f.read(4))
                env_end, = struct.unpack('<L', f.read(4))

                vmstruct = Structure(start_code = start_code, end_code = end_code,
                                     start_data = start_data, end_data = end_data,
                                     start_brk = start_brk, brk = brk,
                                     arg_start = arg_start, arg_end = arg_end,
                                     start_stack = start_stack,
                                     env_start = env_start, env_end = env_end)

                return vmstruct


        def getAllVmAreas(self, f):               # A list of all Memory Regions

                seglist = []
                f.seek(self.mmap_addr)
                address, = struct.unpack('<L', f.read(4))
                vm = VmArea(address)
                check_addr = address - kernel_start
                vmarea = vm.getVmArea(f)
                stop = True
```

```
                while stop or vmarea.addr != check_addr :
                        seglist.append(vmarea)
                        vm = vm.getNext(f)
                        if vm == None :
                                break
                        vmarea = vm.getVmArea(f)
                        stop = False

                return seglist

        def getPgd(self, f):

                pgd_addr = self.pgd_addr
                f.seek(pgd_addr)
                pgd, = struct.unpack('<L', f.read(4))

                return pgd


class VmArea:                                   # vm_area_struct in Linux

        def __init__(self, address):

                self.base_addr = address - kernel_start
                self.next = None
                self.prev = None

        def getNext(self, f):                           # Get next VmArea : vm_next

                f.seek(self.base_addr + vm_next_offset)
                addr, = struct.unpack('<L', f.read(4))

                if addr == 0:
                        return None

                self.next = VmArea(addr)
                return self.next


        def getVmArea(self, f):                                 # Get a memory region

                f.seek(self.base_addr + vm_start_offset)
                vm_start, = struct.unpack('<L', f.read(4))
                vm_end, = struct.unpack('<L', f.read(4))

                f.seek(self.base_addr + vm_pgprot_offset)
                vm_pgprot, = struct.unpack('<L', f.read(4))
                vm_flags, = struct.unpack('<L', f.read(4))

                f.seek(self.base_addr + vm_pgoff_offset)
                vm_pgoff, = struct.unpack('<L', f.read(4))
                vm_file, = struct.unpack('<L', f.read(4))  # file struct address or None

                vmarea = Structure(addr = self.base_addr, vm_start = vm_start,
                                   vm_end = vm_end,
                                   vm_pgprot=vm_pgprot, vm_flags=vm_flags,
                                   vm_pgoff = vm_pgoff,
                                   vm_file=vm_file)

                return vmarea


class File:

        def __init__(self, address):

                self.base_address = address - kernel_start
                self.start = None
                self.end = None

        def getFilePath(self, f):         # Get File path and Name from File Descriptor

                filename = ""

                f.seek(self.base_address + dentry_offset)
                dentry, = struct.unpack('<L', f.read(4))
                dentry -= kernel_start

                f.seek(dentry + d_parent_offset)
                parent, = struct.unpack('<L', f.read(4))
```

```
                f.seek(dentry + name_len_offset)
                size, = struct.unpack('<L', f.read(4))
                name_addr, = struct.unpack('<L', f.read(4))
                name_addr -= kernel_start
                f.seek(name_addr)
                name = f.read(size)

                if name == "" :
                        return ""

                while name != "/" and name != "anon_inode:":

                        if filename == "" :
                                filename = name
                        else:
                                filename = name + '/' + filename

                        if len(filename) > 100:
                                break
                        dentry = parent - kernel_start
                        f.seek(dentry + d_parent_offset)
                        parent, = struct.unpack('<L', f.read(4))

                        f.seek(dentry + name_len_offset)
                        size, = struct.unpack('<L', f.read(4))
                        name_addr, = struct.unpack('<L', f.read(4))
                        name_addr -= kernel_start
                        f.seek(name_addr)
                        name = f.read(size)

                        if name == "" or name == "anon_inode:":
                                return filename

                filename = '/' + filename

                return filename

        def getFilevmAreas(self, seglist):                 # Return vm_areas of a file from list of
all vm_Areas

                l = []
                for vmarea in seglist :
                        if vmarea.vm_file == (self.base_address + kernel_start) :
                                l.append(vmarea)
                                if self.start == None :
                                        self.start = vmarea.vm_start
                                if self.end == None :
                                        self.end = vm_area.vm_end

                                if self.start + vmarea.vm_pgoff * page_size > self.end :
                                        self.end = vmarea.vm_end

                return l
```

********** END STRUCTURES.PY **********

********** GETTSK.PY **********

```
#!/usr/bin/python

import os
import sys
import struct
import getopt
import base64                                       # For data encoding
from xml.dom.minidom import Document

from config import *
from structures import *


def usage(arg):

        print " Usage: %s -f <memory_file> [-o <output_file>] [-v] -p <pid>" % (os.path.basename(arg))
        print ""
        print "       -f <memory_file> :  Memory image file."
        print "       -o <output_file> :  XML output file."
```

```
        print "       -v              :  Verbose."
        print "       -p <pid>        :  PID of process to extract."
        print ""


def check_options(argv):          # Verify Command Line Options

        v = 0                             # verbose
        p = 0                             # pid
        f = ""                            # Memory Image File
        out = "output.xml"                # Output File


        o = "hvf:o:p:"
        try:

                options = getopt.getopt(argv[1:], o)
                opts = dict(options[0])
                extra = options[1]

                if ((len(opts) == 0) or ("-h" in opts)):
                        usage(argv[0])
                        sys.exit(0)

                if ("-v" in opts):
                        v = 1

                if (not "-f" in opts):
                        raise Exception("Memory Image File not specified, use -h for help!")
                f = opts["-f"]

                if (not "-p" in opts):
                        raise Exception("PID not specified, use -h for help!")
                p = int(opts["-p"])

                if ("-o" in opts):
                        if os.path.exists(opts["-o"]):
                                raise Exception("The output filename already exists!")
                        out = opts["-o"]

        except getopt.GetoptError, error:
                print " Error: " + str(error)
                usage(argv[0])
                sys.exit(0)

        except Exception, ex:
                print " Error: " + str(ex)
                sys.exit(0)

        return (v, f, p, out)


def taskByPid(pid, f):              # Returns task_struct address from PID

        init = init_task - kernel_start
        current_pid = 0

        current_task = init

        f.seek(current_task + task_offset)
        val, = struct.unpack('<L', f.read(4))
        current_task = (val - kernel_start) - task_offset
        f.seek(current_task + pid_offset)
        current_pid, = struct.unpack('<i', f.read(4))

        while (current_task != init and current_pid != pid):

                f.seek(current_task + task_offset)
                val, = struct.unpack('<L', f.read(4))
                current_task = (val - kernel_start) - task_offset
                f.seek(current_task + pid_offset)
                current_pid, = struct.unpack('<i', f.read(4))

        if (current_task == init):
                return None
        return current_task


def getCred(addr, f):                     # Retrieve process' credentials;
```

```
                                                                # addr = cred address from
task_struct
        addr = addr - kernel_start
        f.seek(addr + 4)
        uid, = struct.unpack('<i', f.read(4))
        gid, = struct.unpack('<i', f.read(4))
        suid, = struct.unpack('<i', f.read(4))
        sgid, = struct.unpack('<i', f.read(4))
        euid, = struct.unpack('<i', f.read(4))
        egid, = struct.unpack('<i', f.read(4))
        fsuid, = struct.unpack('<i', f.read(4))
        fsgid, = struct.unpack('<i', f.read(4))

        cred = Structure (uid=uid, gid=gid, suid=suid, sgid=sgid, euid=euid,
                          egid=egid, fsuid=fsuid, fsgid=fsgid)

        return cred


def getOpenFiles(addr, f):                          # Returns list of open files from file table

        l = []
        open_files = []
        filename = ""

