A Survey of Main Memory Acquisition and Analysis Techniques for the Windows Operating System

Stefan Vömel, Felix C. Freiling

Chair for IT Security Infrastructures, University of Erlangen-Nuremberg, Am Wolfsmantel 46, 91058 Erlangen-Tennenlohe, Germany

Abstract

Traditional, persistent data-oriented approaches in computer forensics face some limitations regarding a number of technological developments, e.g., rapidly increasing storage capabilities of hard drives, memory-resident malicious software applications, or the growing use of encryption routines, that make an in-time investigation more and more difficult. In order to cope with these issues, security professionals have started to examine alternative data sources and emphasize the value of volatile system information in RAM more recently. In this paper, we give an overview of the prevailing techniques and methods to collect and analyze a computer's memory. We describe the characteristics, benefits, and drawbacks of the individual solutions and outline opportunities for future research in this evolving field of IT security.

Keywords: memory forensics, memory acquisition, memory analysis, live forensics, Microsoft Windows

1. Introduction

With the widespread use of computer systems and network architectures, digital cyber crime has, unfortunately, aggravated as well. According to a recent publication by the Internet Crime Complaint Center (2010), a partnership between the National White Collar Crime Center (NW3C) and the Federal Bureau of Investigation (FBI), the number of complaints filed to the institution has almost gone up by the factor 20 within less than a decade. In 2009, more than 336,000 reports about different types of illicit activity such as online fraud, identity theft, and economic espionage were registered. The yearly monetary loss of complaints referred to law enforcement was estimated to be nearly $560 million. As a survey by the Computer Security Institute (2009) shows, companies may lose up to several hundred thousand dollars in the course of an incident. In such cases, a forensic investigation of the affected machines may prove helpful

Email addresses: `stefan.voemel@informatik.uni-erlangen.de` (Stefan Vömel), `felix.freiling@informatik.uni-erlangen.de` (Felix C. Freiling)

for reconstructing the actions that led to the security breach, finding relevant pieces of evidence, and possibly taking legal actions against the adversary.

Traditional approaches in computer forensics mostly described the acquisition and analysis of *persistent* system data. Respected procedures usually involved powering off the suspect machine, creating an exact bit-by-bit image of the corresponding hard disks and other storage media, and performing a postmortem examination of the collected information (U.S. Secret Service, 2006; U.S. Department of Justice, 2008). Obtaining a copy of physical Random Access Memory (RAM) was, on the other hand, frequently neglected by first responders (Shipley and Reeve, 2006; Hoglund, 2008), even though guidelines stressed the necessity of securing digital evidence with regard to the *order of volatility*, i.e., from the volatile to the less volatile, as early as in 2002 (Brezinski and Killalea, 2002; Casey, 2004; Farmer and Venema, 2005). In the face of ever-growing hard drive storage capabilities (Oswald, 2010), and correspondingly, tremendous efforts to analyze media in time (Mrdovic et al., 2009; Walters and Petroni, 2007; Shipley and Reeve, 2006) as well as a rising number of memory-resident malicious software applications (Moore et al., 2003; Rapid7 LLC, 2004; Sparks and Butler, 2005; Bilby, 2006), the restoration of transient and system state-specific information has, however, also moved more gradually into the focus of current research, beginning with the *Digital Forensic Research Workgroup* (DFRWS) challenge in 2005 (DFRWS, 2005).

This shift in practices has been driven and inspired by several other developments, too: First, "pulling the plug" on a company server may negatively affect productivity in certain cases and cause substantial losses due to unexpected down times. Furthermore, depending on the configuration of the system, file system journals may be damaged during the shutdown process, or the machine may be difficult to restart. Thus, there is a demand to minimize interferences with existing business and enterprise processes. Second, some programs are explicitly designed to make no or preferably as little persistent changes as possible on the hard disk of the user. Contemporary examples for this type of software are the *Mozilla Firefox* web browser with its private browsing capability (see Mozilla Foundation, 2008; Aggarwal et al., 2010) or utilities included in the *PortableApps.com* project (Rare Ideas, 2010). For this reason, forensic analysts must adapt their strategies and also search in volatile system storages for traces and data remnants, including usernames, passwords, and text fragments. Moreover, many modern operating systems include support for file or even full disk encryption (Microsoft Corporation, 2009; Apple Inc., 2010; Saout, 2006). Similar functions are provided by freely available open source tools, e.g., *TrueCrypt* (TrueCrypt Foundation, 2010), or commercial products such as *SafeGuard Easy* (Sophos Plc, 2010) or *PGP* (PGP Corporation, 2010). Because of the transparent design and ease of use of these software products, security professionals are likely to face an increasing number of encrypted drives that make traditional investigations infeasible (Getgen, 2009). Restoring a cryptographic key from memory might be the only possibility to get access to the protected data area in this case. The same holds true for packed malicious binaries. Malware writers typically employ compression, armoring, and obfuscation techniques to make reverse engineering and static analysis of their code more difficult (Sharif et al., 2009; Rolles, 2009; Brosch and Morgenstern, 2006; Young and Yung, 2004). Memory inspection is a viable solution to cope with these issues and extract the unpacked and decrypted executable directly from RAM.

As can be seen, a myriad of valuable information is stored in volatile memory that is usually lost when the target computer is powered off. Failing to preserve its contents may thus destroy a significant amount of evidence.

Motivation for this Paper. Over the last 5 1/2 years, considerable research has been conducted in the field of memory forensics, and various methods have been published for capturing and examining the volatile storage of a target machine. However, many techniques solely apply to specific versions of operating systems and architectures or only work under certain conditions. Moreover, depending on the technology used, the reliability and trustworthiness of generated results may vary. For these reasons, security professionals must have a thorough understanding of the capabilities and limitations of the respective solutions in order to successfully retrieve pieces of evidence and complete a case. A complete description of the current state of the art appears to be missing at the time of this writing though, restricting (research) activities in this area to a number of renowned experts.

In this paper, we give a comprehensive and structural overview of proven approaches for obtaining and inspecting a computer's memory. We explain the technical foundation of existing tools and methodologies and outline their individual strengths and weaknesses. Based on these illustrations, security analysts and first responders may choose an adequate acquisition and analysis strategy. In addition, we give an extensive summary of the relevant literature. This review serves as a good starting point for own future studies.

Please note that our explanations refer to the product family of *Microsoft Windows* operating systems. We assume that due to their high popularity and dominant market position (Net Applications, 2010), investigators are particularly likely to face Windows-based machines in practice. In addition, as we will see, a deep knowledge of internal system structures is required to collect digital evidence from a volatile storage. Covering other platforms such as Linux or Mac OS is therefore out of the scope of this paper. Interested readers are referred to Movall et al. (2005) and Suiche (2010) for more information on these topics.

Outline of the Paper. This paper is outlined as follows: In Section 2, we briefly describe the memory management process and give an overview of the most important data structures that are required for this task. Current techniques and methods for creating a memory image from the target system are presented in Section 3, followed by a detailed illustration of the different investigative procedures in Section 4. A special framework for memory analysis activities, *Volatility*, is subject of Section 5. We conclude with a summary of our work and indicate opportunities for future research in this area in Section 6.

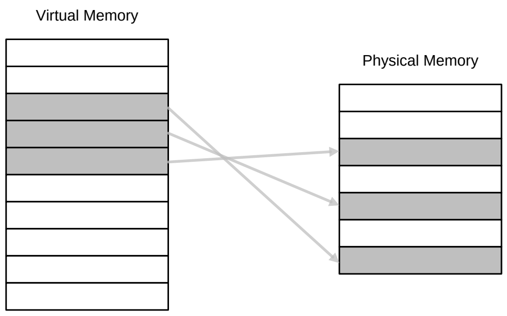


Figure 1: Mapping Virtual to Physical Memory
(Source: Based on Russinovich et al., 2009, p. 15)

2. Technical Background

Modern multi-tasking operating systems typically do not access physical memory directly, but rather operate on an abstraction called *virtual memory*. This abstraction of physical RAM needs specific hardware support (the so-called *Memory Manager* or *Memory Management Unit*) and offers several inherent advantages, e.g., the possibility of providing each process with its own protected view on system memory as well as monitoring and restricting read and write activities with the help of privilege rules (Intel Corporation, 2011). The layout between the virtual and physical address space may differ though, and blocks of virtual memory do not necessarily map to contiguous physical addresses as illustrated in Figure 1.

2.1. Memory Address Space Layout

On Microsoft Windows operating systems, each process has its own *private virtual address space* that is non-accessible to other running applications unless portions of memory are explicitly shared. Thereby, collisions and privilege violations between different executables are prevented.

A 32-bit x86 user process is equipped with 2 GB of virtual memory by default (Russinovich et al., 2009). The other half of the address space (`0x80000000` to `FFFFFFFF`) is reserved for system usage.[1] Kernel space is shared between and available to all system components. Thus, as Russinovich et al. (2009, p. 17) argue, it is vital that kernel-mode applications "be carefully designed and tested to ensure that they don't violate system security and cause system instability". Critical memory regions include, for example, the system pools that store volatile data that must not/may be paged out to hard disk as well as the

[1]For reasons of simplicity, we assume that advanced memory management features such as large address spaces, Address Windowing Extension (AWE), and Physical Address Extension (PAE) are turned off. For more information on these concepts as well as details about the 64-bit address space layout, please refer to Russinovich et al. (2009).

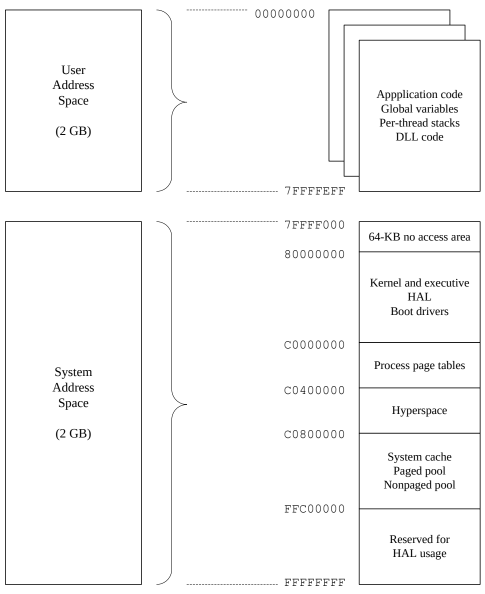


Figure 2: Virtual Address Space Layout
(Source: Based on Russinovich et al., 2009, p. 677)

process page tables that are required for virtual-to-physical address translation as we will see in the following section (see also Figure 2).

2.2. Virtual Address Translation

As we have already explained, programs usually operate on virtual memory regions only. Therefore, to manipulate the respective physical data, the Memory Manager must continuously translate (*map*) virtual into physical addresses. This procedure works as follows (Intel Corporation, 2011; Russinovich et al., 2009): At the hardware level, volatile storage is organized into units called *pages*. A common size of such pages is 4 KB on x86-platforms. To reference a page, the operating system implements a two-level approach: For every process, the operating system maintains a *page directory* that saves pointers (*Page Directory Entries*, PDEs, 4 bytes each, containing a pointer and several flags) to 1,024 different *page tables*. Page tables, in turn, contain up to 1,024 links (*Page Table Entries*, PTEs, 4 bytes each) to the corresponding page in main memory. Thus, in order to translate a virtual to a physical address, the Memory Manager first needs to recover the base address of the page directory. It is stored in the `CR3` register of the processor and reloaded from the kernel process (`_KPROCESS`) block of the executable at every context switch. The first 10 bits of the virtual address can then be used as an index into the page directory to retrieve the desired PDE. With the help of the PDE and the *page table index*, i.e., the subsequent 10 bits of the virtual address, the page table and PTE in question are identified in the

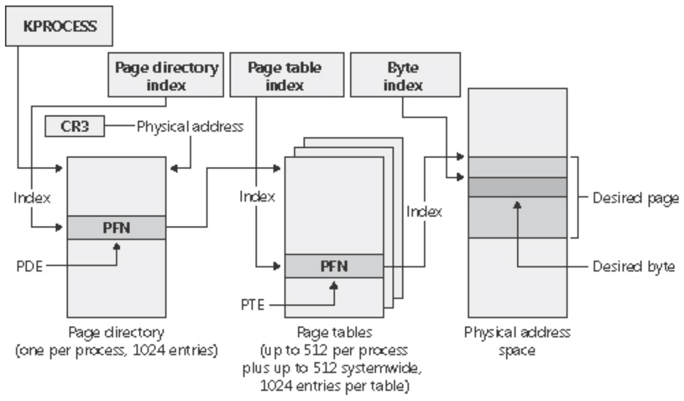


Figure 3: Virtual to Physical Address Translation
(Source: Russinovich et al., 2009, p. 699)

next step. To find the appropriate page and data in RAM eventually, the PTE and the 12-bit *byte index* of the virtual address are parsed. A summary of the entire process is presented in Figure 3.

