Digital Investigation 10 (2013) S105–S115

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

# Anti-forensic resilient memory acquisition

Johannes Stüttgen [a,*], Michael Cohen [b]

[a] Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstraße 3, 91058 Erlangen, Germany
[b] Google Inc., Brandschenkestrasse 110, Zurich, Switzerland

*Keywords:*
Memory forensics
Memory acquisition
Anti forensics
Live forensics
Malware
Computer security
Information security
Incident response

A B S T R A C T

Memory analysis has gained popularity in recent years proving to be an effective technique for uncovering malware in compromised computer systems. The process of memory acquisition presents unique evidentiary challenges since many acquisition techniques require code to be run on a potential compromised system, presenting an avenue for anti-forensic subversion. In this paper, we examine a number of simple anti-forensic techniques and test a representative sample of current commercial and free memory acquisition tools. We find that current tools are not resilient to very simple anti-forensic measures. We present a novel memory acquisition technique, based on direct page table manipulation and PCI hardware introspection, without relying on operating system facilities - making it more difficult to subvert. We then evaluate this technique's further vulnerability to subversion by considering more advanced anti-forensic attacks.



## 1. Introduction

Since host-based memory forensics was first proposed, rapid advances in the analysis techniques for memory images have taken place. Modern tools are capable of extracting detailed information about system state, configuration, and anomalies. In particular, memory analysis has proven useful for the detection of rootkits and other malware infecting the host, as well as the analysis of malicious software.

As this analytical capability matures, applications are emerging for application of memory analysis in many contexts, such as remote forensics (Cohen et al., 2011), malware classification, and even self healing of compromised systems (Grizzard, 2006). Direct memory access can be used in the forensic context to obtain a complete point-in-time static forensic image or to enable an external memory analysis module to perform runtime live analysis. In this paper we refer to the process of accessing the physical memory as "memory acquisition", regardless of its intent.

Memory analysis is attractive for malware analysis as it is seen as a way to examine the system from an external and impartial point of view. While malware may attempt to hide by hooking operating system services (Florio, 2005), analysis of memory images offers the opportunity to examine the rootkit's hooks and code outside of the path of the ordinary operating system functionality.

Anti-Forensics has been broadly defined as "any attempt to compromise the availability or usefulness of evidence to the forensic process" (Harris, 2006). Thus anti-forensic attacks fall into two broad categories – those techniques which prevent evidence from being acquired, and those techniques which remove data from the collected evidence such that the collected evidence can not be suitably analyzed.

A number of effective anti-forensic techniques against memory acquisition have been proposed. Substitution attacks, in which data fabricated by the attacker is substituted in place of valid data during the acquisition process have been implemented (Bilby, 2006; Milkovic, 2012). Alternatively a rootkit might disrupt the acquisition process altogether (e.g. hang the hardware) when detecting the

* Corresponding author.
*E-mail addresses:* johannes.stuettgen@cs.fau.de (J. Stüttgen), scudette@google.com (M. Cohen).

1742-2876/$ – see front matter 
http://dx.doi.org/10.1016/j.diin.2013.06.012

presence of a forensic agent. This approach is especially effective against memory acquisition, since the volatility of the evidence does not permit the investigator to reacquire the memory under the same conditions.

Although there have been efforts to test memory forensic acquisition tools (Inoue et al., 2011), and even solidify the criteria by which these tools can be tested (Carrier and Grand, 2004; Vömel and Freiling, 2012), robustness of the tools against anti-forensic interference is not yet explored during these tests (Wundram et al., 2013). Thus, while one can gain assurances about the forensic soundness of acquisition tools under ideal lab conditions, it is impossible to extrapolate this to acquisition of a hostile system, potentially employing anti-forensic techniques.

Due to lack of research and understanding of anti-forensic techniques in memory acquisition, current commercial or free memory acquisition tools do not appear to implement mechanisms to protect their operations against anti-forensic attacks. Due to the increasing popularity of these tools, there is currently an invigorated research interest in developing anti-forensic techniques specifically targeting these tools (Milkovic, 2012; Haruyama and Suzuki, 2012).

*Related Work*: A number of memory acquisition techniques have been proposed in the literature (Vömel and Freiling, 2011). In assessing the exposure of different memory acquisition techniques to anti-forensic subversion, we can broadly divide techniques into those which rely on the operating system software integrity and those who rely on the hardware.

Many operating systems already present a view of physical memory through a special device or kernel API. For example, in Windows the operating system presents the section object `\\.\PhysicalMemory`, to allow reading from physical memory. Earlier memory acquisition tools directly opened this device from user-space (Garner, 2006). More recent versions of Windows deny direct access to the device from user space, necessitating a kernel driver to open the device from kernel space. A number of more direct kernel API routines are utilized in current tools, such as `MmMapIOSpace` and the undocumented `MmMapMemoryDumpMdl` (MoonSols, 2012).

Bypassing memory acquisition tools that depend on the operating system was demonstrated by the ddefy tool (Bilby, 2006). This tool hooks the physical memory device and filters certain pages from being read through this interface, providing instead a cached copy (prior to kernel modification). In principle, any OS facility can be hooked in a similar manner in order to subvert the acquisition tool. Additionally, many acquisition tools have a user mode process to write and process the image. This increases the attack surface of the tool, by allowing standard user space hooks to modify the memory image as it is written to disk (Milkovic, 2012).

Hardware-based solutions were proposed as being resilient to rootkit manipulation. Dedicated hardware can access the memory bus directly without CPU management (Carrier and Grand, 2004). It is even possible to re-purpose existing hardware to extract physical memory. For example, the Firewire hardware may be used for direct memory access (DMA) to the physical address space (Boileau, 2006).

Unfortunately, even hardware-based acquisition can be defeated using very low level manipulation of the memory controller's hardware registers (Rutkowska, 2007). By remapping some parts of the physical address space into an IO device, the CPU's view of this range is different from the hardware DMA view.

A more subversion resistant approach is taken by `BodySnatcher` (Schatz, 2007). It involves loading a new, trusted OS for acquisition. While avoiding rootkit interference and guaranteeing the atomicity of the image, the technique has severe drawbacks. For example the operating system of the host is halted, making it unsuitable for production environments. Additionally, the acquisition OS needs to have drivers for the device used to extract the memory image to (e.g. the network interface), making it highly platform dependent.

Recent advances in hardware virtualization allow running the acquisition software on a higher privilege level than the operating system. For example the Hypersleuth (Martignoni et al., 2010) and VIS (Yu et al., 2011) tools leverage the Intel VMX instruction set to virtualize the operating system on the fly. Running in VMX root-mode, the acquisition software essentially acts as a thin hypervisor and thus is not prone to subversion by operating system level rootkits. Also, the hypervisor based acquisition tool can guarantee the atomicity of the image, by adopting a copy-on-write based imaging approach. However, this approach depends on the ability to load a new hypervisor on the fly. In environments where a hypervisor is already running, this will not work unless nested hardware virtualization is supported and active. With Hyper-V being shipped with Windows 8 and many web servers being virtual instances, this is increasingly often the case. Also, the ability to virtualize the operating system on the fly means a rootkit can do the same thing, defeating the acquisition hypervisor (Rutkowska, 2006; Zovi, 2006; King and Chen, 2006).

