U-HIPE: hypervisor-based protection of user-mode processes in Windows

Andrei Luțaș, Adrian Coleșa, Sándor Lukács & Dan Luțaș

Journal of Computer Virology and Hacking Techniques

ISSN 2274-2042

J Comput Virol Hack Tech
DOI 10.1007/s11416-015-0237-z

Springer

Your article is protected by copyright and all rights are held exclusively by Springer-Verlag France. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

J Comput Virol Hack Tech
DOI 10.1007/s11416-015-0237-z

ORIGINAL PAPER

U-HIPE: hypervisor-based protection of user-mode processes in Windows

Andrei Luțaș · Adrian Coleșa · Sándor Lukács · Dan Luțaș

Received: 11 September 2014 / Accepted: 16 January 2015
© Springer-Verlag France 2015

Abstract We propose a method to protect user-processes against malicious software attacks running an introspection and protection tool (U-HIPE) inside a hypervisor. Our solution is based on hardware virtualization support, imposing "no-write" and/or "no-execution" restrictions on different guest virtual machine's (VM) memory pages. Protected components include process' thread stacks, heaps and loadable modules. This way most attempts to execute malicious code in a process are detected and blocked. We propose a method to deal with swappable pages. We inject page-fault exceptions in the guest VM when trying to read swapped-out pages for introspection. We also intercept all swap-in and swap-out events to correctly maintain protection on needed memory pages. We implemented a testing prototype for protecting user-processes in several Microsoft Windows operating systems. Tests we performed proved the effectiveness of our solution against attacks like polymorphic/packed viruses, hook injection and injected code execution. The introduced overhead is acceptable for most applications.

A. Luțaș · S. Lukács · D. Luțaș
Bitdefender, Cuza Voda str., no. 1, 400107 Cluj-Napoca, Romania
e-mail: vlutas@bitdefender.com

S. Lukács
e-mail: slukacs@bitdefender.com

D. Luțaș
e-mail: dlutas@bitdefender.com

A. Coleșa (✉)
Technical University of Cluj-Napoca, Baritiu str., no. 26–28, 400027 Cluj-Napoca, Romania
e-mail: adrian.colesa@cs.utcluj.ro

1 Introduction

The *below-OS* implementation [5,9,17,38] of security solutions was a logical consequence of the large-scale adoption of virtualization used to run more virtual machines (VM) on the same physical system. The security tools in such cases are based on the control capabilities of the virtualization system, usually called *hypervisor* or *VM monitor* (VMM). The main advantages of the below-OS strategy over the inside-OS alternative are: (1) increased security, resulted from the isolation of the security tool from the possible malware in the VM, and (2) simpler and more efficient deployability for a large number of VMs.

On the other hand, because the below-OS security tools use *introspection* [12,21,23] of the monitored VM's memory, they face some challenges, among which possible performance penalties and the *semantic gap* [5,8].

The performance was greatly improved due to the latest advancement in the hardware virtualization support [16]. There were also proposed numerous solutions to reduce the semantic gap. The overall purpose was to do it having no knowledge about the guest OS. Still, the resulting solutions could be divided in two classes: (1) (more or less) guest *OS-dependent* [8,11,13,22,24], and (2) totally guest *OS-independent* [18–20,34]. The OS-dependent solutions require knowledge about the internal data structures of the guest OS, which are generally obtained through reverse engineering techniques. So, for the sake of generality an OS-independent solution is desired. Unfortunately, the meaning of the VM contents based only on the virtual hardware information is very limited and also lacks precision. This is why on production systems a pure OS-independent approach is not realistic, if applicable at all.

Therefore, we adopted a combined strategy: OS-dependent method when precision and performance are required,

Published online: 04 February 2015

Author's personal copy

and OS-independent one, otherwise. Even so, our OS-dependent method is *generic* enough, as it uses basic data structures existent in some particular form in most OSes (e.g. Windows, Linux), like the list of processes/threads, allocated heaps, loaded modules etc.

