Digital Investigation 24 (2018) S18–S28

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2018 Europe — Proceedings of the Fifth Annual DFRWS Europe

Styx: Countering robust memory acquisition

Ralph Palutke*, Felix Freiling

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

A B S T R A C T

Images of main memory are an increasingly important piece of evidence in cybercrime investigations, especially against advanced malware threats, and software tools that dump memory during normal system operation are the most common way to acquire memory images today. Of all proposed methods, Stüttgen and Cohen's *robust memory acquistion* (as implemented in the *pmem* tool) can be considered the most advanced technique today. This paper presents *Styx*, of a proof-of-concept system that perfectly covers its traces against *pmem* and other tools that perform software-based forensic memory acquisition. *Styx* is implemented as a loadable kernel module and is able to subvert running 64-bit Linux systems using Intel's VT-x hardware virtualization extension, without requiring the system to reboot. It further uses the second address translation via Intel's EPT to hide behind hidden memory. While exhibiting the limitations of robust memory acquisition, it also shows the potential of undetectable forensic analysis software.

Introduction

Malicious software (malware) is the enabling technology for almost all forms of modern cybercrime. Therefore, there is a great demand for methods to detect, acquire and analyze such software in a forensically sound manner. Since modern malware avoids traces on disk and operates in memory only (O'Murchu and Gutierrez, 2015), memory acquisition and analysis are vital within digital investigations. Among the many ways to acquire physical memory (Vömel and Freiling, 2011), we focus on software-based methods that execute on the target system and attempt to write a copy of physical memory to external storage. Software tools like *pmem* (Stüttgen and Cohen, 2013, 2014) are very common in practice because of their ease of use, but they are known to not always produce sound memory images (Vömel and Freiling, 2012). While there have been some advances in robust techniques for memory acquisition (Stüttgen and Cohen, 2014), investigators are faced with two new developments that threaten their defensive foothold: (1) malware in hidden memory, and (2) hypervisor-based malware.

Malware in hidden memory. While the physical address space is defined as the entire range of memory addresses that can appear on the memory bus, not all of it actually contains RAM. To distinguish free from reserved memory, the BIOS creates a memory map during the boot sequence, and thereby determines which memory regions are available to the OS and which ones are reserved for PCI devices, e.g., as buffers used for memory-mapped input/output (MMIO), or ROM. Commonly, reserved memory regions are avoided by memory acquisition tools, but some of these areas can still be backed by RAM. Stüttgen and Cohen (2013) demonstrated that it is possible to hide data in these areas which they called *hidden memory*.

Even though malware can potentially reside in hidden memory, it still needs to set its hooks in non-hidden memory to gain control. Furthermore, memory acquistion tools can enumerate hidden memory in the same way as malware does. So at least in principle such malware can also be detected.

Hypervisor-based malware. The second development refers to a new breed of stealthy malware that takes advantage of recent processors' virtualization technologies in order to migrate a running system into a virtual machine (VM). Since the virtualization happens on-the-fly, no reboot is required (Rutkowska, 2006). The malware thus acts as a thin hypervisor for the original system. In case the hypervisor uses a second level address translation to isolate its physical memory from the guest, a (guest) physical memory dump will not reveal the contents used by the malware. Proof-of-concept examples of such malware abound (see for example early work by Zovi (2006), Rutkowska and Tereshkin (2007, 2008); Rutkowska (2008), King et al. (2006) and Athreya (2010)).

* Corresponding author.
E-mail addresses: ralph.palutke@fau.de (R. Palutke), felix.freiling@cs.fau.de (F. Freiling).

https://doi.org/10.1016/j.diin.2018.01.004
1742-2876/© 2018 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Although hiding in the hypervisor appears to be an effective mitigation to memory acquistion, such malware still may be detected. One approach is to simply detect whether a system is virtualized (Ptacek et al., 2007; Garfinkel et al., 2007) However, many systems run within a hypervisor today and so this is no reliable detection criterion anymore. However, if the amount of physical memory within the system is known, then a malicious hypervisor may also be detected indirectly, as the memory image should result in the same size.

To see this, observe that the malware needs at least some pages of (host) physical memory to operate on. To hide these pages from a memory image, the hypervisor can redirect guest accesses to a set of *guard pages*. Therefore, it benefits from using a second level address translation mechanism like Intel's EPT in order to virtualize the guest's physical memory. However, these guard pages consume host memory, too. As hiding these pages would require their own guards and physical RAM is finite, a hypervisor can potentially be detected using heuristics (Stüttgen and Cohen, 2013).

In conclusion hypervisor-based rootkits are not able to effectively hide their existence even if they make use of advanced second level address translations.

Research goal and contributions

Overall, both streams of work do not have the potential of hiding malware completely from forensic analysis. However, if both approaches are combined, the weaknesses from one method can be counteracted by the strengths of the other:

1. Hidden memory can easily be simulated as device memory, not backed by accessible RAM. It could be a plausible explanation for pages that are not writeable and thus serve as a perfect hideout for guard pages.
2. A hypervisor-based rootkit approach virtualizes a running system on-the-fly. It allows to control the guest's view to its physical memory due to the possibility to use a second level address translation. The rootkit therefore has the chance to protect its own memory by redirecting accesses. Furthermore, the rootkit is able to control a virtualized system without installing a single hook inside the guest.

In this paper we combine both approaches and present the results from the development of a proof-of-concept system called *Styx* that we locate in hidden memory, which is concealed from the guest. *Styx* perfectly covers its traces from any current tool that performs software-based forensic memory acquisition.

While these results appear to be negative from a forensics point of view, the developed techniques can also be used to create undetectable forensic analysis tools. Example systems in this direction are *Hypersleuth* (Martignoni et al., 2010) and *VIS* (Yu et al., 2012). Both projects make use of forensic tools like a memory dumper or a system call tracer from within a hypervisor after virtualizing a running system. Such tools could also be integrated into *Styx*. They would thus operate from hidden memory, entirely isolated and undetectable from within the guest.

Styx is implemented as a *loadable kernel module* (LKM) and is able to subvert running 64-bit Linux systems using Intel's VT-x hardware virtualization extension, without requiring the system to even reboot. We developed an engine that locates hidden memory and manages its individual ranges. During installation, all of *Styx*'s code and data segments, as well as its dynamically allocated memory, are migrated to hidden memory. After the system was virtualized, we ensure to erase all remaining traces residing in the guest's address space. We set up a second address translation via Intel's EPT to refuse the guest from accessing hidden memory.

To evaluate the stealthiness of our implementation, we enhanced *pmem* (Stüttgen and Cohen, 2013, 2014), one of the most sophisticated memory acquisition tools available, to acquire hidden memory with the same algorithm we used for *Styx*. The evaluation did not reveal any indicators, which could have helped to identify or analyze *Styx* in memory. Thereby, we examined the entire hidden memory as well as selected memory ranges in the guest kernel space which at some point in time contained data of our prototype. We are not aware of any other software-based tool that is capable to acquire hidden memory. As a result, we conclude that *Styx* successfully hides from state of the art memory acquisition tools which do not rely on DMA-based acquisition techniques.