2.3. Paging

With respect to the address translation sequence outlined in the previous section, we have implicitly assumed that the requested information is always available in main memory. However, in some cases, the total amount of virtual memory that is consumed by running processes is larger than the size of the entire physical storage. To cope with these scenarios, the operating system needs to temporarily swap out (*page*) memory contents to hard disk in order to free required space. When a thread attempts to operate on a swapped-out page, the memory management unit generates *page fault* interrupt that is handled by the operating system such that the requested information can be transferred back into RAM.

Whether or not a virtual address has been paged to disk is indicated by the *Valid* flag, the least significant bit, of a PDE or PTE. When it is set to `1`, the entry is regarded as valid, and the respective data is accessible in memory. In contrast, when the flag is cleared and both the *Transition* (`11`) and *Prototype* (`10`) bit are set to `0`, an address in a *page file* is referenced (see Figure 4). The default page file `pagefile.sys` is saved in the root directory of the primary disk. However, Microsoft Windows is capable of supporting 16 different page files with a size of up to 4,095 MB on x86-based platforms. The name and location of the files are specified in the registry string `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles`, a field in the PTE of the virtual address, the *Page Frame Number* (PFN, bits 1 to 4), denotes the current file in use.

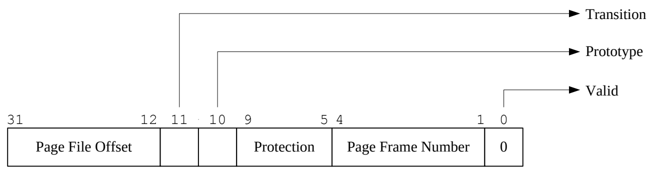


Figure 4: Invalid Page Table Entry (PTE)
(Source: Based on Russinovich et al., 2009, p. 710)

Please note that we have only given a brief overview of the flags and elements of a PDE and PTE, respectively. A more detailed explanation of the individual components and attributes can be found in the work of other authors (Russinovich et al., 2009; Savoldi and Gubian, 2008; Kornblum, 2007b; Maclean, 2006). The concepts described in this section need to be thoroughly understood though, because they form the basis for many memory acquisition and analysis tools presented in later parts of this paper.

3. Acquisition of Volatile Memory

Techniques for capturing volatile data are conventionally divided into *hardware-* and *software*-based solutions in the literature (e.g., see Vidas, 2006; Maclean, 2006; Garcia, 2007; Libster and Kornblum, 2008; Vidas, 2010). While the latter ones depend on functions provided by the operating system, hardware-based approaches directly access a computer's memory for creating a forensic image and, therefore, have long been regarded to be more secure and reliable. A publication by Rutkowska (2007) indicates, however, that these assumptions no longer hold true. Moreover, several concepts that have been proposed more recently rely on a combination of both hardware and software mechanisms and cannot be clearly categorized with the existing terminology (Libster and Kornblum, 2008; Halderman et al., 2009). For this reason, we believe that a classification solely on implementation-specific attributes is obsolete and is not capable of properly characterizing the latest developments any more.

A more viable suggestion is assessing the different methods with respect to the requirements that are necessary to obtain a (sound) memory copy of the target machine. Inspired by the work of Schatz (2007a,b), we consider the two factors *atomicity* and *availability* as prevalent. The latter refers to the applicability of a certain technique on arbitrary system platforms for an arbitrary scenario. On the other hand, atomicity intuitively reflects the demand to produce an accurate and consistent image of a host's volatile storage. More precisely, an atomic snapshot is a snapshot obtained within an "uninterrupted" *atomic action* in the sense of a *critical section* as it is used in operating systems and concurrent programming (Lynch et al., 1993). It is free of the signs of concurrent system activity.

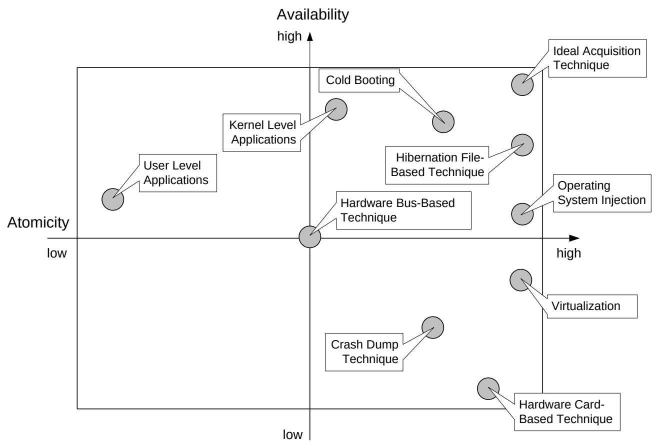

Availability
high
low
Atomicity
low
high
Ideal Acquisition Technique
Cold Booting
Kernel Level Applications
Hibernation File-Based Technique
User Level Applications
Operating System Injection
Hardware Bus-Based Technique
Virtualization
Crash Dump Technique
Hardware Card-Based Technique

Figure 5: Classification of Memory Acquisition Techniques
(Source: Based on Schatz, 2007b)

Based on these two assessment criteria we are able to broadly classify and compare the most common memory acquisition approaches. The corresponding decision matrix is visualized in Figure 5. Please note that while the exact placement of the methods within the fields of the matrix may certainly be subject to discussion at parts and is not the primary intention of the authors, we feel that our illustration may give investigators an insight into when (not) to choose a specific solution. Vital points to keep in mind are in particular:

- An ideal acquisition method is characterized by both a high atomicity and availability and is therefore located in the right upper corner of the matrix.
- Techniques that are listed in the right half of the matrix must generally be favored upon techniques that are grouped on the left side, because they are superior concerning atomicity (and possibly availability as well).
- Methods that are located in the bottom field on the left side of the matrix (currently: none) are generally not suitable for obtaining volatile data in a forensically sound manner and must not be considered further.
- Approaches that are categorized in the right bottom field of the matrix are applicable for scenarios where an investigator has sufficient time for pre-incident preparation.
- Techniques that are listed in the right upper field of the matrix are especially suited for *smoking gun* situations, i.e., where little time between the incident and investigation phase has passed.

In the following, we describe the different approaches in detail and outline their individual benefits, peculiarities, and drawbacks.

3.1. Memory Acquisition Using a Dedicated Hardware Card

One of the first propositions for obtaining a forensic image of a computer's RAM storage was the use of a special hardware card. Carrier and Grand (2004) presented a proof-of-concept (PoC) solution called "Tribble" that makes use of *Direct Memory Access* (DMA) operations to create a copy of physical memory, thereby bypassing possibly subverted structures of the operating system. The card is installed as a dedicated PCI device and is capable of saving volatile information to an attached storage medium. Upon pressing an external switch, the card is activated, and the imaging procedure is initiated. During this process, the CPU of the target host is temporarily suspended to prevent an attacker from executing malicious code and illegitimately modifying the status of the system. Once all operations are completed, control is given back to the operating system, and the acquisition card returns to an idle state again. Two less known but comparable implementations are proposed by Petroni et al. (2004) with their "Copilot" prototype and BBN Technologies (2006) in the form of "FRED", the *Forensic RAM Extraction Device*.

As the described solutions do not rely on functions provided by the operating system, they are assumed to be generally suitable for acquiring an accurate image of volatile memory. Since the processor of the host in question is successfully halted, the imaging operation can be atomically completed without interference by other processes as well. Last but not least, because all information is directly retrieved from physical RAM, the procedure is believed to act outside the view of malicious applications such as rootkits, resulting in a "true" picture of a target's memory (Kornblum, 2006). Rutkowska (2007) has, however, recently proven that it is possible to present a different view of physical memory to the peripherals by reprogramming the chipset of the NorthBridge. Due to this innovation, several authors conclude that hardware cards can no longer be fully trusted and must not be regarded as forensically sound any more, making the development of more robust and reliable memory collecting techniques necessary (Libster and Kornblum, 2008; Ruff, 2008; Rutkowska, 2007).

In addition to these concerns, it is also important to emphasize that a PCI card must be set up *prior* to its use. This characteristic limits its application to special scenarios only. Carrier and Grand (2004, p. 12) clarify that "the device has not been designed for an incident response team member to carry in his toolkit", but "rather needs to be considered as part of a forensic readiness plan". According to the authors, a card is thus most beneficial when installed on (business-)critical servers, "where an attack is likely and a high-stake intrusion investigation might occur". Alternative options for deployment are, for instance, within a *honeypot* environment[2]. Thereby, it is possible to capture volatile information after a machine has been compromised and learn more about the tools, tactics, and motives of adversaries.

[2] A honeypot is "an information system resource whose value lies in unauthorized or illicit use of this resource" (Spitzner, 2003b,a). It acts as an electronic decoy for studying the behavior of (Internet) miscreants (Honeynet Project, 2004).

3.2. Memory Acquisition via a Special Hardware Bus

As an alternative to PCI cards, several authors suggest reading volatile memory via the IEEE 1394 (*FireWire*) bus (Dornseif and Becher, 2004; Becher et al., 2005). A corresponding forensic-related application is described by Piegdon and Pimenidis (2007). While the original code targets Linux and MacOS platforms, Boileau (2006b, 2008), Panholzer (2008), and Böck (2009) demonstrate the feasibility of the approach for different versions of Microsoft Windows. Ruff (2008, p. 84) adds that "any hardware bus can potentially be used for physical memory access". For instance, a proof-of-concept utility that illustrates Direct Memory Access (DMA) operations with the help of the PCMCIA (PC Card) bus was published at the ShmooCon 2006 conference (Hulton, 2006). The sample code has, however, not been published yet.

Retrieving volatile information via the IEEE 1394 bus can address some of the issues we have outlined in the previous section. For instance, the interface is present by default in a great part of systems, especially laptops. However, as Vidstrom (2006) points out, the use of the technique may cause random system crashes and similar reliability problems when accessing regions in the Upper Memory Area (UMA). Other authors have also indicated inconsistencies after comparing created images with raw memory dumps (Carvey, 2007; Boileau, 2006a). For this reason, similar to the hardware card-oriented approach illustrated above, FireWire-based techniques are not sufficiently reliable to obtain a precise copy of a computer's RAM.

3.3. Memory Acquisition with the Help of Virtualization

With the help of *virtualization*, it is possible to emulate complete, isolated, and reliable system environments, so-called *virtual machines*, on top of a host computer (Smith and Nair, 2005). A special software layer, the *virtual machine monitor* (VMM), is responsible for sharing as well as managing and restricting access to the available hardware resources. By emulating replicas of the different physical components, each virtual machine is equipped with its own virtual processor, memory, graphics adapter, network and I/O interface and may run in parallel to other *guest systems*.