The main contributions of our paper are:

1. *Protection against intrusion attacks on user-mode processes*. Many existent hypervisor-conducted security solutions introspect certain guest OS data structures to either protect them against kernel rootkit attacks or detect already injected malicious code inside the kernel [28,31]. Just few of them [15,20,24,34] focus also on user-mode process protection. Notwithstanding, there are attacks that target only user processes, which cannot be detected by only protecting the OS kernel. For instance, attacks using application exploits to gain access on user processes that manipulate important user data (e.g. the mail client, the Internet browser) do not need at all controlling the OS kernel to be able to steal and manipulate user secret data, like passwords, personally identifiable information (PII), emails, contacts etc. Our proposed solution aims protection against such types of attacks, applying introspection and protection on both the kernel and user memory spaces.
2. *Protection against an extended range of attacks on user-mode processes*. The existent user-mode process security solutions are limited regarding the attack types they could fight against. For instance, [34] only protects against stack smashing attacks, [20] reports the existence of running malicious processes or user-process code, but cannot block the corresponding attacks when they start, and [24] blocks any malicious execution, but requires the user to specify and maintain a database with authorized code. We propose a solution that protects user-mode processes against an extended range of attacks, like polymorphic/packed viruses, hook injection and injected code execution. The protected user-process components are the thread stacks, heaps and the loadable modules.
3. *A generic hardware-based user-mode attack prevention method*. Our security tool (U-HIPE) actively monitors user-processes, preventing the attack occurrences. It is based on hardware virtualization support commonly provided by a large range of modern processors, like Intel, AMD, ARM Cortex-A15.
4. *An OS-independent method to deal with swapped-out pages*. Our security tool transparently manage swappable pages for purposes of introspection and protection maintenance, independent of the fact that they are in memory or swapped out.
5. *Detailed description of an implementation of a testing prototype of U-HIPE in several Microsoft Windows operating systems*. Real-life tests we performed proved the effectiveness of our solution against attacks like polymorphic/packed viruses, hook injection and injected code execution.

Section 2 compares U-HIPE with other solutions. Section 3 describes the environment assumptions we make and the threats our solution could face successfully. Section 4 presents the Intel virtualization support we use. Section 5 presents the user-process protection strategy. Section 6 describes the page-fault injection mechanism used to access swapped-out memory pages. Section 7 details the implementation of U-HIPE on a Windows OS. Section 8 analyzes the security capabilities of U-HIPE. Section 9 illustrates the evaluation results, and Sect. 10 concludes the paper.

2 Related work

Some below-OS security solutions require either changes to the guest OS source code, e.g SymCalls [22], Overshadow [6], InkTag [15], or installing custom drivers in the guest OS, e.g. SecVisor [31] and Lares [27]. Even if such a strategy could lead to a better performance, it is either not applicable for closed-source OSes, or too intrusive, being easily detectable by malware and potentially vulnerable to attacks. We chose the non-intrusive alternative, although it faces the semantic gap problem.

Different by systems like Geiger [19], Antfarm [18], Lycosid [20], Ether [7], Virtuoso [8] that try to reduce the semantic gap by inferring guest OS semantic information relying only on virtualized hardware, we adopted an OS-dependent solution. Even if not so general, we based our protection method on general enough data structures (e.g. process/thread list, loaded modules, heaps) that exist in most OSes (e.g. Windows and Linux) and could easily be located by an introspection engine, making it easily portable. We found our approach more appropriate for production systems, where performance and precision are important requirements. Yet, we also use OS-independent techniques to deal with swappable pages.

Some solutions like NICKLE [28], SecVisor [31], Lares [27], RTKDSM [14] deal with protection of the guest OS' code or data structures, others like VMST [10,11] and Virtuoso [8] introspect guest OS structures, but none could deal with the user-space. This is complementary to our work. Still, we rely on the kernel code being protected. Besides, we also use similar techniques but for protecting user-process structures of interest.