Currently, the only way to spot our system is to acquire its memory via either direct memory access (DMA) or by leveraging Intel's system management mode (SMM). However, DMA-based approaches could be easily defeated by restricting device accesses with the help of Intel's VT-d mechanism and the usage of an IOMMU. In theory, even DMA-based methods would then fail to detect our system. On the contrary, software running in SMM is resilient to subversion from the kernel or hypervisor level. Deploying memory acquisition software into the SMM was first proposed by Wang et al. (2011). A first proof-of-concept prototype presented by Reina et al. (2012) followed closely. Since installing code in SMM requires a cryptographically signed firmware update (and thus vendor support) as well as a reboot, it appears impractical for most cases at the moment.

Paper outline

Section Technical background recalls some background on virtual machine extensions (VMX) and the extended page table (EPT) mechanism supported by modern Intel processors. In Section Design we give an architectural overview of *Styx* and explain how it virtualizes a running system. More details about the actual implementation can be found in Section Implementation. In Section Evaluation we evaluate *Styx*'s memory footprint by utilizing an enhanced version of Rekall's *pmem* acquisition tool. The final section concludes the paper with a short summary and known limitations of our research.

Technical background

This section provides fundamental knowledge about features and mechanisms that played an important role in the implementation of *Styx*. We briefly introduce the concept of hidden memory and provide insight into Intel's virtualization features referred to as VT-x.

Hidden memory

During startup the BIOS creates a memory layout called the *E820 map* by requesting BIOS interrupt 15 while still running in real mode. The map classifies physical memory regions into two types:

- *Free* regions are available for the OS to use.
- *Reserved* regions are areas that might be used for device memory like MMIO buffers.

As depicted in Fig. 1, this layout is redefined every time a PCI device maps its buffers within a reserved region (PCI, 2004). Devices, however, do not have to consume the entire reserved area they claimed. The term hidden memory refers to these unused offcuts in the reserved regions that are still backed by RAM (allowing read, write and execute accesses). Accesses to these regions are usually avoided by the kernel in fear of creating system

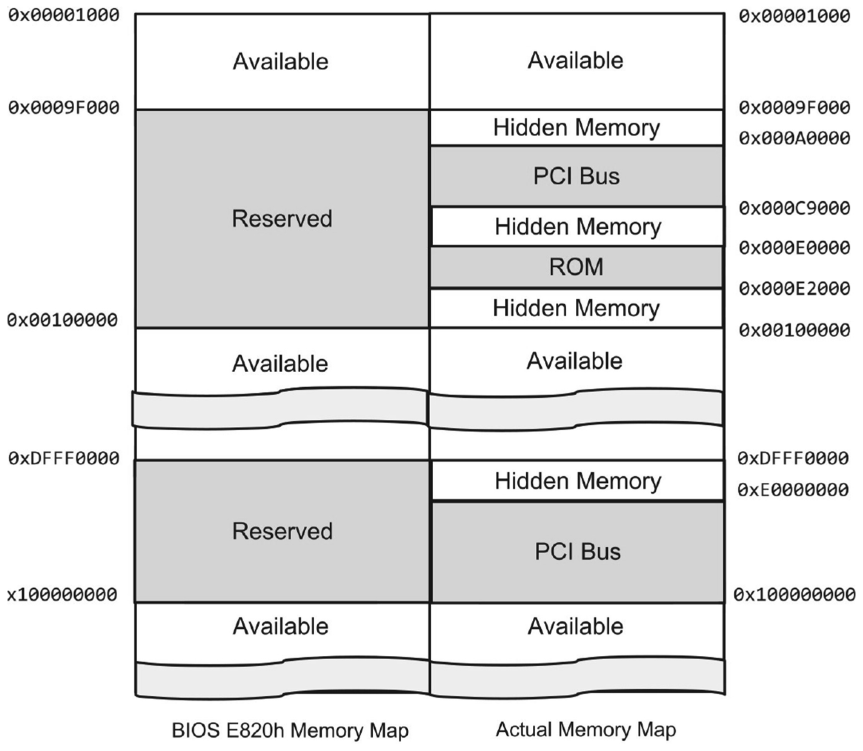

Fig. 1. Hidden memory layout on a 4 GB test system (Stüttgen and Cohen, 2013), Fig. 2.

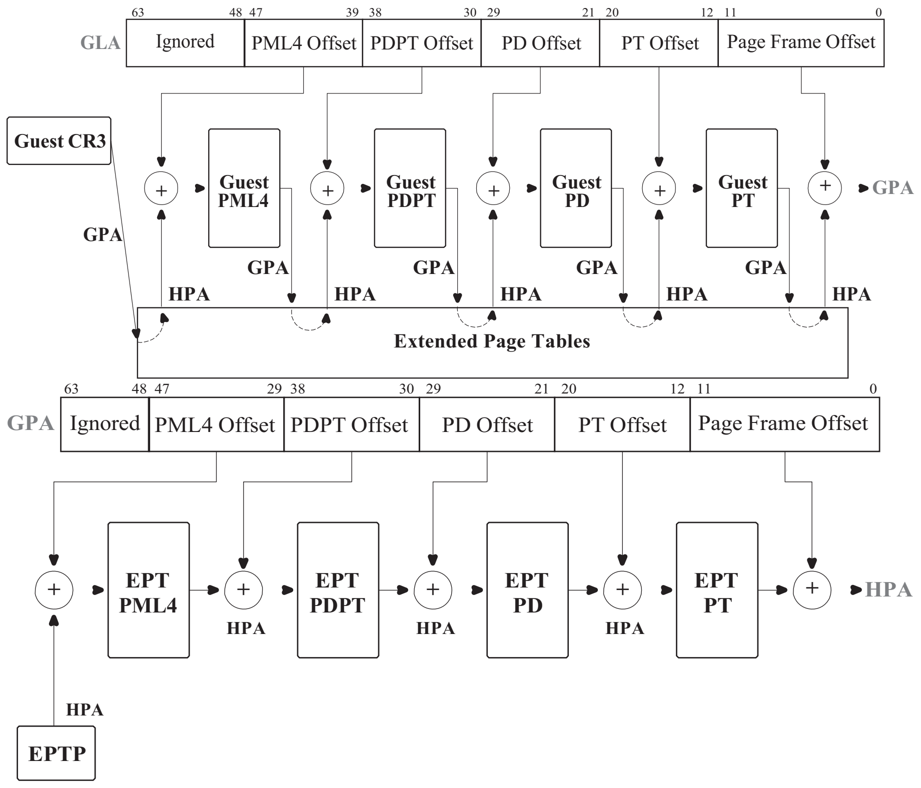

Fig. 2. Address translation via Intel EPT.

instabilities. Therefore, hidden memory regions are usually unknown to the kernel or peripheral devices. Although hidden memory seems like a perfect hideout for advanced stealth malware, to the best of our knowledge, there is no forensic acquisition tool that considers these areas on Linux systems.

Intel VT-x

With the emergence of hardware-supported virtualization, Intel introduced new features which are summarized under the term VT-x. These features, called *virtual machine extensions* (VMX), allow to run unmodified *guests* inside VMs. VT-x also provides the possibility to virtualize a guest's physical address space via *extended page tables* (EPT). The remaining part of this section provides a deeper understanding on the subject. For further information see Intel's processor manual (Intel, 2017).

