Hypervisor Memory Forensics

Mariano Graziano, Andrea Lanzi, and Davide Balzarotti

Eurecom, France
graziano,lanzi,balzarotti@eurecom.fr

Abstract. Memory forensics is the branch of computer forensics that aims at extracting artifacts from memory snapshots taken from a running system. Even though it is a relatively recent field, it is rapidly growing and it is attracting considerable attention from both industrial and academic researchers.
In this paper, we present a set of techniques to extend the field of memory forensics toward the analysis of hypervisors and virtual machines. With the increasing adoption of virtualization techniques (both as part of the cloud and in normal desktop environments), we believe that memory forensics will soon play a very important role in many investigations that involve virtual environments.
Our approach, implemented in an open source tool as an extension of the Volatility framework, is designed to detect both the existence and the characteristics of any hypervisor that uses the Intel VT-x technology. It also supports the analysis of nested virtualization and it is able to infer the hierarchy of multiple hypervisors and virtual machines. Finally, by exploiting the techniques presented in this paper, our tool can reconstruct the address space of a virtual machine in order to transparently support any existing Volatility plugin - allowing analysts to reuse their code for the analysis of virtual environments.

Keywords: Forensics, Memory Analysis, Intel Virtualization

1 Introduction

The recent increase in the popularity of physical memory forensics is certainly one of the most relevant advancements in the digital investigation and computer forensics field in the last decade. In the past, forensic analysts focused mostly on the analysis of non-volatile information, such as the one contained in hard disks and other data storage devices. However, by acquiring an image of the volatile memory it is possible to gain a more complete picture of the system, including running (and hidden) processes and kernel drivers, open network connections, and signs of memory resident malware. Memory dumps can also contain other critical information about the user activity, including passwords and encryption keys that can then be used to circumvent disk-based protection. For example, Elcomsoft Forensic Disk Decryptor [3] is able to break encrypted disks protected with BitLocker, PGP and TrueCrypt, by extracting the required keys from memory.

Unfortunately, the increasing use of virtualization poses an obstacle to the adoption of the current memory forensic techniques. The problem is twofold. First, in presence of an hypervisor it is harder to take a complete dump of the physical memory. In fact, most of the existing tools are software-based solutions that rely on the operating system to acquire the memory. Unfortunately, such techniques can only observe what the OS can see, and, therefore, might be unable to access the memory reserved by the virtual machine monitor itself [31]. Second, even when a complete physical image is acquired by using an hardware-based solution (e.g., through a DMA-enable device [2]), existing tools are not able to properly analyze the memory image. While solutions exist for the first problem, such as a recently proposed technique based on the SMM [25], the second one is still unsolved.

Virtualization is one of the main pillars of cloud computing but its adoption is also rapidly increasing outside the cloud. Many users use virtual machines as a simple way to make two different operating systems coexist on the same machine (e.g., to run Windows inside a Linux environment), or to isolate critical processes from the rest of the system (e.g., to run a web browser reserved for home banking and financial transactions). These scenarios pose serious problem for forensic investigations. Moreover, any incident in which the attacker try to escape from a VM or to compromise the hypervisor in a cloud infrastructure remain outside the scope of current memory forensic techniques.

In this paper we propose a new solution to detect the presence and the characteristics of an hypervisor and to allow existing memory forensic techniques to analyze the address space of each virtual machine running inside the system. Nowadays, if an investigator takes a complete physical snapshot of Alice computer's memory while she is browsing the Internet from inside a VMware machine, none of the state of the art memory analysis tools can completely analyze the dump. In this scenario, Volatility [6], a very popular open source memory forensic framework, would be able to properly analyze the host operating system and observe that the VMware process was running on the machine. However, even though the memory of the virtual machine is available in the dump, Volatility is currently not able to analyze it. In fact, only by properly analyzing the hypervisor it is possible to gain the information required to translate the guest virtual addresses into physical addresses, the first step required by most of the subsequent analysis. Even worse, if Alice computer was infected by some advanced hypervisor-based rootkit, Volatility would not even be able to spot its presence.

In some way, the problem of finding an hypervisor is similar to the one of being able to automatically reconstruct information about an operating system in memory, even though that operating system may be completely unknown. The number of commodity hypervisors is limited and, given enough time, it would be possible to analyze all of them and reverse engineer their most relevant data structures, following the same approach used to perform memory forensics of known operating systems. However, custom hypervisors are easy to develop and they are already adopted by many security-related tools [15,22,28,29]. Moreover,

malicious hypervisors (so far only proposed as research prototypes [12,19,26,33]) could soon become a reality - thus increasing the urgency of developing the area of virtualization memory forensics.

The main idea behind our approach is that, even though the code and internals of the hypervisors may be unknown, there is still one important piece of information that we can use to pinpoint the presence of an hypervisor. In fact, in order to exploit the virtualization support provided by most of the modern hardware architectures, the processor requires the use of particular data structures to store the information about the execution of each virtual environment. By first finding these data structures and then analyzing their content, we can reconstruct a precise representation of what was running in the system under test.

Starting from this observation, this paper has three main goals. First, we want to extend traditional memory forensic techniques to list the hypervisors present in a physical memory image. As it is the case for traditional operating systems, we also want to extract as much information as possible regarding those hypervisors, such as their type, location, and the conditions that trigger their behaviors. Second, we want to use the extracted information to reconstruct the address space of each virtual machine. The objective is to be able to transparently support existing memory analysis techniques. For example, if a Windows user is running a second Windows OS inside a virtual machine, thanks to our techniques a memory forensic tool to list the running processes should be able to apply its analysis to either one or the other operating system. Finally, we want to be able to detect cases of nested virtualization, and to properly reconstruct the hierarchy of the hypervisors running in the system.