There are some security solutions that explicitly aim protecting the user processes. Systems like SP^3 [39], Overshadow [6] and InkTag [15] protect trusted user-processes by a totally untrusted guest OS. They base their method on encrypting the protected pages of the user-processes and pre-

Author's personal copy

senting the OS with the encrypted version of those pages, while the trusted application with the decrypted version. While an appealing method, it must control the way the guest OS changes the paging structures of protected processes. We consider this to interfere too much with the guest OS' internal policy, leading to unpredictable performance results.

Patagonix [24] intercepts execution attempts to uncover any user-space page containing code. Based on executable file formats they check the detected executable pages against a user-defined database, blocking any attempt to execute unauthorized code. Even if less general, our solution could perform better, since we directly identify from guest OS structures the user-process pages that must be protected. This way we could block directly any execution on certain process area and do not need checking the code against a database, which is difficult to maintain in practice.

In [34] hardware performance counters are used to monitor certain types of instructions. They propose a method to protect the user/kernel stacks against stack smashing attacks. Contrary, we protect other process areas, as well. Besides, their method is more or less about monitoring instructions and could be difficult to adapt for different other attack types.

Very few [21,22,25] of the below-OS systems protecting either the guest OS or user-processes take into account the problem of swappable pages. Systems dealing exclusively with the kernel space consider it to be non-swappable, which is not always the case. Obviously, user-process pages could be often swapped-out. Having to introspect them could be required even for kernel protection systems. Our solution to this problem is totally OS-independent, relying only on the paging structures and mechanisms supported by the Intel architecture.

IntroVirt [21] is similar to our system in the way they deal with user-process introspection. They detect attacks running known-vulnerability predicates inside the guest VM, taking advantage of the guest OS functionality, like address translations and access to swapped-out pages. They place breakpoints on different low-level guest OS functions to detect swap in and out events, like we do. Similar to the way we deal with swappable pages, their solution is OS-dependent, while our is OS-independent.

3 Threat model and assumptions

We make the following *assumptions* regarding the characteristics of the environment our security tool will run into, such that to be effective:

- The processors provides hardware virtualization support (e.g. Intel VT-x, AMD-V, ARM Virtualization Extensions) with the following extensions: second layer address translation (SLAT) (e.g. Intel EPT, AMD RVI), I/O memory management unit (IOMMU) (e.g. Intel VT-d, AMD-Vi, ARM SMMU) and trusted execution (e.g. Intel TXT, ARM TrustZone TEE). SLAT is used to protect memory pages. IOMMU is needed for protection against DMA attacks [37], which can, otherwise, easily bypass EPT protection. Trusted execution support is used to measure the integrity of a launched software.
- The trusted computing base (TCB) is formed by the physical hardware, the firmware, the hypervisor and our security tool.
- The booting process (firmware, hypervisor loader, hypervisor itself, and our security tool) is measured for integrity using the trusted execution technology. This way we are sure the security environment we build is the authentic trusted one. The integrity of the protected VM's OS kernel is also measured at the moment it is launched. This is important for our method, since we suppose we start protecting an uncompromised guest OS.
- Certain guest OS's sections (e.g. kernel's code, kernel driver's code, critical kernel and driver data structures, system call dispatch table) are protected by techniques like those described in [28,31]. Our solution also applies similar techniques to protect the kernel data structures and code it uses. However this is not the topic of this paper so we will not concentrate on it.

The *threat model* we consider consists in:

- The host machine can undergo a secure boot, checking for integrity of the hypervisor.
- During the guest OS initialization phase, kernel security protection could be activated, with no attack risks.
- During the runtime, the guest OS could be attacked by kernel exploits.
- The security attacks we must face are: (1) hook insertion, (2) return oriented programming (ROP) based on heap or a stack overflow and execution of injected code, (3) unpacked code execution.

4 Intel virtualization support

4.1 General overview

Intel VT-x [16] enables a hypervisor to fully emulate the underlying hardware for one or more guest VMs. The hypervisor runs in a new more privileged *VMX root* operation mode, while the guest code runs in a less privileged *VMX non-root* operation mode. VM control structure (VMCS) fields allow the hypervisor to enable *VM exits*, i.e. traps of the CPU from guest to host (hypervisor), to be triggered on certain events happening in the VM, like execution of a privileged

Author's personal copy

instruction. After handling the VM exit, the hypervisor passes the control back to the VM by doing a *VM entry*.