One exceptional characteristic of a virtual machine is its capability to be suspended, i.e., to pause its execution process. Thereby, the state of the guest operating system is temporarily frozen, and its virtual memory is saved to hard disk on the underlying host. For instance, in the case of a *VMware*-based machine, all volatile data is saved to a `.vmem` file located in the working directory of the virtual machine (VMware, Inc., 2010). Thus, by simply copying the respective file, an atomically-generated snapshot of main memory can be easily retrieved.

In 2007, (Carvey, 2007, p. 95) pointed out that "virtualization technologies do not seem to be widely used in systems that require the attention of a first responder". With the growing importance of Internet-hosted services, this situation is likely to change though, and investigators will increasingly have to examine incidents on virtual machines. With respect to these scenarios, the memory acquisition approach is forensically sound and only requires little effort.

3.4. Memory Acquisition Using Software Crash Dumps

All versions of Microsoft Windows 2000 and above can be configured to write debugging information, so-called *memory dump files*, to hard disk when the machine unexpectedly stops working (Microsoft Corporation, 2010e). In case of a critical failure, the system state is frozen, and the main memory as well as relevant CPU information are saved in the system root directory for later examination. Preserving the contents of processor registers during this operation is a unique characteristic of this method (Russinovich et al., 2009; Carvey, 2007)[3]. A dump file can then either be opened with the *Microsoft Debugging Tools for Windows* (Microsoft Corporation, 2010a) or be manually analyzed as explained by Schuster (2006a, 2008a).

With regard to a forensic investigation, the system service may be intentionally interrupted by using a third-party application (Microsoft Corporation, 2010g; Open System Ressources, 2009) or the built-in *CrashOnCtrlScroll* feature. In the latter case, the dump is generated upon pressing the keyboard shortcut `Right Ctrl + ScrollLock + ScrollLock` (Microsoft Corporation, 2010d). It is important to note that this capability is disabled by the operating system by default though. Activation requires editing a certain value in the Windows registry (Russinovich et al., 2009), a subsequent reboot[4], and only works with PS/2 keyboards initially. For universal serial bus (USB) devices, a separate hotfix must be installed (Microsoft Corporation, 2010f). Due to these modifications, the applicability of the technique is forensically only suitable in specific situations, analogously to the use of a dedicated PCI card as explained in Section 3.1. Moreover, even though the memory snapshot is atomically created by halting the processing unit, parts of the system pagefile are overwritten during this operation (Russinovich et al., 2009). These changes on the hard disk of the target system are not in accordance with best practices in computer forensics and may possibly lead to a later legal discussion.

3.5. Memory Acquisition with User Level Applications

Especially at the beginning of memory-related forensic research, various software solutions have been published by different third parties to acquire a copy of physical memory directly from user space. A prominent example is *Data-Dumper* (dd) by Garner (2009) which is part of the *Forensic Acquisition Utilities* (FAU) suite. When run with administrative privileges, dd invokes the internal `\\.\Device\PhysicalMemory` section object to create a full memory dump of the target machine (Crazylord, 2002). Due to security reasons, user

[3]We assume the system is configured to save at least a full kernel memory dump. For more information on this subject, please refer to Microsoft Corporation (2010e).

[4]A technique for omitting the mandatory reboot of the operating system was outlined by Ruff (2008).

level access to this object was restricted in Windows Server 2003 and later operating systems though (Microsoft Corporation, 2010b). With regard to the configuration of modern computer platforms, the technique is thus only of very limited value to date.

Rather than creating a (much larger and more time-consuming) forensic duplicate of the *entire* volatile storage of a host, several other utilities permit saving the address space of a single process: With *PMDump*, it is possible to "dump the memory contents of a process to a file" (Vidstrom, 2002). The *Process Dumper* utility (PD) by Klein (2006b) obtains a process's environment and state by retrieving the data and code mappings of an executable. Since all collected information is redirected to standard output by default, the final image can be easily transferred over a remote connection for further investigation, e.g., with *netcat* (nc), a freely-available network administration utility (Craton, 2009)[5].

Both PMDump and Process Dumper also have various drawbacks though: First, they are closed source and use a proprietary data format that make the development of additional parsing tools more difficult. Second, as the programs require the specification of a process ID, a corresponding process listing utility must be run in the first place. This operation further affects the level of contamination on the target host, however.

As the techniques described above neither rely on specific hardware setups or devices nor do they require a pre-defined system configuration, they are generally suitable for most incident scenarios and permit capturing a forensic image even in case a first responder only has little time for preparation. However, while the imaging process is comparatively easy to perform, the approaches still suffer from various inherent weaknesses: First, as we have already indicated, many programs only work on specific operating systems or are bound to specific architectures, e.g. the x86 32-bit platform. Second, as all applications must be loaded into memory before they can be executed, valuable information may be destroyed before it can be preserved. Sutherland et al. (2008) have shown in a study that the impact of the tools on a target machine can be significant. What is worse, other processes are usually not impeded from altering the volatile storage during the imaging operation, resulting in a potentially non-atomic and "fuzzy snapshot" (Libster and Kornblum, 2008, p. 14) of the source data. Last but not least, as all methods depend on functions provided by the operating system, they are vulnerable to subversion. A rootkit might, for instance, deny direct access to the physical memory object or return a modified representation of RAM during the image generation operation in order to evade detection (Vidas, 2006; Bilby, 2006; Sparks and Butler, 2005). Although these types of manipulations are likely to indicate the presence of a malicious software application to the eyes of a skilled analyst (Kornblum, 2006), relying on an untrusted

[5]Netcat does not establish an encrypted communication channel by default. To securely transfer data over the network, the connection can be tunneled over the SSH (*Secure Shell*) protocol. Alternatively, the *cryptcat* implementation may be used as well. For more information on these topics, please see Farmer and Venema (2005).

operating system eventually "decreases the reliability of the evidence" (Carrier and Grand, 2004, p. 6), too.

3.6. Memory Acquisition with Kernel Level Applications

To mitigate the limitations of user space acquisition utilities described in the previous section, software vendors increasingly provide kernel level drivers to create a forensic image of physical memory. Freely-available distributions include Mantech's *Memory DD* (mdd, ManTech CSI, Inc. (2009)), Moonsols' *Windows Memory Toolkit* (MoonSols, 2010), and *Memoryze* by Mandiant (Mandiant, 2010). Commercial alternatives are offered by Guidance Software (*WinEn* which is a part of *EnCase*, Guidance Software (2010)), AccessData (*Forensic Toolkit*, AccessData (2010)), GMG Systems (*KnTDD* which is part of the *KnTTools*, GMG Systems, Inc. (2007)), and HBGary (*Fastdump Pro*, HBGary (2009)). Even though the different solutions are usually available for various versions of the Windows product family, they still cannot overcome the inherent weaknesses we have already outlined. These include manipulations due to concurrently running (system) processes and, consequently, a not sufficiently precise representation of a host's memory as well as the possibility of falling prey to a compromise attempt (see Section 3.5).

In order to cope with these concerns, Libster and Kornblum (2008) propose integrating the capturing mechanism as an operating system module into the system core which is loaded at boot time and invoked by a special keyboard trigger. According to the authors, distinctive characteristics of the module would be the capability to halt active system processes, thereby ensuring atomic operations. In addition, support for several storage dump locations, e.g., a remote network drive or an externally attached media, is encouraged. For increased security, the use of hardware-side, read-only memory flags or encrypted Trusted Platform Modules (TPMs (Group, 2007)) is suggested. These guarantee the integrity of the imaging operation and prevent attacks on the executing code.

3.7. Memory Acquisition via Operating System Injection

Schatz (2007a) has introduced a proof-of-concept application called *Body-Snatcher* which injects an independent operating system into the possibly subverted kernel of a target machine. By freezing the state of the host computer and solely relying on functions provided by the acquisition OS, an atomic and reliable snapshot of the volatile data may be created. The presented prototype is, however, platform-specific to a high degree due to its low level approach and high complexity. In addition, it is limited to single processor mode at the time of this writing, consumes a significant amount of memory, and only supports the serial port for I/O operations. Therefore, even though the concept is very promising, its technical constraints still need to be resolved, before it can be truly applied in real-world situations.

3.8. Memory Acquisition via Cold Booting

Another encouraging approach has been outlined by Halderman et al. (2009). It is based on the observation that volatile information is not immediately erased after powering off a machine, but may still be recovered for a non-negligible amount of time (Chow et al., 2005). By artificially cooling down the respective RAM modules, e.g., with liquid nitrogen, remanence times may even be substantially prolonged. The target machine can then be restarted (*cold booted*) with a custom kernel to access the retained memory in the next step. The usability of this approach has been proven in a number of recent works: Vidas (2010) has published *AfterLife*, a proof-of-concept demonstration that copies the contents of physical RAM to an external storage medium after rebooting. Chan et al. (2008, 2009) implemented a special booting device that revives the host system environment after cutting power and provides the investigator with an interactive shell to retrieve state-related system data. Most of the projects are still in a state of flux though and are not designed to be used in daily practice yet.

3.9. Memory Acquisition Using the Hibernation File

The Windows hibernation file (`hiberfil.sys`) is increasingly regarded as another source of valuable information (Carrier and Grand, 2004; Schatz, 2007a; Libster and Kornblum, 2008; Ruff, 2008; Zhao and Cao, 2009). It is stored in the root directory of the local hard drive and contains runtime-specific information. When the operating system is about to enter an energy-saving sleep mode and is suspended to disk, the system state is frozen, and a snapshot of the existing working set is preserved. Thus, in contrast to many other software-based solutions we have described, investigators are able to capture a consistent and atomically-generated image of volatile data. However, the file is compressed in order to save disk space and uses a proprietary format which has long impeded thorough understanding and applicability in forensic investigations. Details about its structure have been discussed in more recent publications though (Suiche, 2008a). A working prototype that is capable of converting the file to raw dump format as well as reading and writing specific memory sections was developed in the course of the *SandMan* project (Ruff and Suiche, 2007; Suiche, 2008b). At the time of this writing, it has been superseded by *MoonSols* (MoonSols, 2010). The commercial version supports the entire family of Windows operating systems, including both 32- and 64-bit versions.

4. Analysis of the Acquired Memory Image

After a forensic copy of physical memory has been generated with one of the techniques outlined in the previous sections, an in-depth analysis of the acquired data can begin. Primitive approaches that are described in the literature rely on simple string searches, e.g., with command line utilities such as `strings` and `grep` or more powerful applications such as *WinHex* (X-Ways Software Technology AG, 2010), to look for suspicious patterns, usernames, passwords, and other textual representations in the created image (Stover and Dickerson, 2005; Zhao and Cao, 2009). While these methods are easy to apply, they are also noisy, cause a huge overhead, and lead to a large number of false positives (Beebe and Dietrich, 2007). Beebe and Clark (2007, p. S49) argue that "[f]requently, investigators are left to wade through hundreds of thousands of search hits for even reasonably small queries [...] - most of which [...] are irrelevant to investigative objectives". Even when string searches produce quite accurate results, they typically do not take into account the context of the respective information and, as a consequence, are of limited help to the investigation process. For instance, as Savoldi and Gubian (2008, p. 16) point out, "the retrieval of a potentially compromising string (e.g. 'child porn') certainly provides evidence if found in the memory assigned to a process l[a]unched by the user, but it would be likely rejected by jury if that memory belonged to a piece of malware". In addition, Hejazi et al. (2009, p. S123) note that the "existence of unknown sensitive data in the memory is the main important limitation of this method". Thus, a forensic analyst may, for example, solely look for specific keywords, but at the same time, disregard "names, addresses, user IDs, and strings that are not present in the list the investigator is looking for while they are present in the memory dump and are of paramount importance for the investigative case" (Hejazi et al., 2009, p. S123). In sum, these procedures are not sufficently reliable with regard to a forensic investigation. Efficient solutions to these problems have been developed over the last years but are not further illustrated in the remainder of this paper since they fall in the area of information retrieval (IR) and text mining tasks. Good introductions to these topics have been compiled by Beebe and Clark (2007) as well as Roussev and Richard III (2004) though.

