Digital Investigation 12 (2015) S50–S60

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2015 Europe

Acquisition and analysis of compromised firmware using memory forensics

Johannes Stüttgen*, Stefan Vömel, Michael Denzel

Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstraße 3, 91058 Erlangen, Germany

Keywords:
Memory forensics
Memory acquisition
Live forensics
Firmware rootkits
Incident response

A B S T R A C T

To a great degree, research in memory forensics concentrates on the acquisition and analysis of kernel- and user-space software from physical memory to date. With the system *firmware*, a much more privileged software layer exists in modern computer systems though that has recently become the target in sophisticated computer attacks more often. Compromise strategies used by high profile rootkits are almost completely invisible to standard forensic procedures and can only be detected with special soft- or hardware mechanisms. In this paper, we illustrate a variety of firmware manipulation techniques and propose methods for identifying firmware-level threats in the course of memory forensic investigations. We have implemented our insights into well-known open-source memory forensic tools and have evaluated our approach within both physical and virtual environments.

Introduction

Over the last decade, forensic practitioners have increasingly started to perceive the value of volatile information in system RAM for the outcome of a case. Particularly when suspecting potentially-installed malicious applications on a compromised host, so-called *memory forensic* investigation techniques can frequently confirm assumptions and yield results more quickly than traditional, persistent data-oriented approaches (Walters and Petroni, 2007). The majority of research has thereby concentrated on *memory analysis*, i.e., identifying relevant data structures and subsequently extracting pieces of evidence (for an overview of current practices, please refer to Ligh et al. (2010) and Vömel and Freiling (2011)). The process of securing said data in a forensically-sound manner on the other hand, i.e., *memory acquisition*, has been described to a significantly lesser degree in the literature. This fact is quite surprising, considering that recent works have indicated quality deficiencies in popular imaging solutions under certain conditions that may severely impede or even prevent thorough artifact examination at a later time (Vömel and Stüttgen, 2013; Stüttgen and Cohen, 2013).

In order to mitigate this research gap, we will discuss several scenarios in this paper that exemplify various hurdles and pitfalls modern memory acquisition technologies must overcome. With respect to this, we will focus on specific system manipulation strategies on the *firmware* level, in contrast to other studies that illustrate issues in kernel or user space (see Ligh et al., 2010; White et al., 2013). Although only a comparatively small amount of malicious programs distributed in the wild are known of explicitly attacking base components and features of a system, the lower machine layers have received broader attention by malware authors in the last years, especially with regard to the development of *Advanced Persistent Threats* (APTs) and special *rootkits* that are remarkably difficult to detect and remove (Shields, 2008; Davis et al., 2010). As we will see, however, the vast number of

* Corresponding author.
E-mail addresses: johannes.stuettgen@cs.fau.de (J. Stüttgen), stefan.voemel@cs.fau.de (S. Vömel), m.denzel@cs.bham.ac.uk (M. Denzel).

http://dx.doi.org/10.1016/j.diin.2015.01.010

forensic programs and toolkits available on the market to date are unfortunately ill-prepared for dealing with such threats and fail at properly duplicating all sources of a computer's physical address space in which malicious code may reside.

Contributions of the paper

In the scope of the paper, we will see that most firmware code and data can be accessed over the memory bus. We present a broad survey of firmware rootkit techniques and the traces such threats leave in a system. To facilitate acquisition and analysis of potential infections, we have developed methods for enumerating, imaging, and examining firmware components and implemented them into free, open-source utilities. The reliability of these utilities is evaluated with the help of a proof-of-concept ACPI rootkit as well as manipulated BIOS ROMs.

Related work

As we have already indicated, the development of malicious programs that are directly stored and executed on the hardware or firmware layer has increased over the last years. Stewin and Bystrov (2012), for instance, outline techniques for silently capturing keystrokes or extracting sensitive information such as cryptographic keys from a running machine with the help of *Direct Memory Access* (DMA). Embleton et al. (2008) illustrate how the processor's *System Management Mode* (SMM) may be misused for storing malevolent code, while Tereshkin and Wojtczuk (2009) manipulate the chipset of the *Memory Controller Hub* (MCH) for these purposes.

With respect to forensic investigations, other authors have attempted to leverage specific features or components of the hardware. Most notably, the IEEE 1394 (*FireWire*) interface or the more recent *Thunderbolt* port permit duplicating the contents of memory from a target system quite easily (see Zhang et al., 2011; Maartmann-Moe, 2012). In contrast, Wang et al. (2011) as well as Reina et al. (2012) describe a prototype implementation for creating a consistent copy of the physical address space by entering the System Management Mode. Memory acquisition based on DMA operations over a network card is subject of the work of Balogh and Mydlo (2013). Last but not least, obtaining the contents of firmware ROMs is the specific goal of the *Copernicus* project (Butterworth et al., 2013). Copernicus permits extracting firmware data directly over the *Serial Peripheral Interface* (SPI) bus. Because this approach is vulnerable to malicious software running in System Management Mode, the latest implementation utilizes the *Intel TXT* extensions (Intel Corporation, 2014c) to isolate the acquisition module from other parts of the system (Kovah et al., 2014).

Most of the previously described technologies have not or only insufficiently been evaluated. First, more extensive studies concerning the quality of imaging applications were presented by Vömel and Stüttgen (2013) and Stüttgen and Cohen (2013). We will further elaborate upon these works and assess the performance of common imaging products, specifically when sophisticated malware species are present.

Outline of the paper

The remainder of this paper is outlined as follows: In Section Background, we briefly define the type of malicious program we will concentrate on throughout our discussion. We also describe standard approaches for acquiring a copy of main memory from different operating systems and give an overview of several firmware features and technologies that help the reader better understand later parts of our work. A survey of current firmware rootkit techniques and their implications for memory forensic investigations is presented in Section Rootkit Strategies for Compromising Firmware. Methods for enumerating and acquiring firmware code and data are subject of Section Firmware Acquisition Using Memory Forensics. In Section Firmware Analysis, we discuss the analysis of the acquired data, followed by an evaluation of how well these approaches are already incorporated in common forensic suites and applications in Section Evaluation. Aspects and limitations that need to be considered when applying the respective concepts in real-world investigations are discussed in Section Discussion. We conclude with a short summary of our work and indicate various opportunities for future research in Section Conclusion.