4.2 Memory virtualization

The *Extended Page Tables* (EPT) mechanism was added to assist the memory virtualization. The EPT is configured by the hypervisor and acts as a second layer of address translation. When paging is activated in a non-virtualized environment, the *Memory Management Unit* (MMU) uses paging structures to translate a virtual address to a physical address.

When the EPT are configured by an hypervisor, a second translation phase is added. In the first phase the CPU translates a *guest virtual address* (GVA) into a *guest physical address* (GPA), using paging structures inside the guest OS. In the second phase the CPU translates the resulted GPA into a *host physical address* (HPA), using the EPT.

The EPT also control the access rights on a certain GPA. Three different access types can be configured individually: *read*, *write* and *execute* access. If any of these is denied, the corresponding type of access on that page will trigger a VM exit with the reason "*EPT violation*". The hypervisor will handle this violation: it might emulate or simply skip the faulting instruction. This mechanism sits at the heart of our solution to provide memory protection that is otherwise very difficult to achieve.

The way EPT are used is illustrated in Fig. 1. Details about protection settings will be given below.

4.3 Event injection

Sometimes, certain events need to be injected inside one guest VM from the hypervisor, for various reasons. The most common are the external interrupts. Injection of faults, traps or exceptions is also supported, allowing the hypervisor to simulate events that did not actually take place inside the VM. The event injection mechanism allows the delivery of an event together with an error code, if necessary. The event will be handled by the CPU when the control is passed back to the guest OS. Injection of certain events requires additional settings. For example, in case of a page-fault exception injection, the CR2 register must contain the address the page-fault was "generated" on. This mechanism is also crucial in our U-HIPE, since it allows us to access swapped-out guest virtual pages.

5 User-process protection mechanisms

5.1 Challenges

Whenever a new process is created, U-HIPE should be able to decide, based on user-defined policies, whether to activate protection on it or not. It must then start monitoring the process for thread stacks creation, heaps creation and modules loading, to activate protection on them. In order to do that, U-HIPE must be able to *intercept* the following events:

1. process/thread creation and termination;
2. heap creation and termination;
3. module loading and unloading.

Correspondingly, it must *identify*, in each case, the memory pages that must be protected.

There is, though, a significant challenge in achieving this for user-space memory, because it is normally pageable. This means that at any given moment, a virtual page needed for introspection and protection may be swapped out, i.e. non-present in memory. This poses two new issues:

1. How to access a page if it is not present inside the physical memory?
2. How to achieve protection on pages that are being swapped in and out of the physical memory due to the guest OS page replacement policy?

5.2 Events interception and memory protection

Event detection is based on placing hooks on certain kernel functions or write-protecting certain kernel or user-space structures. Function hooking is not the scope of our paper. It is described in papers like [27,36].

The most important events U-HIPE must intercept are process creation and termination. The protection should be activated before the process is visible and attackable by malicious processes. When a process terminates, U-HIPE must cleanly remove its protection. These actions could done by either: (1) hooking the kernel functions that activate/deactivate a process (we will present in Sect. 7 such functions for Windows OS), or (2) intercepting memory events that are specific to the process creation, such as changes to the guest OS' process list.

The later technique could be done activating "*read-only*" hardware protection on the corresponding pages. Whenever a new node will be added or an old node removed, writes will take place inside this list, and the corresponding memory violation will indicate U-HIPE that a new process has been created or a process has been terminated.

Similar techniques are used to intercept thread creation and termination for a protected process. At thread creation, its user-mode stack must be identified, usually using kernel or user specific data structures. The type of protection U-HIPE places on the stack pages is "*no-execution*". When a thread terminates, the protection of its stack is removed.