As a viable alternative to string searching algorithms, security professionals recommend a more *structured* methodology to find valuable traces in memory. It involves examining *what types* of data may be contained within a captured image, *how* these types are defined, and *where* they are located. Relevant pieces of information are generally stored in the system or user address space, either directly in RAM or in the local page file, and include (Hoglund, 2008; Sutherland et al., 2008)

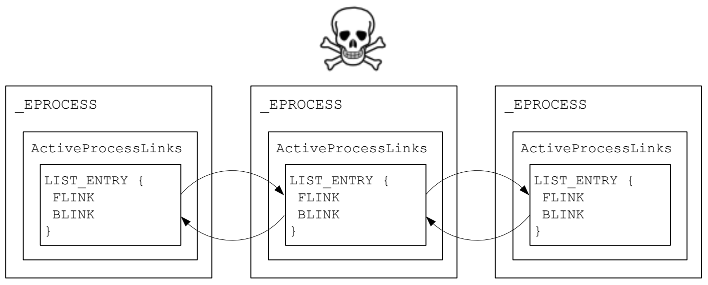
- the list of running system processes and possibly installed malicious software applications,
- cryptographic keys,
- the system registry,
- established network connections and network-related data such as IP addresses and port information,
- open files, and
- system state- and application-related data such as the command history, date and timestamp information.

The prevailing techniques to recover the specified artifacts are subject of the following sections.

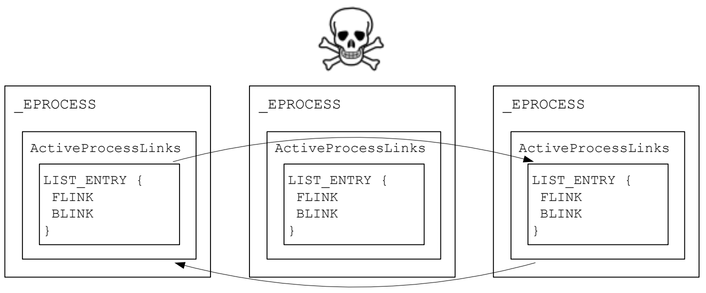
4.1. Process Analysis

An essential part of a forensic investigation is checking the integrity of the operating system on the suspect machine and distinguishing legitimate components from suspicious and potentially malicious applications. Thus, the identification and sound examination of running processes becomes “a basic security task that is a prerequisite for many other types of analysis” (Dolan-Gavitt et al., 2009, p. 2).

Early approaches attempted to parse internal system resources to enumerate the list of loaded programs (Burdach, 2005; Betz, 2006; Garner, 2007). However, as various authors point out, an increasing number of malware executables use so-called *rootkit* techniques and subvert integral system structures to avoid detection (Aquilina et al., 2008; Ligh et al., 2010). For instance, the *FU* rootkit (Butler, 2005) implements a method known as *Direct Kernel Object Manipulation* (DKOM) to unlink itself from the `ActiveProcessLinks` list, a member of the `_EPROCESS` (*executive process*) block every process is represented by (Bilby, 2006). This operation is illustrated in Figure 6(b), the skull icon over the second structure indicates the malicious rootkit process. Due to these interventions, list and table walking solutions are likely to fail and must not be seen as reliable.



(a) Structure of the Process List Before a DKOM Attack



(b) Structure of the Process List After a DKOM Attack

Figure 6: Subversion of a Process List Using *Direct Kernel Object Manipulation* (DKOM)

In order to cope with these issues, Schuster (2006d) developed a signature-based scanner. It uses a set of rules that precisely describe the structure of a system process or thread, respectively. A sample signature is illustrated in Listing 1. In the given example, it is checked that the page directory is aligned at a page boundary, the control structures of the working thread are contained in kernel space, and the fields of the `DISPATCHER_HEADER` structure match predefined values.

The results of the scanner can be compared with the output of the standard process list in the next step. Differences and anomalies potentially indicate the presence of a malicious program and need to be inspected in more detail. Similar strategies are proposed by Walters and Petroni (2007), Carr (2006), and Rutkowska (2005). Walters and Petroni (2007, p. 14) note, however, that "a reliable pattern may rely on characteristics of an object that are not essential". Consequently, an adversary may change the value of a *non-essential* field of the process object without affecting the stability of the underlying operating system. For instance, in the case of Listing 1, it is also possible to set the value of the `Size` field to zero. Thereby, the rule of the scanner is circumvented, and the respective process becomes invisible. To prevent such scenarios, Dolan-Gavitt et al. (2009) have created so-called *robust* signatures which are solely based on fields that are critical to system functionality. A manipulation of such a field leads to an automatic system crash and, thus, renders an attack useless. With regard to the core `_EPROCESS` structure outlined above, 72 fields (out of 221) suffice this requirement and, therefore, form strong candidates for a process signature[6].

An alternative method is suggested by Zhang et al. (2009, 2010). The authors use a combination of scanning and list traversing techniques that rely on the *Kernel Processor Control Region* (KPCR). This region stores processor-specific data and contains a separate block, the *KPRCB*, that, for instance, saves CPU-related statistics as well as scheduling information about the current and next thread (Russinovich et al., 2009). With the help of this information, the corresponding process list can be restored as follows (Barbosa, 2005; Ionescu, 2005; Dolan-Gavitt, 2008b): The KPCR includes a field `KdVersionBlock` that points to a `_DBGKD_GET_VERSION64` structure that, in turn, references the undocumented `_KDDEBUGGER_DATA64` structure. Part of this structure is the global variable `PSActiveProcessHead` that serves as a pointer to the list of active `_EPROCESS` structures.

In Microsoft Windows XP, both the KPCR and KPRCB are located at fixed addresses (`0xFFDFF000` and `0xFFDFF120`, respectively). In later versions, the base addresses are dynamically computed. Since the KPCR structure is self-referencing (offset `0x1C`) and the KPRCB starts at offset `0x120`, a simple signature can be generated though to discover the structures and enumerate the running programs in the next step (Aumaitre, 2009).

[6]After performing a fuzzing test, the number of strong signature candidates was reduced from 72 fields to 43 (Dolan-Gavitt et al., 2009).

```
PageDirectoryTable !=0
PageDirectoryTable % 4096 == 0
(ThreadListHead.FLink > 0x7FFFFFFF) &&
   (ThreadListHead.Blink > 0x7FFFFFFF)
DISPATCHER_HEADER.Type == 0x03
DISPATCHER_HEADER.Size == 0x1b
```

Listing 1: Example of a Process Signature (Schuster, 2006d)

4.2. Cryptographic Key Recovery

Due to the spread of (freely) available encryption technologies, security professionals are likely to increasingly encounter secured and password-protected files and disks in the future (see Section 1). In case an investigator is unable to obtain the desired information from the suspect, e.g., through social engineering, the restoration of cryptographic keys from volatile memory may therefore become a central task in the course of a forensic analysis.

To solve this problem, different methods are proposed in the literature: Hargreaves and Chivers (2008) describe a linear memory scanning technique that moves through RAM one byte at a time, using a block of bytes as the possible decryption key for the volume in question. The brute force-oriented procedure does not require a deep understanding of the underlying operating system and, thus, can be easily generalized according to the authors. However, it cannot be directly applied if the key is split, i.e., is not stored in a contiguous pattern in memory.

Shamir and van Someren (1999) seek for sections of high entropy to locate an RSA key within "gigabytes of data". The solution exploits the mathematical properties of the cryptographic material. In contrast, the attack described by Klein (2006a) is based on the observation that both private keys and certificates are stored in standard formats. Klein (2006a) constructs a simple search pattern to easily dump the secret information from RAM. A pattern-like approach is also implemented by Kaplan (2007). His idea stems from the fact that, for reasons of security, cryptographic keys are generally not paged out to disk and, as a result, are typically saved in the *non-paged pool* of the operating system (see Section 2.1). The non-paged pool consists of a range of virtual addresses that always reside in physical memory (see also Russinovich et al., 2009; Schuster, 2006b, 2008b). Since in kernel space, memory is a shared resource, allocated regions may be associated with an identifier, the so-called *pool tag* (Microsoft Corporation, 2010c). Consequently, "a cryptosystem-specific signature, consisting of the driver specific pool tag and pool allocation size are all that is necessary to extract pool allocations containing key material from a memory dump with an acceptably small number of false positives" (Kaplan, 2007, p. 18). As an alternative to the non-paged pool, it is possible to prevent regions of virtual address space from being written out to disk by calling the `VirtualLock` function. Thereby, "[p]ages that a process has locked remain in physical memory until the process unlocks them or terminates" (Microsoft Corporation, 2010h). Whether these mechanisms are relevant to the forensic recovery of cryptographic keys has not been investigated to the best knowledge of the authors though.

Walters and Petroni (2007) outline a different concept that relies on an analysis of publicly available source code. They identify internal data structures that are responsible for holding the master key. As Maartmann-Moe et al. (2009, p. S133) point out, Walter and Petroni "do, however, not describe how to locate the different structures in memory, and neither do they discuss the fact that some of these may be paged out, thereby breaking the chain of data structures that leads to the master key if only the memory dump is available for analysis".

Last but not least, Halderman et al. (2009) suggest parsing a computer's memory for key schedules. The authors leverage remanence effects in DRAM modules and launch a *cold boot* attack on the target machine (see Section 3.8). After loading a custom operating system, the volatile data is extracted, the key material is retrieved, and the hard drive automatically decrypted. The performance of this process is improved in the works of Heninger and Shacham (2009). Likewise, Tsow (2009) presents an algorithm that is capable of recovering cryptographic information from memory images that are significantly decayed and is magnitudes faster than the original method. Finally, Maartmann-Moe et al. (2009) extend the conducted research on additional ciphers and illustrate the vulnerability of several well-known whole-disk and virtual-disk encryption utilities for these types of side-channel operations. However, it is important to note that these attacks can be mitigated by implementing cryptographic algorithms entirely on the microprocessor. A corresponding proof-of-concept application has been published by Müller et al. (2010) and has been further sophisticated more recently (Müller et al., 2011).

4.3. System Registry Analysis

The Windows *registry* is a central, hierarchically-organized repository for configuration options of the operating system and third party applications (Microsoft Corporation, 2008). It is internally structured into a set of so-called *hives*, i.e., discrete, treelike databases that hold groups of registry keys and corresponding values (see also Russinovich et al., 2009). Most registry hives (e.g., `HKLM\SYSTEM`, `HKLM\SOFTWARE`) are persistently stored in the `\system32\config` folder of the operating system[7]. However, a number of *volatile* hives (e.g., `HKLM\HARDWARE`) are entirely maintained in RAM only and are created every time the system is booted.

While the examination of on-disk registry data is a quite established procedure for a forensic investigation to find possible pieces of evidence (Carvey, 2005; Mee et al., 2006; Chang et al., 2007), a solely memory-based approach has been documented by Dolan-Gavitt (2008c) only recently. In the following, we give a short overview about the prevailing techniques used in his work.

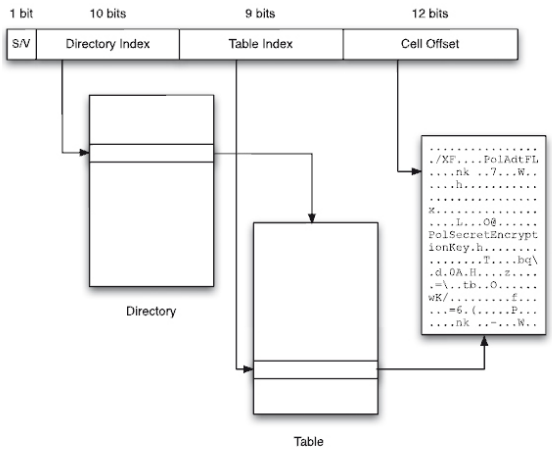
[7]User-specific settings are saved in the file `NTuser.dat` that is located in the `%SystemDrive%\Documents and Setttings\<username>` folder.