        addr -= kernel_start
        f.seek(addr + fdt_offset)
        addr, = struct.unpack('<L', f.read(4))              # Pointer to File table

        addr -= kernel_start
        f.seek(addr + fd_offset)
        addr, = struct.unpack('<L', f.read(4))     # Pointer to array of fds

        addr -= kernel_start
        f.seek(addr)
        addr, = struct.unpack('<L', f.read(4))

        while addr != 0 :
                l.append(addr)
                addr, = struct.unpack('<L', f.read(4))

        for addr in l :
                """
                f.seek(addr + dentry_offset)
                dentry, = struct.unpack('<L', f.read(4))
                dentry -= kernel_start
                """
                file = File(addr)
                filename = file.getFilePath(f)

                if filename == "" :
                        continue

                # print filename
                open_files.append(filename)

        return open_files


def getKernelModeStack(addr, f):                    # Get Top of stack containing process CPU registers

        addr -= kernel_start - 4                            # -4 to include top of stack
        f.seek (addr - regs_size)
        buf = f.read(regs_size)
        # print buf
        return buf


def VirtToPhysical(addr, f, pgd) :                  # Translate from user mode Virtual to Physical
Address.
                                                            #
                                                            # Returns the triplet (Paget table flags,
Type of page entry, Physical address of frame if any).
                                                            #
                                                            # The type of page table is:
                                                            #       - 0 for a page directory entry
                                                            #       - 1 for a page table entry
                                                            #
                                                            # This is used to rebuild the page tables
correctly.
```

```
        ret = ((pgd >> 12) << 12) + (addr >> 22)*4
        f.seek(ret - kernel_start)
        pgd_entry, =  struct.unpack('<L', f.read(4))

        if (pgd_entry & 0b1 != 0b1):
                #print "PGD_ENTRY present flag is 0: Page frame not in memory ",
                #print hex(pgd_entry)
                return (pgd_entry & 0xfff, 0, 0)

        ret = ((pgd_entry >> 12) << 12) + ((addr >> 12) & (1 << 10)-1)*4
        f.seek(ret)
        pt_entry, =  struct.unpack('<L', f.read(4))

        if (pt_entry & 0b1 != 0b1):                          # Not in Memory
                #print "PT_ENTRY present flag is 0: Page frame not in memory "
                #print hex(pt_entry)
                return (pt_entry & 0xfff, 1, 0)


        final = (((pt_entry >> 12) << 12) + (addr & (1 << 12) - 1))

        return (pt_entry & 0xfff, 1,final)



def dumpVmArea(vmarea, f, pgd):                              # Dump the contents of a memory region,
NOT USED NOW!!!

        addr = vmarea.vm_start
        dump = ""
        i = 0
        while addr < vmarea.vm_end :

                (flags, page_type, res) = VirtToPhysical(addr, f, pgd)
                if (not res):
                        #print "Page could not be found: start = 0x%08x , end = 0x%08x" %
(vmarea.vm_start, addr)
                        #print ""
                        addr += page_size
                        continue

                else :
                        if (res & page_mask) :
                                print " Oops! Page address 0x%08x not aligned" % addr
                                print ""

                        f.seek(res)
                        dump += f.read(page_size)

                addr += page_size
                i += 1

        #print i
        return (dump, i)



def buildXml(task, output, f):                               #  Build XML file from Process Structure


        out = open (output, "w")
        dirs = 'tsk_files/'
        regstring = 'region'

        if dump_map:

                if not os.path.isdir("./" + dirs):
                        os.makedirs(dirs)
                if not os.path.isdir("./" + dirs):
                        print "Cannot create \"tsk_files\" directory!! All memory segment related
files will be in current directory!"
                        #dirs = './'

        doc = Document()

        proc = doc.createElement("process")
        proc.setAttribute("magic_number", MAGIC)
        proc.setAttribute("kernel_version", "2.6.35")
        proc.setAttribute("arch", "x86")
        doc.appendChild(proc)
```

```
old_pid = doc.createElement("old_pid")
proc.appendChild(old_pid)
pid_t = doc.createTextNode(str(task.pid))
old_pid.appendChild(pid_t)

name = doc.createElement("name")
proc.appendChild(name)
comm = doc.createTextNode(task.name)
name.appendChild(comm)

state = doc.createElement("state")
proc.appendChild(state)
state_t = doc.createTextNode(str(task.state))
state.appendChild(state_t)

flags = doc.createElement("flags")
flags.setAttribute("value", "0x%08x" % (task.flags))
proc.appendChild(flags)

cred = doc.createElement("credentials")
proc.appendChild(cred)
for (attr, value) in task.cred.__dict__.items():
        cred.setAttribute(attr, "%d" % value)


buf = getKernelModeStack(task.kmstack, f)

#print "%s" %buf

buf = base64.standard_b64encode(buf)
reg_set = doc.createElement("reg_set")
proc.appendChild(reg_set)
reg_set.setAttribute("encoding", "binary-base64")
reg_stack = doc.createTextNode(buf)
reg_set.appendChild(reg_stack)


vmstruct = doc.createElement("vmstruct")
proc.appendChild(vmstruct)
for (attr, value) in task.vmstruct.__dict__.items():
        vmstruct.setAttribute(attr, "0x%08x" % value)


############################ seglist   ########################
segments = doc.createElement("segments")
segments.setAttribute("number", "%d" % (len(task.seglist)))
sp = task.sp_addr - kernel_start

count = 0

for vmarea in task.seglist :

        pgoff = 0                # Offset of page in file representing a region.
        #name = 'vm%d'%count

        if dump_map:
                regfilename = '%s%s%d'%(dirs, regstring, count)
                reg_file = open (regfilename, "w")
                #print "%s" %name

        region = doc.createElement("vm_region")
        region.setAttribute("vm_start", "0x%08x" % vmarea.vm_start)
        region.setAttribute("vm_end", "0x%08x" % vmarea.vm_end)
        region.setAttribute("vm_pgprot", "0x%08x" % vmarea.vm_pgprot)
        region.setAttribute("vm_flags", "0x%08x" % vmarea.vm_flags)
        region.setAttribute("vm_pgoff", "0x%08x" % vmarea.vm_pgoff)
        if dump_map:
                region.setAttribute("local_file", "%s" %regfilename)

        #(dmp, num) = dumpVmArea(vmarea, f, task.pgd)


        addr = vmarea.vm_start
        num = 0

        if vmarea.vm_file != 0 :
                file = File(vmarea.vm_file)
                filename = file.getFilePath(f)
                region.setAttribute("filename", filename)

        while addr < vmarea.vm_end :   # For each frame, Present in memory or not!
```

```
                        (flgs, ptype, res) = VirtToPhysical(addr, f, pgd)

                        if (not res):
                                #print "Page not present in memory: start = 0x%08x , end = 0x%08x" %
(addr, addr+page_size)
                                present = 0
                                addr += page_size
                                continue

                        #fname = name + '_frame%d'%num

                        frame = doc.createElement("vm_frame")

                        frame.setAttribute("vm_start", "0x%08x" % addr)
                        frame.setAttribute("vm_end", "0x%08x" % (addr + page_size))
                        frame.setAttribute("flags", "0x%08x" % flgs)
                        frame.setAttribute("offset", "0x%08x" % pgoff)

                        #if (ptype):
                        #       frame.setAttribute("entry_type", "pte")
                        #else:
                        #       frame.setAttribute("entry_type", "pgd")

                        present = 1
                        if (res & page_mask) :
                                print " Oops! Page address 0x%08x not aligned!" % addr

                        f.seek(res)
                        dmp = f.read(page_size)
                        if dump_map:
                                reg_file.seek(pgoff)
                                reg_file.write(dmp)

                        if vmarea.vm_file == 0 :
                                dmp = base64.standard_b64encode(dmp)
                                frame.setAttribute("encoding", "binary-base64")
                                mem_contents = doc.createTextNode(dmp)
                                frame.appendChild(mem_contents)