Inside the user realm, we need to intercept heap creation and termination and user mode modules loading and unload-

Author's personal copy

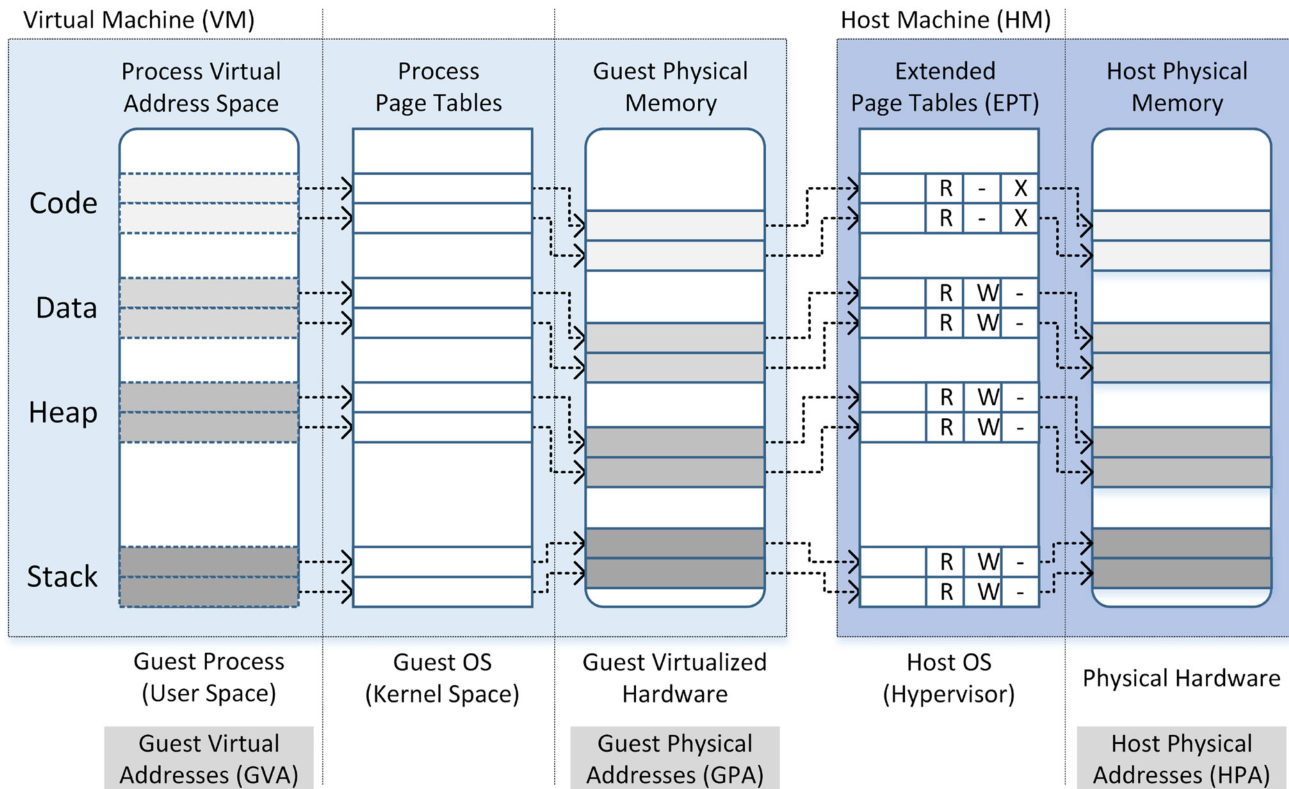

Fig. 1 EPT protection settings for different pages in the user process. Code pages refer to both statically loaded code and dynamically loaded libraries. It could be noted that the code pages are restricted for being changed (i.e. "no-write" protection), while other pages are restricted for execution (i.e. "no-execution" protection)

ing. U-HIPE intercepts these events by placing "*read-only*" protection on the structures describing the process: the list of loaded modules and the list of heaps. If necessary, interception of other events, such as user-mode library APIs, could also be done.