A registry hive consists of a so-called *base block* and a number of *hive bins* (hbins). The base block with a fixed size of 4 KB defines the start of the hive and contains a unique signature (`regf`) as well as additional meta information such as a time stamp that saves the time of the last access operation, an index to the first key, the *RootCell*, the internal file name of the hive, and a checksum. A hive bin is typically 4 KB wide (or a multiple of it) and serves as a container for *cells* that, in turn, store the actual registry data. Thus, to read a certain key or value from RAM, the correct hive and corresponding cell must be found first. The former task can be quite easily solved by creating a hive-specific signature: Internally, a hive is represented by a `_CMHIVE` structure which is allocated from the paged pool of the system (see Section 2.1) and referenced by the tag `CM10`. Furthermore, it embeds a sub-structure `_HHIVE` with the string constant `0xbee0bee0`. These pieces of information are sufficient to construct a unique search pattern and locate a hive in memory. To enumerate the entire list of loaded hives, the `HiveList` member structure must be simply followed in the last step. It acts as a link between the individual registry repositories.

Retrieving pre-defined keys or values from a memory image is slightly more complex: Dolan-Gavitt (2008a) notes that because "space for the hives is allocated out of the paged pool, there is no way of guaranteeing that the memory allocated for the hive will continue to be contiguous". For this reason, a strategy is used that is similar to the virtual-to-physical address translation mechanism we have described in Section 2.2. A cell is generally referenced by a *cell index*. This field can be split into four different components. The first element, a one-bit flag, determines the stable (on-disk) or volatile *storage map* for the hive which are both saved in the `_HHIVE` sub-structure. With the help of the 10-bit directory index, the correct entry in the cell map directory can be located. The cell map directory refers to a cell map table with 512 (2^9) entries that, in turn, point to the virtual address of the target bin. Using the third and fourth component of the cell index, the correct cell and cell information can be finally discovered (see Figure 7). Dolan-Gavitt (2008e) has published a proof-of-concept utility that is capable of extracting the list of open keys and displaying the corresponding registry data. The software is freely available for download and integrated into the *Volatility* analysis framework that we will introduce in more detail in a later section of this paper. It is, however, important to emphasize that the techniques described above only supplement traditional investigation methods. As later versions of Microsoft Windows map only 16-KB portions of a hive into RAM when they are needed (Russinovich et al., 2009), it is possible that "parts of the registry may have never been brought into memory in the first place" and, consequently, "it cannot be assumed that the data found in memory is complete" (Dolan-Gavitt, 2008c, p. S30).

4.4. Network Analysis

Analyzing open network connections as well as inspecting incoming and outgoing network traffic is an integral part of a forensic investigation and becomes particularly important in the face of a potentially compromised system. Malicious applications typically bind to pre-defined ports and enable attackers to



```
................
./XF....PolAdtFL
....nk ..7...W..
....h...........
................
x...............
....L...O@......
PolSecretEncrypt
ionKey.h........
........T....bq\
.d.0A.H....z....
.=\..tb..O......
wK/.........f...
...=6.(.....P...
....nk ..-...W..
```

Figure 7: Structure of a Cell Index
(Source: Dolan-Gavitt, 2008c, p. S28)

execute arbitrary commands on the target machine, disable security protections, and upload or download files (Aquilina et al., 2008; Ligh et al., 2010). Contemporary examples also include launching *Distributed Denial of Service* (DDoS) attacks and consuming the resources of a host in order to degrade its performance (Peng et al., 2007). A vast number of utilities, either for live response (e.g., *TCPView* (Russinovich, 2010), *FPort* (Foundstone, Inc., 2000)) or post-mortem examination (e.g., *PyFlag* (Cohen, 2008)), have been developed to reveal such threats. Since many network-related data are also temporarily stored in RAM, memory forensic techniques can help correlate results with the tools described above in order to gain a thorough understanding of an incident.

The first solution was suggested by Schuster (2006b). He scans the non-paged pool of the operating system to find allocations for listening sockets. His algorithm is based on a unique pool tag (`TCPA`, saved in little-endian format) and a pre-defined pool size (368 bytes) that can be recovered after disassembling the `tcpip.sys` driver where the respective operating system functions are implemented in. Once the appropriate addresses have been identified, the socket list can be easily created. With a similar procedure, it is possible to retrieve the list of open network connections.

A different methodology is described by Ligh et al. (2010) and Okolica and Peterson (2010): The authors locate two internal hash tables (`_AddrObjTable`, `_TCBTable`) in the `tcpip.sys` driver file. Each of the hash tables references a singly linked list of objects that include information about the IP address and port bindings[8]. By traversing the lists as shown in Figure 8, existing sockets and open network connections can be enumerated.

[8]Details about the structure of these objects can be found in the work of Ligh et al. (2010).

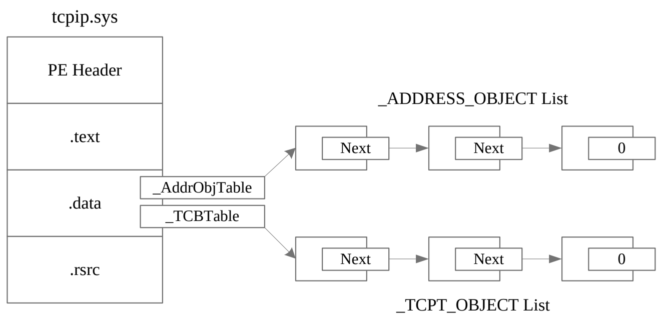


Figure 8: Enumerating the List of Socket and Connection Objects
(Source: Based on Ligh et al., 2010, p. 681)

It is interesting to note that a malicious manipulation of these lists, similar to the DKOM attack described in Section 4.1, appears to hamper system functionality and, thus, is likely to be detected in practice. For demonstration purposes, Ligh et al. (2010) manually unlinked an object to hide the presence of a listening socket. This operation disrupted the communication process though, and connections could no longer be initiated. From a forensic point of view, list crawling-based approaches can therefore be seen as reliable to date and permit a legit and unaltered view of network activities.

4.5. File Analysis

Similar to the examination of running system processes (see Section 4.1), inspecting the list of open files and dynamically loaded libraries (DLLs) a program is referencing may be crucial to reveal suspicious activities in the course of an investigation. For instance, an adversary might inject a malicious DLL in the address space of a legitimate process by exploiting a remote vulnerability to hide her presence on the target machine (Miller and Turkulainen, 2006; Walters, 2006). As Walters (2006, p. 2) points out, these techniques "thwart conventional forms of filesystem forensic analysis" and "are commonly labeled as 'anti-forensic', since data is not written to any non-volatile storage and [...] would typically be lost during common incident response procedures". To counter such threats, security professionals recommend analyzing the *Process Environment Block* (PEB), a structure which lives in the user address space and is part of every process (see Russinovich et al., 2009). The Process Environment Block contains a `Ldr` member with three doubly linked lists that save the full name, size, and base address of all loaded libraries[9]. Simply enumerating the individual lists may thus suffice to identify an injection attack. It is critical

[9]The three doubly linked lists (`InLoadOrderModuleList`, `InMemoryOrderModuleList`, `InInitializationOrderModuleList`) contain the same modules but in different order. For more information, please refer to Russinovich et al. (2009).

to stress though that the approach can be subject to manipulation as various rootkit authors have demonstrated (Darawk, 2005; Kdm, 2004). For this reason, Dolan-Gavitt (2007a) proposes an alternative methodology which is based on Virtual Address Descriptors (VADs).

A *Virtual Address Descriptor* is a kernel data structure that is maintained by the memory manager to keep track of allocated memory ranges (Russinovich et al., 2009). It stores information about the start and end of the respective addresses as well as additional access and descriptor flags. Each time a process reserves address space, a VAD is created and added to a self-balancing tree for maintenance reasons. A node in the tree is associated with a pool tag and is of type `_MMVAD_SHORT` ("VadS"), `_MMVAD` ("Vad"), or `_MMVAD_LONG` ("Vadl") (Dolan-Gavitt, 2007a). The latter two store a pointer to a `_Control_Area` structure that, in turn, points to a `_File_Object` that holds the unique file name (see the lower half of Figure 9). Consequently, the entire list of loaded modules can be retrieved by traversing the VAD tree from top to bottom and following the corresponding `_Control_Area` and `_File_Object` references[10]. Furthermore, it is possible to reconstruct the contents of memory-mapped files, most importantly, executables. While it is unlikely to completely rebuild the exact form of a binary due to changes in code variables at runtime (Kornblum, 2007a), the recovered copy can frequently be reverse engineered and inspected upon malicious behavior (see also Schuster, 2006c). These operations supplement traditional file carving techniques that are implemented in utilities such as *Foremost* (United States Air Force Office of Special Investigations, 2001) and *Scalpel* (Golden G. Richard III, 2006) and are suited for scenarios where an on-disk restoration has failed. A proof-of-concept application has been published by Dolan-Gavitt (2007b) and has been integrated into the analysis framework *Volatility* that we will describe in depth in Section 5. Van Baar et al. (2008) have extended these works and developed a prototype that is capable of finding file remnants in RAM even if the associated process has terminated.

In spite of these features, it is important to emphasize that the VAD tree itself may be the target of a compromise attempt. An adversary with system privileges might, for instance, remove a node from the tree, similar to the DKOM attack presented in Section 4.1, or overwrite the pointer to the `_Control_Area` structure, thereby effectively hiding a malicious artifact from the forensic practices described above (Dolan-Gavitt, 2007a; Ligh et al., 2010). Even though we are not aware of any malware species that leverage these types of modifications, sample code has been written that proves the deficiencies of many security products offered on the market to date (NT Internals, 2009). Therefore, investigators should evaluate data from various sources to obtain views from multiple angles of the compromised system.

[10]The root of the tree is saved in the `VADRoot` member of a process's `_EPROCESS` structure. Strategies to find this structure in memory are explained in Section 4.1.

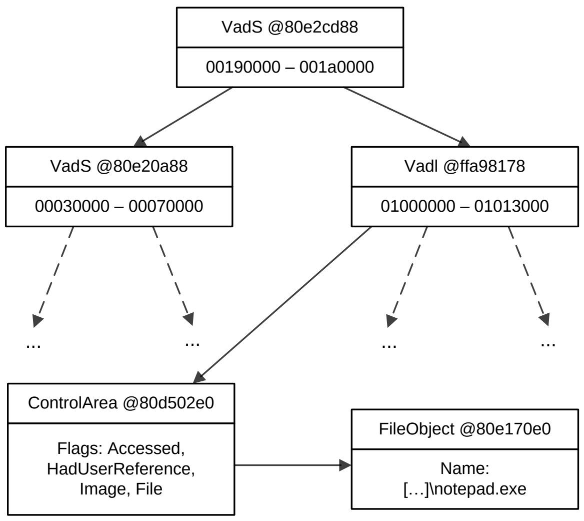


Figure 9: Virtual Address Descriptor (VAD) Tree
(Source: Based on Dolan-Gavitt, 2007a)

4.6. System State- and Application-Specific Analysis

In addition to the pieces of evidence an investigator may collect with the techniques described in the previous sections, a memory image frequently contains a myriad of information about the system state that can be of great benefit to an investigation. Especially the aforementioned `_EPROCESS` block is a source of valuable data. For instance, the `StartTime` and `ExitTime` fields indicate the start and respective end time of a process and may be parsed to create forensic timelines. In addition, the periods an application has spent in system and user mode can be derived from members of the kernel process (`_KPROCESS`) block, a structure which is part of the `_EPROCESS` environment (see Russinovich et al., 2009). Schuster (2008b) has proven that these types of artifacts may be recovered from RAM even after a program has terminated for more than 24 hours. Furthermore, with the help of the `Token` member, it is possible to reconstruct the *security context* of an application. As Russinovich et al. (2009, p. 473) explain, "a security context consists of information that describes the privileges, accounts, and groups associated with the process or thread". Of particular interest is the list of user and group SIDs (*Security Identifiers*) that eventually reveal the name of the user and corresponding group account the executable was run as (see Dolan-Gavitt, 2008d)).