Intel VMX

Processors which support VMX allow virtualization to be assisted by the hardware. This is because guests running inside VMs can now be executed directly on a physical CPU. In the context of virtualization we distinguish between two different roles: The hypervisor or *virtual machine monitor* (VMM) and the VM or *guest*. The hypervisor represents the host software, which is in full control over the physical CPU and schedules physical resources between virtualized guests. A VM on the other hand serves a guest system as a transparent virtual execution environment.

To establish *hardware-based virtual machines* (HVM), VMX introduces a new processor mode called *VMX operation*.[1] The mode splits up into *VMX root operation* and *VMX non-root operation* which can be switched via *VMX transitions*. While the first describes the privileged mode a hypervisor is running in, the latter appears as a restricted mode to run guest systems. With the appearance of certain events (execution of restricted instructions, access to specific registers, or interacting with emulated devices) in VMX non-root mode, the processor triggers a *VM-exit* which transfers control to the hypervisor. It now has the chance to handle the event and return control back to the guest by generating a *VM-entry*. This allows to run unmodified operating systems inside a VM.

To setup, launch and control a VM, the hypervisor has to configure a *virtual machine control structure* (VMCS) for each logical processor. The VMCS contains the entire state of a VM, including information on how and when to perform VMX transitions.

Intel EPT

Intel's *extended page tables* (EPT) support the virtualization of physical memory. In use, certain physical addresses are treated as *guest physical addresses* (GPA), which restrict a guest to directly access physical memory. Instead, GPAs are translated into *host physical addresses* (HPA) using a set of EPT paging structures. A HPA can then be used to access physical memory. The translation itself is fully controlled by the hypervisor.

The translation of a GPA to a HPA is determined by a set of EPT paging structures which are similar to the ones used in IA-32e mode. The whole translation mechanism now splits up into two parts, the *first level address translation* (FLAT), followed by the *second level address translation* (SLAT). Therefore, two different paging hierarchies are required. The FLAT-phase uses a guest's paging structures, now called *guest paging structures*, to translate a *guest linear address* (GLA) into a GPA. Unlike the usual translation, the physical addresses of the guest paging structures are now also treated as GPAs. Before they can be accessed, they thus need to be translated into HPAs, too. The SLAT-phase, however, refers to the new EPT paging structures. A GPA is then used to traverse these structures to compute the final HPA. In case the guest possesses sufficient privileges, it can now access the memory. Similar to conventional paging, traversing the EPT hierarchy splits up a GPA in different parts to use these as offsets in the respective EPT table. To locate the root of the EPT hierarchy, the processor consults a hardware register called the *extended page table base pointer* (EPTP), which acts as the counterpart to the FLAT's `cr3` register. Fig. 2 depicts the translation of a GLA to its corresponding HPA during both the FLAT and the SLAT phase.

[1] In this paper we interchangeably use the terms VM and HVM.

Like page table entries in IA-32e mode, EPT entries have several privilege bits that prohibit unauthorized memory accesses. If an EPT paging-structure entry does not meet the required privileges during traversal, the processor generates a VM-exit due to an *EPT violation*. Hence, the hypervisor retains control and is able to handle the trap.

Design

Styx can be classified as a small *hardware-based hypervisor rootkit* that hides its complete memory footprint in hidden memory. Note that we do not consider a rootkit as inherently malicious, but rather as a neutral tool which purpose is to hide both malicious and benign software.

With the help of Intel's VT-x features, *Styx* migrates a running Linux operating system into a VM, without requiring the system to ever reboot. In addition, it protects its own memory by virtualizing the guest's physical address space using Intel's EPT. Furthermore, we enhanced the memory acquisition tool *pmem* to be capable of accessing hidden memory. In the following, we shed light on the design architecture of our system.

Styx consists of three parts: a user mode component, called the *Installer*, and two *loadable kernel modules* (LKM). The first LKM, referred to as the *Rootkit LKM* is used to virtualize the system, while the second LKM's (*Eraser LKM*) purpose is to erase all remaining traces in guest memory. Fig. 3 illustrates the steps taken during *Styx*'s entire setup.

The installation process

The installation process is controlled by the Installer running in user mode. As the installation of kernel modules requires elevated process privileges, we need to launch the Installer as *root*. Notice that *Styx* is not able to subvert systems, which already utilize VMX as these can only be claimed exclusively. Since this is not the default case on Linux systems, it does not appear as a major concern.

Based on a clean or uninfected system (see Fig. 3, part 1), the Installer loads the Rootkit LKM (see Fig. 3, part 2). The module then starts to enumerate all hidden memory ranges and sets up EPT for later usage. Both LKMs are implemented as miscellaneous devices, which allow direct communication to the user land via *IOCTLs* by referring to its corresponding device entries located under `/dev`. To migrate the rootkit to hidden memory later on, it is mandatory to know both the base addresses and sizes of each of its loaded segments. Because this would be harder discovered by the LKM itself, we decided to shift this task into user mode and let the Installer handle it. Therefore, the Installer parses all ELF segments when the Rootkit LKM has finished its loading routine. Information about a LKM's segment base addresses can be found in the virtual file system mounted under `/sys`. In the case of our Rootkit LKM this means under `/sys/module/styx/sections`. The directory contains a file for each segment, named after the section it belongs to. Each file in turn holds the linear base address of its corresponding segment. The most convenient way to get their corresponding sizes was to parse the rootkit's ELF header. As the segment sizes do not change during runtime, the Installer directly parses the ELF header in the compiled binary of the Rookit LKM. These segment information are then send from the Installer to the Rootkit LKM (Fig. 3, part 3). After receiving the data, Styx is able to determine its own memory regions and start to migrate its segments to previously enumerated hidden memory. Thereafter, the rootkit can finally virtualize the running system (Fig. 3, part 4). From thereon, the system runs inside a fully controllable VM, while Styx completely resides in hidden memory, entirely isolated from the guest. To prevent the guest from accessing its memory, Styx