                        region.appendChild(frame)
                        addr += page_size
                        pgoff += page_size
                        num += 1

                region.setAttribute("num_pages", "%d" % num)
                segments.appendChild(region)
                count += 1


                # break

        proc.appendChild(segments)

        ############################## end seglist  ########################

        open_files = doc.createElement("open_files")
        open_files.setAttribute("Number", "%d" % (len(task.open_files)))
        proc.appendChild(open_files)

        for p in task.open_files:
                var = doc.createElement("file")
                var.setAttribute("pathname", p)
                open_files.appendChild(var)

        signals = doc.createElement("signals")
        proc.appendChild(signals)

        sig_act = doc.createElement("sig_actions")
        sig_act.setAttribute("encoding", "binary-base64")
        signals.appendChild(sig_act)

        sig_hand_addr = task.sig_actions - kernel_start
        f.seek(sig_hand_addr)
        buf = f.read(sig_action_size)
        buf = base64.standard_b64encode(buf)

        sighand = doc.createTextNode(buf)
        sig_act.appendChild(sighand)
```

```
        blocked = doc.createElement("sig_blocked")
        signals.appendChild(blocked)
        blocked.setAttribute("value", "0x%016x" % (task.sig_blocked))

        pending = doc.createElement("sig_pending")
        signals.appendChild(pending)
        pending.setAttribute("value", "0x%016x" % (task.sig_pending))

        xml = doc.toprettyxml(indent="    ")
        out.write(xml)
        out.close()

        return 0



        ########################################################
        #                ***** MAIN ********                    #
        ########################################################

print ""
print " Gettsk version 0.1 "
print " By Ernest Mougoue <mougoued@dukes.jmu.edu>"
print ""

(verbose, fileString, pid, outString) = check_options(sys.argv)


# print "Test: pid = %d, file = %s, out = %s, verb = %d " % (pid, fileString, outString, verbose)

dump_map = 0                    # dump memory mappings to external files?

dumpfile = open(fileString, "rb")

if (pid == 0):
        print "Cannot extract \"swapper\", Please choose another process!"
        print ""
        sys.exit(0)

task_addr = taskByPid(pid, dumpfile)                    # Get the address of task_struct

if (task_addr == None):
        print "There is no process with PID: %d, Please choose another process!" % pid
        print ""
        sys.exit(0)

dumpfile.seek(task_addr + comm_offset)                          # Get Process' Name
i = 16
comm = ""
read = dumpfile.read(1)
while (i > 0) and (read != "\x00"):
        comm += read
        read = dumpfile.read(1)
        i -= 1

if (verbose):
        print ""
        print "  The chosen Process is:"
        print "  PID: %d" % pid
        print "  Name: %s @ 0x%08x" % (comm, task_addr)
        print "  Output File : %s" % outString
        print ""

if (verbose):
        print "  + Getting Process' State....."

dumpfile.seek(task_addr + state_offset)
state, = struct.unpack('<L', dumpfile.read(4))              # Get Process' Running State
# print "State = %02d" % state



if (verbose):
        print "  + Getting Process' Flags....."

dumpfile.seek(task_addr + flags_offset)
flags, = struct.unpack('<L', dumpfile.read(4))              # Get Process' flags
# print "Flags = 0x%08x" % flags
```

```
if (verbose):
        print "  + Getting Process' Memory Mappings....."

dumpfile.seek(task_addr + mm_offset)

mm_addr, = struct.unpack('<L', dumpfile.read(4))    # Get Process' mm_struct Base Address
if mm_addr < kernel_start :
        print "This Process has an invalid memory map!!!"
        print "Exiting ......"
        print ""
        sys.exit(0)

mm_struct = AddressSpace(mm_addr)
vmstruct = mm_struct.getVmStruct(dumpfile)          # Get Memory Map Structure of process
pgd = mm_struct.getPgd(dumpfile)                        # Get Process' PGD

#print hex(pgd)

seglist = mm_struct.getAllVmAreas(dumpfile)                 # Get a list of vm_area_structs



if (verbose):
        print "  + Getting Process' Credentials....."
dumpfile.seek(task_addr + cred_offset)
cred_addr, = struct.unpack('<L', dumpfile.read(4))  # Get Process' Credentials Base Address
cred = getCred(cred_addr, dumpfile)
# print cred


if (verbose):
        print "  + Getting Process' Kernel Mode Stack....."
dumpfile.seek(task_addr + sp0_offset)
kmstack_addr, = struct.unpack('<L', dumpfile.read(4))       # Get Process' Kernel Mode Stack Base
Address

if (verbose):
        print "  + Kernel Mode Stack Pointer:    0x%08x" %kmstack_addr

sp_addr, = struct.unpack('<L', dumpfile.read(4))    # Get Process' Stack Pointer

if (verbose):
        print "  + Stack Pointer:    0x%08x" %sp_addr

ip_addr, = struct.unpack('<L', dumpfile.read(4))
ip_addr, = struct.unpack('<L', dumpfile.read(4))

dumpfile.seek(ip_addr - kernel_start)
ip_addr, = struct.unpack('<L', dumpfile.read(4))


#print hex(sp_addr)
#print hex(kmstack_addr)

# dumpfile.seek(task_addr + fs_offset)
# fs_addr, = struct.unpack('<L', dumpfile.read(4))  # Get Process' fs_struct Base Address



if (verbose):
        print "  + Getting Process' Open Files....."
dumpfile.seek(task_addr + files_offset)
files_addr, = struct.unpack('<L', dumpfile.read(4)) # Get Process' files_struct Base Address
open_files = getOpenFiles(files_addr, dumpfile)     # Get a list of opened files
open_files = open_files[3:]                         # The first 3 are stdin, stdout and stderr



if (verbose):
        print "  + Getting Process' Signals....."

dumpfile.seek(task_addr + sighand_offset)
sighand_addr, = struct.unpack('<L', dumpfile.read(4))       # Get Process' Signal Handlers address
sighand_addr += 4                                   # Address of Signal Actions
# sighand = getSigHand(sighand_addr, dumpfile)


dumpfile.seek(task_addr + sigblocked_offset)
blocked, = struct.unpack('<d', dumpfile.read(8))    # Blocked signals
#  print "0x%016x" % blocked
```

```
dumpfile.seek(task_addr + sigpending_offset + 8)
pending, = struct.unpack('<d', dumpfile.read(8))    # Get Process' pending signals



# -----All the infos about a Process in a structure--------------------

process = Structure(task_addr = task_addr, pid = pid, name = comm, state = state,
                         flags = flags,seglist = seglist, cred = cred, vmstruct = vmstruct, kmstack =
                         kmstack_addr,
                         open_files = open_files, sp_addr = sp_addr, pgd = pgd,
                         sig_actions = sighand_addr, sig_blocked = blocked,
                         sig_pending = pending)



if (verbose):
         print "  + Building XML File....."
done = buildXml(process, outString, dumpfile)                          # Build XML File

if (done):
         print "An error Occured while saving the process!!!"
         dumpfile.close()
         sys.exit(0)

dumpfile.close()