To summarize, in this paper we make the following contributions:

- We are the first to design a forensics framework to analyze hypervisor structures in physical memory dumps.
- We implemented our framework in a tool named Actaeon, consisting of a Volatility plugin, a patch to the Volatility core, and a standalone tool to dump the layout of the Virtual Machine Control Structure (VMCS) in different environments.
- We evaluate our framework on several open source and commercial hypervisors installed in different nested configurations. The results show that our system is able to properly recognize the hypervisors in all the configuration we tested.

2 Background

Before presenting our approach for hypervisor memory forensics we need to introduce the Intel virtualization technology and present some background information on the main concepts we will use in the rest of the paper.

2.1 Intel VT-x Technology

In 2005, Intel introduced the VT-x Virtualization Technology [18], a set of processor-level features to support virtualization on the x86 architecture. The main goal of VT-x was to reduce the virtualization overhead by moving the implementation of different tasks from software to hardware.

VT-x introduces a new instruction set, called Virtual Machine eXtension (VMX) and it distinguishes two modes of operation: VMX *root* and VMX *non root*. The VMX root operation is intended to run the hypervisor and it is therefore located below "ring 0". The non root operation is instead used to run the guest operating systems and it is therefore limited in the way it can access hardware resources. Transitions between non root and root modes are called `VMEXIT`, while the transition in the opposite direction are called `VMENTRY`. As part of the VT-x technology, Intel introduced a set of new instructions that are available when the processor is operating in VMX root operation, and modified some of the existing instructions to trap (e.g., to cause a `VMEXIT`) when executed inside a guest OS.

2.2 VMCS Layout

VMX transitions are controlled by a data structure called Virtual Machine Control Structure (VMCS). This structure manages the transitions from and to VMX non root operation as well as the processor behavior in VMX non root operation. Each logical processor reserves a special region in memory to contain the VMCS, known as the VMCS region. The hypervisor can directly reference the VMCS through a 64 bit, 4k-aligned physical address stored inside the *VMCS pointer*. This pointer can be accessed using two special instructions (`VMPTRST` and `VMPTRLD`) and the VMCS fields can be configured by the hypervisor through the `VMREAD`, `VMWRITE` and `VMCLEAR` commands.

Theoretically, an hypervisor can maintain multiple VMCSs for each virtual machine, but in practice the number of VMCSs normally matches the number of virtual processors used by the guest VM. The first word of the VMCS region contains a revision identifier that is used to specify which format is used in the rest of the data structure. The second word is the `VMX_ABORT_INDICATOR`, and it is always set to zero unless a VMX abort is generated during a `VMEXIT` operation and the logical processor is switched to shutdown state. The rest of the structure contains the actual VMCS data. Unfortunately, the memory layout (order and offset) of the VMCS fields is not documented and different processors store the information in a different way.

Every field in the VMCS is associated with a 32 bit value, called its *encoding*, that needs to be provided to the `VMREAD/VMWRITE` instructions to specify how the values has to be stored. For this reason, the hypervisor has to use these two instructions and should never access or modify the VMCS data using ordinary memory operations.

The VMCS data is organized into six logical groups: 1) a *guest state area* to store the guest processor state when the hypervisor is executing; 2) a *host state*

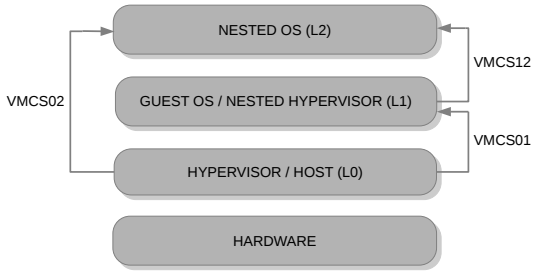

Fig. 1. VMCS structures in a Turtle-based nested virtualization setup

area to store the processor state of the hypervisor when the guest is executing; 3) a *VM Execution Control Fields* containing information to control the processor behavior in VMX non root operation; 4) *VM Exit Control Fields* that control the VMEXITs; 5) a *VM Entry Control Fields* to control the VMENTRIES; and 6) a *VM Exit Info Fields* that describe the cause and the nature of a VMEXIT.

Each group contains many different fields, but the offset and the alignment of each field is not documented and it is not constant between different Intel processor families[1].

2.3 Nested Virtualization

Nested virtualization has been first defined by Popek and Goldberg [16, 24] in 1973. Since then, several implementation has been proposed. In a nested virtualization setting, a guest virtual machine can run another hypervisor that in turn can run other virtual machines, thus achieving some form of recursive virtualization. However, since the x86 architecture provides only a single-level architectural support for virtualization, there can only be one and only one hypervisor mode and all the traps, at any given nested level, need to be handled by this hypervisor (the "top" one in the hierarchy). The main consequence is that only a single hypervisor is running at ring -1 and has access to the VMX instructions. For all the other nested hypervisors the VMX instructions have to be emulated by the top hypervisor to provide to the nested hypervisors the illusion of running in root mode.