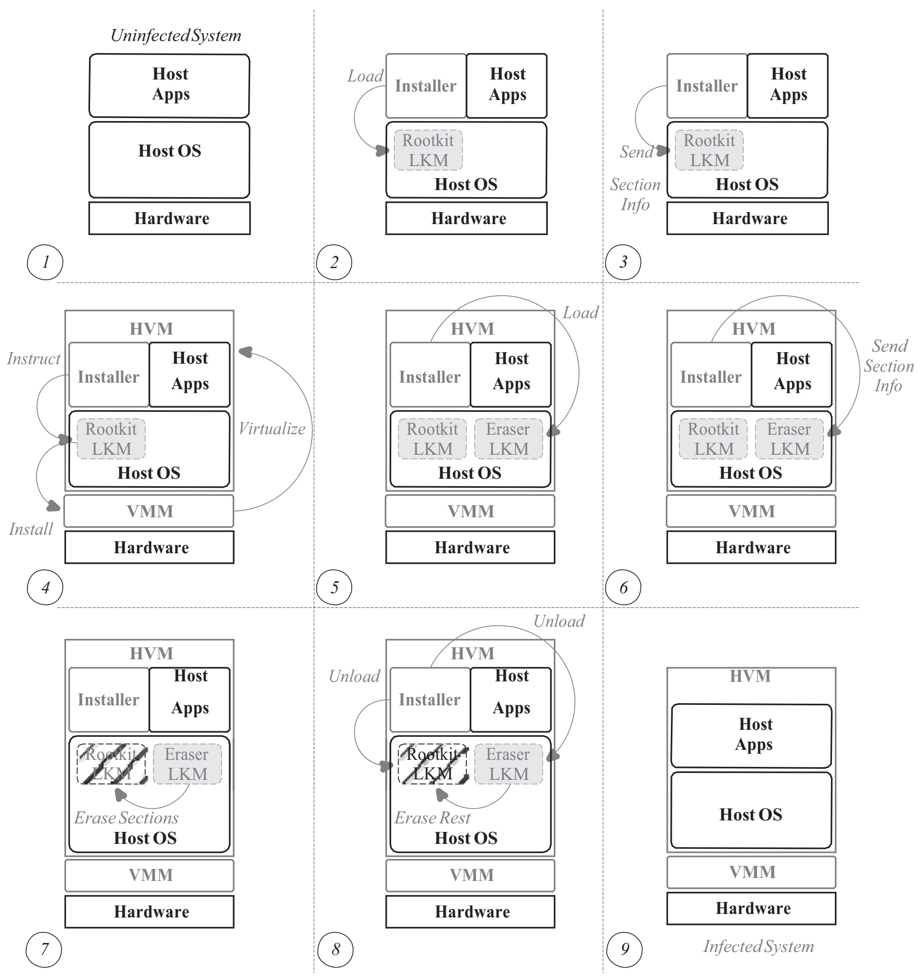

Fig. 3. *Styx*: System setup.

protects itself by redirecting accesses to a prior allocated guard page by utilizing EPT. The guard page, also located in hidden memory, mimics a page in device memory which is not backed by accessible RAM. Any read attempts therefore do not return any meaningful data, while write accesses are intercepted by the hypervisor and simply disregarded.

The erasing procedure

Styx's second kernel component is called the Eraser LKM. Its purpose is to delete all remaining traces that occurred during the hypervisor's installation process. Once again, the Installer sends the prior parsed segment information to kernel space. This time, however, to the Eraser LKM (Fig. 3, parts 5 and 6). Since at this point the Rootkit LKM was not unloaded by the Installer, yet, its memory could not have been claimed elsewhere. Therefore, the Eraser LKM can safely delete the rootkit's segments after it received the required information from user space (Fig. 3, part 7). However, some of these segments are required during the Rootkit LKM's unloading routine and hence must remain valid. Any attempt to delete them prematurely causes the kernel to panic when unloading the module. When the Rootkit LKM finished to transfer its segments to hidden memory, the Installer can safely unload the LKM without terminating the rootkit's execution (Fig. 3, part 8). Afterwards, the leftover segments can be erased, if they were not already claimed elsewhere (Fig. 3, part 8).

As a result, we have installed a thin-hypervisor, that lives in hidden memory, beneath the original system (see Fig. 3, part 9). No traces of the rootkit remain in the guest's address space.

Implementation

Based on the design outlined in the previous section, we developed a prototype of our approach called *Styx*. This section provides several implementation details. The majority of the code base is written in C, with a few exceptions implemented in both inline and pure assembly. Upon request, we will provide the code for research purposes.

The rootkit LKM

The Rootkit LKM represents the core of *Styx*. Its main purpose is to virtualize a running Linux system and migrate its own code and data to hidden memory. Furthermore, it sets up a second level address translation to virtualize the guest's physical memory, so it can hide and protect itself from guest accesses.

System virtualization

Initially, the Rootkit LKM is loaded by the Installer via the `init_module(2)` system call and control flow is diverted to the module's initialization routine. As we want our rootkit to be located in hidden memory later on, we first need to find all of its ranges (we explain later how this is done). This also allows dynamic allocations to be restricted to hidden memory only. Furthermore, our management engine provides some basic functionality to allocate and free chunks and accomplishes address translations in regard to hidden memory.

The Rootkit LKM sets up a second level address hierarchy by using Intel's EPT mechanism. As hidden memory was already enumerated at this point, every EPT paging structure is allocated directly in hidden memory. The EPTs are later used to hide and protect the hypervisor rootkit's memory by redirecting or intercepting guest accesses.

Styx, at its core, represents a thin-hypervisor that makes use of Intel's VMX technology to virtualize a running Linux system on-the-fly. To support the virtualizion of multi-core systems, we followed a symmetrical hypervisor design. To be able to fully virtualize a multi-core system, every logical processor needs to be switched to VMX operation by setting the `VMXE` bit in control register `cr4`. Therefore, the kernel thread, executing our installation routine, is pinned to each logical core. A symmetric design does not preclude asymmetry in its operations, however. Hence, each logical processor has its own set of hypervisor specific data to avoid synchronization problems.

Unfortunately, the amount of hidden memory, found on our test system, was insufficient to hold all of these data structures. This was mostly because each logical processor requires its own paging hierarchy. Therefore, *Styx* could only be tested on single core machines. Although we did not further investigate multi-processor issues, we believe that our implementation will generally allow the virtualization of multi-core systems if the required amount of hidden memory is available.

Once the current kernel thread was pinned to a logical processor, the actual setup of the thin-hypervisor begins. Notice that all following actions are executed on every logical processor installed on the system. The LKM starts off by allocating hypervisor specific data structures. Each allocation is executed by the hidden memory manager (see below) and thus guaranteed to be located in hidden memory. These data structures include a private IDT, GDT, TSS, MSR-bitmaps, VMXON region, guest VMCS and a paging hierarchy to isolate from the kernel. Since we do not support interrupts during the hypervisor's execution, we set up the IDT with dummy entries which simply return to the hypervisor by executing the `iret` instruction.

The virtualization process mainly consists of three parts: It checks if the processor fulfills the necessary requirements to enter VMX operation, initializes relevant data structures like the VMCS and finally launches the guest into a VM. The VMCS allows to declare important information for both the guest and the hypervisor. For example the hypervisor's entry point, its stack, control register values, etc. are specified in the VMCS's host state area. To lift the running system into a VM, the guest state fields of the VMCS (like instruction pointer, stack, segment registers, control register, etc.) need to match the guest's current state. The VMCS also allows us to specify which events should be intercepted by the hypervisor. As this is totally dependent to the use case *Styx* is required for, our prototype limits guest interceptions to a minimum. An exception makes the interception of EPT violations, as these are required to be handled during write accesses to hidden memory (see below). To hide the previously enabled `VMXE` bit, we make use of shadow masks provided by the VMCS. These fields are located in the guest state area and allow the virtualization of both the `cr0` and `cr4` control registers. That way, we can fool the guest view to the `cr4` register and intercept critical write accesses.

To erase traces regarding allocations in hidden memory, we need to eliminate its corresponding mappings in the guest's page tables. Although the guest would not able to access these areas due to the rootkit's EPT protection mechanism, it could still detect its suspicious PTEs in its own page tables. Therefore, it is mandatory to invalidate these PTEs. Finally, we can enter the virtualized guest by executing the `vmlaunch` instruction. Because we renounced to map the kernel into the hypervisor's page tables, we cannot use any kernel functionality from within the hypervisor. While this prevents us from using kernel functionality, it simultaneously grants protection against code that would try to reveal the rootkit's existence.