print "Done. "
print " "
```

APPENDIX D: XML SCHEMA FOR GETTSK'S OUTPUT

```
<? xml version = "1.0"?>
<xs: element name="process">
    <xs:complexType>
        <xs:sequence>
          <xs:element name="pid" type="xs:integer"/>
          <xs: element name="name" type="xs:string"/>
          <xs:element name="state" type="xs:integer"/>
          <xs:element name="flags" type="xs:integer"/>
          </xs:element>
          <xs:element name="credentials">
             <xs:attribute name="uid" type="xs:integer"/>
             <xs:attribute name="gid" type="xs:integer"/>
             <xs:attribute name="euid" type="xs:integer"/>
             <xs:attribute name="egid" type="xs:integer"/>
               <xs:attribute name="fsuid" type="xs:integer"/>
             <xs:attribute name="fsgid" type="xs:integer"/>
             <xs:attribute name="suid" type="xs:integer"/>
             <xs:attribute name="sgid" type="xs:integer"/>
          </xs:element>
          <xs:element name="reg_set" type="xs:string">
             <xs:attribute name="encoding" type="xs:string" default="binary-base64"/>
          </xs:element>
          <xs:element name="vmstruct">
             <xs:attribute name="start_code" type="xs:string"/>
             <xs:attribute name="end_code" type="xs:string"/>
             <xs:attribute name="start_data" type="xs:string"/>
             <xs:attribute name="end_data" type="xs:string"/>
             <xs:attribute name="start_brk" type="xs:string"/>
             <xs:attribute name="brk" type="xs:string"/>
             <xs:attribute name="arg_start" type="xs:string"/>
             <xs:attribute name="arg_end" type="xs:string"/>
             <xs:attribute name="start_stack" type="xs:string"/>
             <xs:attribute name="env_start" type="xs:string"/>
             <xs:attribute name="env_end" type="xs:string"/>
          </xs:element>
          <xs:element name="segments">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="vm_region" type"xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                        <xs:attribute name="encoding" type="xs:string" default="binary-base64"/>
                     <xs:attribute name="vm_start" type="xs:string"/>
                     <xs:attribute name="vm_end" type="xs:string"/>
                     <xs:attribute name="vm_pgprot" type="xs:string"/>
                     <xs:attribute name="vm_pgoff" type="xs:string"/>
                     <xs:attribute name="vm_flags" type="xs:string"/>
                     <xs:attribute name="num_pages" type="xs:integer"/>
```

```
                    <xs:attribute name="filename" type="xs:string" use="optional"/>
                  </xs:element>
                </xs:sequence>
                <xs:attribute name="number" type="xs:integer">
              </xs:complexType>
            </xs:element>
            <xs:element name="open_files">
              <xs:complexType>
                <xs:element name="file" minOccurs="0" maxOccurs="unbounded">
                  <xs:attribute name="pathname" type="xs:string"/>
                </xs:element>
                <xs:attribute name="Number" type="xs:integer">
              </xs:complexType>
            </xs:element>
            <xs:element name="signals">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="sig_actions" type="xs:strings">
                    <xs:attribute name="encoding" type="xs:string" default="binary-base64"/>
                  </xs:element>
                  <xs:element name="sig_blocked" type="xs:strings">
                    <xs:attribute name="value" type="xs:string"/>
                  </xs:element>
                  <xs:element name="sig_pending" type="xs:strings">
                    <xs:attribute name="value" type="xs:string"/>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:elements>
            <xs:attribute name="arch" type="xs:string" default="x86"/>
            <xs:attribute name="kernel" type="xs:string" default="2.6.35.8"/>
            <xs:attribute name="magic_number" type="xs:string" default="FFC0">
          </xs:sequence>
        </xs:complexType>
</xs:element>
```

APPENDIX E: MEMEXEC SOURCE CODE

This is the source code of the 3 files constituting the `memexec` tool: `memexec.c`, `memexec.h` and `memlkm.c`. Also included is the Makefile used to compile them.

```
******************** memexec.h ******************************
#include <asm/ptrace.h>
#include <linux/ipc.h>
#include <asm/param.h>

#define MEMPROC_SIZE sizeof(struct memproc)
#define MAX_FILENAME                    256
#define MAX_PATH                256
#define PAGES                           4096 // Page Size

/* IOCTL */

#define MAGIC_NUMBER 0xCC
#define IOCTL_RESTART                   _IOW(MAGIC_NUMBER, 1, struct memproc *)
#define MAX_IOC_NR 1
#define DEV_FILE    "/dev/memexec"

struct credentials {
        int uid, gid, suid, sgid, euid, egid, fsuid, fsgid;
};

struct registers {
        int size;
        char *regs;
};

struct signals {
        int sig_actions_size;
        char *sig_actions;
        unsigned long sig_blocked, sig_pending;
};

struct vmstruct {
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack, arg_start, arg_end;
        unsigned long env_start, env_end;
};

struct frames {
        unsigned long present;
        unsigned long flags;
        unsigned long offset;
        char entry_type[3];

        unsigned long vm_start, vm_end;
        char *contents;
};


struct segments {
        unsigned long vm_start, vm_end, prot, flags, pgoff;
        int num_pages;
        int vm_file;
        char local_file[MAX_PATH];              // Local file storing the contents of the segment.
        char filename[MAX_FILENAME];
        //char *contents;
        struct frames *framelist; /* Frames associated with each segment */
};
```

```
struct open_file {
        /* Not Fully supported - Now only shows open file's path */
        char * path;
};


struct memproc {
        unsigned long old_pid, state, flags;
        int seg_num;              /* Number of Segments */
        int file_num;       /* Number of Open Files */

        char name[16];
        struct registers reg_set;
        struct credentials memcred;
        struct vmstruct vm_map;
        struct segments *seglist;
        struct open_file *filelist;
        struct signals sig;
};

static inline void mem_strncpy(char * src, const char * dest, int size) {
  int i;
  for (i = 0; i < size; i++) {
    src[i] = dest[i];
  }
}
```

********** End memexec.h **********

********** memelkm.c **********

```
/* Standard in kernel modules */
#include <linux/smp.h>
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* For character devices */
#include <linux/fs.h>       /* The character device
                             * definitions are here */
#include <linux/cdev.h>
#include <linux/binfmts.h>
#include <asm/mman.h>
#include <asm/uaccess.h> /* for KERNEL_DS and get_fs() */
#include <linux/file.h>
#include <linux/sys.h>
#include <linux/syscalls.h>
#include <asm/page.h>
#include <asm/msr.h>
#include <linux/mount.h>
//#include <linux/swapops.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>

#include "memexec.h"



/* verbose printks */
const int verbose = 1;



static inline unsigned long get_mmap_flags(unsigned short vm_flags) {

  return MAP_FIXED |
    (vm_flags & VM_MAYSHARE? MAP_SHARED : MAP_PRIVATE) |
    (vm_flags & VM_GROWSDOWN) |
    (vm_flags & MAP_FIXED) |
```

```
    (vm_flags & VM_EXECUTABLE)
        ;
}


/*
    Restart a process.
*/

static int restart(unsigned long clone_flags, unsigned long arg,
                   unsigned long stack_size)
{

        struct memproc *mem;
        struct pt_regs *regs;
        struct pt_regs *tmp;
        struct pt_regs *tmp1;
        int ret;
        int err, i, k;
        mm_segment_t fs;
        sigset_t * blocked;
        struct sighand_struct *sig;



        mem = (struct memproc*)kmalloc(sizeof(struct memproc), GFP_KERNEL);

        // arg is the pointer to param
        if (copy_from_user(mem, (struct memproc *)arg, sizeof(struct memproc)))
                return -EFAULT;

        fs = get_fs();
        err = -ENOMEM;

        // Restore command name
        mem_strncpy(current->comm, mem->name, 16);
        printk(KERN_INFO "MEMEXEC: %s \n", current->comm);

        if (verbose)
          printk(KERN_INFO "MEMEXEC: Restoring Flags\n");

        current->flags = mem->flags;

        current->state = TASK_INTERRUPTIBLE;


        if (verbose)
          printk(KERN_INFO "MEMEXEC: Restoring vm areas\n");

        /* Map all the segments */

        for (i=0; i<mem->seg_num; i++)