Background

In the scope of this paper, we discuss challenges and pitfalls for memory acquisition solutions that operate within a hostile environment, i.e., the target system is likely to have been affected by malicious programs, and the integrity of the machine cannot be trusted. In Section Rootkits, we specify the conditions the respective acquisition technologies must cope with in more detail and give a more thorough definition of the *rootkit* term. In Section Memory Acquisition Process, we outline basic approaches for duplicating volatile information on different operating systems. The characteristics of major firmware technologies are subject of Sections System Firmware–ACPI. The information presented in these sections is mainly based on the work of Salihun (2006), Dice (2013) as well as the respective vendor specifications (Intel Corporation, 2014b; PCI-SIG, 2010) and is required to better understand the technical complexities illustrated in later parts of this paper. Readers who are already familiar with rootkits and have a solid knowledge of the BIOS, PCI, and ACPI environment may safely skip these explanations and directly proceed to Section Rootkit Strategies for Compromising Firmware.

Rootkits

A particularly sophisticated type of malicious programs are so-called *rootkits*. Rootkits are defined to consist of a set of programs and code that allows a permanent or consistent, undetectable presence on a computer (Hoglund and Butler, 2005, p. 4). They are intentionally designed to project a manipulated view to the system user in order to achieve the primary objectives *concealment*, *surveillance* as well as

command and control of the target host (Blunden, 2009). Depending on the execution mode they operate in, rootkits are commonly divided into distinct classes. These include the *user* as well as *kernel* space, the *system library* level, the *hypervisor* level (sometimes also referred to as *ring -1*, in reference to the ring architecture of the x86 processor family), and finally, the *firmware* level (see Shields, 2008). In this paper, only the latter layer will be in the focus of our discussion. Introductions concerning samples running in higher levels of the system abstraction can be found in the works of Vieler (2007) and Davis et al. (2010).

Memory acquisition process

One of the most common and convenient methods for obtaining a copy of volatile memory is the use of an imaging utility. Due to security restrictions, the respective programs can usually not be entirely executed in user space (e.g., see Microsoft Corporation, 2013), but must rather inject a special kernel driver or module into the core of the operating system. With regard to Microsoft Windows, frequently-applied procedures involve either reading from the `\\.\Device\PhysicalMemory` section object or calling one of the memory-mapping APIs of the kernel, i.e., `MmMapIOSpace()` or `MmMapMemoryDumpMdl()`. On Linux platforms, the predominant approach is similar and relies on parsing the `iomem_resource` tree and mapping memory using the `kmap()` API (Sylve, 2012).

One distinctive disadvantage of software-based imagers is that they may easily fall prey to manipulation (Stüttgen and Cohen, 2013). Moreover, because they run in parallel to other system processes, the state of memory is in a constant state of flux, leading to a drastically reduced level of *atomicity* of the generated snapshot (Vömel and Freiling, 2012). To mitigate these drawbacks, researchers have outlined various alternatives that explicitly leverage the capabilities of specific *hardware* features and technologies. These include, for instance, the Intel VMX instruction set for accessing physical memory from an isolated *hypervisor* environment (Martignoni et al., 2010; Yu et al., 2012) as well as patching the system BIOS for transparently and reliably writing out the contents of memory via the processor's *System Management Mode* (SMM, Wang et al., 2011; Reina et al., 2012). An overview and comparison of the individual propositions can be found in the work of Vömel and Freiling (2011).

System firmware

The system firmware, i.e., the *Basic Input Output System* (BIOS) or the *Extensible Firmware Interface* (EFI) on more modern systems, is the first program that runs on the CPU when a computer is turned on. The respective code is saved in a non-volatile storage area, usually an *EEPROM*, on the mainboard of the machine. The north- and southbridge are initially configured to map the contents of this ROM into the physical address space at an architecture-specific location inside the first megabyte of the address space.[1] The ROM is also aliased in a way that 16 bytes are mapped to the physical address `0xFFFFFFF0`, at the CPUs *reset vector* (Intel Corporation, 2014b).

The firmware code initially runs on the ROM chip. It then moves from ROM into RAM by manipulating special registers, the so-called *Programmable Attribute Map* (PAM), to shadow the ROM-mapped regions with RAM and uncompressing its code image into memory. Firmware then starts initializing devices on the PCI bus and maps their registers and memory into the physical address space as required. It is during this phase that the BIOS and EFI start to differ greatly, and the final layout of the physical address space is determined (Dice, 2013; UEFI Forum, 2014).

BIOS

The BIOS runtime environment operates in 16-bit real mode. It is responsible for creating an *Interrupt Vector Table* (IVT) in order to support a set of simple operations, e.g. sending output to the screen or reading data from a hard disk. The latter functionality is required to drive the bootstrapping process and load the boot manager as well as the operating system eventually. Precisely, the BIOS reads the code of the boot manager into memory and hands over control.[2] The boot manager, in turn, loads the operating system and further prepares the system environment. For these tasks, the primary BIOS services are used. In the last step, the operating system takes over interrupt handling by replacing the IVT with an appropriate *Interrupt Descriptor Table* (IDT). Thereby, direct access to the BIOS services is lost.

EFI

Contrary to BIOS-based firmware, EFI operates in 32-bit protected mode. The boot process comprises a distinct security phase (*SEC*) in which the integrity of the firmware is explicitly checked, and secure booting is facilitated. In a second, so-called *Pre-EFI Initialization* (PEI) phase, similar tasks as during early BIOS initialization are performed. However, at the end of this phase, EFI provides a structured *Driver Execution Environment* (DXE) for drivers and services. These are not loaded from the MBR but from the file system of a designated *EFI System Partition*. The location of the respective images are specified in Non-Volatile RAM (NVRAM) on the mainboard. Drivers, bootloaders, and the OS can interact with the firmware through specific protocols. After the operating system has started, it still has access to some firmware interfaces through the so-called *EFI runtime services*.

PCI option ROMs

Because the system firmware has no internal knowledge of the functionality and specifics of attached devices, complementary code that is required for unique device initialization is stored on a separate chip. With regard to hardware cards that are connected via the PCI bus, the respective instruction set is saved in a read-only area on the device, the so-called *PCI Option ROM*. In order to be

[1] On most x86 systems this range spans from 0xF0000 to 0xFFFFF.

[2] The boot manager is resident in the *Master Boot Record* (MBR) on the first hard disk.

successfully accessed, a PCI card must implement a *configuration space*, i.e., 256 bytes of registers that control the behavior of the PCI device at all times (Salihun, 2013, p. 9). Part of this space are two special registers at offsets `0x04` (Command Register) and `0x30` (Expansion ROM Base Address Register), each of them contains a bit that must be explicitly set before operations on the ROM can be executed (Salihun, 2013).