Because of these limitations, the support for nested virtualization needs to be implemented in the top hypervisor. KVM has been the first x86 virtual machine monitor to fully support nested virtualization using the Turtle technology [9]. For this reason, in the rest of this paper we will use the KVM/Turtle nomenclature when we refer to nested hypervisors. Recent versions of Xen also adopted the same concepts and it is reasonable to think that also proprietary hypervisors (such as VMware and Hyper-V) use similar implementations.

[1] For more information on each VMCS section please refer to the Intel Manual Vol 3B Chapter 20

The Turtle architecture is depicted in Figure 1. In the example, the top hypervisor (L0) runs a guest operating system inside which a second hypervisor (L1) is installed. Finally, this second hypervisor runs a nested guest operating system (L2). In this case the CPU uses a first VMCS (VMCS01) to control the top hypervisor and its guest. The nested hypervisor has a "fake" VMCS (VMCS12) to mange the interaction with its nested OS (L2). Since this VMCS is not real but it is emulated by the top hypervisor, its layout is not decided by the processor, but can be freely chosen by the hypervisor developers. The two VMCSs are obviously related to each other. For example, in our experiments, we observed that for KVM the VMCS12 Host State Area corresponds to the VMCS01 Guest State Area.

The Turtle approach also adds one more VMCS (VMCS02), that is used by the top hypervisor (L0) to manage the nested OS (L2). In theory, nested virtualization could be implemented without using this additional memory structure. However, all the hypervisors we analyzed in our tests adopted this approach.

Another important aspect that complicates the nested virtualization setup is the memory virtualization. Without nested virtualization, the guest operating system has its own page tables to translate the Guest Virtual Addresses (GVAs) to the Guest Physical Addresses (GPAs). The GPA are then translated by the hypervisor to Host Physical Addresses (HPAs) that are pointing to the actual physical pages containing the data. This additional translation can be done either in software (e.g., using shadow page tables [30]) or in hardware (e.g., using the Extended Page Tables (EPT) described later in this section). The introduction of the nested virtualization adds one more layer of translation. In fact, the two dimensional support is no longer enough to handle the translation for nested operating systems. For this reason, Turtle introduced a new technique called multidimensional-paging in which the nested translations (from L2 to L1 in Figure 1) are multiplexed into the two available layers.

2.4 Extended Page Table

Since the introduction of the *Nehalem* microarchitecture [5], Intel processors adopted an hardware feature, called Extended Page Tables (EPT), to support address translation between GPAs and HPAs. Since the use of this technology greatly alleviated the overhead introduced by memory translation, it quickly replaced the old and slow approach based on shadow pages tables.

When the EPT is enabled, it is marked with a dedicated flag in the *Secondary Based Execution Control Field* in the VMCS structure. This tells the CPU that the EPT mechanism is active and it has to be used to translate the guest physical addresses.

The translation happens through different stages involving four EPT paging structures (namely PML4, PDPT, PD, and PT). These structures are very similar to the ones used for the normal IA-32e address mode translation. If the paging is enabled in the guest operating system the translation starts from the guest paging structures. The PML4 table can be reached by following the corresponding pointer in the VMCS. Then, the GPA is split and used as offset to

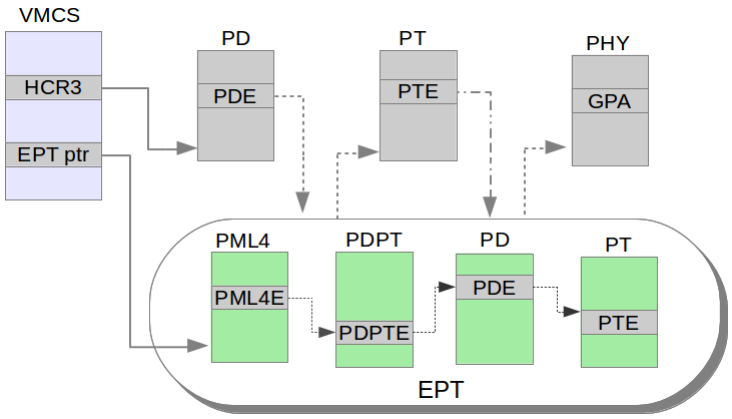

Fig. 2. EPT-based Address Translation

choose the proper entry at each stage of the walk. The EPT translation process is summarized in Figure 2.4. [2]

3 Objectives and Motivations

Our goal is to bring the memory forensic area to the virtualization world. This requires the introduction of new techniques to detect, recognize, and analyze the footprint of hypervisors inside the physical memory. It also requires to support previous techniques, so that existing tools to investigate operating systems and user-space programs could be easily applied to each virtual machine inside a memory image.

Locate Hypervisors in Memory

If an hypervisor is known, locating it in memory could be as simple as looking for a certain pattern of bytes (e.g., by using a code-based signature). Unfortunately, this approach have some practical limitations. In fact, given a snapshot of the physical memory collected during an investigation, one of the main question we want to ask is "Is there *any* hypervisor running on the system?". Even though a signature database could be a fast way to detect well-known products, custom hypervisors are nowadays developed and used in many environments. Moreover, thin hypervisor could also be used for malicious purposes, such as the one described by Rutkowska [26], that is able to install itself in the system and intercept critical operations. Detecting this kind of advanced threats is also going to become a priority for computer forensics in the near future.