In the end, the hypervisor resumes the guest by returning to the Rootkit LKM's `module_init()` function. There, final preparations like de-registering from the virtual `/proc` file system take place. Besides, the rootkit closes the IOCTL channel to user mode and thus refuses to receive further instructions. The LKM can then be unloaded by the Installer via `delete_module(2)`. This has the effect that all of the rootkit's file system entries are deleted and its kernel data structures are deregistered. Hence, tools like `lsmod` fail to display any suspicious information. Due to its prior migration to hidden memory, the rootkit still operates as its intended, however.

As a result, *Styx* secretly runs beneath the guest and stealthily awaits further instructions.

Hidden memory manager

The *hidden memory manager*'s main purpose is to enumerate all hidden memory ranges in physical memory. It also enables the hypervisor to allocate or free chunks of hidden memory and provides functionality for linear to physical address translations (and vice versa) with the help of its TLBs.

Initially, we assume the whole physical memory to be one consecutive hidden memory region. Over time, the layout is redefined by excluding areas that do not belong to hidden memory. Thus, we start out with a single region ranging from the base of the physical memory to the maximum address supported by the processor. To ascertain these addresses, we make use of the `iomem_resource` tree. It is created during system boot and contains a set of `resource` structures, which represent non-overlapping physical memory ranges. Each of these structures contains the base and the end address of a particular region, as well as a pointer to next resource structure. Its root is referenced by the `iomem_resource` symbol provided by the kernel. Because the resources are sorted, finding out both the beginning and the end of physical memory is simply a matter of traversing the tree. Our enumeration algorithm is described below:

1. **Exclusion of PCI device ranges:** To exclude the memory regions of PCI devices, we manually enumerate the PCI bus. This presupposes that some firmware configured the PCI buses properly. Our enumeration function probes each of the possible 256 PCI buses, as well as every one of its 32 possibly connected devices. Furthermore, it iterates over eight functions a device could potentially offer. It then reads the vendor ID from each of their corresponding configuration spaces and checks if the device is present. Otherwise, we continue our search at the next function. In case the vendor ID is valid, we exclude the MMIO buffers of the device from the hidden memory layout. We therefore disable I/O and memory bus accesses, before we access the corresponding *base address register (BAR)* and reenable them afterwards.
2. **Exclusion of** `iomem_resource` regions: The next step is to exclude all memory ranges, provided by the `iomem_resource` tree, that are not marked as reserved. Example ranges to exclude are System RAM and Adapter ROM. Although the `iomem_resource` tree contains PCI ranges, it was necessary to manually exclude them in the first step. This is because the tree layout could already be outdated, as new devices could have been

added via hotplug. Since any access to a PCI device range could lead to undefined behavior (including system crashes), we absolutely must ensure to exclude them correctly. Up to this moment, we managed to exclude all memory regions apart from those, which are marked as reserved and do not belong to a PCI device.

3. **Content validation:** On our test system, the preceding step reduced the remaining memory to a few 1000 pages as the *reserved* ranges make up only a small part of physical memory. Thereat, we check if any of the remaining regions contains meaningful data. This is because reserved regions include to store read-only data (*System ROM*). Validation is important, because overwriting existing data could cause fatal damage to the system's stability. We thus run through each region's pages and "OR" their content quardword-wise to a zero initialized variable. Afterwards, we check if the variable still equals its initial value. If it does not, we know that at least one quadword had to hold a value unequal to zero. We then can exclude the particular page from the layout. By examining the regions page-wise, we ensure that not the whole range is excluded, although only a few of its pages contain data. Until now, only reserved regions that do not contain data are left in the layout.
4. **Write access verification:** The regions found so far do not contain data and are not mapped by the kernel. Hidden memory basically represents the offcuts in the reserved regions that are not used by a device, but are still backed by accessible RAM. Therefore, we need to verify their accessibility. Like in the last step, we page-wise review each range and thus ensure to exclude only non-accessible pages instead of the entire range. The actual verification is done by a manual memory probing algorithm, which only requires a few lines of assembly code (see Listing 1). We used a cache flush to ensure to test the actual physical address instead of the cache. In case the probed value did change, we managed to find a page backed by RAM. We had to insert a dummy write to an accessible address between overwriting the value and flushing the cache, as writes to "non-RAM" could be retained due to bus capacitance. Including the dummy write ensures to read back the dummy value instead of the tested data, in case no RAM exists at the probed address. Finally, we restore the page's original content.

Listing 1. Hidden memory probing.

```
u64 dummy;
bool ret = false;
__asm__ __volatile__ (
  "orq  $0xff8,%1\n\t"    /* last quadword */
  "movq (%1),%%rax\n\t"  /* original value */
  "movq %%rax,%%rdx\n\t" /* original value */
  "notq %%rax\n\t"       /* reversed value */
  "movq %%rax,(%1)\n\t"  /* modify value */
  "movq %%rdx,(%2)\n\t"  /* dummy write */
  "wbinvd\n\t"           /* cache flush */
  "movq (%1),%%rcx\n\t"  /* new value */
  "movq %%rdx,(%1)\n\t"  /* restore original */
  "cmpq %%rdx,%%rcx\n\t"
  "setz %0"
     : "=r" (ret)
     : "r" (block), "r" (&dummy)
     : "%rax", "%rdx", "%rcx", "%rsi", "memory"
);
return ret;
```

5. **Data migration:** To keep track of the hidden memory ranges, we set up several management structures. Although we tried to limit hypervisor-specific data allocations to hidden memory only, we first had to allocate these structures in ordinary kernel space. After completing the enumeration phase, we made sure to relocate these data to hidden memory, too. Therefore, we simply instruct the management engine to allocate a chunk of hidden memory and copy the corresponding data. We then erase these management structures to prevent remaining traces in the guest's kernel space.
6. **Hidden memory TLB setup:** At last, we set up a translation lookaside buffer (TLB) which is used to efficiently manage hidden memory linear to physical address mappings. As we do not map the kernel into the hypervisor's page tables (see 4.1.3), we cannot rely on kernel functionality to translate linear to physical addresses and vice versa. To prevent the overhead of page table walks, we use our own TLB to speed up address translations regarding hidden memory. Like all the other data structures of the hypervisor, the TLB itself is located in hidden memory.

On our test system we were able to find two hidden memory ranges. While the first goes from `0xcd000` to `0xedfff`, the second ranges from `0xbffe2000` to `0xbfffffff`. Therefore, we found 63 pages ($\sim$ 252 kilobytes) that serve our rootkit as a hiding spot. This small amount of hidden memory appears as the major limitation of the usefulness of our system.

Private paging hierarchy setup

To fully isolate the hypervisor from the guest system, it is essential to use a separate paging hierarchy. Its corresponding paging structures must be located in hidden memory. Otherwise, software would have the chance to detect them. Because the rookit's amount of memory is fairly small, the paging hierarchy results in a very sparse layout.