The ROM itself is typically composed of a number of separated code *images* that apply to different processor architectures and platforms. For a given machine, always only one particular image is chosen and run though (Salihun, 2012). The format and contents of an image are standardized and incorporate a header and PCI data structure (see Salihun, 2006). The header includes a signature bytestring and a pointer that references the adjacent data structure. The PCI data structure, in turn, is located in the first 64 kilobytes and contains several identifiers with device and vendor information as well as parameters for the image and code size. These values are later needed for dynamically fitting the code base into memory. Precisely, device-specific tasks are not carried out in-place, but the system firmware rather transfers the image file into RAM below the 1-MB boundary in the course of the *Power-On Self Test* (POST).[3] A custom `INIT` function that is invoked during this process is responsible for adjusting the image size to the actual runtime requirements. Thereby, available memory can be used as efficiently as possible.

For further explanations concerning the previously described steps, the reader is referred to the official specification (PCI-SIG, 2010).

ACPI

The *Advanced Configuration and Power Interface* (ACPI) defines a platform-independent interface specification comprised of both software and hardware elements for device configuration and power management (ACPI Promoters Corporation, 2013, p. 4). The interface consists of three components, i.e., the *ACPI System Description Tables*, the *ACPI Registers*, and the *ACPI BIOS*. Moreover, system-specific drivers facilitate communication between the hardware and software layer and implement methods for easily accessing the provided functionality.

Of the three interface components, the ACPI System Description Tables are most interesting. They contain structured information about the computing environment, available devices as well as their capabilities, and may be read out by the *operating system-directed configuration and power management* (OSPM) unit. The information is stored in so-called definition blocks and encoded in a special language, the *ACPI Machine Language* (AML). AML may be converted into the *ACPI Source Language* (ASL) format for better readability, a step we will describe in more detail in a later part of this paper.

In order to process a definition block, the OSPM must first locate a pointer to the main ACPI table, i.e., the *Root System Description Table* (RSDT).[4] For this purpose, a signature-based scan in memory is applied.[5] Once the table has been found, pointers to various other important system tables can be identified, e.g., to the *Fixed ACPI Description Table* (FADT) or to the *Differentiated System Description Table* (DSDT). While the former saves the addresses of a number of power management-related register blocks, the latter supplies the implementation and configuration information about the base system (ACPI Promoters Corporation, 2013, p. 19).

Rootkit strategies for compromising firmware

In this section, we present a survey concerning the current state of the art in x86 firmware exploitation. We illustrate attacks on the BIOS and EFI environment, respectively, on PCI Option ROMS as well as ACPI, and outline the respective traces that may be recoverable during memory forensics.

While firmware rootkits are highly target-specific and require a lot of in-depth knowledge to develop, malware authors have demonstrated that building working prototypes is feasible, and various approaches have already been adopted by different species “in the wild” (Giuliani, 2013).

BIOS- and EFI-Based attacks

As Bulygin et al. (2014) report, a significant number of BIOS/EFI attacks were successfully carried out in the past. Despite *update signature verification*, *secure boot*, and other security measures at the firmware level, many feasible attack vectors still exist. In the following, we give a brief overview of common system compromise strategies.

Flash protection registers

When an x86 computer is first switched on, the ROM containing the firmware is initially writable through the SPI bus. This functionality is necessary to permit legitimate installation of new firmware updates. On the other hand, before control is handed to the operating system, SPI flash must be properly locked down to prevent software from overwriting the ROM. However, many vendors fail at these tasks and leave the respective areas open for manipulation (Bulygin, 2013; Bulygin et al., 2013). As a consequence, malicious code may flash the firmware ROM directly from kernel space and incorporate malevolent functionality.

Insecure firmware updates

Most BIOS update implementations do not require a cryptographic signature. They process any source file as long as it matches a given format. This flaw was exploited by the *Mebromi* rootkit to infect versions of *Award BIOS* (Bulygin et al., 2014). In contrast, modern firmware technologies based on EFI are more wary of such attack vectors

[3] For legacy devices, the image is mapped in the address space between regions 0xC0000 and 0xDFFFF (see Salihun, 2006).

[4] On modern systems, this structure is superseded by the *Extended System Description Table* (XSDT).

[5] For more information concerning the exact memory locations, please refer to the ACPI specification (ACPI Promoters Corporation, 2013, p. 107).

and attempt to verify update requests more rigorously. However, the respective algorithms may contain unintentional errors and, thus, be susceptible themselves as Wojtczuk and Tereshkin (2009) argue.

Physical attacks

Even with all software measures perfectly implemented, a malicious adversary at an arbitrary position in the supply chain can modify a system's firmware with the help of a flash programmer. As recently outlined by Brossard (2012), these tasks can be easily accomplished using open firmware like *Coreboot* (Minnich, 2014), *SeaBIOS* (O'Connor, 2014), or *iPXE* (Brown, 2014).

Memory forensic implications

All of the previously described attacks ultimately result in reprogramming of the firmware flash ROM. As laid out in Section System Firmware, the ROM chip is mapped on the memory bus from `0xF0000` to `0xFFFFF`. It is therefore possible to include said region in a memory image and analyze the duplicated pages in a later step.

PCI option ROM-Based attacks

Because some PCI devices require custom initialization, system firmware loads and executes any option ROM provided by devices during boot time. This code runs in firmware context while SPI flash is unlocked and can therefore patch the firmware ROM effortlessly. For instance, as Brossard (2012) points out, it is possible to load a bootkit over the built-in Wifi or WiMax devices of the system by flashing a malicious option ROM onto a network card. Thereby, firewalls or intrusion detection systems can be bypassed.

A vulnerable firmware version can also be directly exploited over the network: Triulzi (2010) outlines techniques for remotely reflashing the firmware of specific network cards. Even worse, because PCI devices have unrestricted access to physical memory, additional malicious code may be downloaded in order to further propagate into the local network.

Last but not least, a system may also be compromised using a malicious device that is attached over a hardware port and initiating a subsequent reboot. For example, Loukas (2012) shows how an Apple computer may be infected with malware by connecting a small ethernet adapter to the Thunderbolt port. Because Thunderbolt hardware has direct access to the PCI bus and, thus, to physical memory, the machine is prone to attack, in correspondence to our previous explanations.

Memory forensic implications

The different attacks outlined above result in the introduction of one or more new PCI Option ROMs into the system. Firmware maps these ROMs somewhere into the physical address space and stores a pointer to their location in PCI configuration space. Similarly to the firmware ROM, Option ROMs can also be read over the memory bus, and thus, their code can also be included in a memory image. In addition, because firmware copies Option ROMs for execution purposes into the memory area between addresses `0xC0000` and `0xE0000` (see Section PCI option ROM's), it is possible to consider the respective pages during memory acquisition as well.