For these reasons, we decided to design a **generic** hypervisor detector. In order to be generic, it needs to rely on some specific features that are required

[2] For more detail about EPT look at Vol 3B, Chapter 25 Intel Manuals.

by all hypervisors to run. As explained in the previous section, to provide hardware virtualization support, the processor requires certain data structures to be maintained by the hypervisor. For Intel, this structure is called VMCS, while the equivalent for AMD is called VMCB. If we can detect and analyze those structures we could use them as entry points to find all the other components: hypervisors, hosts, and guest virtual machines.

To show the feasibility of our approach, we decided to focus our effort on the Intel architecture. There are two reasons behind this choice. First, Intel largely dominates the market share (83% vs 16% in the second quarter of 2012 [1]). Second, the AMD virtualization structures are fixed and well documented, while Intel adopts a proprietary API to hide the implementation details. Even worse, those details vary between different processor families. Therefore, it provided a much harder scenario to test our techniques.

A limitation of our choice is that our approach can only be applied to hardware assisted hypervisors. Old solutions based on para-virtualization are not supported, since in this case the virtualization is completely implemented in software. However, these solution are becoming less and less popular because of their limitations in terms of performance.

Analysis of Nested Virtualization

Finding the top hypervisor, i.e. the one with full control over the machine, is certainly the main objective of a forensic analysis. But since now most of the commodity hypervisors support nested virtualization, extracting also the hierarchy of nested hypervisors and virtual machines could help an analyst to gain a better understanding of what is running inside the system.

Unfortunately, developing a completely generic and automated algorithm to forensically analyze nested virtualization environments is - in the general case - impossible. In fact, while the top hypervisor has to follow specific architectural constraints, the way it supports nested hypervisors is completely implementation specific. In a nested setup, the top hypervisor has to emulate the VMX instructions, but there are no constraints regarding the location and the format in which it has to store the fields of the nested VMCS. In the best-case scenario, the fields are recorded in a custom VMCS-like structure, that we can reverse engineer in an automated way by using the same technique we use to analyze the layouts of the different Intel processor families. In the worse case, the fields could be stored in complex data structures (such as hash tables) or saved in an encoded form, thus greatly complicating the task of locating them in the memory dump.

Not every hypervisor support nested virtualization (e.g. VirtualBox does not). KVM and Xen implement it using the Turtle [9] approach, and a similar technique to multiplex the inner hypervisors VT-x/EPT into the underlying physical CPU is also used by VMware [7].

By looking for the nested VMCS structure (if known) or by recognizing the VMCS02 of a Turtle-like environment (as presented in Figure 1 and explained

in details in Section 4), we can provide an extensible support to reconstruct the hierarchy of nested virtualization.

Virtual Machine Forensic Introspection

Once a forensic analyst is able to list the hypervisors and virtual machines in a memory dump, the next step is to allow her to run all her memory forensic tools on each virtual machine. For example, the Volatility memory forensic framework ships with over 60 commands implementing different kinds of analysis - and many more are available through third-party plugins. Unfortunately, in presence of virtualization, all these commands can only be applied to the host virtual machine. In fact, the address spaces of the other VMs require to be extracted and translated from guest to host physical addresses.

The goal of our introspection analysis is to parse the hypervisor information, locate the tables used by the EPT, and use them to provide a transparent mechanism to translate the address space of each VM.

4 System Design

Our hypervisor analysis technique consists of three different phases: memory scanning, data structure validation, and hierarchy analysis. The Memory Scanner takes as input a memory dump and the database of the known VMCS layouts (i.e., the offset of each field in the VMCS memory area) and outputs a number of candidate VMCS. Since the checks performed by the scanner can produce false positives, in the second phase each structure is validated by analyzing the corresponding page table. The final phase of our approach is the hierarchy analysis, in which the validated VMCSs are analyzed to find the relationships among the different hypervisors running on the machine.

In the following sections we will describe in details the algorithms that we designed to perform each phase of our analysis.

4.1 Memory Scanner

The goal of the memory scanner is to scan a physical memory image looking for data structures that can represent a VMCS. In order to do that, we need two types of information: the memory layout of the structure, and a set of constraints on the values of its fields that we can use to identify possible candidates. The VMCS contains over 140 different fields, most of which can assume arbitrary values or they can be easily obfuscated by a malicious hypervisors. The memory scanner can tolerate false positives (that are later removed by the validation routine) but we want to avoid any false negative that could result in a missed hypervisor. Therefore we designed our scanner to focus only on few selected fields:

- `Revision ID`: It is the identifier that determines the layout of the rest of the structure. For the VMCS of the top hypervisor, this field has to match the value of the `IA32_VMX_BASIC MSR` register of the machine on which the image was acquired (and that changes between different micro-architecture). In case of nested virtualization, the revision ID of the VMCS12 is chosen by the top hypervisor. The `Revision ID` is always the first word of the VMCS data structure.
- `VMX ABORT INDICATOR`: This is the VMX abort indicator and its value has to be zero. The field is the second entry of the VMCS area.
- `VmcsLinkPointerCheck`: The values of this field consists of two consecutive words that, according to the Intel manual, should always be set to `0xffffffff`. The position of this field is not fixed.
- `Host_CR4`: This field contains the host CR4 register. Its 13th bit indicates if the VMX is enabled or not. The position of this field is not fixed.