        {
                unsigned long size;
                unsigned long mmap_prot, mmap_flags;
                struct file *file;
                int omode = O_RDONLY;

                set_fs(KERNEL_DS);

                size = mem->seglist[i].vm_end - mem->seglist[i].vm_start;

                mmap_prot = mem->seglist[i].flags & 7;
                mmap_flags = get_mmap_flags(mem->seglist[i].flags);

                set_fs(fs);
                if ( (mmap_prot&PROT_WRITE) && (mmap_flags&MAP_SHARED) &&
                        !(mmap_flags&MAP_DENYWRITE))
                        omode = O_RDWR;


                if (mem->seglist[i].num_pages == 0)
                        continue;


                if (mem->seglist[i].vm_file == 0)
                {

                        down_write(&current->mm->mmap_sem);
```

```
                if ((mmap_flags&MAP_GROWSDOWN) || !(mmap_flags&MAP_EXECUTABLE))
                ret = do_mmap(NULL, mem->seglist[i].vm_start, size, mmap_prot|PROT_WRITE,
                        mmap_flags, 0);

                up_write(&current->mm->mmap_sem);

                if (ret != mem->seglist[i].vm_start)
                        printk(KERN_INFO "Error in mmap %08lx!! Received %d !\n",
                                mem->seglist[i].vm_start, ret);

        for (k=0; k<=mem->seglist[i].num_pages - 1; k++)
        {
                set_fs(KERNEL_DS);

                if (mem->seglist[i].num_pages == 0)
                                break;

                if ((ret = copy_to_user((void *)mem->seglist[i].framelist[k].vm_start,
                                        mem>seglist[i].framelist[k].contents,
                                        PAGE_SIZE)) > 0)
                printk(KERN_INFO "Fail Copy??...  0x%08lx, %d \n",
                                mem->seglist[i].framelist[k].vm_start, ret);

        }

        }

        else
        {

                set_fs(KERNEL_DS);
                file = filp_open(mem->seglist[i].filename, omode, 0);

                size = mem->seglist[i].vm_end - mem->seglist[i].vm_start;

                down_write(&current->mm->mmap_sem);

                ret = do_mmap_pgoff(file, mem->seglist[i].vm_start, size, mmap_prot,
                        mmap_flags, mem->seglist[i].pgoff);

                up_write(&current->mm->mmap_sem);

                fput(file);

                if (ret != mem->seglist[i].vm_start)
                        printk("Error in mmap %08lx!! Received %08x !\n",
                                mem->seglist[i].vm_start, ret);

        }

}

set_fs(fs);

        /* Restore the memory mapping */
if (verbose)
  printk(KERN_INFO "MEMEXEC: Restoring vm structure\n");


down_write(&current->mm->mmap_sem);

current->mm->start_code  = mem->vm_map.start_code;
current->mm->end_code    = mem->vm_map.end_code;
current->mm->start_data  = mem->vm_map.start_data;
current->mm->end_data    = mem->vm_map.end_data;
current->mm->start_brk   = mem->vm_map.start_brk;
current->mm->brk         = mem->vm_map.brk;
current->mm->start_stack = mem->vm_map.start_stack;
current->mm->arg_start   = mem->vm_map.arg_start;
current->mm->arg_end     = mem->vm_map.arg_end;
current->mm->env_start   = mem->vm_map.env_start;
current->mm->env_end     = mem->vm_map.env_end;

up_write(&current->mm->mmap_sem);


if (verbose)
  printk(KERN_INFO "MEMEXEC: Restoring registers\n");

regs = task_pt_regs(current);
```

```
        tmp = (struct pt_regs *)mem->reg_set.regs;
        tmp1 = (struct pt_regs *)kmalloc(sizeof(struct pt_regs), GFP_KERNEL);

        *regs = *tmp;



        /*   FILES   */
        //if (verbose)
        //  printk("Restoring file table\n");


        /******************/
        /*    SIGNALS     */
        /******************/

        if (verbose)
          printk(KERN_INFO "MEMEXEC: Restoring signal handlers\n");

        sig = (struct sighand_struct *) mem->sig.sig_actions;
        atomic_set(&sig->count, 1);

        spin_lock_irq(&current->sighand->siglock);


        for (i = 0; i < _NSIG; i++)
        {
                current->sighand->action[i] = sig->action[i];
                if (i== SIGSTOP-1 || i == SIGKILL -1)
                {
                        current->sighand->action[i].sa.sa_handler = SIG_DFL;
                        current->sighand->action[i].sa.sa_flags = 0;
                        sigemptyset(&current->sighand->action[i].sa.sa_mask);
                }

                sigdelsetmask(&current->sighand->action[i].sa.sa_mask,
sigmask(SIGKILL)|sigmask(SIGSTOP));
        }

        spin_unlock_irq(&current->sighand->siglock);

        blocked = (sigset_t *)&mem->sig.sig_blocked;
        for (i = 0; i < _NSIG_WORDS; i++)
                current->blocked.sig[i] = blocked->sig[i];

        // Restore Process' State
        current->state = TASK_RUNNING; //mem->state;

        if (verbose)
          printk(KERN_INFO "MEMEXEC: *** RESTART: done ***\n");

        err = 0;

        return err;

}



/* Device Declarations ************************** */

/* The name for our device, as it will appear
 * in /proc/devices */
#define DEVICE_NAME "memexec"

int memexec_open(struct inode *inode,
                    struct file *file) {
  return 0;
}

int memexec_release(struct inode *inode,
                        struct file *file) {
  return 0;
}

/*
 * ioctl impl. for checkpoint. Primary means to interact with device.
 */
int memexec_ioctl(struct inode * inode_i, struct file * file,
```

```
               unsigned int cmd, unsigned long arg) {

  /* Quick error checking. */
  if(_IOC_TYPE(cmd) != MAGIC_NUMBER) return -ENOTTY;
  if(_IOC_NR(cmd) > MAX_IOC_NR) return -ENOTTY;

  // do restart here
  if (cmd == IOCTL_RESTART)
    return restart(SIGCHLD, arg, 0);

    return -EINVAL;
}

/* Module Declarations *************************** */

static int major;

struct file_operations memexec_fops = {
 owner:          THIS_MODULE,
 ioctl:          memexec_ioctl,
 open:           memexec_open,
 release:        memexec_release,
};

int memexec_init(void)
{
  int result;

  result = register_chrdev(0, DEVICE_NAME, &memexec_fops);

  if(result < 0){
    printk(KERN_ALERT "Registering Device Failed w/ major %d.\n", major);
    return result;
  }

  if(verbose)
        printk(KERN_INFO "*****Device Memexec Loaded!!!*****");

  major = result;

  return 0;
}

void memexec_cleanup(void)
{
  unregister_chrdev(major, DEVICE_NAME);

}

module_init(memexec_init);
module_exit(memexec_cleanup);