A possible solution to this problem is to go even deeper, and execute memory acquisition software on a firmware level. By running in System Management Mode (SMM), the program is isolated from any operating system and even hypervisor based malware (Wang et al., 2011), while still being able to create an atomic memory image without completely halting the host. Unfortunately, only the BIOS can load code into SMM. The acquisition software thus has to be installed by flashing a new BIOS onto the target machine. This requires a reboot, making this technique unsuitable for ad-hoc analysis.

*Contributions*: In this paper, we advance the field of forensic memory acquisition by considering the efficacy of forensic tools when facing determined and skilled adversaries, willing to use anti-forensic techniques. We find that the current generation of forensic memory acquisition tools are ill equipped to face this adversarial challenge. By understanding the weaknesses present in current tools we are able to further the state of the art by developing more robust solutions, thereby increasing the complexity required by the attacker to effectively bypass forensic tools.

Forensic tool testing is a contemporary research topic. There have been attempts to quantify testable criteria by which to assess the correctness of memory acquisition

tools. However, this line of reasoning completely ignores the fact that memory acquisition tools are running in a hostile environment. The following paper demonstrates that whilst many acquisition tools produce correct and forensically sound images under idealized lab conditions, they completely fail when simple anti-forensic attacks are present. This makes it hard to gain assurances about the tool's correctness from simple tool testing procedures.

In this work, we experimentally implement a range of previously published and novel anti-forensic techniques. We then apply these techniques against a representative sample of contemporary popular memory acquisition tools. Unfortunately, we find that all tools tested can be subverted by simple anti-forensic attacks.

We then introduce a novel technique to combat these attacks, and still acquire a usable memory image. We incorporated this novel acquisition technique into the open source "WinPmem" memory acquisition tool (Cohen, 2012a), making it resilient to current published anti-forensic attacks. We critically analyze potential countermeasures to our novel technique and offer scope for further research.

## 2. Physical memory acquisition

The physical address space is the entire range of memory addresses that can appear on the memory bus. However, the physical address space contains more than simply RAM, as Fig. 1 illustrates. When the system boots, the BIOS creates a physical memory layout which consists of segments of usable memory interleaved with ranges reserved for ROM, PCI resources and the BIOS use itself. Most modern motherboard chipsets are able to route memory access around some reserved regions such that all available RAM chips are utilized. Therefore, usually the highest physical memory address (and hence the size of the raw memory image acquired) is larger than the total amount of RAM installed.

In order to learn about the BIOS allocated memory layout, an operating system must issue a BIOS service interrupt (`INT 0x15` with `AX` holding the value `0xE820` (Microsoft Corporation et al., 2006)) early in the boot sequence, while the processor is still running in real mode.

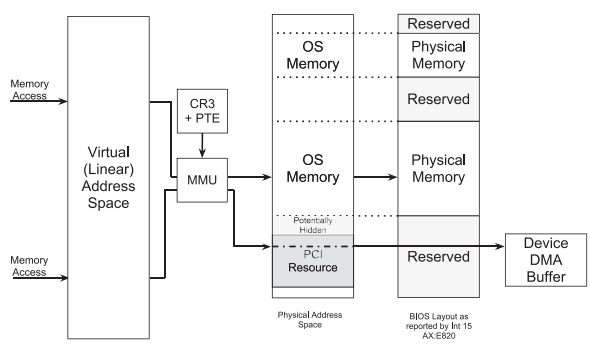


**Fig. 1.** Physical Memory utilization on modern architectures. The MMU translates memory access from the virtual address space into the physical address space using page tables and the Control Register CR3. The physical address space consists of reserved and available regions configured by the hardware BIOS. The operating system configures hardware DMA buffers within reserved regions for direct access to device memory.

This BIOS configured memory layout is subsequently refined by configuring PCI plug and play devices and remapping their DMA buffers within the reserved regions (PCI Special Interest Group, 2005). Although the operating system itself does not utilize reserved regions, they may still be backed by RAM. An acquisition tool failing to acquire reserved regions which are not mapped to DMA devices potentially allows malware to hide code or data there. This is explored further in Section 3.2.

When running in protected mode, the CPU can not access physical memory directly. Instead, the kernel creates a continuous virtual address space that is transparently mapped by the Memory Management Unit (MMU). Fig. 1 illustrates at a high level how memory is accessed with modern Intel/AMD based architectures. The control register, `CR3`, contains the addresses of page tables set up in memory. These tables are maintained by the operating system, for the hardware Memory Management Unit (MMU) to use in translating memory access. The details of this page translation process are described elsewhere (Intel, 2013a), however for our purposes it is important to note that virtual addresses can be translated to any region of the physical address space by suitable page table entries (PTE).

For efficiency, peripheral device communication can be conducted over the system memory bus by routing some addresses to registers or buffers on a device's board (e.g., Video hardware often has its own memory chips, allowing direct memory copy to the video buffers over the high speed system memory bus). This mapping is termed Direct Memory Access (DMA) and is configured via the PCI controller during kernel initialization. The physical address space therefore, contains directly mapped device DMA buffers in addition to physical memory.

Hence, reading these DMA mapped regions from the physical address space does not result in a read operation from system memory, rather it activates various hardware devices connected to the system. These read operations may trigger interrupts, system crashes, or even data corruption if not performed according to the device's specific PCI bus protocols.

For the memory acquisition tool, it is extremely important to ensure that DMA regions are avoided during acquisition in order to minimize the chance of unrecoverable system crashes. From a forensic point of view, we are usually most interested in acquiring the physical memory which only forms a part of the physical address space.

As illustrated in Fig. 1, the acquisition tool has two problems to solve:

1. **Enumeration of address space layout:** The tool has to determine which parts of the physical address space are backed by physical memory as opposed to peripheral device DMA buffers. DMA regions must be avoided to prevent system crashes.
2. **Physical memory mapping:** Since the tool must access physical memory via the virtual address space, the tool must create a page table mapping between a region in the physical address space and the virtual address space.

### 2.1. Common software solution

There are common solutions for both problems, which are very similar among available tools even on different operating systems.

*Enumeration of address space layout*: Unfortunately, it is impossible to issue the BIOS service interrupt *Int 15* while running in protected mode once the operating system is booted. Therefore, when memory acquisition tools attempt to obtain the physical memory layout, they must rely on data structures or APIs within the operating system kernel, which are easily susceptible to manipulation.