During the setup, we also migrate the rootkit's memory segments from kernel space to hidden memory. When loading a module, the kernel uses `vmalloc()` to request memory for its segments. The function does not guarantee those regions to be continuous in physical memory, however. Therefore, we copied each segment page-wise to hidden memory. We took care that the mappings of the copied segments match their counterpart in the guest's original page tables. That way, we did not have to accomplish any relocations within the segments. For example, a page with a GLA of `0xdeadbeef` needs to be mapped with the same host linear address (HLA). That means, that the page must occupy the entries `PML4[0]`, `PDPT[3]`, `PDE[245]` and `PTE[219]` in both the guest's and the hypervisor's paging hierarchies. Instead of the physical addresses of the original segments, we assign the addresses of the copies, located in hidden memory, to the hypervisor's corresponding PTEs. Fig. 4 illustrates the remapping process. To prevent time consuming page table traversals during address translations, we additionally include the established mappings into the hidden memory manager's TLB (see above).

After all segments were copied and included in the appropriate page tables, we map the hypervisor's dynamically allocated data structures. Since all of these are executed by the hidden memory manager, they are guaranteed to be located in hidden memory and thus do not need to be migrated from kernel space. However, it is important that all dynamic allocations were done up to this point. Otherwise, a particular data structure would not be mapped in the appropriate page table and the hypervisor would not be able to access it later on. Any attempt would lead to fatal system crashes.

In conclusion, we created a paging hierarchy that exclusively maps hidden memory that is used by the hypervisor. In our test system, we had to allocate one page map level 4 table (PML4), two page-directory pages tables (PDPT), three page directories (PD) and five page tables (PT), to map the hypervisor's entire memory.

Extended page table setup

During the hypervisor's setup, *Styx* creates a second level address translation in order to virtualize a guest's physical address

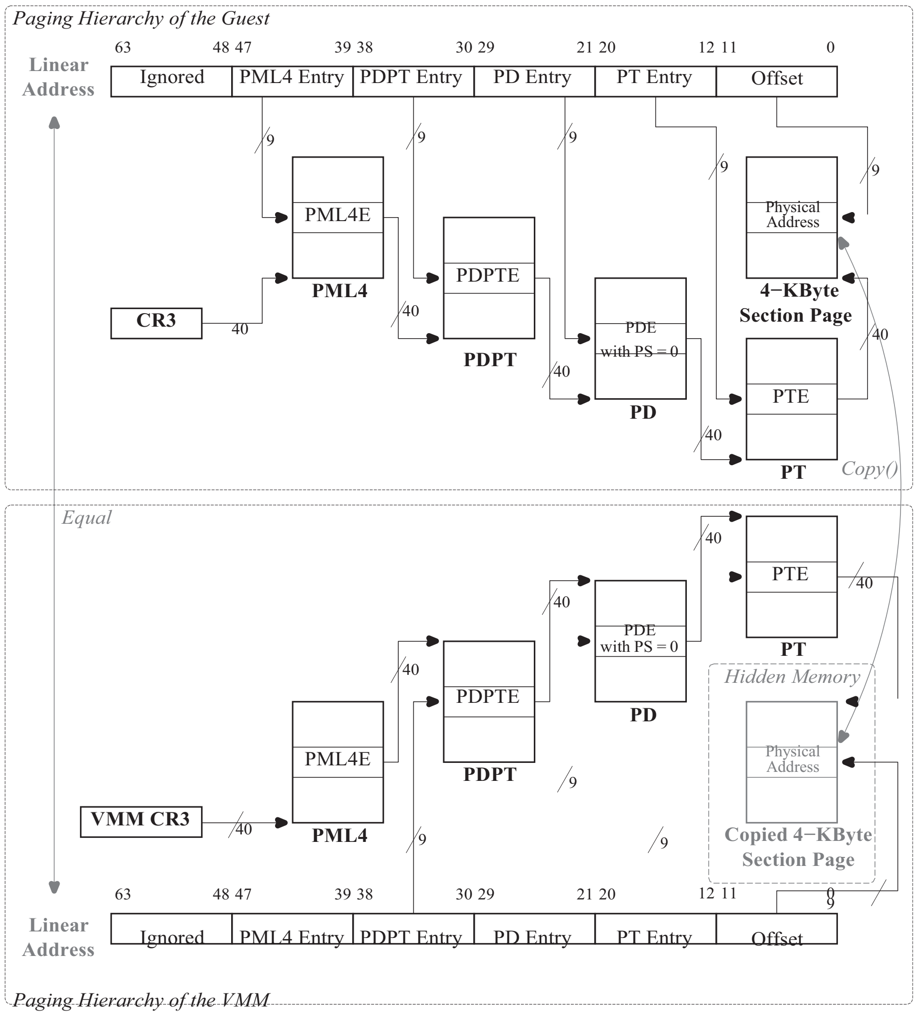

Fig. 4. Segment remapping to hidden memory.

space. We therefore use processor support by using Intel's EPT mechanism. That way, we are able to hide and protect our rootkit in hidden memory. For multi-core systems, the memory layout looks the same for all logical processors, so we only need to establish one EPT hierarchy to virtualize the whole physical memory.

The EPT hierarchy is created using an identity mapping for the entire physical memory. This means, we assign every GPA to the exact same HPA. Like every other data of the hypervisor, we want to locate EPT paging structures in hidden memory. As the amount found on our system was very limited, we reduced the memory required by the EPT paging structures by using 2-megabyte mappings. This resulted in the use of a three-level paging hierarchy. We chose page mappings of 2 megabytes instead of 1 gigabyte, because these constitute a better compromise between the consumed amount of memory and the level of control due to a finer granularity.

To setup the EPT leaf structures, we had to assign the correct memory type for each individual physical address range. These information can be gathered by consulting the corresponding fixed-range and variable-range *memory type range registers* (MTRR). MTRR support is determined by bit 12 in the feature information register (`edx`) after consulting CPUID leaf 1. If supported, additional information regarding memory types can be retrieved from the MSRs `IA32_MTRRCAP` and `IA32_MTRR_DEF_TYPE`. If, however, a 2-megabyte page contains multiple regions with different memory types, we must remap it with a 4-kilobyte granularity. Only so, we can specify the correct memory type for each page individually. Entering the guest with incorrect EPT settings would lead to immediate system freezes due to EPT misconfigurations. Typically, the first megabyte of physical memory is split into several regions with different memory types. On our test system, this region was the only range that exhibits distinguishing memory types. Furthermore, we configure a leaf's access rights to full access (readable, writable and executable) for any physical mapping except hidden memory and assign its HPA to match the same GPA. Fig. 5 illustrates the EPT memory layout found on our test system.