MODULE_LICENSE("GPL");
```

********** End memlkm.c **********

********** memexec.c **********

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/mman.h>

#include <libxml/xmlmemory.h>
#include <libxml/tree.h>
```

```
#include <libxml/parser.h>

#include "memexec.h"


/*
** Translation Table as described in RFC1113
*/
static const char cb64[]="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

/*
** Translation Table to decode (created by author)
*/
static const char
cd64[]="|$$$}rstuvwxyz{$$$$$$$>?@ABCDEFGHIJKLMNOPQRSTUVW$$$$$$XYZ[\\]^_`abcdefghijklmnopq";

/*
** decodeblock
**
** decode 4 '6-bit' characters into 3 8-bit binary bytes
*/
void decodeblock( unsigned char in[4], unsigned char out[3] )
{
    out[ 0 ] = (unsigned char ) (in[0] << 2 | in[1] >> 4);
    out[ 1 ] = (unsigned char ) (in[1] << 4 | in[2] >> 2);
    out[ 2 ] = (unsigned char ) (((in[2] << 6) & 0xc0) | in[3]);
}

/*
** decode
**
** decode a base64 encoded stream discarding padding, line breaks and noise
*/
static int b64decode(char *src, char *dst )
{
        unsigned char in[4], out[3], v;
        int i, len, size;
        char *tmpptr;
        FILE *pFile;

        pFile = fopen(".out", "wb+");
        tmpptr = src;

        while( *tmpptr )
        {
        for( len = 0, i = 0; i < 4 && *tmpptr; i++ )
                {
                        v = 0;
                        while( *tmpptr && v == 0 )
                        {
                        v = (unsigned char) (*tmpptr);
                        v = (unsigned char) ((v < 43 || v > 122) ? 0 : cd64[ v - 43 ]);
                        if( v )
                                {
                                        v = (unsigned char) ((v == '$') ? 0 : v - 61);
                        }

                                tmpptr++;
                        }
                if( *tmpptr )
                {
                len++;
                if( v )
                        {
                                in[ i ] = (unsigned char) (v - 1);
                }
                }
                else
                {
                in[i] = 0;
                }
        }
        if( len )
         {
         decodeblock( in, out );
         for( i = 0; i < len - 1; i++ )
                putc( out[i], pFile );
        }
    }
```

```
        fseek(pFile, 0, SEEK_END);
        size = ftell(pFile);
        fseek(pFile, 0, SEEK_SET);


        fread(dst, size, 1, pFile);
        fclose(pFile);
        return size;
}

void dump_buffer(void *buffer, int buffer_size)
{
  int i;

  for(i = 0;i < buffer_size;++i)
     printf("%c", ((char *)buffer)[i]);
}

static void mem_copy(char * dst, char * src, size_t size)
{
        char *tmp;
        int i = 0;

        for (tmp = src;i<size;++tmp)
        {
                dst[i] = *tmp;
                i++;
        }
        dst[i] = '\0';

}

static int stripSpaces(char* in)    // Remove White spaces
{

        char *ret;
        char *tmp;
        int i = 0;
        int j = 0;

        ret = malloc(strlen(in)+1);
        for (tmp = in;*tmp!='\0';++tmp)
        {
                if (!isspace(*tmp))
                {
                        ret[j] = *tmp;
                        j++;
                }
        }
        ret[j] = '\0';

        mem_copy(in, ret, j);
        free(ret);

        return 0;

}

static void parseDoc(char *docname, struct memproc * input) //Parse XML document
{

        xmlDocPtr doc;
        xmlNodePtr cur, child;
        xmlChar *key;
        char *test, *str;
        int size, err;

        doc = xmlParseFile(docname);

        if (doc == NULL ) {
                fprintf(stderr,"Document not parsed successfully. \n");
                return;
        }
        cur = xmlDocGetRootElement(doc);

        if (cur == NULL) {
                fprintf(stderr,"empty document\n");
                xmlFreeDoc(doc);
                return;
        }
```

```
if (xmlStrcmp(cur->name, (const xmlChar *) "process")) {
        fprintf(stderr,"document of the wrong type, root node != process");
        xmlFreeDoc(doc);
        return;
}

cur = cur->xmlChildrenNode;


/************* GET OLD PID *************/

child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"old_pid")))
        {
                key = xmlNodeGetContent(child->xmlChildrenNode);
                input->old_pid = atoi(key);
                //printf("OLD_PID: %ld  \n", input->old_pid);
                xmlFree(key);
                break;
        }

child = child->next;
}


/*********** GET NAME ***************/


child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"name")))
        {
                key = xmlNodeListGetString(doc, child->xmlChildrenNode, 1);
                test = (char *)key;
                err = stripSpaces(test);
                strcpy(input->name, test);
                //printf("NAME: %s\n", input->name);
                xmlFree(key);
                break;
        }
child = child->next;
}

/*********** GET STATE ***************/

child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"state")))
        {
                key = xmlNodeGetContent(child->xmlChildrenNode);
                input->state = atoi(key);
                //printf("STATE: %ld  \n", input->state);
                xmlFree(key);
                break;
        }

child = child->next;
}

/*********** GET FLAGS ***************/

child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"flags")))
        {
                key = xmlGetProp(child, "value");
                input->flags = strtoul((char *)key, NULL, 16);  // String to unsigned long
                //printf("FLAGS: 0x%08x  \n", (int) input->flags);
                xmlFree(key);
                break;
        }

child = child->next;
}
```

```
/*********** GET CREDENTIALS ***************/

child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"credentials")))
        {
                key = xmlGetProp(child, "uid");
                input->memcred.uid = atoi(key);
                key = xmlGetProp(child, "gid");
                input->memcred.gid = atoi(key);

                key = xmlGetProp(child, "euid");
                input->memcred.euid = atoi(key);
                key = xmlGetProp(child, "egid");
                input->memcred.egid = atoi(key);

                key = xmlGetProp(child, "suid");
                input->memcred.suid = atoi(key);
                key = xmlGetProp(child, "sgid");
                input->memcred.sgid = atoi(key);

                key = xmlGetProp(child, "fsuid");
                input->memcred.fsuid = atoi(key);
                key = xmlGetProp(child, "fsgid");
                input->memcred.fsgid = atoi(key);

                //printf("CREDENTIALS - UID: %d  \n", input->memcred.uid);
                xmlFree(key);
                break;
        }

child = child->next;
}

/*********** GET REGISTERS ***************/

child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"reg_set")))
        {
                key = xmlNodeListGetString(doc, child->xmlChildrenNode, 1);
                test = (char *)key;
                err = stripSpaces(test);

                str = (char *)malloc(strlen(test) + 1);
                input->reg_set.regs = (char *)malloc(strlen(test) + 1);
                str[0] = '\0';
                size = b64decode(test, str);
                mem_copy(input->reg_set.regs, str, size);
                input->reg_set.size = size;
                xmlFree(key);
                free(str);
                break;
        }
child = child->next;
}


/*********** GET VMSTRUCT ***************/

child = cur;
while (child != NULL) {

        if ((!xmlStrcmp(child->name, (const xmlChar *)"vmstruct")))
        {
                key = xmlGetProp(child, "start_code");
                input->vm_map.start_code = strtoul((char *)key, NULL, 16);
                key = xmlGetProp(child, "end_code");
                input->vm_map.end_code = strtoul((char *)key, NULL, 16);

                key = xmlGetProp(child, "start_data");
                input->vm_map.start_data = strtoul((char *)key, NULL, 16);
                key = xmlGetProp(child, "end_data");
                input->vm_map.end_data = atoi(key);

                key = xmlGetProp(child, "start_brk");
                input->vm_map.start_brk = strtoul((char *)key, NULL, 16);
                key = xmlGetProp(child, "brk");
                input->vm_map.brk = atoi(key);
```

```
                        key = xmlGetProp(child, "arg_start");
                        input->vm_map.arg_start = strtoul((char *)key, NULL, 16);
                        key = xmlGetProp(child, "arg_end");
                        input->vm_map.arg_end = strtoul((char *)key, NULL, 16);

                        key = xmlGetProp(child, "env_start");
                        input->vm_map.env_start = strtoul((char *)key, NULL, 16);
                        key = xmlGetProp(child, "env_end");
                        input->vm_map.env_end = strtoul((char *)key, NULL, 16);

                        key = xmlGetProp(child, "start_stack");
                        input->vm_map.start_stack = strtoul((char *)key, NULL, 16);

                        xmlFree(key);
                        break;
                }

        child = child->next;
        }

        /*********** GET SEGMENTS ***************/

        child = cur;
        while (child != NULL) {

                if ((!xmlStrcmp(child->name, (const xmlChar *)"segments")))
                {
                        xmlNodePtr ptr1, ptr2;
                        int k1 = 0;

                        key = xmlGetProp(child, "number");
                        input->seg_num = atoi(key);


                        input->seglist = (struct segments *)malloc(sizeof(struct segments[input-
>seg_num]));

                        ptr1 = child->xmlChildrenNode;
                        while (ptr1 != NULL)                                                    // Parse
each region of a segment
                        {