For example, on Microsoft Windows systems the symbol `MmPfnDatabase` stores an array with information on every page frame on the system and how it is utilized by the kernel. However, this symbol is usually not exported, and is only accessible through the KdDebugBlock structure (which is also not exported but used by the kernel debugger.). The exported API `MmGetPhysicalMemoryRanges()` (Russinovich, 1999) returns the list of physical address ranges which are available for use, as prepared by the BIOS, and most windows memory acquisition software we tested use this API.

On Linux, the symbol `iomem_resource`, populated during system boot, contains a tree of resource structures representing system RAM regions, as well as DMA regions assigned. Resource regions are named using a simple string, with regions used by the system named "System RAM". Linux acquisition tools (Sylve, 2012; Cohen, 2011) directly parse this data structure and only acquire regions named "System RAM", again allowing for the possibility of hidden data in non-system regions.

*Physical memory mapping*: As illustrated in Fig. 1, software running in protected mode can not access physical memory directly. Instead, the required physical memory region must be mapped into the virtual address space, by setting up the appropriate page tables. There are a number of OS provided APIs to achieve this.

For example, on Windows, the API `ZwMapViewOfSection` can be used to map a file into the virtual address space such that any read operations from the mapped regions are serviced from the file contents. This technique is then used to map parts of the section object `\\.\PhysicalMemory`, which provides a controlled operating system view of the physical memory.

Alternatively, Windows also provides the API `MmMapIOSpace()` to allow drivers to map regions of IO or Physical Memory space into the kernel address space (for example, in order to directly access the PCI-configured DMA buffers). Additionally, undocumented APIs exist, such as `MmMapMemoryDumpMdl()`, normally used during the system's crash dump handling.

The Linux operating system has a non-paging kernel, allowing it to have a permanent, linear mapping between the kernel virtual address space and the physical address space. This means that the entire physical address space is always mapped into the virtual address space at a constant offset. Therefore, the API `kmap()` on many platforms is simply a macro which translates the physical page frame number to that linearly mapped kernel virtual address that corresponds to it.

## 3. Anti-forensic techniques

Recent interest in anti-forensics has concentrated on leveraging standard rootkit techniques such as API hooking in both userspace and kernelspace to disable memory acquisition tools, or selectively redact evidence from the acquired image (Milkovic, 2012; Bilby, 2006).

Strategically, the rootkit's task is to remain hidden, causing as little noticeable interruption to normal system activity. However, once the rootkit detects a memory acquisition tool is running, it requires that acquisition to be thwarted or redacted. Therefore, the rootkit must trade-off more intrusive, system wide techniques which might destabilize the system with simple techniques which would attack the acquisition software alone.

Due to the reliance of acquisition tools on obscure and undocumented exported system APIs, it is quite easy to differentiate the forensic agent from normal system processes. This may allow the rootkit to direct its anti-forensic action towards specific forensic agents, while leaving regular software unaffected.

In the following sections we examine experimentally how simple anti-forensic techniques affect several of the most popular memory acquisition tools.

### 3.1. Active anti-forensics

We have created a small Python kernel patcher (shown in Listing 1) that demonstrates a few simple techniques.

```
from volatility import session
from volatility.plugins.overlays import windows

def KernelApiPatch(session, symbol, patch):
    kernel_image = windows.pe_vtypes.PE(
        session.kernel_address_space,
        session.kdbg.KernBase)
    offset = kernel_image.GetProcAddress(symbol)
    session.kernel_address_space.write(offset,
        patch)

if __name__ == "__main__":
    s = session.Session(filename = r"\\.\pmem")
    return_null_shellcode = "\x48\x31\xc0\xc3"
    KernelApiPatch(s, "MmGetPhysicalMemoryRanges",
                   return_null_shellcode)
    KernelApiPatch(s, "MmMapMemoryDumpMdl",
                   return_null_shellcode)
    s.kdbg.Header.OwnerTag = "MOOF"
```

**Listing 1.** A kernel patcher script based on the Technology Preview Edition of the Volatility Memory Analysis Framework (Cohen, 2012b), is able to locate arbitrary kernel functions in the running system and patch them.

In a few lines, this script utilizes the Technology Preview edition of the Volatility Framework (Walters, 2007; Cohen, 2012b) and the WinPmem driver with enabled write support (Cohen, 2012a), to subvert memory acquisition in three ways:

*Kernel debugger block hiding*: The static kernel structure `KdDebugBlock` is used to find the base address of the kernel image and several non-exported symbols. It can be found by scanning for the `OwnerTag` member, which is the static string "KDBG". Haruyama and Suzuki already demonstrated that overwriting this tag is effective in

thwarting analysis by frameworks like Volatility (Haruyama and Suzuki, 2012). As we show in Section 3.1.1, this technique can even disrupt memory acquisition, as some drivers rely on the KDBG to resolve some symbols.

*Hooking of memory enumeration APIs*: As mentioned in Section 2, memory acquisition drivers need to enumerate the physical address space prior to acquisition. On the Microsoft Windows family of operating systems, all tested drivers use the undocumented symbol `MmGetPhysicalMemoryRanges()` to obtain a map of the physical address space. By patching this function to always return `NULL`, which is the failure indicator for this function, we prevent drivers from learning about the physical address space layout. As reading from device memory can crash the kernel, this effectively prevents memory acquisition. The patch is relatively stable, as usage of this API is discouraged by Microsoft, so regular drivers don't use it. An actual rootkit could of course simply return a modified version of the memory map, which excludes ranges it is trying to hide. Acquisition would then appear successful, while being incomplete.

*Hooking of memory mapping APIs*: To actually access physical memory, acquisition drivers need to map it into the kernels virtual address space (see Section 2). The three kernel APIs commonly used for this purpose are `ZwMapViewOfSection()`, `MmMapIOSpace()` and the undocumented symbol `MmMapMemoryDumpMdl()`. For demonstration purposes, we patch `MmMapMemoryDumpMdl()` to return `NULL`. As this symbol is also undocumented and usage is discouraged, this patch is relatively stable. Because the other two APIs are often used by drivers, patching them will quickly result in system instability. However, a more sophisticated rootkit can easily install hooks that filter mapping operations on hidden pages. This would also be a very stable modification, subverting any memory acquisition tools using the other two APIs.

Note that because of Kernel Patch Protection this script will not work on 64-bit kernels without disabling Patch Guard (Microsoft Corporation, 2006). However, as rootkits more and more start to subvert this protection (Rusakov, 2012, 2011), we believe it is safe to assume an attacker is able to do this. For testing purposes, we have enabled debug mode on our test systems, which disables Patch Guard.