After locating our rootkit in hidden memory, we configure EPT to hide/protect its appropriate address ranges. This can be achieved by disabling certain access rights in their corresponding EPT leaf entries. Since hidden memory ranges are not guaranteed to match 2 megabytes in size, these regions need to be mapped with a page-size granularity, too. This allows an exclusive protection of hidden memory instead of preventing guest accesses to the entire 2-megabyte range. Basically, EPT allows to control read, write and execute accesses to each memory range mapped by a leaf structure. Any attempt to access a region without the appropriate EPT privileges causes an EPT violation which we configured to be intercepted by the hypervisor. To minimize VMX transitions, we redirect guest reads to a *guard page* by modifying the HPA in the appropriate EPT PTEs. The guard page is a 4-kilobyte page that only contains zeros (thus also called a *zero page*). Still, we need to prevent the guest from write accesses, as this would also affect the remaining hidden memory protected by the same guard page. For example, if

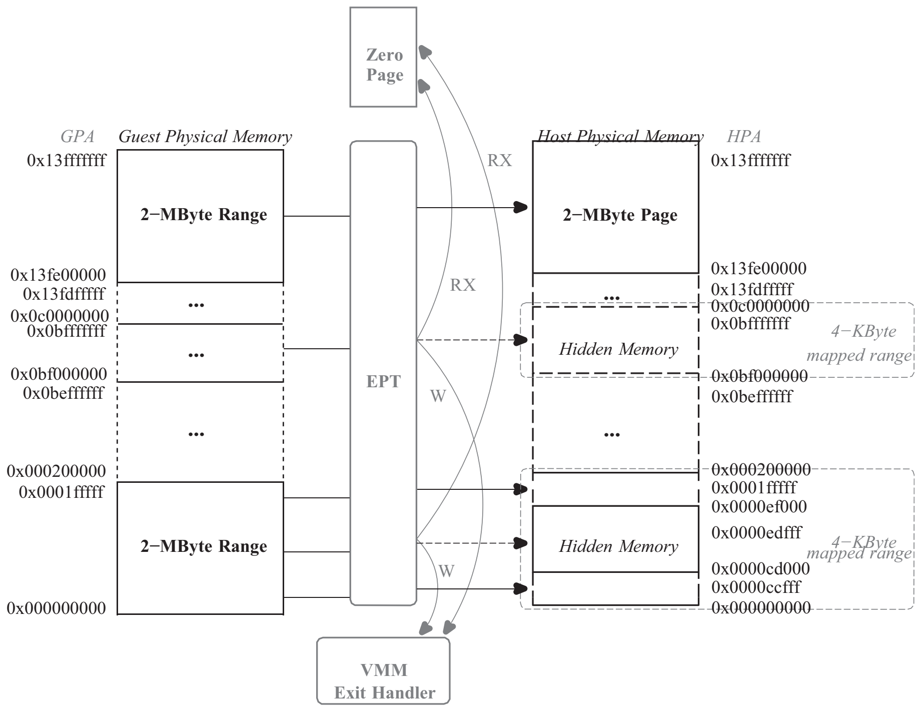

Fig. 5. EPT-based Second Level Address Translation (the abbreviations *R*, *W*, *X* symbolize read, write and execute accesses).

the guest reads both the contents of the two protected pages A and B, it would retain zeros in both cases. If it writes a specific pattern to page A, however, and again reads the content of page B, it would contain A's pattern, as both pages are redirected to the same guard page. Since this would raise suspicion, we intercept write accesses to hidden memory. To simulate writes to memory that is not backed by accessible RAM, the hypervisor simply ignores them and resumes the guest's execution. That way, we can share the guard page between all appropriate mappings which allows to virtualize an arbitrary amount of memory with single a 4-kilobyte page. The EPT protection mechanism is also depicted in Fig. 5.

From within the guest system, not even our own enumeration algorithm could find the hidden memory regions claimed by the hypervisor anymore. This is because the hypervisor prevents its ranges from being probed as described above. Hence, not even memory acquisition tools that consider hidden memory, would be able to detect *Styx*.

The Eraser LKM

During the hypervisor's setup, various traces arise in the guest's address space. These traces comprise the rootkit's original memory segments located in kernel space. Since a LKM cannot erase its own sections, *Styx* fulfills this task by using a second LKM. This Eraser LKM is loaded by the Installer as soon as the Rootkit LKM finished the system virtualization, but before its unloading routine was executed. At this point, all of the rootkit's segments as well as its dynamically allocated memory were already copied to hidden memory. Since the Rootkit LKM was not unloaded, yet, its original kernel memory cannot be claimed elsewhere. Like with the Rootkit LKM, the Eraser retrieves its segment information via IOCTL from *Styx*'s user mode component. The Eraser then zeros out every segment with the kernel's `memset()` function. Note that despite erasing the rootkit's segments, their appropriate PTEs remain intact. This, however, does not imply an issue as they are invalidated by the kernel during the Rootkit LKM's unloading process.

Some segments, however, are mandatory for the successful unloading of the Rootkit LKM and can thus only be erased afterwards. The rootkit's `module_exit()` function, placed in the .exit.text segment, is one example of such a segment. To ensure a complete erasure of all segments, these must be overwritten after the Rootkit LKM was unloaded. This time however, its necessary to verify the segments were still not claimed elsewhere. Otherwise, overwriting their contents, could lead to fatal kernel crashes, as we could potentially delete required code or data. Because the rootkit's segments would have been overwritten in this case anyways, we do not have to take any further considerations. If the PTEs are still unused, however, we `kmap()` them into kernel space and erase them afterwards.

As a LKM's segments are allocated via `vmalloc()`, they are not guaranteed to be located in one continuous physical chunk. Therefore, we must map each of its pages individually before we can entirely overwrite a segments content.

Evaluation

We use KVM with enabled nested virtualization support to run a 64-bit Ubuntu Server 14.04.1 LTS, which serves as the guest system. The kernel version of the guest is 3.13.0–37-generic. At the time we started the project, these seemed the predominant server systems. We configured KVM with a series of different CPU configurations and assigned the guest varying amounts of RAM. As a result, the located hidden memory differed depending on the system's configuration and was not always sufficient to install *Styx*. Assigning 4 gigabytes of RAM showed the best results in terms of available hidden memory. Further research is required to evaluate the amounts of hidden memory for a wide range of both virtualized and unvirtualized system configurations. To date, *Styx* is exclusively tested under KVM because of our fear to overwrite sensitive memory regions on our real machine, as their write protection could potentially be deficient. Nevertheless, we expect *Styx* to be functional on physical hosts and also on newer kernel versions with only minimal code adaptions.

Based on our implementation, we now describe how we evaluated the stealthiness of *Styx*. We searched for suspicious traces that are left over in both the file system and physical memory after the installation had completed.

During its installation, *Styx* deletes all of its corresponding files. Therefore, it overwrites their data blocks on disk, before it removes corresponding inode references from the file system. That way, its files cannot be recovered even with specialized forensic software. Since both LKMs were unloaded from the system, all corresponding file system entries were removed, too. These entries include `/dev/styx`, `/dev/eraser`, as well as `/sys/module/styx` and `/sys/module/eraser`. In addition, unloading the modules also unlinks them from the kernel's module list or similar management data structures. Therefore, tools like `lsmod` cannot display perfidious information anymore.

To evaluate traces in physical memory, we used the memory acquisition tool *pmem*, which is part of the forensic tool suite *Rekall* (Cohen, 2014). Pmem is an in-guest memory acquisition tool that comprises both a user mode and a kernel mode component. It accesses physical memory by remapping every single frame to a selected PTE. This allows pmem to acquire the entire physical memory without any suspicious kernel support. Otherwise, conventional kernel rootkits would have the chance to tamper with the acquired data. After loading pmem's LKM, we can instruct it to dump the content of selected physical memory ranges. We specify the Rootkit LKM's original segment ranges in order to verify they were indeed overwritten by the Eraser LKM. On our test system, every single byte of these regions was successfully deleted.