To be sure that our choice is robust against evasions, we implemented a simple hypervisor in which we tried to obfuscate those fields during the guest operation and re-store them only when the hypervisor is running, a similar approach is described in [14]. This would simulate what a malicious hypervisor could do in order to hide the VMCS and avoid being detected by our forensic technique. In our experiments, any change on the values of the previous five fields produced a system crash, with the only exception of the Revision ID itself. For this reason, we keep the revision ID only as a key in the VMCS database, but we do not check its value in the scanning phase.

The memory scanner first extracts the known VMCS layouts from the database and then it scans the memory looking for pages containing the aforementioned values at the offsets defined by the layout. Whenever a match is found, the candidate VMCS is passed over to the validation step.

4.2 VMCS Validation

Our validation algorithm is based on a simple observation. Since the `HOST_CR3` field needs to point to the page table that is used by the processor to translate the hypervisor addresses, that table should also contain the mapping from virtual to physical address for the page containing the VMCS itself. We call this mechanism self-referential validation.

For every candidate VMCS, we first extract the `HOST_CR3` field and we assume that it points to a valid page table structure. Unfortunately, a page table can be traversed only by starting from a virtual address to find the corresponding physical one, but not vice-versa. In our case, since we only know the physical address of the candidate VMCS, we need to perform the opposite operation. For this reason, our validator walks the entire page tables (i.e., it tries to follow every entry listed in them) and creates a tree representation where the leaves represent the mapped physical memory pages and the different levels of the tree represent the intermediate points of the translation algorithm (i.e., the page directory, and the page tables).

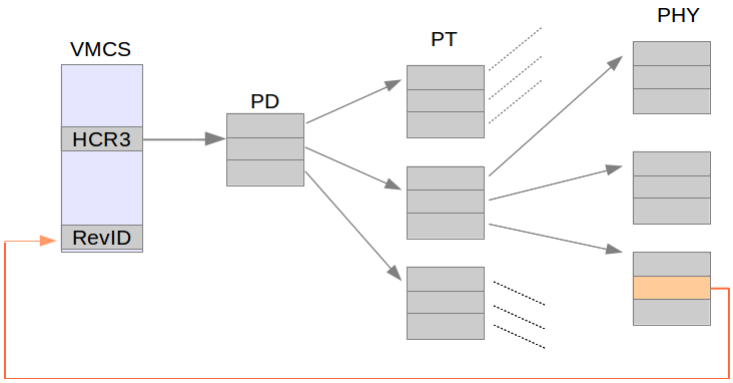

Fig. 3. Self-referential Validation Technique

This structure has a double purpose. First, it serves as a way to validate a candidate VMCS, by checking that one of the leaves points to the VMCS itself (see Figure 3). If this check fails, the VMCS is discarded as a false positive. Second, if the validation succeeded, the tree can be used to map all the memory pages that were reserved by the hypervisor. This could be useful in case of malicious hypervisors that need an in-depth analysis after being discovered.

It is important to note that the accuracy of our validation technique leverages on the assumption that is extremely unlikely that such circular relationship can appear by chance in a memory image.

4.3 Reverse Engineering The VMCS Layout

The previous analysis steps are based on the assumption that our database contains the required VMCS layout information. However, as we already mentioned in the previous sections, the Intel architecture does not specify a fix layout, but provides instead an API to read and write each value, independently from its position.

In our study we noticed that each processor micro-architecture defines different offsets for the VMCS fields. Since we need these offsets to perform our analysis, we design and implement a small hypervisor-based tool to extract them from a live system.

More in detail, our algorithm considers the processors microcode as a black box and it works as follows. In the first step, we allocate a VMCS memory region and we fill the corresponding page with a 16 bit-long incremental counter. At this point the VMCS region contains a sequence of progressive numbers ranging from 0 to 2048, each representing its own offset into the VMCS area. Then, we perform a sequence of `VMREAD` operations, one for each field in the VMCS. As a result, the processor retrieves the field from the right offset inside the VMCS page and returns its value (in our case the counter that specifies the field location).

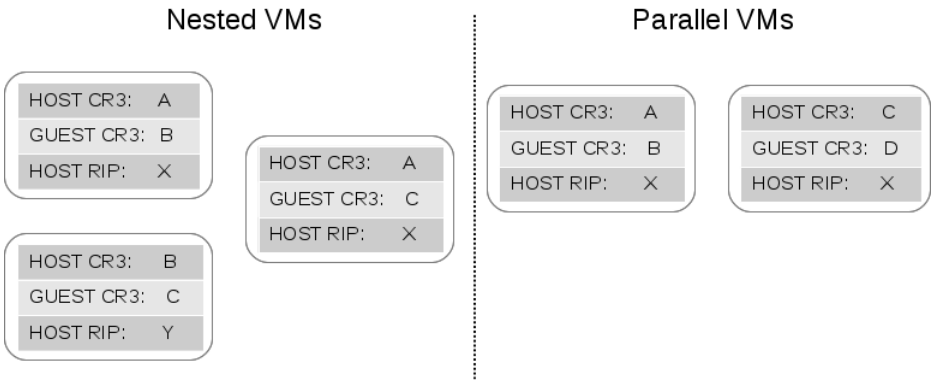

Fig. 4. Comparison between different VMCS fields in nested and parallel configurations

The same technique can also be used to dump the layout of nested VMCSs. However, since in this case our tool would run as a nested hypervisor, the top hypervisor could implement a protection mechanism to prevent write access to the VMCS region (as done by VMware), thus preventing our technique to work. In this case we adopt the opposite, but much slower, approach of writing each field with a `VMWRITE` and then scan the memory for the written value.