### 3.1.1. Evaluation against active anti-forensics

We evaluated our proof-of-concept techniques against several popular memory acquisition tools. For this study, we have requested evaluation copies of "Moonsols Dumpit", "HBGary Fastdump Pro", "GMG Systems' Kntdd" and "Guidance's WinEn" for the purpose of forensic tool testing. Only Moonsols responded positively to our request. Additionally, we included open source or free tools such as "WindowsMemoryReader", "Winpmem", "Mandiant Memoryze" and "Access Data's FTK Imager". We believe that most other tools exhibit similar deficiencies. However, since we are unable to test these, readers are encouraged to use Listing 1 to reproduce these tests themselves.

Our test system is an ×86-64 Intel computer, running a fully patched Windows 7 ×86-64 with Service Pack 1. We have tested Memoryze (Mandiant, 2011), FTK Imager (AccessData, 2012), Win64dd (MoonSols, 2012), WinPmem (Cohen, 2012a), and WindowsMemoryReader (ATC-NY, 2012b), using their default settings to produce a raw image. In cases where the tool could produce a crashdump format image, the tests were repeated for this format. All patches in Listing 1 were tested individually, as well as simultaneously. A summary of the evaluation results is depicted in Table 1, where a PASS means the acquisition tool was able to create an image of memory despite the employed anti-forensic method.

The data shows that every tested acquisition tool was subverted by at least one of the tested anti-forensic methods. After employing all anti-forensic techniques simultaneously, none of the tools were able to acquire a single byte of memory. Some tools even crashed the kernel while trying, a very undesirable effect when analysing production systems. This may be due to missing error checking within the acquisition tool which may assume that Kernel APIs can never fail.

*Mandiant Memoryze*: Destroying the KDBG Owner Tag had no impact on the performance of Memoryze. Also, hooking `MmMapMemoryDumpMdl()` had no effect, as Memoryze only supports the `\\.\PhysicalMemory` and `MmMapIOSpace()` methods for mapping physical memory. Hooking `MmGetPhysicalMemoryRanges()` caused Memoryze to crash the kernel immediately, making it impossible to acquire any memory at all and forcing the target machine to reboot without an error message.

*Accessdata FTK Imager*: Similarly to Memoryze, destroying the KDBG Owner Tag or Hooking `MmMapMemoryDumpMdl()` did not affect FTK Imager, as it maps memory by calling `ZwMapViewOfSection()` on the `\\.\PhysicalMemory` device. However, hooking `MmGetPhysicalMemoryRanges()` will result in an empty image, without any apparent warnings.

**Table 1**
Successful acquisitions with active anti-forensics.

| Acquisition tool | Version | Format | KDBG | MmGetPhysical memory-ranges() | MmMap-MemoryDump-Mdl() |
|---|---|---|---|---|---|
| Memoryze | 2.0 | raw | PASS | FAIL | PASS |
| FTK Imager | 3.1.2 | raw | PASS | FAIL | PASS |
| Win64dd | 1.4.0 | raw | PASS/FAIL | FAIL | FAIL |
| Win64dd | 1.4.0 | dmp | FAIL | FAIL | FAIL |
| DumpIt | 1.4.0 | raw | PASS | FAIL | FAIL |
| WinPmem | 1.3.1 | raw | FAIL | FAIL | PASS |
| WinPmem | 1.3.1 | dmp | FAIL | FAIL | PASS |
| WindowsMemoryReader | 1.0 | raw | PASS | FAIL | PASS |
| WindowsMemoryReader | 1.0 | dmp | PASS | FAIL | PASS |

*Moonsols Win64dd*: When creating a raw image, trashing of the KDBG Owner Tag resulted in spontaneous reboots during acquisition with Win64DD. In our tests, an incomplete dump of 100 MB was created before the fault occurred. The log did not include any error messages. Similar behaviour was experienced when hooking `MmGetPhysicalMemoryRanges()` or `MmMapMemoryDumpMdl()`, which is the default memory mapping method of Win64DD. The tool behaved in the same way when creating a crash dump (`dmp`). However, when providing all arguments on the command-line and creating a raw image, the `KDBG` method did not cause Win64dd to crash anymore. It was still impossible to create a crash-dump, though. We presume Win64dd's interactive mode queries the driver for some information, that triggers it to search for the `KDGB`, regardless of the image format.

*Moonsols DumpIt*: Moonsols offers a packaged version of it's memory acquisition tools called DumpIt. This tool only supports the raw output format and does not seem to be affected by overwriting of the KDBG-Owner-Tag. It is still vulnerable to the other two anti-forensic methods.

*WinPmem*: Overwriting the KDBG Owner Tag causes WinPmem to fail. In contrast to other tools we tested, it does not crash the kernel. However, there is no error message indicating the reason for the failure. The hooking of `MmGetPhysicalMemoryRanges()` also causes an abort, displaying the error message to obtain memory geometry. Hooking `MmMapMemoryDumpMdl()` does not affect WinPmem, as it utilizes the `\\.\PhysicalMemory` and `MmMapIOSpace()` methods for memory mapping.

*ATC-NY WindowsMemoryReader*: The KDBG method did not affect WindowsMemoryReader at all. It was even able to create a crash-dump. However, the resulting dump could not be parsed by WinDBG completely, as the contained KDBG block was corrupted. Hooking of `MmMapMemoryDumpMdl()` had no effect, as it is not used by WindowsMemoryReader. The `MmGetPhysicalMemoryRanges()` method however completely disabled both, `raw` and `dmp` output. It caused an error in the driver to crash the kernel, immediately rebooting the host.

#### 3.1.2. Platform dependence

The demonstrated problems are not Windows specific. We have also conducted experiments with other operating systems, with similar results. On Mac OSX 10.8 Mountain Lion, we have tested MacMemoryReader in version 3.0.2 (ATC-NY, 2012a), as well as OXSPmem (Stüttgen, 2012) version RC1. Both function in a similar way, with the same inherent problems malicious software can exploit.

On EFI enabled systems, rather than using the BIOS Interrupt 15 routine, memory geometry is obtained by calling an EFI routine while the CPU is still in real mode. This physical memory map is handed to the kernel on startup, and subsequently cached in kernel data structures.

The *platform expert* component of the OSX kernel stores a pointer to this structure in the symbol `PE_state.bootArgs`. Zeroing this structure, or simply zeroing the size member, will prevent acquisition drivers from obtaining a map of physical address space, effectively preventing acquisition. Of course a more sophisticated rootkit could modify this map to exclude any data it protects. Acquisition will then succeed, without any indication of subversion. However, hidden data will not be included in the image, reducing its evidentiary efficacy.