The main focus, however, was to evaluate the content of hidden memory, which, as we have discussed before, now contains all of *Styx*'s code and data. To achieve respectable evaluation results, we enhanced pmem's kernel mode component to include hidden memory into its memory dumps by implementing the same algorithm we used for *Styx*. Our test system included two physical hidden memory ranges: one from `0xcd000` to `0xedfff` and the other from `0xbffe2000` to `0xbfffffff`. To be able to dump its contents, we need to specify both their physical base addresses (`-b` switch) as well as their amount of bytes (`-s` switch). Therefore, the first range has a size of `0x1e000` bytes, whereas the second amounts up to `0x21000` bytes. In addition, we need to declare the name of the file, that should be used to write the memory dump to. This can be done with the `-f` option. As you can see in Listing 2, the acquired data of both ranges did not contain any meaningful data. This is due to the fact that pmem's read accesses were redirected to the rootkit's guard page via EPT. As mentioned in the last section, this guard page was initialized with zeros what is conform to the results of our dumps. We also evaluated our rootkit's protection features by overwriting these two hidden memory ranges. As we suspected, Styx still remained fully functional afterwards.

Listing 2. Hidden memory dumps.

```
$ ./pmem -b 0xbffe2000 -s 0x1e000 -f out.raw
[pmem] Dump physical memory from 0xbffe2000 to 0xbfffffff
    (size: 0x1e000).
[pmem] Copy physical memory from 0xbffe2000 to 0xbfffffff .

99%       0xbffe2000 .
100% Done.
$ hexdump -C out.raw
0xbffe2000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
0xbfffffff
$ ./pmem -b 0xcd000 -s 0x21000 -f out.raw
[pmem] Dump physical memory from 0xcd000 to 0xedfff
    (size: 0x21000).
[pmem] Copy physical memory from 0xcd000 to 0xedfff .

86%          0xcd000 .
100% Done.
$ hexdump -C out.raw
0xcd000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
0xedfff
```

We are not aware of any other software-based tool that is capable to acquire hidden memory. We therefore conclude, that we successfully achieved to hide *Styx*'s entire memory footprint from state-of-the-art in-guest memory acquisition tools which do not rely on DMA-based acquisition techniques or make use of Intel's SMM.

Conclusions

We presented *Styx*, a proof-of-concept system to show that even robust memory acquisition (using the example of pmem) can be subverted by a combination of run-time virtualization and use of hidden memory. We described the installation process, which migrates a running Linux system into a hardware-supported virtual machine. We pointed out how to retrieve information about the segments of a LKM and presented an algorithm to enumerate hidden memory. We also pointed out how to remap our rootkit from kernel space to hidden memory during run time. In order to hide its own memory footprint, we protect these regions with the help of Intel's EPT by simulating device memory. Since the installation process leaves various traces in the memory of the guest, we developed a second LKM, which is responsible for their erasure.

We furthermore enhanced Rekall's pmem, a state of the art memory acquisition tool, to search for remaining traces in the physical memory. We then evaluated the results to make a statement on *Styx*'s ability to hide from modern memory forensics. As it turned out, the tool was not able to find any traces, which could be used to identify or analyze our prototype.

Our implementation currently does not support protection against DMA accesses. Therefore, a tool, that uses DMA to acquire physical memory, would be able to detect our rootkit. This, however, appears only as a minor pitfall, as DMA accesses can be virtualized with Intel's VT-d extension. The feature enables a hypervisor to restrict DMA devices to access certain parts of memory. *Styx* could thus easily block device accesses to hidden memory.

Styx can be used both for malicious and benign purposes. It thus can be used by malware to hide its payloads and serve forensics software for out-of-band guest analysis. Although we have shown our techniques to be effective against today's memory forensics, we doubt to see this kind of rootkit spreading broadly in the wild. This is because the amount of hidden memory depends very much on the configuration of a target system. These techniques are more probable to show up in software used for selected systems where their specific configuration is well known. Our work therefore shows that alternative methods to software-based memory acquisition will remain to be relevant in such situations.

Acknowledgments

We would like to thank Johannes Stüttgen his great prior work on hidden memory that was the inspiration for this work. We are grateful for his continued interest and his (technical and moral) support on this topic. We also wish to thank the anonymous reviewers for their helpful comments.

References

Athreya, M.B., 2010. Subverting Linux On-the-fly Using Hardware Virtualization Technology (Ph.D. thesis). Georgia Institute of Technology.
Cohen, M., 2014. Rekall memory forensic framework. http://www.rekallforensic.com.
Garfinkel, T., Adams, K., Warfield, A., Franklin, J., 2007. Compatibility is not transparency: Vmm detection myths and realities. In: Hunt, G.C. (Ed.), Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems, May 7-9, 2005. California, USA, USENIX Association, San Diego.
Intel, 2017. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, December.

King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R., 2006. Subvirt: implementing malware with virtual machines. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, pp. 314–327.

Martignoni, L., Fattori, A., Paleari, R., Cavallaro, L., 2010. Live and Trustworthy Forensic Analysis of Commodity Production Systems. In: RAID. Springer, pp. 297–316.

O'Murchu, L., Gutierrez, F.P., 2015. The Evolution of the Fileless Click-fraud Malare Poweliks, Symantec Security Response Version 1.0. Symantec Corp.

PCI Local Bus Specification, Production Version, Revision 3.0, 2004.

Ptacek, T., Lawson, N., Ferrie, P., 2007. Don't tell Joanna, the virtualized rootkit is dead. Black Hat USA.

Reina, A., Fattori, A., Pagani, F., Cavallaro, L., Bruschi, D., 2012. When hardware meets software: a bulletproof solution to forensic memory acquisition. In: Proceedings of the 28th Annual Computer Security Applications Conference. ACM, pp. 79–88.

Rutkowska, J., 2008. Security challenges in virtualized environments. In: RSA Conference.

Rutkowska, J., 2006. Subverting VistaTM kernel for fun and profit. Black Hat Briefings. USA.

Rutkowska, J., Tereshkin, A., 2007. IsGameOver( ) anyone. Black Hat USA.

Rutkowska, J., Tereshkin, A., 2008. Bluepilling the Xen hypervisor. Black Hat USA.

Stüttgen, J., Cohen, M., 2013. Anti-forensic resilient memory acquisition. Digit. Invest. 10, S105–S115.

Stüttgen, J., Cohen, M., 2014. Robust linux memory acquisition with minimal target impact. Digit. Invest. 11, S112–S119.

Vömel, S., Freiling, F.C., 2011. A survey of main memory acquisition and analysis techniques for the windows operating system. Digit. Invest. 8 (1), 3–22.

Vömel, S., Freiling, F.C., 2012. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. Digit. Invest. 9 (2), 125–137.

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

Yu, M., Qi, Z., Lin, Q., Zhong, X., Li, B., Guan, H., 2012. Vis: virtualization enhanced live forensics acquisition for native system. Digit. Invest. 9 (1), 22–33.

Zovi, D.A.D., August 2006. Hardware virtualization rootkits. Black Hat USA.