A different research focus is set by Stevens and Casey (2010). They dissect the structure of the `DOSKEY` utility that is integrated into the command shell and permits editing past commands as well as displaying the command history. The latter is solely maintained in memory and is only accessible as long as the command prompt is open. As Stevens and Casey (2010, p. S58) point out, "[i]n practice, this makes recovering the command history difficult due to its volatile nature and the low likelihood of finding an open command window during an investigation", even though major parts "may be recoverable from memory for some time after the windows has been closed". The authors generate a unique signature for various `DOSKEY` elements and succeed in restoring command history objects from a number of reference data sets, including intact lists of entered commands. As these "might contain the only retrievable traces of a deleted file or suspect activity", such types of examinations support other system state-oriented investigation methods and "can provide significant context into how and what occurred on [a] system" (Stevens and Casey, 2010, p. S57).

While most approaches we have described so far target the operating system architecture, "one of the issues currently faced in the analysis of physical memory is the recovery and use of application-level data" (Simon and Slay, 2009, p. 996). As Simon and Slay (2010) argue, these steps become necessary due to the increasing spread of anti-forensic technologies and a trend towards online and network applications. However, research results in this area still remain sparse to date which may be partially due to the low lifespan of data loaded in the user address space (Solomon et al., 2007). Published solutions mainly comprise instant messaging and Voice over IP (VoIP) applications (Gao and Cao, 2010; Simon and Slay, 2010). Other security professionals have concentrated on finding remnants of social communication platforms in RAM (Bryner, 2009; Drinkwater, 2009). With the growing use of these services, additional and more advanced techniques are likely to be developed in the future.

5. Volatility: A Memory Analysis Framework

While most memory analysis utilities we have described in the previous sections sophistically solve a specific problem, they were typically not designed with a holistic forensic process in mind but rather as a proof-of-concept demonstration. As a consequence, many programs have their own user interface, must be invoked with different command line options, and generally neglect inter-process communication with other applications. In addition, some tools are OS-dependent and only work with specific platforms and configurations. Vidas (2006) notes that, for instance, the structure of the `_EPROCESS` block significantly differs across operating system versions and service pack levels, rendering hard-coded address offsets in process scanners ineffective. In sum, these characteristics force security professionals to thoroughly understand the scope and limitations of a myriad of individual solutions, and great efforts must be made to correlate results and generate homogenous reports. Case et al. (2008, p. S65) argue, "[a]s the complexity of systems grows rapidly, it becomes ever more difficult for an investigator to perform thorough, reliable, and timely investigations".

In order to cope with these issues, Walters and Petroni (2007) suggest integrating memory forensic techniques with the digital investigation process model proposed by Carrier and Spafford (2003). As Walters and Petroni (2007, p. 2) point out, the model "allows us to organize the way we think about, discuss, and implement the procedure that are typically performed during an investigation" and "forces us to think about all phases of an investigation and how the tools and techniques that are used during a digital investigation fit together to support the process". Their work builds the foundation for the forensic framework *Volatility* (Volatile Systems, LLC, 2008) which, in turn, has been embedded in several other application suites such as *PyFlag* (Cohen, 2008), *DFF* (ArxSys, 2009), or *PTK* (DFlabs, 2010). Haruyama (2009) created a collection of scripts to port Volatility to the commercially distributed *Encase* environment.

With respect to its architecture, Volatility consists of several core modules that are written in Python. The functionality of the application can be further extended with plug-ins that are provided by the developer community[11]. Even though earlier releases were solely capable of examining images of Microsoft Windows XP, the most recent branch (`1.4`) also supports current operating systems, including Windows Vista and Windows 7. An overview of important program features can be seen in Appendix A, the complete list of modules can be found in the work by Ligh et al. (2010) or when running Volatility with the `-h` switch.

In sum, the framework implements a great part of the concepts and methods we have outlined in this paper and has already been successfully leveraged in different scenarios (DFRWS, 2008; Smith and Cote, 2010). Its architecture makes it suitable for a high degree of memory forensic-related tasks. However, it is also important to note that a high level of expertise is required to interpret extracted information and correlate results. Therefore, at the time of this writing, Volatility predominantly aims at academic researchers and security professionals with a strong technical background who are able to adapt the framework to their needs on a case-specific basis.

6. Conclusion and Future Work

The volatile storage of a computer contains a plethora of valuable information that may be key in the course of an investigation. Data found in RAM or in the system page file may significantly contribute to the reconstruction of an incident and therefore, supplement hard disk- and persistent media-oriented approaches in computer forensics. For this reason, best practices should generally include preserving those artifacts for later examination, correspondent to the guidelines as proposed by Brezinski and Killalea (2002). In sum, "during the forensic process as much attention must be paid to volatile memory as is paid to the more traditional sources of evidence" (Maclean, 2006, p. 29).

We have illustrated the prevalent concepts for creating a memory snapshot of a running machine. However, security professionals must also be aware of the scope and limitations of the different technologies. Understanding the individual benefits and drawbacks is crucial for choosing an adequate acquisition strategy. Particularly the impact of software-based solutions has been assessed

[11] A comprehensive list of available plug-ins is maintained by members of the *Forensics Wiki* project (Forensics Wiki, 2010).

only marginally at the time of this writing and must be evaluated in more detail in the future.

The documentation and analysis of evidence found in memory has received broad attention in research more recently. As we have outlined in Section 5, existing solutions are mainly *problem-oriented* to date though and frequently fail to provide appropriate interfaces for data import/export tasks and correlating results. The *Volatility* framework we have briefly outlined in the previous section addresses some of these issues and is capable of processing images in raw, crash dump, and hibernation file format. Due to its modular design, the functionality of the application can be easily extended if necessary. Case et al. (2008, p. S65) emphasize, however, that "merely swamping the user with all available data [...] falls well short of what is actually needed" and "is not, by itself, very useful". Thus, it is also important to develop suitable visualization techniques in order to properly present the collected evidence. Only then will analysts fully recognize memory forensics as "a critical component of the digital crime scene" (Walters and Petroni, 2007, p. 15) and not as an additional burden that impedes the timely and cost-effective completion of a case.

Acknowledgments

We would like to thank Andreas Schuster for reading a previous version of this paper and giving us detailed and helpful suggestions for improvements. We would also like to thank Tilo Müller as well as Thomas Schreck for additional feedback on our work.

A. Integral Modules of the *Volatility* Memory Forensics Framework

Module Name	**Description**
`connections`	Locates the `_TCBTable` hash table in the `tcpip.sys` driver file and traverses the singly linked list of `_TCPT_OBJECT` entries to enumerate open network connections on the target system (see Section 4.4).
`connscan2`	Scans the non-paged pool of the operating system for allocations that contain information about open network connections (see Section 4.4).
`dlllist`	Retrieves the base address, size, and path of all dynamically loaded libraries (DLLs) that are referenced by a running application. For this operation, the `Ldr` structure of the *Process Environment Block* (PEB) is parsed. It stores three doubly linked lists of `_LDR_DATA_TABLE_ENTRY` types that hold the respective information (see Section 4.5).
`files`	Retrieves the list of open file handles that is maintained by a process (see Section 4.5).
`getsids`	Reconstructs the security context of a process to retrieve the list of user and group SIDs (*Security Identifiers*) the application is associated with (see Section 4.6).
`hivelist`	Prints the virtual address and name of hive structures that internally represent parts of the system registry (see Section 4.3).
`hivescan`	Uses a signature-based approach to search for `_CMHIVE` structures in memory. These data types internally represent parts of the system registry (see Section 4.3).
`imageinfo`	Displays various meta data about the image, including the image type as well as the creation date and time.
`kpcrscan`	Scans a memory image for a `_KPCR` structure that defines the *Kernel Processor Control Region*. With the help of the structure, it is for instance possible to enumerate the list of running processes on a machine (see Section 4.1).
`modscan2`	Scans a memory image for `_LDR_DATA_TABLE_ENTRY` objects that save the base address, size, and path of all dynamically loaded libraries (DLLs) that are referenced by a process (see Section 4.5).
`procexedump`	Creates an executable (`.EXE`) file of a process. While the recovered file is unlikely to run due to changes in its `.data` section, the binary can frequently be successfully disassembled for further analysis and reveal malicious activities (see Section 4.5).
	Table continues on the following page.

Module Name	Description
`pslist`	Follows the `ActiveProcessLinks` list that is part of the `_EPROCESS` block to enumerate running processes on the system (see Section 4.1).
`psscan`	Applies a signature-based search algorithm to locate `_EPROCESS` structures within a memory image and reveal potential *Direct Kernel Object Manipulation* (DKOM) attacks (see Section 4.1).
`pstree`	Generates a tree-like view of running system processes. The tree is derived from the output of the `pslist` command (see Section 4.1).
`regobjkeys`	Enumerates the list of open registry keys that are referenced by a process (see Section 4.3).
`sockets`	Finds the `_AddrObjTable` hash table in the `tcpip.sys` driver file and follows the singly linked list of `_ADDRESS_OBJECT` entries to enumerate listening sockets on the target system (see Section 4.4).
`sockscan`	Scans the non-paged pool of the operating system for allocations that contain information about open network sockets (see Section 4.4).
`vaddump`	Traverses the *Virtual Address Descriptor* (VAD) tree of a process and dumps the allocated memory segments to a file (see Section 4.5).
`vadinfo`	Prints detailed information about a process's *Virtual Address Descriptor* (VAD) tree that contains the range of memory addresses that are allocated by the application as well as references to loaded modules and memory-mapped files (see Section 4.5).
`vadtree`	Creates a graphical structure of a process's *Virtual Address Descriptor* (VAD) tree (see Section 4.5).

Table A.1: Integral Modules of the *Volatility* Memory Forensics Framework

References

AccessData. 2010. Forensic Toolkit 3. http://www.accessdata.com/forensictoolkit.html.

Aggarwal, G., Bursztein, E., Jackson, C., and Boneh, D. 2010. An Analysis of Private Browsing Modes in Modern Browsers. In *Proceedings of the USENIX Security Symposium.*

Apple Inc. 2010. Encrypting Your Home Folder with FileVault. http://docs.info.apple.com/article.html?path=Mac/10.5/en/8736.html.

Aquilina, J. M., Casey, E., and Malin, C. H. 2008. *Malware Forensics: Investigating and Analyzing Malicious Code.* Syngress Publishing.

ArxSys. 2009. DFF: Digital Forensics Framework. http://www.arxsys.fr/.

Aumaitre, D. 2009. A Little Journey Inside Windows Memory. *Journal in Computer Virology 5,* 2, 105–117.

Barbosa, E. 2005. Finding some Non-Exported Kernel Variables in Windows XP. http://www.rootkit.com/vault/Opc0de/GetVarXP.pdf.

BBN Technologies. 2006. FRED: Forensic RAM Extraction Device. http://www.ir.bbn.com/~vkawadia/.

Böck, B. 2009. Firewire-Based Physical Security Attacks on Windows 7, EFS and BitLocker. http://www.securityresearch.at/publications/windows7_firewire_physical_attacks.pdf.

Becher, M., Dornseif, M., and Klein, C. N. 2005. FireWire - All Your Memory Are Belong To Us. In *Proceedings of the Annual CanSecWest Applied Security Conference.*

Beebe, N. and Dietrich, G. 2007. A New Process Model for Text String Searching. In *Advances in Digital Forensics III.* IFIP International Federation for Information Processing. Springer Boston, 179–191.