This procedure is very easy to implement, for example, a simple proof of concept is depicted in Listing 2. A malicious OSX kernel extension calling this 2-line function can completely prevent both tools from acquiring even a single byte of memory.

```
void destroy_efi_memory_map(void) {
  // Access boot arguments through platform export,
  // and zero size member of EFI Memory Map.
  boot_args * ba = (boot_args *)(PE_state.bootArgs);
  ba->MemoryMapSize = 0;
}
```

**Listing 2.** OSX Memory-Map Overwriting.

Similarly to the Windows acquisition tools, OSX physical memory mapping can also be easily subverted. On OSX, physical memory mapping is achieved by creating an object of `IOMemoryDescriptor`, and then calling it's `createMappingInTask()` method. By either hooking the constructor or mapping method, malicious software can perform the exact same attacks as with the above mentioned Windows memory mapping functions.

### 3.2. Passive techniques

As already mentioned in Section 2, the BIOS memory map divides the physical address space into two categories: available and reserved. Available means the memory is backed by RAM and usable by the operating system. Reserved indicates regions where devices might create DMA mappings. The left part of Fig. 2, depicts the layout as it exists on one of our test machines.

The BIOS memory map is subsequently refined during device configuration, and parts of the address space in the reserved regions finally contain device memory mapped IO.

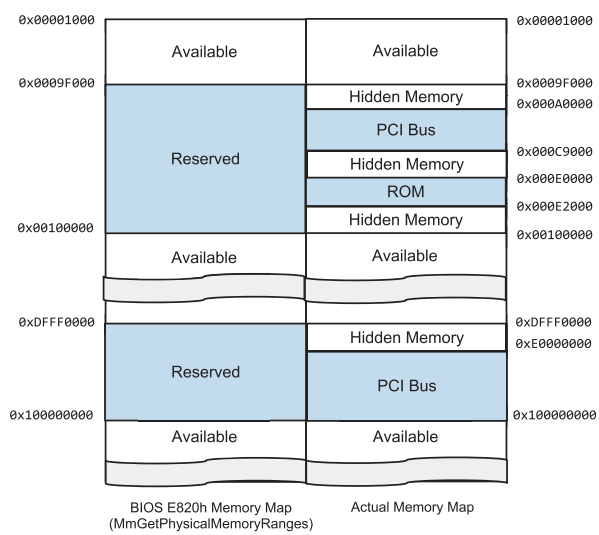


**Fig. 2.** Hidden memory on a test system with 4 GB of RAM.

However, because of the way the BIOS set up the reserved regions, small segments in the reserved regions can still be backed by RAM. We call these segments "Hidden Memory", as they can be used by rootkits to hide data from memory acquisition tools.

We have located these regions by identifying segments in reserved memory that are not mapped by devices, writing known values to them and then reading these back to ensure the write persisted. On our test machine, the final configuration contains 4 easily locatable segments of hidden memory, as shown on the right side of Fig. 2. They have a total size of 52 pages (1, 23, 12 and 16), which sums up to 212,992 bytes of hidden memory.

#### 3.2.1. Evaluation against passive anti-forensics

For this test, all hidden memory segments were filled with a known string. We then acquired a raw memory image and compared the data at the corresponding offsets with the known string. Note that some tools provide multiple settings on which memory regions to acquire. Because the hidden memory segments lie inside regions reported as memory mapped IO by the operating system, we have acquired the test images using the most extended setting possible.

Mandiant Memoryze, AccessData FTK Imager, Winpmem and Moonsols DumpIt don't allow the acquisition of anything other than the "available" regions. In the resulting image, these regions are zero-padded, except for Memoryze, which uses the `0xBA` byte for padding. The known string was not acquired.

ATC-NY WindowsMemoryReader allows very fine tuning on the parts of memory that are acquired. It even resolves all device DMA mappings and provides options to include them in the image. Unfortunately, it regards regions that are neither "available" nor memory mapped IO as nonexistent, so they can not be selected for acquisition. They are zero padded in the image, the known string could not be acquired. When using the most extensive setting -"r" to acquire all resources, the system crashed before the entire memory could be acquired.

Moonsols Win64dd is an exception in this test, because it allows to choose a mode that acquires the entire physical address space. In our test this did acquire the known string from the first 3 hidden memory segments. However the machine crashed and rebooted while imaging the reserved memory region containing the 4th segment. This resulted in the image file being incomplete, missing the last hidden segment.

## 4. Improving memory acquisition

Previous sections illustrated some common weaknesses in current memory acquisition tools. Specifically, the use of non-standard or undocumented API imports makes the tool easily identifiable, allowing malware to install simple, stable hooks specifically targeting forensic agents, while leaving the rest of the system unaffected.

Our goal is to make memory acquisition more resilient to malware subversion by utilizing the hardware itself, rather than relying on kernel APIs. Our driver is therefore not vulnerable to the simple anti-forensic techniques demonstrated above. Additionally, not using exotic APIs, makes it harder to differentiate our acquisition driver from ordinary drivers without thorough code analysis.

### 4.1. Hardware-based physical memory layout detection

As discussed in Section 2.1, obtaining the physical memory map via BIOS or EFI service routines can only be run in real mode, and this can only be done early in the operating system's boot sequence.

Section 2.1 also points out that data may be hidden in reserved regions which are not used by the operating system. Forensic memory acquisition tools should aim to recover all available data, including data in reserved regions. However the danger with reading DMA mapped device memory is that the hardware may become activated, crash the system or corrupt data.

Therefore, rather than finding the memory regions which are safe to read (e.g. via the `MmGetPhysicalMemoryRanges()` routine), we instead directly enumerate the memory ranges which are not safe to read, and avoid those.

The process of configuring PCI plug and play devices is well documented and follows a standard configuration protocol (PCI Special Interest Group, 2002). The legacy PCI configuration protocol, uses two special IO ports (`PCI_CONFIG_DATA` is at $0\times$CFC and `PCI_CONFIG_ADDRESS` is at $0\times$CF8) to read the configuration space of all devices and secondary PCI buses on the main system bus. A modern and more efficient configuration method is provided by PCI Express (PCI Special Interest Group, 2005), but most hardware also supports the old protocol.

It is therefore possible to enumerate all active PCI devices, and retrieve their Base Address Register (BAR) configuration and DMA buffer sizes. Secondary buses on the main PCI bus must also reserve memory ranges for themselves, which can also be read using this method (PCI Special Interest Group, 1998). It should be noted here, that querying the PCI controller involves IO port assembly instructions and not operating system routines - hence this can not be hooked in the usual way.

In addition to DMA buffers, standard memory regions that are assigned to hardware, such as the ISA bus hole ranges, are automatically added to the list of excluded memory ranges.