ACPI-based attacks

ACPI programs run in kernel space and therefore have full permission to operate on the physical address space. Even though sensitive data structures could theoretically be protected efficiently by filtering the respective instructions in the AML virtual machine, such restrictions have not yet been implemented in any major operating system to the best of our knowledge. Neither Linux up to kernel 3.15 nor Windows up to version 8 seem to have security measures in place to prevent ACPI programs from subverting the system core. Because the ACPI tables are provided by the firmware, they are implicitly trusted. In the presence of a skilled adversary, this assumption may be potentially devastating.

The vulnerability we have just outlined can be exploited in several ways: First, it is possible to patch the ACPI tables directly in the firmware image. In addition, because the tables are copied to memory and must be identified by the operating system, a malicious *bootkit* has the chance of modifying them prior to this process. Alternatively, a manipulated version of the tables can be placed right in front of the firmware-provided copy. Since the location of the pointer to the Root Description System Table is not strictly defined and must be retrieved by the operating system with the help of a signature-based scan (see Section ACPI), only the manipulated version is found, while the original and legitimate code is never executed. As a consequence, an ACPI rootkit may be embedded in either

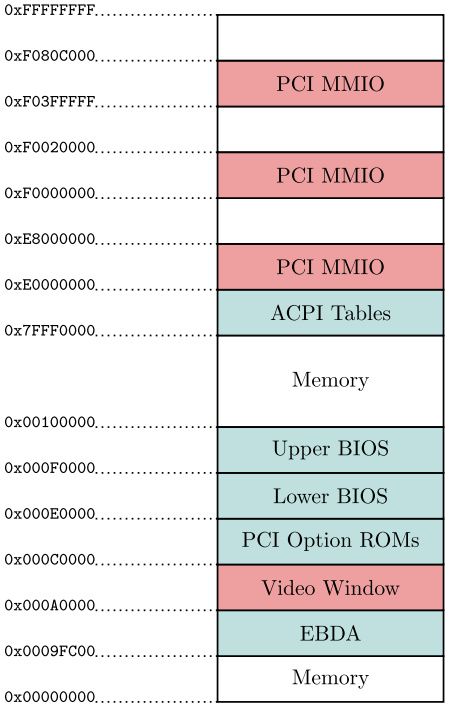

Fig. 1. Firmware memory ranges.

the firmware ROM on the mainboard, in any PCI option ROM, on a connected PCI device, or even as part of an EFI driver module. Detection and removal of such a threat is cumbersome, and most of the described methods even survive a complete wipe of the hard disk.

A proof of concept implementation of an ACPI rootkit for the Linux kernel has already been published (see Heasman, 2006). The rootkit hooks all unused system calls by overwriting the `sys_ni_syscall()` function with the instructions `call ebx; ret;`. Because the `ebx` register is controlled by code running in user space, effectively all programs with an arbitrary privilege level are able to execute code in kernel space. The concept can be used to, e.g., illegitimately gain additional permissions or load additional kernel rootkits even in case kernel module loading has been disabled. However, at the point of this writing, we are not aware that these insights are being actively abused by malicious programs in the wild.

Memory forensic implications

Irrespectively of the original attack vector, the ACPI tables must be placed into RAM in order for the operating system to find and execute them. As a consequence, they can be included into a memory image. If the tables are supplied by the firmware or a malicious Option ROM, all firmware ROMs should be included in the memory image.

Firmware in the physical address space

As we have indicated in the previous sections, there are many regions in the physical address space that contain firmware code and data. A typical layout of the address space, taken from a test machine, is illustrated in Fig. 1. In the figure, regions that are not highlighted contain physical RAM and are marked as “Memory”. Such regions are generally considered by standard physical memory acquisition solutions. In contrast, regions colored in blue contain firmware code or data. As we have pointed out, these regions can also be accessed through the memory bus and have to be duplicated if firmware analysis should be included in the memory forensic investigation. Finally, regions marked in red represent memory-mapped I/O areas and must not be touched as even read operations on these addresses may trigger interrupts on the device and, thus, lead to data corruption or unintentional system crashes. In the worst case, the interrupt may cause the device to exceed its operational parameters, and physical damage is caused.

Firmware acquisition using memory forensics

As pointed out in Section Memory Acquisition Process, memory acquisition software commonly relies on the operating system to identify and map physical memory. Precisely, imaging programs duplicate solely those parts of the address space that are explicitly marked as *RAM*. On Microsoft Windows, the respective regions are usually identified using the `MmGetPhysicalMemoryRanges()` API (Stüttgen and Cohen, 2013). However, further but less common methods do exist: On systems with a BIOS, for instance, the firmware memory map may be queried in *real mode* by setting the `eax` register to `0xE820` and repeatedly invoking interrupt `0x15`. This method is usually applied by the boot manager, and the retrieved information is passed to the operating system for further processing. During runtime, it is not advisable to manually switch to real mode from a driver as this can cause system instabilities. Fortunately, since Windows Vista, the kernel's *Hardware Abstraction Layer* (HAL) includes an undocumented BIOS emulation module that permits drivers accessing BIOS services directly (Chappell, 2010).

Enumeration of the physical address space

Each memory enumeration method provides a unique view of the physical address space. None of them is entirely accurate though, because most devices (especially on the PCI bus) are not directly managed by the operating system but by a vendor-supplied driver. In Fig. 2, a direct comparison of three major sources of information on the physical address space is shown.

The most incomplete view of the physical address space is returned when querying the `MmGetPhysicalMemoryRanges()` API in the windows memory manager. As we have argued in the previous section, memory imaging programs only acquire those ranges that are identified as being *available* by the operating system. For safety reasons, other areas are ignored, including regions of memory that are used by the firmware. For this reason, memory images obtained through this method are not suited for *firmware* examinations. With respect to a test system that we have analyzed, a created memory image only contains two ranges of physical memory (see Fig. 2 on the right). The remaining regions in the image are either zero-padded or not part of the image at all (e.g., when using the *crash dump* approach (Microsoft Corporation, 2011)).

As depicted in the center part of Fig. 2, the BIOS provides a better view of the physical address space. Additionally to the memory regions identified by the memory manager, the BIOS also keeps track of memory used by ACPI. Furthermore, there are 3072 bytes of memory right at the end of the first memory region that the operating system does not know about. This area represents hidden memory that is simply neither used by the BIOS nor the OS (Stüttgen and Cohen, 2013).