Beebe, N. L. and Clark, J. G. 2007. Digital Forensic Text String Searching: Improving Information Retrieval Effectiveness by Thematically Clustering Search Results. *Digital Investigation 4,* S1, 49–54.

Betz, C. 2006. Memparser Analysis Tool. http://sourceforge.net/projects/memparser/.

Bilby, D. 2006. Low Down and Dirty: Anti-Forensic Rootkits. In *Proceedings of Ruxcon 2006.*

Boileau, A. 2006a. FireWire, DMA & Windows. http://www.storm.net.nz/projects/16.

Boileau, A. 2006b. Hit by a Bus: Physical Access Attacks with Firewire. In *Proceedings of Ruxcon 2006.*

Boileau, A. 2008. winlockpwn. http://www.storm.net.nz/static/files/winlockpwn.

Brezinski, D. and Killalea, T. 2002. RFC3227 - Guidelines for Evidence Collection and Archiving. http://www.faqs.org/rfcs/rfc3227.html.

Brosch, T. and Morgenstern, M. 2006. Runtime Packers - The Hidden Problem? http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.

Bryner, J. 2009. Facebook Memory Forensics. http://computer-forensics.sans.org/blog/2009/11/20/facebook-memory-forensics#.

Burdach, M. 2005. An Introduction to Windows Memory Forensic. http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf.

Butler, J. 2005. FU Rootkit. http://www.rootkit.com/board_project_fused.php?did=proj12.

Carr, C. 2006. GREPEXEC: Grepping Executive Objects from Pool Memory. http://uninformed.org/?v=4&a=2&t=pdf.

Carrier, B. and Spafford, E. H. 2003. Getting Physical with the Digital Investigation Process. *International Journal of Digital Evidence 2,* 2, 1–20.

Carrier, B. D. and Grand, J. 2004. A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation 1,* 1, 50–60.

Carvey, H. 2005. The Windows Registry as a Forensic Resource. *Digital Investigation 2,* 3, 201–205.

Carvey, H. 2007. *Windows Forensic Analysis*. Syngress Publishing.

Case, A., Cristina, A., Marziale, L., Richard, G. G., and Roussev, V. 2008. FACE: Automated Digital Evidence Discovery and Correlation. *Digital Investigation 5,* 1, S65–S75.

Casey, E. 2004. *Digital Evidence and Computer Crime - Forensic Science, Computers, and the Internet*, 2nd ed. Academic Press.

Chan, E., Wan, W., Chaugule, A., and Campbell, R. 2009. A Framework for Volatile Memory Forensics. In *Proceedings of the16th ACM Conference on Computer and Communications Security.*

Chan, E. M., Carlyle, J. C., David, F. M., Farivar, R., and Campbell, R. H. 2008. BootJacker: Compromising Computers Using Forced Restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security.*

Chang, K., Kim, G., Kim, K., and Kim, W. 2007. Initial Case Analysis Using Windows Registry in Computer Forensics. In *Proceedings of the Future Generation Communication and Networking.*

Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. 2005. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium.*

Cohen, M. 2008. PyFlag - An Advanced Network Forensic Framework. In *Proceedings of the Digital Forensic Research Workshop (DFRWS).*

Computer Security Institute. 2009. 14th Annual CSI Computer Crime and Security Survey.

Craton, J. 2009. Netcat for Windows. http://joncraton.org/blog/netcat-for-windows.

Crazylord. 2002. Playing with Windows /dev/(k)mem. *Phrack Magazine 59,* 16, 1–14.

Darawk. 2005. CloakDll. http://www.darawk.com/code/CloakDll.cpp.

DFlabs. 2010. PTK Forensics. http://ptk.dflabs.com/.

DFRWS. 2005. DFRWS 2005 Forensics Challenge. http://www.dfrws.org/2005/challenge/index.shtml.

DFRWS. 2008. DFRWS 2008 Forensics Challenge Results. http://www.dfrws.org/2008/challenge/results.shtml.

Dolan-Gavitt, B. 2007a. The VAD Tree: A Process-Eye View of Physical Memory. *Digital Investigation 4,* 1, 62–64.

Dolan-Gavitt, B. 2007b. VADTools. http://vadtools.sourceforge.net/.

Dolan-Gavitt, B. 2008a. Cell Index Translation. http://moyix.blogspot.com/2008/02/cell-index-translation.html.

Dolan-Gavitt, B. 2008b. Finding Kernel Global Variables in Windows. http://moyix.blogspot.com/2008/04/finding-kernel-global-variables-in.html.

Dolan-Gavitt, B. 2008c. Forensic Analysis of the Windows Registry in Memory. *Digital Investigation 5,* 1, S26–S32.

Dolan-Gavitt, B. 2008d. Linking Processes to Users. http://moyix.blogspot.com/2008/08/linking-processes-to-users.html.

Dolan-Gavitt, B. 2008e. Reading Open Keys. http://moyix.blogspot.com/2008/02/reading-open-keys.html.

Dolan-Gavitt, B., Srivastava, A., Traynor, P., and Giffin, J. 2009. Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security.*

Dornseif, M. and Becher, M. 2004. Feuriges Hacken - SpaSS mit Firewire. In *Proceedings of the 21st Chaos Communication Congress.*

Drinkwater, R. 2009. Facebook Chat Forensics. http://forensicsfromthesausagefactory.blogspot.com/2009/03/facebook-chat-forensics.html.

Farmer, D. and Venema, W. 2005. *Forensic Discovery.* Addison Wesley.

Forensics Wiki. 2010. List of Volatility Plugins. http://www.forensicswiki.org/wiki/List_of_Volatility_Plugins.

FOUNDSTONE, INC. 2000. FPort. http://www.foundstone.com/us/resources/proddesc/fport.htm.

GAO, Y. AND CAO, T. 2010. Memory Forensics for QQ from a Live System. *Journal of Computers 5,* 4, 541–548.

GARCIA, G. L. 2007. Forensic Physical Memory Analysis: An Overview of Tools and Techniques. Tech. rep., Helsinki University of Technology.

GARNER, G. M. 2007. KnTTools with KnTList. http://gmgsystemsinc.com/knttools/.

GARNER, G. M. 2009. Forensic Acquisition Utilities. http://gmgsystemsinc.com/fau/.

GETGEN, K. 2009. 2009 Encryption and Key Management Industry Benchmark Report. http://iss.thalesgroup.com/Resources/Webinars/Benchmarksurvey-November52009-Webinar.aspx.

GMG SYSTEMS, INC. 2007. KnTTools with KnTList. http://gmgsystemsinc.com/knttools/.

GOLDEN G. RICHARD III. 2006. Scalpel: A Frugal, High Performance File Carver. http://www.digitalforensicssolutions.com/Scalpel/.

GROUP, T. C. 2007. Trusted Platform Module - Design Principles. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

GUIDANCE SOFTWARE. 2010. EnCase Enterprise Platform. http://www.guidancesoftware.com/computer-forensics-fraud-investigation-software.htm.

HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. 2009. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Communications of the ACM 52,* 5, 91–98.

HARGREAVES, C. AND CHIVERS, H. 2008. Recovery of Encryption Keys from Memory Using a Linear Scan. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security.*

HARUYAMA, T. 2009. EnCase EnScript "Memory Forensic Toolkit" Part1. http://tharuyama.blogspot.com/2009/10/encase-enscript-memory-forensic-toolkit.html.

HBGARY. 2009. FastDump - a Memory Acquisition Tool. https://www.hbgary.com/products-services/fastdump/.

HEJAZI, S., TALHIA, C., AND DEBBABI, M. 2009. Extraction of Forensically Sensitive Information from Windows Physical Memory. *Digital Investigation 6,* 1, S121–S131.

HENINGER, N. AND SHACHAM, H. 2009. Reconstructing RSA Private Keys from Random Key Bits. *Cryptology ePrint Archive 2008,* 510, 1–17.

HOGLUND, G. 2008. The Value of Physical Memory for Incident Response. http://www.hbgary.com/wp-content/themes/blackhat/images/the-value-of-physical-memory-for-incident-response.pdf.

HONEYNET PROJECT. 2004. *Know your Enemy - Learning about Security Threats*. Addison Wesley.

HULTON, D. 2006. Cardbus Bus-Mastering: 0wning the Laptop. In *Proceedings of ShmooCon*.

INTEL CORPORATION. 2011. Intel® 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/Assets/PDF/manual/325384.pdf.

INTERNET CRIME COMPLAINT CENTER. 2010. 2009 IC3 Annual Report. http://www.ic3.gov/media/annualreport/2009_IC3Report.pdf.

IONESCU, A. 2005. Getting Kernel Variables from KdVersionBlock, Part 2. http://www.rootkit.com/newsread.php?newsid=153.

KAPLAN, B. 2007. RAM Is Key - Extracting Disk Encryption Keys from Volatile Memory. M.S. thesis, Carnegie Mellon University.

KDM. 2004. NTIllusion: A Portable Win32 Userland Rootkit. *Phrack Magazine 11,* 62, 1–26.

KLEIN, T. 2006a. All Your Private Keys Are Belong to Us - Extracting RSA Private Keys and Certificates from Process Memory. http://trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf.

KLEIN, T. 2006b. Process Dumper. http://www.trapkit.de/research/forensic/pd/index.html.

KORNBLUM, J. 2007a. Recovering Executables with Windows Memory Analysis. http://www.jessekornblum.com/presentations/dodcc07.pdf.

KORNBLUM, J. D. 2006. Exploiting the Rootkit Paradox with Windows Memory Analysis. *International Journal of Digital Evidence 5,* 1, 1–5.

KORNBLUM, J. D. 2007b. Using Every Part of the Buffalo in Windows Memory Analysis. *Digital Investigation 4,* 1, 24–29.

LIBSTER, E. AND KORNBLUM, J. D. 2008. A Proposal for an Integrated Memory Acquisition Mechanism. *ACM SIGOPS Operating Systems Review 42,* 3, 14–20.

LIGH, M., ADAIR, S., HARTSTEIN, B., AND RICHARD, M. 2010. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley.

LYNCH, N. A., MERRITT, M., WEIHL, W. E., AND FEKETE, A. 1993. *Atomic Transactions.* Morgan Kaufmann.

MAARTMANN-MOE, C., THORKILDSEN, S. E., AND ÅRNES, A. 2009. The Persistence of Memory: Forensic Identification and Extraction of Cryptographic Keys. *Digital Investigation 6,* S1, S132–S140.

MACLEAN, N. P. 2006. Acquisition and Analysis of Windows Memory. M.S. thesis, University of Strathclyde, Glasgow.

MANDIANT. 2010. Memoryze. http://www.mandiant.com/products/free_software/memoryze/.

MANTECH CSI, INC. 2009. Memory DD. http://sourceforge.net/projects/mdd/files/.

MEE, V., TRYFONAS, T., AND SUTHERLAND, I. 2006. The Windows Registry as a Forensic Artefact: Illustrating Evidence Collection for Internet Usage. *Digit 3,* 3, 166–173.

MICROSOFT CORPORATION. 2008. Windows Registry Information for Advanced Users. http://support.microsoft.com/kb/256986/.

MICROSOFT CORPORATION. 2009. BitLocker Drive Encryption. http://technet.microsoft.com/en-us/library/cc731549%28WS.10%29.aspx.

MICROSOFT CORPORATION. 2010a. Debugging Tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

MICROSOFT CORPORATION. 2010b. Device PhysicalMemory Object. http://technet.microsoft.com/de-de/library/cc787565%28WS.10%29.aspx.

MICROSOFT CORPORATION. 2010c. ExAllocatePoolWithTag. http://msdn.microsoft.com/en-us/library/ff544520%28VS.85%29.aspx.

MICROSOFT CORPORATION. 2010d. KB244139 - Windows Feature Lets You Generate a Memory Dump File by Using the Keyboard. http://support.microsoft.com/?scid=kb%3Ben-us%3B244139&x=5&y=9, crash dump file generation.