Note that there might be other devices that are not registered on the PCI bus but might have memory mapped into the physical address space. Examples include the High Precision Event Timer (HPET) on the LPC Bus, as well as local APICs, I/O APICs and BIOS ROMs. While it is possible to locate MMIO ranges used by these devices by parsing the MP (Intel, 1997) or ACPI Tables (Hewlett-Packard et al., 2011), excluding them from acquisition might not be wise in this context. These tables are not expected to be updated after the system has booted (Hewlett-Packard et al., 2011; Intel, 1997), making them an easy target for rootkit manipulation. While there are programming rules enforcing register alignment for reads in some of these MMIO regions (Intel, 2004, 2013b), reading them does not violate any of the documented constraints and did not cause any problems in our experiments. Of course we weren't able to

test all available hardware configurations. Some devices might exist that cause problems when being read and don't adhere to the PCI specifications. A broad evaluation of different devices should be focus of future research.

Once these memory ranges are obtained, we need to determine the highest addressable physical memory in the system. Whilst the OS stores this value internally, we do not wish to query the OS. Calculating or obtaining this value from the hardware is not an obvious matter since, as described in Section 2, the memory chipset may not back some reserved regions with RAM at all. Instead the physical address space is extended, leading to the highest physical memory address being much larger than the total memory installed. We therefore allow this setting to be user selectable, and prefer to acquire past the end of physical memory (yielding simply zero blocks).

### 4.2. Hardware-based mapping of physical memory

Rather than rely on kernel APIs to set up the required page table mapping to allow physical memory pages to be accessible from the kernel's virtual address space, we rely on direct manipulation of page tables. Ordinarily, interfering with the kernel's management of the page tables is risky due to the required synchronization requirements and detailed understanding of kernel page table management, especially on multi-core systems, where race conditions can occur by simultaneous manipulation of page table by different cores.

To avoid directly manipulating the kernel's own page tables, we ask the kernel to allocate a single non-pageable page for our own use. This causes the kernel to create a page table entry (PTE) to our own private allocation. Since this memory is non-paged, we can be confident that the PTE mapping to this memory will not be changed while we are using it, guaranteeing that our driver has exclusive access to this PTE.

There are multiple methods to achieve this, depending on the operating system. On Windows, the regular non-paged pool allocations usually have large page PTEs, and hence can not be used for our technique. Instead, we create an unused, page sized, static char array for this purpose within the driver's binary. We then call the `MmProbeAndLockPages` routine to ensure this allocation does not get paged out for the life of the driver. On Linux we use `vmalloc()` and on Mac OS X `IOMallocAligned()`. The created page-sized mapping is further referred to as the "Rogue Page". Of course we could simply use the APIs the operating system offers to drivers that need to manipulate the page tables. However, by doing something very common like allocating memory, we keep a lower profile and make it harder for malware to identify our module as a memory acquisition driver.

The driver then walks the page tables directly using the value of CR3 to find the Directory Table Base (DTB), and determines the PTE's virtual memory address. While page table addresses are usually specified in the physical address space, most operating systems have PTEs permanently mapped into the kernel address space for quick access. As illustrated in Fig. 3, the driver first obtains the address for the Page Map Level 4 (PML4) from the `CR3` register. It then uses parts of the virtual address of the rogue page, to locate the corresponding Page Table Pointer Table entry (PTPTE) and finally the Page Table entry (PTE). The PTE, in turn, refers to the Page Frame Number, which is the physical offset of the page divided by the page size.

For each physical page we wish to access (further referred to as the Target Page), the driver changes the Page Frame Number in the PTE to match the physical address of the target page. It then flushes the virtual address of the rogue page from the Translation Look-aside Buffer (TLB). All further reads from the virtual address of the rogue page will now be performed from the physical target page by the system's MMU. Once the TLB is flushed, the MMU will automatically translate our buffer's virtual address into the physical page in hardware.

This algorithm does not call any operating system functionality once the rogue page has been locked into memory. We simply write to the PTE address directly, and copy memory out of the rogue page to the user space buffers.

Note that depending on the caching type in the PTE that holds the original mapping to a physical page, writing to the *rogue* mapping could cause cache incoherence and is strongly discouraged. Thus, operating systems usually prevent the creation of an incompatible second mapping to the same physical page (Vidstrom, 2006). However, this is not a problem for the purpose of memory acquisition, as we only need to read from this mapping. Of course it is possible that reading from the rogue page results in stale data that has already been replaced in one of the CPU caches. Because of the inherent atomicity issues that come with any software based acquisition procedure (Vömel and Freiling, 2011), we don't believe this to be a problem. By effectively bypassing the operating system in the creation

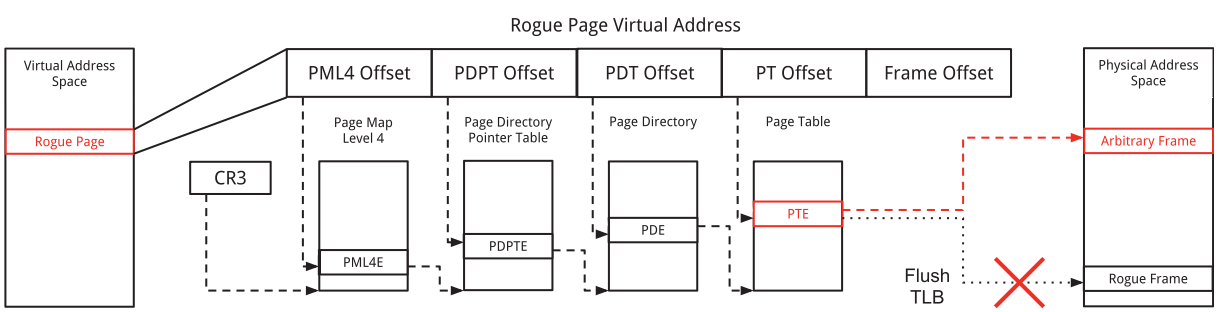


**Fig. 3.** Complete Technique.

of the *rogue* mapping, this approach is even more powerful than using one of the APIs that would prevent the mapping in some situations.

### 4.3. Evaluation

We have integrated these techniques into the open source acquisition tool Winpmem. We then tested it against all anti-forensic techniques presented in Section 3 on a Windows 7 ×64 virtual machine as well as a physical Intel Ivy-Bridge System with 4 GB of RAM running Windows 8 ×64. Both systems were equipped with Intel 510 Series Solid State Drives, to minimize the storage bottleneck when writing the image. The tool was able to acquire the entire address space on both fully compromised systems with a broken `KDBG` and hooks in `MmMapMemoryDumpMdl()` and `MmGetPhysicalMemoryRanges()`. It also correctly acquired the contents of hidden memory.

We didn't have any problems with non-deterministic stability issue, like experienced with Win64dd and WindowsMemoryReader when acquiring the entire address space. Actually, we believe our approach to be generally more stable than current established techniques, because we are in no danger to trigger any bug checks in the kernel, when running on a low IRQL level.