EFI offers a similar service to the BIOS memory map. However, because it is a boot service, it is not available anymore once the boot manager has handed control to the operating system. The layout and classification of memory ranges is the same though.

The most exhaustive map of the physical address space can be constructed by intersecting knowledge from the architecture specifications with an enumeration of PCI configuration space. This view is illustrated on the left side of Fig. 2: As discussed in Section System Firmware, the physical address space layout in the first megabyte is well-defined. Areas that are reported here as being “reserved” (or not available at all) are either firmware code, firmware data, or video memory. Please note that the mentioned firmware ROMs in these regions are not actually mapped ROMs anymore. Due to performance reasons, firmware migrates into memory during initialization (see Sections

System Firmware and PCI option ROM's). It is therefore safe to read from these addresses and perform memory acquisition just like with regions that are explicitly marked as RAM.

The memory layout above the first megabyte is not defined and depends on the amount of installed memory as well as on the number of installed devices. Because the latter map registers and memory into this part of the address space, simply iterating through the entire area would be a dangerous process since the respective operations could trigger interrupts and result in undefined behavior, loss of data, or even physical damage to the device (see Section Firmware in the Physical Address Space). Therefore, in order to avoid instabilities, software needs to consult the firmware or operating system upon what areas are safe to read (Stüttgen and Cohen, 2013).

Because the ACPI tables lie somewhere outside of the memory regions reported by the operating system, it is prudent to acquire as much memory from the upper part as safely possible. Furthermore, it is trivial for malware to hook the kernel memory enumeration APIs and hide from the acquisition. Because the real danger of accessing memory outside the available regions comes from touching PCI device memory, it is best to simply exclude all MMIO regions and acquire all remaining sections.

Stüttgen and Cohen (2013) have shown that it is possible to find all MMIO regions of PCI devices by enumerating the PCI configuration space. As explained in Section PCI option ROM's, all PCI devices must implement such a space with special address registers that specify the exact location and size of all MMIO regions (PCI-SIG, 2010). As long as these regions remain untouched, it is safe to read from any other address in the entire physical address space.[6]

To sum up, the blue regions (in web version) on the left of Fig. 2 do not necessarily contain RAM. Reading from parts of the physical address space that are not mapped simply returns zeroes.[7] The resulting image is significantly larger than an image that solely comprises ranges being marked as available but includes the entire firmware code and data. For this reason, we deem this trade acceptable.

Mapping of memory and firmware regions

Some of the firmware regions in the physical address space we have identified are in fact RAM. The BIOS area, ACPI tables, and the legacy PCI Option ROM area in the first megabyte are stored in memory and can therefore be accessed with the help of conventional methods like `kmap()`. Others are part of the memory-mapped I/O space which may lead to problems with standard memory-mapping functions due to caching constraints. While it is possible to use the `iomap_nocache()` API on Linux or `MmMapIOSpace()` on Windows, respectively, we prefer to completely bypass the operating system for accessing device memory. If an area of memory has already been mapped by a driver or even the kernel itself, care has to be taken to conform to caching attributes in order to avoid memory corruption. The Windows kernel will actually prevent any attempts to map memory that has already been mapped with different caching attributes, thus making use of standard operating system memory-mapping facilities unreliable (Vidstrom, 2006).

Our method of memory acquisition is based on direct remapping of page table entries with a kernel module (see Stüttgen and Cohen, 2013). When a page is about to be acquired, the module remaps a page in its data segment to point to the physical page in question by directly manipulating the corresponding page table entry. After flushing the *Translation Lookaside Buffer* (TLB), the driver can then read the desired page by reading from the data segment. Because this method uses a separate mapping and is guaranteed to only read from this mapping, we can avoid running into problems with cache coherence and alignment requirements (Stüttgen and Cohen, 2013). In addition, because the mapping is manually created and the memory management APIs are bypassed, the operating system is unable to interfere with these operations as already indicated. The resulting memory image contains the entire memory, firmware code and data, and can be analyzed with standard memory forensic suites such as *Rekall* (Cohen, 2013) or *Volatility* (Walters, 2009).

We have implemented the previously described techniques in the free, open-source memory acquisition utilities *Winpmem* and *Pmem* (Cohen, 2013). With the help of these utilities, firmware code and data on both Windows and Linux can be efficiently acquired.

Firmware analysis

Firmware implementation is highly dependent on the platform. Even executable formats and code compression schemes vary wildly from vendor to vendor. It is out of the scope of this paper to present generic firmware code analysis and verification solutions. However, since most of the locations of firmware code are clearly defined, it is trivial to disassemble these, e.g., using the `dis` plugin of the *Recall* framework (Cohen, 2013). ACPI code on the other hand allows for more automation on the analysis side. We have created two plug-ins for both Volatility and Rekall. One plug-in permits extracting the ACPI tables from a memory image, a second plug-in facilitates scanning the respective tables for potential rootkits. To extract the different tables, we have mirrored the process used by the OSPM unit. Specifically, a signature-based scan for the Root System Description Table (RSDT) is performed (see Section ACPI). Once the RSDT is found, we process the respective pointers to locate associated ACPI structures. All tables are finally saved to the file system for further analysis.

With respect to the latter task, we first decompile and, in a second step, examine the tables for signs of malicious behavior. The central technique for manipulating kernel memory from an ACPI program is the definition of so-called operating regions. Such regions determine which part of

[6] There can be non-PCI devices in the address space, e.g., *HPET*, *RTC*, or *APIC*. In our experiments we found no indication that reading from their mapped registers caused any problems.

[7] It is possible that some systems return another pattern or even data that is still on the bus from a previous read. However, we have not witnessed such behavior during our tests.