                                if ((!xmlStrcmp(ptr1->name, (const xmlChar *)"vm_region")))
                                {

                                        key = xmlGetProp(ptr1, "vm_start");
                                      input->seglist[k1].vm_start = strtoul((char *)key, NULL, 16);
                                        key = xmlGetProp(ptr1, "vm_end");
                                        input->seglist[k1].vm_end = strtoul((char *)key, NULL, 16);
                                        key = xmlGetProp(ptr1, "vm_pgprot");
                                        input->seglist[k1].prot = strtoul((char *)key, NULL, 16);
                                        key = xmlGetProp(ptr1, "vm_flags");
                                        input->seglist[k1].flags = strtoul((char *)key, NULL, 16);
                                        key = xmlGetProp(ptr1, "vm_pgoff");
                                        input->seglist[k1].pgoff = strtoul((char *)key, NULL, 16);
                                        key = xmlGetProp(ptr1, "num_pages");
                                        input->seglist[k1].num_pages = atoi(key);


                                        input->seglist[k1].framelist = (struct frames
*)malloc(sizeof(struct frames[input->seglist[k1].num_pages]));

                                        if (key = xmlGetProp(ptr1, "filename"))
                                        {
                                                strcpy(input->seglist[k1].filename, (char *)key);
                                                input->seglist[k1].vm_file = 1;
                                        }
                                        else
                                        {
                                                input->seglist[k1].vm_file = 0;
                                                strcpy(input->seglist[k1].filename, "");
                                        }

                                        ptr2 = ptr1->xmlChildrenNode;
                                        int k2 = 0;

                                        while (ptr2 != NULL)
        // Parse each frame of a region
                                        {
```

```
                                                        if ((!xmlStrcmp(ptr2->name, (const xmlChar
*)"vm_frame")))
                                                        {

                                                                key = xmlGetProp(ptr2, "vm_start");
                                                                input->seglist[k1].framelist[k2].vm_start
= strtoul((char *)key, NULL, 16);
                                                                key = xmlGetProp(ptr2, "vm_end");
                                                                input->seglist[k1].framelist[k2].vm_end =
strtoul((char *)key, NULL, 16);
                                                                key = xmlGetProp(ptr2, "flags");
                                                                input->seglist[k1].framelist[k2].flags =
strtoul((char *)key, NULL, 16);

                                                                key = xmlGetProp(ptr2, "offset");
                                                                input->seglist[k1].framelist[k2].offset =
strtoul((char *)key, NULL, 16);

                                                                if ( (input->seglist[k1].vm_file == 0) &&
                                                                        (key = xmlNodeListGetString(doc,
ptr2->xmlChildrenNode, 1)) )
                                                                {

                                                                                test = (char *)key;
                                                                                err = stripSpaces(test);

                                                                                str = (char
*)malloc(strlen(test) + 1);
                                                                                str[0] = '\0';
                                                                                input-
>seglist[k1].framelist[k2].contents = (char *)malloc(strlen(test) + 1);

                                                                                size = b64decode(test,
str);
                                                                                mem_copy(input-
>seglist[k1].framelist[k2].contents, str, size);

                                                                }
                                                                else
                                                                {
                                                                        input-
>seglist[k1].framelist[k2].contents = (char *)malloc(2);
                                                                        mem_copy(input-
>seglist[k1].framelist[k2].contents, "\0", 2);
                                                                }

                                                                k2++;
                                                        }

                                                        ptr2 = ptr2->next;
                                                }
                                                k1++;
                                        }
                                        ptr1 = ptr1->next;
                                }

                                xmlFree(key);
                        }
                child = child->next;
                }



                /*********** GET OPEN FILES ***************/

                child = cur;
                while (child != NULL) {

                        if ((!xmlStrcmp(child->name, (const xmlChar *)"open_files")))
                        {

                                xmlNodePtr ptr;
                                int k = 0;

                                key = xmlGetProp(child, "Number");
                                input->file_num = atoi(key);

                                if (input->file_num == 0)
                                        break;
```

```
                        struct open_file tmp[input->file_num];

                        ptr = child->xmlChildrenNode;
                        while (ptr != NULL)
                        {
                                if ((!xmlStrcmp(ptr->name, (const xmlChar *)"vm_region")))
                                {
                                        key = xmlGetProp(ptr, "path");
                                        tmp[k].path = malloc(strlen(key) + 1);
                                        strcpy(tmp[k].path, (char *) key);
                                        k++;
                                }
                                ptr = ptr->next;
                        }

                        xmlFree(key);
                        input->filelist = malloc(sizeof(tmp) + 1);
                        input->filelist = tmp;
                        break;
                }
        child = child->next;
        }

        /*********** GET SIGNALS ***************/

        child = cur;
        while (child != NULL) {

                if ((!xmlStrcmp(child->name, (const xmlChar *)"signals")))
                {

                        child = child->xmlChildrenNode;
                        while (child != NULL)
                        {
                                if ((!xmlStrcmp(child->name, (const xmlChar *)"sig_actions")))
                                {
                                        key = xmlNodeListGetString(doc, child->xmlChildrenNode, 1);
                                        test = (char *)key;
                                        err = stripSpaces(test);

                                        str = (char *)malloc(strlen(test) + 1);
                                        str[0] = '\0';
                                        input->sig.sig_actions = (char *)malloc(strlen(test) + 1);

                                        size = b64decode(test, str);
                                        mem_copy(input->sig.sig_actions, str, size);
                                        input->sig.sig_actions_size = size;
                                }

                                if ((!xmlStrcmp(child->name, (const xmlChar *)"sig_blocked")))
                                {
                                        key = xmlGetProp(child, "value");
                                        input->sig.sig_blocked = strtoul((char *)key, NULL, 16);
                                }

                                if ((!xmlStrcmp(child->name, (const xmlChar *)"sig_pending")))
                                {
                                        key = xmlGetProp(child, "value");
                                        input->sig.sig_pending = strtoul((char *)key, NULL, 16);
                                }

                                child = child->next;
                        }

                        xmlFree(key);
                        break;
                }
        child = child->next;
        }
        xmlFreeDoc(doc);
        return;
}

int main(int argc, char **argv) {

        struct memproc *input;
        char *docname;
        int ret;
        int dev_fd = 0;
        int l, k;
```

```
        if (argc <= 1) {
                printf("Usage: %s docname\n", argv[0]);
                return(0);
        }

        input = (struct memproc *)malloc(sizeof(struct memproc));
        docname = argv[1];
        parseDoc (docname, input);

        dev_fd = open(DEV_FILE, 0);
        if (dev_fd<0)
        {
                printf("Cannot open device file.... file_desc = %d\n", dev_fd);
                exit (-1);
        }

        printf("Device file opened: %d\n", dev_fd);

        ret = ioctl(dev_fd, IOCTL_RESTART, input);
        close(dev_fd);


        free(input);
        printf("Done....\n");

        return (1);
}
```