It is not possible to do an exact performance evaluation against other approaches, as we acquire a large amount of memory that current tools simply can't, which is why we obviously have to read and write more data. However, in comparison to current techniques our approach is significantly slower. For example the unpatched version of Winpmem wrote a zero padded image of the 4.8 GB physical address space on our test machine in 22 s at 218 MB/s. Our tool created a 6.3 GB image in 3 min and 20 s, about 9 times slower at 31.5 MB/s. While this does have a negative impact on the atomicity of the image, we believe to be sufficient in real world scenarios, given the benefits the technique provides. Depending on the chosen storage medium, the bottleneck could also be the network or harddisk (We are still almost three times faster than 100BASE-T Ethernet).

Furthermore, we believe IO throughput can be significantly improved in the future, by mapping bigger ranges of memory. Our current implementation writes each page separately, which can not utilize the large file IO buffers of the operating system in an optimal way.

## 5. Discussion

Our technique is simple to implement. Since we do not rely on the OS for mapping physical pages or enumerating memory, simple hooking techniques, such as demonstrated in the ddefy tool are ineffective. By flushing the TLB completely just before copying the memory out, we remove the possibility of a split TLB type attack (Sparks and Butler, 2005). Also, the technique is completely operating system independent and works on all systems with the x86 architecture. We have successfully tested it on Windows, Linux, and Mac OS X systems with implementations based on the Linux pmem (Cohen, 2011) Winpmem (Cohen, 2012a) and OSXPmem (Stüttgen, 2012) drivers.

The following discussion evaluates our solution against possible anti-forensic attacks that a rootkit might implement.

### 5.1. Access to ring 0

Our memory acquisition technique depends upon being able to run in kernel mode. The obvious countermeasure a rootkit may implement is to prevent our driver from being loaded into kernel mode – for example, by hooking the Service Control Manager (SCM) interface.

Although our driver requires access to ring 0, there are few signatures that can be employed to detect our driver's intentions. Currently, it is trivial for a rootkit to identify a memory acquisition driver simply by inspecting the module's import table. This is especially true for a driver that uses undocumented functions which are not usually imported by legitimate drivers (e.g. `MmMapMemoryDumpMdl` as is used by the Win64DD driver (MoonSols, 2012)).

By rejecting the driver from loading, the rootkit reveals its existence, so it must only do this as a last resort, when it is certain that a forensic agent is running. Since our driver does not import any special OS functions, a much more thorough analysis must be conducted to determine its intentions.

### 5.2. Interception of data buffers

Once the physical memory is accessible, memory acquisition drivers typically write it to disk, or copy it to user buffers. A simple anti-forensic technique is to mark certain regions of memory using a magic string and then hooking all kernel file operations and kernel to user space copy operations, searching for the magic strings. If these are found, the rootkit has an opportunity to scrub the data.

This attack can be easily circumvented by encrypting or obfuscating the raw data as it is copied to userspace. Our solution can use simple RC4 encryption to prevent the rootkit from identifying the data as it is passed from kernel space to user space.

### 5.3. Debug registers

An effective anti-forensic technique is the use of the debug registers to alert the rootkit of reading certain memory regions (halfdead, 2008). Modern CPUs have a set of debug registers which can be used to set hardware breakpoints on memory access (Intel, 2013c). The processor can contain four distinct memory access breakpoints stored in debug registers `D0–D3`. Ordinarily, the debug registers contain a virtual address and will trap when the processor accesses the breakpoint in the virtual address space. This kind of breakpoint is ineffective against our imaging driver since, in the kernel's virtual address space, we are accessing our own private memory page. The PTE manipulation simply makes the desired physical memory page available through this virtual page.

However, the Debug Control Register (D7) can configure the breakpoint to be an I/O read or write breakpoint. This has the effect of generating a trap when the CPU executes an `in` or `out` assembler op code with an operand matching the breakpoint. Our acquisition process will not be affected

by this (since we do not use `in/out` instructions to read physical memory). Unfortunately, our PCI introspection routine which is used to read the DMA memory regions does use these instructions as part of obtaining the physical memory layout.

A malicious rootkit can thus hook our PCI enumeration routine and cause a "fake" PCI device to appear on the system bus by returning a pre-fabricated configuration space buffer when querying for a specific device id (PCI Special Interest Group, 2002). This configuration can claim that this fake device is occupying a specific memory region for a DMA buffer, causing our tool to exclude it from the imaging process.

### 5.4. Shadow page tables

Another weakness of our technique is its reliance on the operating system on finding the page tables in the first place. All addresses in CR3 and Page Tables are physical addresses. Hence walking the Page Tables requires a physical-to-virtual translation function, which relies on the operating system. A rootkit could hook this translation function and employ a shadow-paging approach to hook write access to PTEs. This would require removing write access to the page tables and hooking the page fault handler (Ooi, 2009).

There is no way to prevent a rootkit from doing this, nor to detect it has happened. However, there is a simple solution for this problem. If the memory driver creates its own page tables and changes CR3 to point to these custom tables, we can remain in complete control over the translation process without alerting the rootkit. The details of this implementation are left for future research.

## 6. Conclusion

It is commonly believed that running any software, and especially a forensic agent, on a compromised system can not be trusted since the system may be hooked so as to "lie" to the forensic tool. Since memory acquisition tools must run on potentially compromised systems, they must be vulnerable to subversion by a pre-installed malware. However, we believe this view fails to take into account the practical aspects of rootkit engineering - which are to maintain system stability and hide from forensic agents.

We believe that by advancing the robustness of acquisition techniques, we are furthering the "arms race" between malware and forensic tools by raising the level of complexity required to adequately recognize and subvert forensic agents. Although our technique is not perfect, the countermeasures discussed in Section 5 illustrate the level of complexity required to successfully subvert acquisition is much increased. For example, to successfully counter our acquisition technique, the rootkit may need to manipulate kernel page tables directly, adding complexity and reducing the rootkit's ability to keep the system stable after it has been compromised.

Although we found problems with all tested tools, we were able to add our technique to the open source WinPmem acquisition tool and also correct the errors we identified in Section 3.1.1. The flexibility offered by an open source tool is obvious, as the tool can be recompiled with only the needed acquisition methods linked in, controlling its import tables. The tool can also be customized and resigned such that a rootkit is unable to recognize it as a forensic agent. This flexibility is missing in commercial tools, many of which even have clear copyright strings embedded within the driver's binary, making it trivial for a rootkit to identify them and prevent them from loading into the kernel.

## Acknowledgements

We would like to thank Matthieu Suiche for kindly providing us with an evaluation version of Moonsols Win64DD and DumpIt. Furthermore we would like to thank Darren Bilby, Stefan Voemel and Felix Freiling for reading a previous version of this paper and giving us valuable feedback and suggestions for improvement. We would also like to thank Tavis Ormandy and Halvar Flake for their valuable discussions and suggestions.