4.4 Virtualization Hierarchy Analysis

If our previous techniques detect and validate more then one VMCS, we need to distinguish between several possibilities, depending whether the VMCS represent parallel guests (i.e., a single hypervisor running multiple virtual machines), nested guests (i.e, an hypervisor running a machine the runs another hypervisor), or a combination of the previous ones.

Moreover, if we assume one virtual CPU per virtual machine, we can have three different nested virtualization scenarios: Turtle approach and known nested VMCS layout (three VMCSs found), Turtle approach and unknown nested layout (two VMCSs found), and non-Turtle approach and known layout (two or more VMCSs found).

In the first two cases (the only ones we could test in our experiments since all the hypervisors in our tests adopted the Turtle approach), we can infer the hierarchy between the hypervisors and distinguish between parallel and nested VMs by comparing the values of three fields: the `GUEST CR3`, the `HOST CR3`, and the `HOST RIP`. The first two fields represent the CR3 for the guest and for the hypervisor. The third is the pointer to the hypervisor entry point, i.e., to the first instruction to execute when the CPU transfer control to the hypervisor.

Figure 4 show a comparison of the values of these three fields in a parallel and nested configurations. As the diagram shows, in a nested setup we have two different hypervisors (represented by the two different `HOST RIP` addresses) while for parallel virtual machine the hypervisor is the same (same value of `HOST RIP`).

Moreover, by comparing the `GUEST CR3` and `HOST CR3` values we can distinguish among VMCS01, VMCS02, and VMCS12 in a nested virtualization setup. More precisely, the VMCS01 and VMCS02 share the same `HOST CR3`, while the `HOST CR3` of the VMCS12 has to match the `GUEST CR3` of the VMCS01.

Finally, in the third scenario in which the nested virtualization is not implemented following the Turtle approach (possible in theory but something we never observed in our experiments), the previous heuristics may not work. However, also in this case we can still tell that a VMCS belongs to a nested hypervisor if its layout matches the one of a known nested VMCS (e.g., the one emulated by KVM).

4.5 Virtual Machine Introspection

The last component of our system is the algorithm to extract the EPT tables and to provide support for the memory analysis of virtual machines. In this case the algorithm is straightforward. First, we extract the pointer to the EPT from the VMCS of the machine we want to analyze (see Figure 2.4 in Section 2). Then, we simulate the EPT translation by programmatically walking through the PML4, PDPT, PD, and PT tables for each address that need to be translated.

4.6 System Implementation

We implemented the previously described techniques in an open source tool called Actaeon. Actaeon consists of three components: a standalone VMCS layout Extractor derived from HyperDbg [15], an hypervisor Memory Analysis plugin for the Volatility framework, and a patch for the Volatility core to provide a transparent mechanism to analyze the virtual machines address spaces. The tool, along with a number of datasets and usage examples, can be downloaded from `http://s3.eurecom.fr/tools/actaeon`.

VMCS Layout Extractor This component is designed to extract and save into a database the exact layout of a VMCS, by implementing the reverse engineering algorithm described above. The tool is implemented as a small custom hypervisor that re-uses the initialization code of HyperDbg, to which it adds around 200 lines of C code to implement the custom checks to identify the layout of the VMCS.

Hyper-ls This component is implemented as a Python plugin for the Volatility framework, and it consists of around 1,300 lines of code. Its goal is to scan the memory image to extract the candidate VMCSs, run our validation algorithm to filter out the false positives, and analyze the remaining structures to extract the details about the corresponding hypervisors.

The tool is currently able to parse all the fields of the VMCS and to properly interpret them and print them in a readable form. For example, our plugin can show which physical devices and which events are trapped by the hypervisor,

the pointer to the hypervisor code, the Host and Guest CR3, and all the saved CPU registers for the host and guest systems.

The `hyperls` plugin can also print a summary of the hierarchy between the different hypervisors and virtual machines. For each VM, it also reports the pointer to the corresponding EPT, required to further inspect their content.

Virtual Machine Introspection Patch An important functionality performed by Acteon is to provide a transparent mechanism for the Volatility framework to analyze each Virtual Machine address space. In order to provide such functionality, Acteon provides a patch for the Volatility core to add one command-line parameter (that the user can use to specify in which virtual machine he wants to run the analysis) and to modify the APIs used for address translations by inserting an additional layer based on the EPT tables. The patch is currently implemented in 250 lines of Python code.

5 Evaluation

The goal of our experiments is to evaluate the accuracy and reliability of our techniques in locating hypervisors inside physical memory dumps, access their private data, reconstruct the hierarchy in case of nested virtualization, and provide the support for other memory forensic techniques to inspect the guest operating systems. All the experiments have been performed on an Intel Core 2 Duo P8600 and an Intel Core i5-2500 machines running the Ubuntu Linux 12.10 32bit operating system and with one virtual processor per guest.

5.1 Forensic Memory Acquisition

The first step of our experiments consisted in the acquisition of complete snapshots of the physical memory on a computer running a number of different hypervisor configurations.