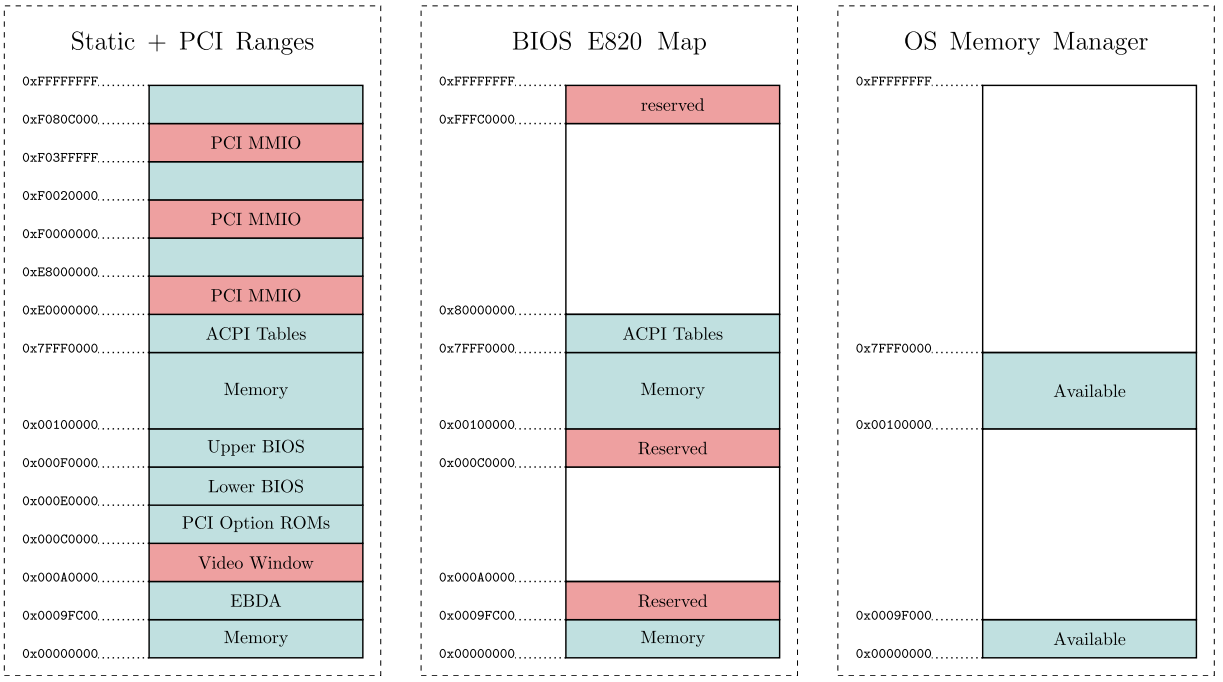

Fig. 2. Physical memory layout views.

the address space will be modified. Our method for detecting malicious behavior is thus to identify all operation regions that reference kernel memory. In the course of investigation, an analyst may then use this information to particularly focus on these parts of the ASL program.

Our plug-in utilizes the official AML decompiler (Intel Corporation, 2014a) to transform the AML code into ACPI Source Language. Subsequently, the resulting ASL code is scanned, and all operation regions referencing *critical memory* are flagged as suspicious, i.e., parts of physical memory that contain kernel code and data.

Evaluation

We have evaluated the created acquisition utilities for stability, correctness, and, in case of the ACPI triage plug-ins, for rate of detection and number of false positives and negatives as well.

Stability and correctness of the acquisition method

To assess the stability and correctness of our approach, we set up several physical as well as virtual machines and created duplicates of their physical address space. The machines comprised the following configuration:

- A *Lenovo* x220 notebook with an Intel Sandy Bridge CPU and 8 GBs of DDR3 RAM running Ubuntu 12.04 x64
- A *Dell* workstation with Intel Ivy Bridge CPU and 8 GBs of DDR3 RAM running Windows 8.1 x64
- A virtual machine based on *VirtualBox* with 4 GBs of RAM running Debian 7 x64 with Kernel 3.2.41
- A virtual machine based on *VirtualBox* with 2 GBs of RAM running Windows 7 SP1 x64

All acquisition operations were successfully completed every time. We could identify the firmware regions in every image with corresponding data. We were not able to verify the firmware though, because we did not have access to EEPROM reprogramming hardware and, thus, did not have access to the original contents of the firmware ROM. Additionally, because most firmware implementations are compressed to save space, proper verification would require reverse engineering of the firmware compression algorithm and analysis of the decompressed ROM image. To establish correct firmware acquisition without access to the ROM nonetheless, we leveraged features of virtualization software. Specifically, *qemu-kvm* (Linux Kernel Organization, 2014) permits loading custom BIOS images over the `-bios` command line option. With the help of the `-option-rom` parameter, it is possible to load a custom Option ROM as well.

We started a qemu-kvm-based virtual machine with a known BIOS version (*SeaBIOS*, O'Connor, 2014) and a known Option ROM (*iPXE*, Brown, 2014). By acquiring memory from inside the virtual machine, we obtained an image with known BIOS and PCI Option ROM code. We were able to find fragments of the iPXE and SeaBIOS images in the created memory images at their expected locations. In addition, we could identify parts of the dumped firmware to come from the supplied ROM images. Other parts were heavily modified though and are likely to have been space-optimized in memory. Further experiments are needed in the future to confirm these assumptions.

Comparison with available memory acquisition solutions

We have evaluated a large group of freely-available memory acquisition solutions to see if they are capable of correctly obtaining firmware code and data. The results of our evaluation are depicted in Table 1. Thereby, an entry labeled with the extension *pci* means that the respective version of the program supports PCI address space enumeration (see Section Enumeration of the Physical Address Space). As can be seen, only those two versions were able to acquire all firmware code and data. All other tools simply imaged the available ranges supplied by the Windows Memory Manager or, on Linux systems, by the `iomem_resource` tree (see Section Memory Acquisition Process), and do not contain any firmware-related content at all.

Detection of ACPI rootkits

We created a simple ACPI rootkit that is capable of modifying the Linux kernel and setting up a hidden backdoor, analogously to the proof of concept application by Heasman (2006) as described in Section ACPI-Based Attacks. The rootkit was installed on five virtual machines running Fedora 19, Ubuntu 12.04, Debian 7, OpenSuse 12.3, and Windows XP as well as two physical Intel Sandy Bridge systems running Ubuntu 12.04. Each system was analyzed with the help of the scanner plug-in we developed for the *Volatility* framework (see Section Firmware Analysis). Further tests were conducted with non-infected ACPI tables of original manufacturers as well as manually-manipulated tables that covered a wide range of malicious accesses to kernel memory. Objective of our experiments was to examine ACPI-related data structures and automatically distinguish potentially infected components from legitimate program parts. In total, 299 operation regions were evaluated. The corresponding results are shown in Table 2.

As can be seen, the scanner flagged 29.4% of all operation regions as malicious. In reality however, only 13% of these regions represented true rootkit activity. The remaining 16.4% were erroneously reported and are due to legitimate memory accesses in the AML virtual machine. In contrast 61.9% of the operation regions were correctly recognized as benign and do not reference any kernel memory. Last but not least, 8.7% of the regions could not be evaluated because their respective arguments were *dynamic*. If the parameters of a region depend on a variable or the result of a function call, it is impossible to determine the target of the operation with static code analysis. Evaluating those would require the state of the runtime environment at the given time they are executed. The missing regions can thus not be classified and have to be manually analyzed.