MICROSOFT CORPORATION. 2010e. KB254649 - Overview of Memory Dump File Options for Windows Vista, Windows Server 2008 R2, Windows Server 2008, Windows Server 2003, Windows XP, and Windows 2000. http://support.microsoft.com/?scid=kb%3Ben-us%3B254649&x=13&y=5.

MICROSOFT CORPORATION. 2010f. KB971284 - A Hotfix is Available to Enable CrashOnCtrlScroll Support for a USB Keyboard on a Computer that is Running Windows Vista SP1 or Windows Server 2008. http://support.microsoft.com/?scid=kb%3Ben-us%3B244139&x=5&y=9, adding support for USB keyboards in Windows Vista and Server 2008 to create a crash dump file.

MICROSOFT CORPORATION. 2010g. Notmyfault. http://download.sysinternals.com/Files/Notmyfault.zip.

MICROSOFT CORPORATION. 2010h. VirtualLock Function. http://msdn.microsoft.com/en-us/library/aa366895%28v=vs.85%29.aspx.

MILLER, M. AND TURKULAINEN, J. 2006. Remote Library Injection. http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf.

MÜLLER, T., DEWALD, A., AND FREILING, F. 2011. TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Security Symposium*.

MÜLLER, T., DEWALD, A., AND FREILING, F. C. 2010. AESSE: A Cold-Boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*.

MOONSOLS. 2010. Windows Memory Toolkit. http://moonsols.com/product.

MOORE, D., PAXSON, V., SAVAGE, S., SHANNON, C., STANIFORD, S., AND WEAVER, N. 2003. Inside the Slammer Worm. *IEEE Security and Privacy 1,* 3, 33–39.

MOVALL, P., NELSON, W., AND WETZSTEIN, S. 2005. Linux Physical Memory Analysis. In *Proceedings of the USENIX Annual Technical Conference*.

MOZILLA FOUNDATION. 2008. Private Browsing. https://wiki.mozilla.org/PrivateBrowsing.

MRDOVIC, S., HUSEINOVIC, A., AND ZAJKO, E. 2009. Combining Static and Live Digital Forensic Analysis in Virtual Environment. In *Proceedings of the XXII International Symposium on Information, Communication and Automation Technologies*.

NET APPLICATIONS. 2010. Operating System Market Share. http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=10.

NT INTERNALS. 2009. Hidden Dynamic-Link Library Detection Test. http://www.ntinternals.org/dll_detection_test.php.

OKOLICA, J. AND PETERSON, G. L. 2010. Windows Operating Systems Agnostic Memory Analysis. *Digital Investigation 7,* 1, S48–S56.

OPEN SYSTEM RESSOURCES. 2009. BANG! – Crash on Demand Utility. http://www.osronline.com/article.cfm?article=153.

OSWALD, E. 2010. Toshiba's Hard Drive Breakthough Could Herald Mega-Capacity Drives. http://www.betanews.com/article/Toshibas-hard-drive-breakthough-could-herald-megacapacity-drives/1282246111.

PANHOLZER, P. 2008. Physical Security Attacks on Windows Vista. https://www.sec-consult.com/files/Vista_Physical_Attacks.pdf.

PENG, T., LECKIE, C., AND RAMAMOHANARAO, K. 2007. Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems. *ACM Computing Surveys 39,* 1, 1–42.

PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. 2004. Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*.

PGP CORPORATION. 2010. PGP Whole Disk Encryption. http://www.pgp.com/products/wholediskencryption/index.html.

PIEGDON, D. R. AND PIMENIDIS, L. 2007. Targeting Physically Addressable Memory. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

RAPID7 LLC. 2004. MetasploitŠs Meterpreter. http://www.metasploit.com/documents/meterpreter.pdf.

RARE IDEAS. 2010. PortableApps.com Platform. http://portableapps.com/.

ROLLES, R. 2009. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*.

ROUSSEV, V. AND RICHARD III, G. G. 2004. Breaking the Performance Wall: The Case for Distributed Digital Forensics. In *Proceedings of the Digital Forensic Research Workshop (DFRWS)*.

RUFF, N. 2008. Windows Memory Forensics. *Journal in Computer Virology 4,* 2, 83–100.

RUFF, N. AND SUICHE, M. 2007. Enter Sandman. In *Proceedings of the 5th Annual PacSec Applied Security Conference*.

RUSSINOVICH, M. E. 2010. TCPView. http://technet.microsoft.com/en-us/sysinternals/bb897437.aspx.

RUSSINOVICH, M. E., SOLOMON, D. A., AND IONESCU, A. 2009. *Microsoft Windows Internals*, 5th ed. Microsoft Press.

RUTKOWSKA, J. 2005. modGREPER. http://www.invisiblethings.org/tools/modGREPER.

RUTKOWSKA, J. 2007. Beyond The CPU: Defeating Hardware Based RAM Acquisition (Part I: AMD Case). http://invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt.

SAOUT, C. 2006. dm-crypt: A Device-Mapper Crypto Target. http://www.saout.de/misc/dm-crypt/.

Savoldi, A. and Gubian, P. 2008. Towards the Virtual Memory Space Reconstruction for Windows Live Forensic Purposes. In *Proceedings of Systematic Approaches to Digital Forensic Engineering (SADFE).* 15–22.

Schatz, B. 2007a. BodySnatcher: Towards Reliable Volatile Memory Acquisition by Software. *Digital Investigation 4*, 126–134.

Schatz, B. 2007b. Recent Developments in Volatile Memory Forensics.

Schuster, A. 2006a. DMP File Structure. http://computer.forensikblog.de/en/2006/03/dmp_file_structure.html.

Schuster, A. 2006b. Pool Allocations as an Information Source in Windows Memory Forensics. In *Proceedings of IT-Incident Management & IT-Forensics.* 15–22.

Schuster, A. 2006c. Reconstructing a Binary. http://computer.forensikblog.de/en/2006/04/reconstructing_a_binary.html.

Schuster, A. 2006d. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *Digital Investigation 3,* 1 (July), 10–16.

Schuster, A. 2008a. 64bit Crash Dumps. http://computer.forensikblog.de/en/2008/02/64bit_crash_dumps.html.

Schuster, A. 2008b. The Impact of Microsoft Windows Pool Allocation Strategies on Memory Forensics. *Digital Investigation 5,* S1, S58–S64.

Shamir, A. and van Someren, N. 1999. Playing Hide and Seek with Stored Keys. In *Proceedings of the Third International Conference on Financial Cryptography.*

Sharif, M., Lanzi, A., Giffin, J., and Lee, W. 2009. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy.*

Shipley, T. G. and Reeve, H. R. 2006. Collecting Evidence from a Running Computer - A Technical and Legal Primer for the Justice Community. http://www.hbgary.com/wp-content/themes/blackhat/images/collectevidenceruncomputer.pdf.

Simon, M. and Slay, J. 2009. Enhancement of Forensic Computing Investigations through Memory Forensic Techniques. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES).*

Simon, M. and Slay, J. 2010. Recovery of Skype Application Activity Data from Physical Memory. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES).*

Smith, J. and Cote, M. 2010. The Honeynet Project Forensic Challenge 3: Banking Troubles. http://honeynet.org/challenges/2010_3_banking_troubles.

Smith, J. E. and Nair, R. 2005. The Architecture of Virtual Machines. *Journal of Computer 38,* 5, 32–38.

Solomon, J., Huebner, E., Bem, D., and Szezynska, M. 2007. User Data Persistence in Physical Memory. *Digital Investigation 4,* 2, 68–72.

Sophos Plc. 2010. SafeGuard Easy. http://www.sophos.com/products/enterprise/encryption/safeguard-easy/.

Sparks, S. and Butler, J. 2005. Shadow Walker - Raising the Bar for Rootkit Detection. http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf.

Spitzner, L. 2003a. Definitions and Value of Honeypots. http://www.spitzner.net/honeypots.html.

Spitzner, L. 2003b. Moving Forward with Defintion of Honeypots. http://www.securityfocus.com/archive/119/321957/30/0/threaded.

Stevens, R. M. and Casey, E. 2010. Extracting Windows Command Line Details from Physical Memory. *Digital Investigation 7,* 1, S57–S63.

Stover, S. and Dickerson, M. 2005. Using Memory Dumps in Digital Forensics. *;login: The USENIX Magazine 30,* 6, 43–48.

Suiche, M. 2008a. Exploiting Windows Hibernation. In *Proceedings of Europol High Tech Crime Expert Meeting*.

Suiche, M. 2008b. SandMan Project. http://sandman.msuiche.net/docs/SandMan_Project.pdf.

Suiche, M. 2010. Advanced Mac OS Physical Memory Analysis. http://www.blackhat.com/presentations/bh-dc-10/Suiche_Matthieu/Blackhat-DC-2010-Advanced-Mac-OS-X-Physical-Memory-Analysis-wp.pdf.

Sutherland, I., Evans, J., Tryfonas, T., and Blyth, A. 2008. Acquiring Volatile Operating System Data Tools and Techniques. *ACM SIGOPS Operating Systems Review 42,* 3, 65–73.

TrueCrypt Foundation. 2010. TrueCrypt. http://www.truecrypt.org/.

Tsow, A. 2009. An Improved Recovery Algorithm for Decayed AES Key Schedule Images. *Lecture Notes in Computer Science 5867*, 215–230.

United States Air Force Office of Special Investigations. 2001. Foremost. http://foremost.sourceforge.net/.

U.S. Department of Justice. 2008. Electronic Crime Scene Investigation: A Guide for First Responders. http://www.ncjrs.gov/pdffiles1/nij/219941.pdf.

U.S. Secret Service. 2006. Best Practices For Seizing Electronic Evidence.

van Baar, R., Alink, W., and van Ballegooij, A. 2008. Forensic Memory Analysis: Files Mapped in Memory. In *Proceedings of the Digital Forensic Research Workshop (DFRWS)*.

Vidas, T. 2006. The Acquisition and Analysis of Random Access Memory. *Journal of Digital Forensic Practice 1,* 4, 315 – 323.

Vidas, T. 2010. Volatile Memory Acquisition via Warm Boot Memory Survivability. In *Proceedings of the 43rd Hawaii International Conference on System Sciences*.

Vidstrom, A. 2002. PMDump. http://ntsecurity.nu/toolbox/pmdump/.

Vidstrom, A. 2006. Memory Dumping over FireWire - UMA Issues. http://ntsecurity.nu/onmymind/2006/2006-09-02.html.

VMware, Inc. 2010. What Files Make Up a Virtual Machine? http://www.vmware.com/support/ws55/doc/ws_learning_files_in_a_vm.html.

Volatile Systems, LLC. 2008. The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. https://www.volatilesystems.com/default/volatility.

Walters, A. 2006. FATKit: Detecting Malicious Library Injection and Upping the "Anti". Tech. rep., 4Tϕ Forensic Research.

Walters, A. and Petroni, N. L. 2007. Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. In *Proceedings of Black Hat DC 2007*.

X-Ways Software Technology AG. 2010. WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk Editor. http://www.x-ways.net/winhex/index-m.html.

Young, A. and Yung, M. 2004. *Malicious Cryptography - Exposing Cryptovirology.* Wiley.

Zhang, R., Wang, L., and Zhang, S. 2009. Windows Memory Analysis Based on KPCR. In *Proceedings of the Fifth International Conference on Information Assurance and Security*.

Zhang, S., Wang, L., Zhang, R., and Guo, Q. 2010. Exploratory Study on Memory Analysis of Windows 7 Operating System. In *Proceedings of the Third International Conference on Advanced Computer Theory and Engineering (ICACTE)*.

Zhao, Q. and Cao, T. 2009. Collecting Sensitive Information from Windows Physical Memory. *Journal of Computers 4,* 1 (January), 3–10.