## References

AccessData. FTK Imager. http://www.accessdata.com/; 2012.
ATC-NY. MacMemoryReader. http://cybermarshal.com/index.php/cyber-marshal-utilities/mac-memory-reader; 2012a.
ATC-NY. WindowsMemoryReader. http://cybermarshal.com/index.php/cyber-marshal-utilities/windows-memory-reader; 2012b.
Bilby D. Low down and dirty: anti-forensic rootkits. In: Proceedings of Black Hat Japan 2006.
Boileau A. Hit by a bus: physical access attacks with Firewire. In: Ruxcon 2006.
Carrier B, Grand J. A hardware-based memory acquisition procedure for digital investigations. Digital Investigation 2004;1(1):50–60.
Cohen M. PMEM – physical memory driver. http://code.google.com/p/volatility/source/browse/branches/scudette/tools/linux; 2011.
Cohen M. The PMEM memory acquisition suite. http://code.google.com/p/volatility/source/browse/branches/scudette/tools/windows/winpmem; 2012a.
Cohen M. Volatility Technology Preview. http://code.google.com/p/volatility/source/browse/branches/scudette/; 2012b.
Cohen M, Bilby D, Caronni G. Distributed forensics and incident response in the enterprise. Digital Investigation 2011;8:S101–10.
Florio E. When malware meets rootkits. Virus Bulletin 2005.
Garner G. Windows implementation of DD. http://gmgsystemsinc.com/fau/; 2006.
Grizzard J. Towards self-healing systems: re-establishing trust in compromised systems. Ph.D. thesis. Georgia Institute of Technology; 2006.
halfdead. Mystifying the debugger for ultimate stealthness. Phrack 2008. 0x0c, 0x08.
Harris R. Arriving at an anti-forensics consensus: examining how to define and control the anti-forensics problem. Digital Investigation 2006;3(Suppl. 0):44–9. Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).
Haruyama T, Suzuki H. One-byte modification for breaking memory forensic analysis. http://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf; 2012.
Hewlett-Packard, Intel, Microsoft, Phoenix-Technologies, Toshiba. ACPI specification 5.0. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf; 2011.
Inoue H, Adelstein F, Joyce RA. Visualization in testing a volatile memory forensic tool. Digital Investigation 2011;8:S42–51.
Intel. MultiProcessor specification, In: http://download.intel.com/design/archives/processors/pro/docs/24201606.pdf; 1997.
Intel. HPET specification. http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf; 2004.
Intel. Intel 64 and and IA-32 Architectures software developer's manual volume 3A: system programming guide 2013. Vol. 3A. Chapter [Chapter 3], Protected-Mode Memory Management.

Intel. Intel 64 and and IA-32 Architectures software developer's manual volume 3A: system programming guide 2013. Vol. 3B. Chapter [Chapter 10] Advanced Programmable Interrupt Controller.

Intel. Intel 64 and and IA-32 Architectures software developer's manual volume 3A: system programming guide 2013. Vol. 3B. Chapter [Chapter 17].2 Debug Registers.

King ST, Chen PM. SubVirt: implementing malware with virtual machines. In: Security and privacy, 2006 IEEE symposium on. IEEE; 2006. p. 14.

Mandiant. Memoryze. http://www.mandiant.com/resources/download/memoryze; 2011.

Martignoni L, Fattori A, Paleari R, Cavallaro L. Live and trustworthy forensic analysis of commodity production systems. In: Recent advances in intrusion detection. Springer; 2010. p. 297–316.

Microsoft Corporation. Kernel patch protection: FAQ. http://msdn.microsoft.com/en-us/library/windows/hardware/gg487353.aspx; 2006.

Microsoft Corporation, Phoenix Corporation, Toshiba Corporation, Hewlett-Packard Corporation, Intel Corporation. Advanced configuration and power interface specification [Chapter 14]. In System address map interfaces 2006;vol. 3. Describes the INT 15H, E820H - Query System Address Map interface.

Milkovic L. Defeating Windows memory forensics. http://events.ccc.de/congress/2012/Fahrplan/events/5301.en.html; 2012.

MoonSols. Windows memory Toolkit. http://moonsols.com/product; 2012.

Ooi T. Stealthy Rootkit: how bad guy fools live memory forensics?. http://www.slideshare.net/a4lg/stealthy-rootkit-how-bad-guy-fools-live-memory-forensics-pacsec-2009; 2009.

PCI Special Interest Group. PCI-to-PCI bridge architecture specification 1998. [Chapter 3].2. PCI-to-PCI Bridge Configuration Space Header Format.

PCI Special Interest Group. PCI local bus specification 2002 Chapter [Chapter 6] Configuration Space.

PCI Special Interest Group. PCI express base specifications 2005. Revision 1.1.

Rusakov V. TDL4 Rootkit. http://www.securelist.com/en/analysis/204792157/TDSS_TDL_4; 2011.

Rusakov V. XPAJ: reversing a Windows x64 Bootkit. http://www.securelist.com/en/analysis/204792235/XPAJ_Reversing_a_Windows_x64_Bootkit#5; 2012.

Russinovich M. New Win2K RC Kernel APIs. The Systems Internals Newsletter 1999;1:5.

Rutkowska J. BluePill 2006.

Rutkowska J. Beyond the CPU: defeating hardware based RAM acquisition. In: Proceedings of BlackHat DC 2007.

Schatz B. BodySnatcher: towards reliable volatile memory acquisition by software. Digital Investigation 2007;4:126–34.

Sparks S, Butler J. Shadow Walker: raising the bar for Rootkit detection. In: Proceedings of Black Hat Japan 2005. p. 504–33.

Stüttgen J. OSXPmem. http://code.google.com/p/pmem/wiki/OSXPmem; 2012.

Sylve J. LiME – Linux memory extractor. In: ShmooCon' 12 2012.

Vidstrom A. Forensic memory dumping intricacies PhysicalMemory, DD, and caching issues. http://ntsecurity.nu/onmymind/2006/2006-06-01.html; 2006.

Vömel S, Freiling FC. A survey of main memory acquisition and analysis techniques for the windows operating system. Digital Investigation 2011;8(1):3–22.

Vömel S, Freiling FC. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. Digital Investigation 2012;9:2.

Walters A. Volatility: an advanced memory forensics framework. https://code.google.com/p/volatility/; 2007.

Wang J, Zhang F, Sun K, Stavrou A. Firmware-assisted memory acquisition and analysis tools for digital forensics. In: Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on. IEEE; 2011. p. 1–5.

Wundram M, Freiling F, Moch C. Anti-forensics: the next step in digital forensics tool testing. In: Proceedings 7th International Conference on IT Security Incident Management & IT Forensics (IMF) 2013.

Yu M, Lin Q, Li B, Qi Z, Guan H. Vis: virtualization enhanced live acquisition for native system. In: Proceedings of the Second Asia-Pacific Workshop on Systems. ACM; 2011. p. 13.

Zovi D. Hardware virtualization Rootkits. http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf; 2006.