As we already mentioned in Section 1, this turned out to be a challenging task. In fact, even though a large number of memory imaging solution exists on the market, the vast majority adopt software-based techniques that uses kernel modules to acquire the memory from the operating system point of view. These approaches have not been designed to work in a virtualization environment where the OS does not have a complete view of the system memory. In fact, if the virtual machine monitor is protecting its own pages, the memory image collected from the host operating system does not contain the pages of the hypervisor. To overcome this limitation, whenever a software approach was not able to properly capture the memory, we resorted to a hardware-based solution. In particular, we used a PCI Firewire card with a Texas Instrument Chipset, and the Inception [4] tool to dump the memory through a DMA attack [23]. In this case, we had to disable the Intel VT-d support from the BIOS, to prevent the IOMMU from blocking the DMA attack.

The main drawback of using the Firewire acquisition is that in our experiments it was quite unstable, often requiring several consecutive attempts before we could obtain a correct dump. Moreover, it is worth noting that in theory even a DMA-based approach is not completely reliable. In 2007 Joanna Rutkowska showed the feasibility of attacks against hardware-based RAM acquistion [27]. The presented attacks are based on the modification of the processor's North-Bridge memory map to denial of service the acquisition tool or to hide some portions of the physical memory. However, we are not aware of any hypervisor that uses these techniques to tamper with the memory acquisition process.

Today, the best solution to acquire a complete system memory in presence of an hypervisor would be to use an acquisition tool implemented in the SMM (therefore running at higher privileges than the hypervisor itself), as proposed by A. Reina et al. [25]. Unfortunately, we were not able to find any tool of this kind available on the Internet.

5.2 System Validation

The first step of our experiments was to perform a number of checks to ensure that our memory acquisition process was correct and that our memory forensic techniques were properly implemented.

In the first test, we wrote a simple program that stored a set of variables with known values and we run it in the system under test. We also added a small kernel driver to translate the program host virtual addresses to host physical addresses and we used these physical addresses as offset in the memory image to read the variable and verify their values.

The second test was designed to assess the correctness of the VMCS layout. In this case we instrumented three open source hypervisors to intercept every VMCS allocation and print both its virtual and physical addresses. These values were then compared with the output of our Volatility plugin to verify its correctness. We also used our instrumented hypervisors to print the content of all the VMCS fields and verify that their values matched the ones we extracted from the memory image using our tool.

Our final test was designed to test the virtual machine address space reconstruction through the EPT memory structures. The test was implemented by instrumenting existing hypervisors code and by installing a kernel debugger in the guest operating systems to follow every step of the address translation process. The goal was to verify that our introspection module was able to properly walk the EPT table and translate every address.

Once we verify the accuracy of our acquisition and implementation we started the real experiments.

5.3 Single-Hypervisor Detection

In this experiment we ran the `hyperls` plugin to analyze a memory image containing a single hypervisor.

Hypervisor	Guests	Candidate VMCS	Validated VMCS
HyperDbg	1	1	1
KVM	2	4	2
Xen	2	3	2
VirtualBox	1	2	1
VMware	3	3	3

Table 1. Single Hypervisor Detection

Top Hypervisor	Nested Hypervisor	VMCS Detection	Hierarchy Inference
KVM	HyperDbg	✓	✓
	KVM	✓	✓
XEN	KVM	✓	✓
	XEN	✓	✓
VMware	HyperDbg	✓	✓
	KVM	✓	✓
	VirtualBox	✓	✓
	VMware	✓	✓

Table 2. Detection of Nested Virtualization

We tested our plugin on three open source hypervisors (KVM 3.6.0, Xen 4.2.0, and VirtualBox 4.2.6), one commercial hypervisor (VMware Workstation 9.0), and one ad-hoc hypervisor realized for debugging purposes (HyperDbg). The results are summarized on Table 1. We run the different hypervisors with a variable number of guests (between 1 and 4 virtual machines). The number of candidate VMCS found by the memory scanner algorithm is reported in the third column, while the number of validated ones is reported in the last column. In all the experiments our tool was able to detect the running hypervisors and all the virtual machines with no false positives.

The performance of our system are comparable with other offline memory forensic tools. In our experiment, the average time to scan a 4GB memory image to find the candidate VMCS structures was 13.83 seconds. The validation time largely depends on the number of matches, with an average of 51.36 seconds in our tests (all offline analysis performed on an Intel Xeon L5420 (2.50Ghz) with 4GB RAM).

In the second experiment, we chose a sample of virtual machines from the previous test and we manually inspect them by running several Volatility commands (e.g., to list processes and kernel drivers). In all cases, our patch was able to transparently extract the EPT tables and provide the address translation required to access the virtual machine address space.

5.4 Nested Virtualization Detection

In the final set of experiments we tested our techniques on memory images containing cases of nested virtualization. This task is more complex due to the

implementation specific nature of the nested virtualization. First of all, only three of the five hypervisors we tested supported this technology. Moreover, not all combinations were possible because of the way the VMX instructions were emulated by the top hypervisor. This turned out to be crucial for the nested hypervisor to work properly, since an imperfect implementation would break the equivalence principle and allow the nested hypervisor to detect that it is not running on bare metal. For example, VMware refuses to run under KVM, while Xen and VirtualBox under KVM start but without any hardware virtualization support.

Because of these limitations we were able to set up eight different nested virtualization installations (summarized in Table 2). In all the cases, `hyperls` was able to detect and validate all the three VMCS structures (VMCS01, VMCS02, and VMCS12) and to infer the correct hierarchy between the different hypervisors.