Table 1
Ability of memory acquisition solutions to acquire firmware code and data.

Acquisition tool	Firmware acquired
Memoryze	✗
FTK Imager	✗
Moonsols DumpIt	✗
WinPmem	✗
WinPmem −3 (pci)	✓
WindowsMemoryReader	✗
LiMe	✗
Pmem	✗
Pmem (pci)	✓

Table 2
Classification of operation regions in the test data set.

	Correctly classified	Falsely classified	∑
Malicious	13.0%	16.4%	29.4%
Benign	61.9%		61.9%
Unknown	8.7%		8.7%
∑	83.6%	16.4%	100%

Our results can be summarized as follows: On the one hand, due to our plug-in, more than three fifths of all memory accesses do not need to be examined in detail and may safely be ignored in the course of an investigation. As such, forensic practitioners benefit from considerable time savings and are able to focus on the relevant sections of an ACPI program. On the other hand, with 16.4%, the number of false positives is still rather high. As we have already indicated, these misclassifications stem from the fact that we were unable to distinguish accesses to regions that belong to legitimate ACPI memory from those that access actual kernel data structures. To decrease the false positive rate, an in-depth analysis of the ACPI environment of the kernel would be necessary. For this task, further research in the future must be conducted.

Discussion

Even though our approach is capable of reliably dumping all firmware code and data and may be easily integrated with existing memory forensic procedures, practitioners have to be aware of several technological limitations and pitfalls. A brief discussion of these will be subject of the following sections.

Technological limitations

Some firmware rootkits cannot be detected with *software-based* memory forensic methods. Any rootkit that completely isolates itself from the CPU-accessible memory falls into this category. *SMM rootkits*, for instance, patch the BIOS to inject code into System Management Mode. This code is run when a *System Management Interrupt* (SMI) is triggered. The System Management Mode comprises its own address space, i.e., *System Management RAM* (SMRAM), and is strictly separated from accesses by kernel or user space applications. The *Memory Controller Hub* (MCH) enforces this restriction. By a similar reasoning, malicious programs running on the *Management Engine* (ME, Stewin and Bystrov, 2012) cannot be discovered.[8] The only way of obtaining a copy of the respective memory regions

[8] The *Management Engine* (ME) is a co-processor on the chipset that is used to perform administrative tasks. For more information on this topic, please refer to Kumar (2009).

would be to perform a RAM transplantation attack (Halderman et al., 2008). For this purpose, physical access to the machine and a system reboot would be required. As a recent study has indicated though, such attacks do not reliably work on modern RAM technologies such as DDR3 anymore (Gruhn and Müller, 2013).

Anti-forensics

It is also possible for firmware to hide or even wipe malicious code and data from RAM before the acquisition process commences. If the only malicious component that is still in memory at runtime resides in SMRAM, it is protected by the MCH and will not appear in the memory image. Any bootstrapping code in the firmware can be wiped from memory after performing its designated task. In this situation, the only way of acquiring the malicious code is by either using a flash programmer to physically read the ROM chip or running a tool like *Copernicus* (Butterworth et al., 2013) if Intel TXT is available.

Conclusion

In this paper we have discussed possibilities for rootkits and other sophisticated malicious applications to compromise x86 systems at the firmware level. Although yet rarely seen in the wild, these types of attacks are highly dangerous and may be particularly devastating because the base of the machine is subverted at a very early point of time and corresponding traces are easily overlooked during typical system investigation routines. As we have seen, common memory forensic solutions distributed on the market to date fail to properly acquire the respective sources of the physical address space and are therefore ill-prepared in the course of an incident. We have therefore suggested alternative techniques for improving current imaging approaches based on PCI introspection and page table entry remapping. Our insights are incorporated in the software products *Winpmem* and *Pmem* and will be made available in the *Rekall* memory forensic framework. We have also created two plug-ins for the *Volatility* and *Rekall* forensic frameworks to facilitate inspection of the ACPI environment and discover traces of malevolent behavior more quickly.

Opportunities for future research

In spite of our plug-ins, automated methods for analyzing the firmware layer for signs of system infections are still sparse. For this reason, a strong focus should be put on identifying and developing additional detection capabilities. Moreover, in order to discover ACPI-related threats more efficiently, creating an emulator for naturally processing the ACPI Machine Language (AML) would prove fruitful.

Acknowledgments

We would like to thank Ben Stock and Prof. Dr.-Ing. Felix Freiling for reading a previous version of this paper and giving us valuable feedback and suggestions for improvement. We also thank our shepherd Dr. Bradley Schatz for his guidance in revising the paper.

References

ACPI Promoters Corporation. Advanced configuration and power interface specification – revision 5.0 errata a. 2013. URL, http://acpi.info/DOWNLOADS/ACPI_5_Errata%20A.pdf.

Balogh Š, Mydlo M. New possibilities for memory acquisition by enabling dma using network card. In: Proceedings of the 7th IEEE international conference on intelligent data acquisition and advanced computing systems (IDAACS); 2013.

Blunden B. The rootkit arsenal: escape and evasion in the dark corners of the system. Wordware Publishing; 2009.

Brossard J. Hardware backdooring is practical. 2012. URL, https://media.blackhat.com/bh-us-12/Briefings/Brossard/BH_US_12_Brossard_Backdoor_Hacking_Slides.pdf.

Brown M. ipxe. 2014. URL, http://ipxe.org/.

Bulygin Y. Evil maid just got angrier – why full-disk encryption with tpm is insecure on many systems. 2013. URL, https://cansecwest.com/slides/2013/Evil%20Maid%20Just%20Got%20Angrier.pdf.

Bulygin Y, Bazhaniuk O, Furtak A, Loucaides J. Summary of attacks against bios and secure boot. 2014. URL, http://www.c7zero.info/stuff/DEFCON22-BIOSAttacks.pdf.

Bulygin Y, Furtak A, Bazhaniuk O. A tale of one software bypass of windows 8 secure boot. 2013. URL, https://media.blackhat.com/us-13/us-13-Bulygin-A-Tale-of-One-Software-Bypass-of-Windows-8-Secure-Boot-Slides.pdf.

Butterworth J, Kallenberg C, Kovah X, Herzog A. Bios chronomancy: fixing the core root of trust for measurement. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security. ACM; 2013. p. 25–36.

Chappell G. The x86 bios emulator. 2010. URL, http://www.geoffchappell.com/studies/windows/km/hal/api/x86bios/call.htm.

Cohen M. Rekall memory forensic framework. 2013. URL, http://www.rekall-forensic.com/docs/Tools/pmem.html.