********** end memexec.c **********

********** Makefile **********

```
KDIR := /lib/modules/$(shell uname -r)/build
PWD  := $(shell pwd)

obj-m += memlkm.o

all:    memlkm
        gcc memexec.c -o memexec -I/usr/include/libxml2 -lxml2

memlkm:
        make -Wall -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
        make -Wall -C  $(KDIR) SUBDIRS=$(PWD) modules clean
        rm memexec
```

REFERENCE LIST

[1] Bayer U., Moser A., Kruegel C., Kirda E., Dynamic Analysis of Malicious Code, *Journal in Computer Virology 2(1): 67-77*, 2006.

[2] Betz, C. *DFRWS 2005 Challenge Report*, August 2005. http://www.dfrws.org/2005/challenge/betzReport.shtml, Last Visited August 2011.

[3] Bovet D., Cesati M., *Understanding the Linux Kernel, Third Edition*, O'Reilly Media, Inc., November 2005.

[4] Burdach M., *An introduction to Windows memory forensic*, July 2005. http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf, Last Visited May 2011.

[5] Burdach, M. *Digital Forensics of the Physical Memory*; March 2005. http://forensic.seccure.net/, Last Visited May 2011.

[6] Burdach M., *Finding Digital Evidence in Physical Memory*, Black Hat Conference Las Vegas, NV, 2006.

[7] Carrier B., Digital Forensics Works, *IEEE Security and Privacy 7:26-29*, March 2009.

[8] Carrier B., *File System Forensic Analysis*, Addison-Wesley Professional, 2005.

[9] Carrier B. D., Risks of Live Digital Forensic Analysis, *Communications of the ACM*, 49(2), 56-61. February 2006.

[10] Carrier B. D., Grand J., A Hardware-based Memory Acquisition Procedure for Digital Investigations, *Digital Investigation 1(2): 50-60*, 2004.

[11] Carrier B., Spafford E.H., Getting Physical with the Digital Investigation Process, *International Journal of Digital Evidence 2(2),* 2003.

[12] Case A., Golden R., and Marziale L., FACE: Automated digital Evidence Discovery and Correlation. *Digital Investigation 5: 65-75*, 2008.

[13] Davidoff S., *Cleartext Passwords in Linux Memory*, Massachusetts Institute of Technology, July 2008.

[14] Digital Forensics Solutions, *Bringing Linux Support to Volatility*, http://dfsforensics.blogspot.com/2011/03/bringing-linux-support-to-volatility.html , Last Visited October 2011.

[15] Distler D., *Malware Analysis: An Introduction*, SANS Institute InfoSec Reading Room, 2007.

[16] Easttom C., Taylor J., *Computer Crime, Investigation, and the Law*, Course Technology PTR, April 2010.

[17] Farmer D., Venema W., *Forensic Discovery*, Addison-Wesley Professional, December 2004.

[18] Foster M., Wilson J., Process Forensics: A Pilot Study on the Use of Checkpointing Technology in Computer Forensics, *International Journal of Digital Evidence 3(1),* 2004.

[19] Garner G. M. Jr., Mora R., *Response to Specific Questions Posed by the DFRWS 2005 Memory Challenge*, August 2005. http://www.dfrws.org/2005/challenge/kntlist.shtml, Last Visited August 2011.

[20] Girault E., *Volatilitux*, http://www.segmentationfault.fr/projets/volatilitux-physical-memory-analysis-linux- systems/, Last Visited August 2011.

[21] Golden R., Case A., and Marziale L., Dynamic Recreation of Kernel Data Structures for Live Forensics. *Digital Forensics Research Workshop 2010*, August 2010.

[22] Gorman M., *Understanding the Linux Virtual Memory Manager*, Prentice Hall, April 2004.

[23] Grugq, Defeating Forensic Analysis on Unix, *Phrack #59*, July 2002. http://www.phrack.org/issues.html?issue=59&id=6#article, Lat Visited May 2011.

[24] Grugq, Remote Exec, *Phrack #62*, July 2004. http://www.phrack.org/issues.html?issue=62&id=8#article, Last Visited May 2011.

[25] Halderman A., Schoen S., Heninger N., Clarkson W., Paul W., Calandrino J., Feldman A., Appelbaum J., and Felten E., Lest we remember: Cold Boot Attacks on Encryption Keys, *Usenix Security Symposium*, 2008.

[26] Hay B., Nance K., Bishop M., Live Analysis: Progress and Challenges, *IEEE Security and Privacy 7(2): 30-37*, March 2009.

[27] Identity Theft Resource Centre (ITRC), *Data Breaches in 2010*, http://www.idtheftcenter.org/artman2/publish/lib_survey/Breaches_2010.shtml, January 2011, Last Visited April 2011.

[28] Ilo, Process Dump and Binary Reconstruction, *Phrack #63*, August 2005. http://www.phrack.org/issues.html?issue=63&id=12#article, Last Visited August 2011.

[29] Kendhall K., *Practical Malware Analysis*, Black Hat Conference Las Vegas, NV, 2007.

[30] Kollar I., Forensic RAM dump image analyzer (Foriana)*, Charles University in Prague Thesis*; August 2010.

[31] Lessing M., von Solms B., Live Alternative Acquisition as Alternative to traditional Forensic Processes, *IT Incident Management & IT Forensics (IMF 2008): 1-9*, 2008.

[32] LXR (The Linux Cross Reference), http://lxr.linux.no/, Last Visited August 2011.

[33] Mandiant, *Memoryze*, http://www.mandiant.com/products/free_software/memoryze/, Last Visited August 2011.

[34] Martin A., Firewire Memory Dump of a Windows XP computer: A Forensic Approach, *Tech. Rep.,* 2007.

[35] Mrdovic S., Huseinovic A. and Zajko E. Combining Static and Live Digital Forensic Analysis in Virtual Environment. *Symposium on Information, Communication and Automation Technologies*, 12, 1-6; October 2009.

[36] Nance, K., M. Bishop, and B. Hay. *Virtual Machine Introspection: Observation or Interference?* IEEE Security and Privacy Virtualization Special Issue, October 2008.

[37] Pikewerks Corp., *SecondLook*, http://pikewerks.com/sl, Last Visited May 2011.

[38] Rantala R., BJS (Bureau of Justice Statistics), *Cybercrime against Businesses*, NCJ 221943, September 2008.http://bjs.ojp.usdoj.gov/index.cfm?ty=pbdetail&iid=769, Last Visited May 2011.

[39] Richardson R., CSI (Computer Security Institute), *Computer Crime and Security Survey*, 2007. http://gocsi.com/SurveyArchive, Last Visited April 2011.

[40] Rodriguez C., Fischer G., Smolski S., *Linux Kernel Primer, The: A Top-Down Approach for x86 and PowerPC Architectures*, Prentice Hall, September 2005.

[41] Rogers M., Seigfried K, The future of Computer Forensics: A needs analysis survey, *Computers & Security Volume 23(1): 12-16*, February 2004.

[42] Schuster A., Searching for processes and Threads in Microsoft Windows Memory Dumps, *Proceedings of the 2006 Digital Forensics Research Workshop (DFRWS)*, August 2006.

[43] Suiche M., NFI (Netherlands Forensic Institute), Advanced Mac OS X Physical Memory Analysis. *Black Hat Briefings DC*. 2010.

[44] Suiche M., NFI (Netherlands Forensics Institute), Wind32dd: Challenges of Windows physical memory acquisition and exploitation, *Shakacon*, June 2009.

[45] Symantec, *W32.Stuxnet*, http://www.symantec.com/security_response/writeup.jsp?docid=2010-0714003123-99, September 2010. Last Visited August 2011.

[46] Urrea JM. An analysis of Linux RAM forensics. *Naval Post Graduate School Thesis*; March 2006.

[47] VMware, *What Files Make Up a Virtual Machine?*, http://www.vmware.com/support/ws55/doc/ws_learning_files_in_a_vm.html, Last Visited September 2011.

[48] W3C – World Wide Web Consortium, *XML Schema Reference*, http://www.w3schools.com/schema/schema_elements_ref.asp , Last Visited October 2011.

[49] Waits C., Akinyele J.A. , Nolan R. , and Rogers L. , *Computer Forensics: Results of Live Response Inquiry vs. Memory Image Analysis*, CERT; 2008.

[50] Walters A., Petroni M. et al., *Volatility*, https://www.volatilesystems.com/default/volatility, Last Visited August 2011.

[51] Wiles J., Reyes A., *The Best Damn Cybercrime and Forensics Book Period*, Syngress, October 2007.

[52] Zhong H., Nieh J., *CRAK: Linux Checkpoint/Restart As a Kernel Module*, Department of Computer Science, Columbia University, Technical Report CUCS-014-01, November 2001.