6 Related Work

The idea to inspect the physical memory to retrieve sensitive information or to find evidence of malicious activities has already been broadly explored in the literature. For example, Alex Halderman et al. [17], described several attacks where they exploited DRAM remanence effects to recover cryptographic keys and other sensitive information. Several works focus their attention on the analysis of user space memory: Memparser [10] was one of the first memory analysis tools that was able to provide information about the modules loaded and the process parameters by leveraging the PEB memory structure. Dolan-Gavitt [13] was the first to allow the analysis of the Windows user-space process by extracting the VADs memory structure from a memory image. Arasteh and Debbabi [8] used the information about the stack memory structures to rebuild the execution history of a process. On the other side, several papers proposed systems to search kernel and user-space memory structures in memory with different methodologies. Dolan-Gavitt et al. [14] presented a research work in which they automatically generated robust signatures for important operating system structures. Such signatures can then be used by forensic tools to find the objects in a physical memory dump.

Other works focused on the generation of strong signatures for structures in which there are no values invariant fields [20, 21]. Even though these approaches are more general and they could be used for our algorithm, they produce a significant number of false positives. Our approach is more ad-hoc, in order to avoid false positives.

Another general approach was presented by A. Cozzie et al. in their system called Laika [11], a tool to discover unknown data structures in memory. Laika is based on probabilistic techniques, in particular on unsupervised Bayesian learning, and it was proved to be very effective for malware detection. Laika is interesting because it is able to infer the proper layout also for unknown structures. However, the drawback is related to its accuracy and the non negligible amount

of false positives and false negatives. Z. Lin et al. have developed DIMSUM [32] in which, given a set of physical pages and a structure definition, their tool is able to find the structure instances even if they have been unmapped.

Even though a lot of research have been done in the memory forensics field, to the best of our knowledge there is no previous works on automatic virtualization forensics. Our work is the first attempt to fill this gap.

Finally, it is important to note that several of the previously presented systems have been implemented as a plugin for Volatility [6] - the standard the facto for open source memory forensics. Due to the importance of Volatility, we also decided to implement our techniques as a series of different plugins and as a patch to the main core of its framework.

7 Conclusion

In this paper, we presented a first step toward the forensics analysis of hypervisors. In particular we discussed the design of a new forensic technique that starts from a physical memory image and is able to achieve three important goals: locate hypervisors in memory, analyze nested virtualization setups and show the relationships among different hypervisors running on the same machine, and provide a transparent mechanism to recognize and support the address space of the virtual machines.

The solution we propose is integrated in the Volatility framework and it allows forensics analysts to apply all the previous analysis tools on the virtual machine address space. Our experimental evaluation shows that Actaeon is able to achieve the aforementioned goals, allowing for a real-world deployment of hypervisor digital forensic analysis.

Acknowledgment

The research leading to these results was partially funded by the European Union Seventh Framework Programme (contract N 257007) and by the French National Research Agency through the MIDAS project. We would also like to thank Enrico Canzonieri, Aristide Fattori, Wyatt Roersma, Michael Hale Ligh and Edgar Barbosa for the discussions and their support to the Actaeon development.

References

1. Amd's market share drops. `http://www.cpu-wars.com/2012/11/amds-market-share-drops-below-17-due-to.html`.
2. Documentation/dma-mapping.txt.
3. Elcomsoft forensic disk decryptor. `http://www.elcomsoft.com/edff.html`.
4. Inception memory acquisition tool. `http://www.breaknenter.org/projects/inception/`.
5. Nehalem architecture. `http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf`.

6. Volatility framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.
7. Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.
8. Ali Reza Arasteh and Mourad Debbabi. Forensic memory analysis: From stack and code to execution history. *Digit. Investig.*, 4:114–125, September 2007.
9. Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: design and implementation of nested virtualization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
10. Chris Betz. Memparser. `http://www.dfrws.org/2005/challenge/memparser.shtml`.
11. Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.
12. Anthony Desnos, Eric Filiol, and Ivan Lefou. Detecting (and creating !) a hvm rootkit (aka bluepill-like). *Journal in Computer Virology*, 7(1):23–49, 2011.
13. Brendan Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digit. Investig.*, 4:62–64, September 2007.
14. Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 566–577, New York, NY, USA, 2009. ACM.
15. Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 417–426, September 2010.
16. R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM.
17. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.
18. Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Aug 2012.
19. Samuel T. King, Peter M. Chen, Yi min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *In IEEE Symposium on Security and Privacy*, pages 314–327, 2006.
20. Bin Liang, Wei You, Wenchang Shi, and Zhaohui Liang. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 217–227, New York, NY, USA, 2011. ACM.
21. Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

22. Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and Trustworthy Forensic Analysis of Commodity Production Systems. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2010.
23. Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.
24. Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.
25. Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. When Hardware Meets Software: a Bulletproof Solution to Forensic Memory Acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, USA, December 2012.
26. Joanna Rutkowska. Subverting Vista Kernel for Fun and Profit. *Black Hat USA*, aug 2006.
27. Joanna Rutkowska. Beyond The CPU: Defeating Hardware Based RAM acquisition. *Black Hat USA*, 2007.
28. Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.
29. Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.
30. Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
31. Xiantao Zhang and Eddie Dong. Nested Virtualization Update from Intel. *Xen Summit*, 2012.
32. Lin Zhiqiang, Rhee Junghwan, Wu Chao, Zhang Xiangyu, and Xu. Dongyan. Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS'12, 2012.
33. Dino A. Dai Zovi. Hardware Virtualization Rootkits. *Black Hat USA*, aug 2006.