Davis M, Bodmer S, Lemasters A. Hacking exposed – malware & rootkits. McGraw Hill; 2010.

Dice P. Quick boot: a guide for embedded firmware developers. Intel Press; 2013.

Embleton S, Sparks S, Zou C. Smm rootkits: a new breed of os independent malware. In: Proceedings of the 4th international conference on security and privacy in communication networks; 2008.

Giuliani M. *Mebromi*: the first bios rootkit in the wild. 2013. URL, http://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/.

Gruhn M, Müller T. On the practicability of cold boot attacks. In: Proceedings of the 8th international conference on availability, reliability and security (ARES). IEEE; 2013. p. 390–7.

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold-boot attacks on encryption keys. In: Proceedings of the 17th USENIX security symposium; 2008.

Heasman J. Implementing and detecting an acpi bios rootkit. 2006. URL, http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf.

Hoglund G, Butler J. Rootkits: subverting the windows kernel. Addison Wesley; 2005.

Intel Corporation. Acpi component architecture. 2014. URL, https://acpica.org/.

Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. 2014. URL, http://www.intel.de/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

Intel Corporation. Trusted compute pools with intel® trusted execution technology (intel® txt). 2014. URL, http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/malware-reduction-general-technology.html.

Kovah X, Butterworth J, Kallenberg C, Cornwell S. Copernicus 2: senter the dragon. 2014. URL, http://www.mitre.org/publications/technical-papers/copernicus-2-senter-the-dragon.

Kumar A. Active platform management demystified: unleashing the power of intel VPro (TM) technology. Intel Press; 2009.

Ligh MH, Adair S, Hartstein B, Richard M. Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Wiley Publishing; 2010.

Linux Kernel Organization. Kernel virtual machine. 2014. URL, git://git.kernel.org/pub/scm/virt/kvm/kvm.git.

Loukas K. De mysteriis dom jobsivs—mac efi rootkits. 2012. URL, http://hoax/De_Mysteriis_Dom_Jobsivs_Black_Hat_Paper.pdf.
Maartmann-Moe C. Adventures with daisy in thunderbolt-dma-land: hacking macs through the thunderbolt interface. 2012. URL, http://www.breaknenter.org/2012/02/adventures-with-daisy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface/.
Martignoni L, Fattori A, Paleari R, Cavallaro L. Live and trustworthy forensic analysis of commodity production systems. In: Proceedings of the 13th international conference on recent advances in intrusion detection (RAID); 2010.
Microsoft Corporation. Windows feature lets you generate a memory dump file by using the keyboard. 2011. URL, http://support.microsoft.com/?scid=kb%3Ben-us%3B244139&x=5&y=9.
Microsoft Corporation. Device\physicalmemory object. 2013. URL, http://technet.microsoft.com/en-us/library/cc787565%28v=ws.10%29.aspx.
Minnich R. Coreboot. 2014. URL, http://www.coreboot.org/.
O'Connor K. Seabios. 2014. URL, http://www.seabios.org/SeaBIOS.
PCI-SIG. Pci firmware 3.1 specification. 2010. URL, https://www.pcisig.com/specifications/conventional/pci_firmware/.
Reina A, Fattori A, Pagani F, Cavallaro L, Bruschi D. When hardware meets software: a bulletproof solution to forensic memory acquisition. In: Proceedings of the 28th annual computer security applications conference; 2012.
Salihun D. BIOS disassembly Ninjutsu uncovered. A-List Publishing; 2006.
Salihun D. Malicious code execution in pci expansion rom. 2012. URL, http://resources.infosecinstitute.com/pci-expansion-rom/.
Salihun D. System address map initialization in x86/x64 architecture part 1: pci-based systems. 2013. URL, https://sites.google.com/site/pinczakko/bios-articles/System%20Address%20Map%20Initialization%20in%20x86_x64%20-%20Part%201.pdf?attredirects=0&d=1.
Shields T. Survey of rootkit technologies and their impact on digital forensic. 2008. URL, http://www.donkeyonawaffle.org/misc/txs-rootkits_and_digital_forensics.pdf.
Stewin P, Bystrov I. Understanding dma malware. In: Proceedings of the 9th international conference on detection of intrusions and malware, and vulnerability assessment (DIMVA); 2012.
Stüttgen J, Cohen M. Anti-forensic resilient memory acquisition. In: Proceedings of the 13th annual DFRWS conference; 2013.
Sylve J. Lime – linux memory extractor. In: Proceedings of the 7th ShmooCon conference; 2012.
Tereshkin A, Wojtczuk R. Introducing ring -3 rootkits. 2009. URL, http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf.
Triulzi A. The jedi packet trick takes over the deathstar. 2010. URL, http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf.
UEFI Forum. Unified extensible firmware interface specification. 2014. URL, http://www.uefi.org/sites/default/files/resources/2_4_Errata_B.pdf.
Vidstrom A. Forensic memory dumping intricacies – physicalmemory, dd, and caching issues. 2006. URL, http://ntsecurity.nu/onmymind/2006/2006-06-01.html.
Vieler R. Professional rootkits. Wrox Press; 2007.
Vömel S, Freiling FC. A survey of main memory acquisition and analysis techniques for the windows operating system. Digit Investig 2011;8(1):3–22.
Vömel S, Freiling FC. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. Digit Investig 2012;9(2):125–37.
Vömel S, Stüttgen J. An evaluation platform for forensic memory acquisition software. In: Proceedings of the 13th annual DFRWS conference; 2013.
Walters A. Volatility framework. 2009. URL, https://github.com/volatilityfoundation/volatility.
Walters A, Petroni NL. Volatools: integrating volatile memory forensics into the digital investigation process. In: Proceedings of black hat DC; 2007.
Wang J, Zhang F, Sun K, Stavrou A. Firmware-assisted memory acquisition and analysis tools for digital forensics. In: Proceedings of the sixth international workshop on systematic approaches to digital forensic engineering (SADFE); 2011.
White A, Schatz B, Foo E. Integrity verification of user space code. In: Proceedings of the 13th annual DFRWS conference; 2013.
Wojtczuk R, Tereshkin A. Attacking intel® bios. 2009. URL, https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf.
Yu M, Lin Q, Li B, Qi Z, Guan H. Vis: virtualization enhanced live forensics acquisition for native system. Digit Investig 2012;9(1):22–33.
Zhang L, Wang L, Zhang R, Zhang S, Zhou Y. Live memory acquisition through firewire. Forensics Telecommun Inf Multimedia 2011;56:159–67.