All of these events will eventually be used in order to activate: (1) "no-execution" protection on the newly created heaps, and (2) "no-write" protection on the newly loaded shared libraries.

The EPT protection on the different page types is illustrated in Fig. 1. In order to keep the picture clear, we did not figure the "read-only" restrictions on the EPT entries that map the VM pages containing the process' page tables (see below in relation to the technique we use to keep page protection on swappable pages). Though, they are similar to the illustrated EPT entries and could be easily imagined taking into account that page tables themselves are also stored in guest physical memory frames, which need to be mapped onto the host physical memory through EPT entries.

5.3 Memory protection maintenance

Inside EPT, U-HIPE manipulates only GPAs and HPAs. Although, protection must be applied on GVAs. Therefore, if U-HIPE must protect a guest virtual page, that page's GVA must be translated to its corresponding GPA using the guest OS' paging structures. The found GPA can then be protected in its corresponding entry in EPT.

Some problems could arise when the guest virtual page (at GVA) is swappable, because the guest physical page (at GPA) it is mapped on could be changed due to the swapping operations in guest OS. Let us assume the followings: (1) U-HIPE must protect a virtual page, whose address is GVA; (2) GVA translates to GPA, via the guest OS' page tables; (3) U-HIPE protects GPA using the EPT. We could have the following problematic situations:

1. *False protection*. The guest OS swaps out GVA and replaces it with GVA^*, which is mapped now on GPA. In this case, U-HIPE could still protect GPA, even if the target virtual page's GVA does not point to it any more. Actually, the protected page would be in the new configuration the arbitrary GVA^*.
2. *Lost protection*. The guest OS swaps out GVA and maps nothing else to GPA. Later, it swaps back in GVA, but maps it to GPA^*. In this case, U-HIPE could lose the protection on GVA, since it could still protect GPA instead of GPA^*.

Author's personal copy

3. *Wrong protection*. The guest OS remaps GVA from GPA to GPA^*, without actually swapping the virtual page. This case is a sort of a combination of the previous two ones: as long as GPA is not allocated to another GVA^*, U-HIPE would suffer by lost protection, whereas as long as the GPA would be allocated to another GVA^*, U-HIPE would also suffer by false protection.

Figure 2 illustrates the three cases of protection problems mentioned above: Fig. 2a illustrates the normal protection mechanism, Fig. 2b the lost protection situation, and Fig. 2c both lost and wrong protection.

In order to avoid such problems, U-HIPE applies EPT "no-write" protection on the guest OS's paging structures themselves. This way, every time a certain guest virtual page is being swapped in or out and its corresponding page-table entry (PTE) must be changed, EPT violations will be triggered, which U-HIPE intercepts and handles to adapt its protection correspondingly. It decodes the faulty instruction, extracts the new value of the PTE that is tried to be written and compares it with the current (old) value, to infer if the corresponding page is:

1. *swapped-in*: the present bit has a transition from 0 to 1;
2. *swapped-out*: the present bit has a transition from 1 to 0;
3. *remapped*: no transition of the present bit, but a change of the GPA.

Other times, the guest OS might only try modifying other bits in a PTE (e.g. read/write bits or OS specific bits). In such cases, the corresponding write operation will be simply ignored by U-HIPE, since there is no present bit transition and no GPA change.

There are still some other problems that must be taken into account:

1. *Non-present page tables*. The guest OS page table containing the PTE of a protected page could be freed and made non-present itself whenever there is no page mapped using it. In such a case, U-HIPE must place EPT "no-write" protection on the upper level paging structure (page directory, in Intel terminology). This method must be extended to the full hierarchy of the translation paging structures.
2. *Shared Pages*. This is the case, for instance, of shared libraries, which are physically allocated one time, but mapped inside the virtual address space of every process using them. When such libraries contain writable pages, those pages are dealt with by the guest OS using the "copy-on-write" strategy. Every write on a "copy-on-write" page, triggers a page-fault exception inside the guest OS, letting it allocate a private copy of that page for the writing process and change correspondingly the process' page table. Based on the "no-write" protection placed on the guest OS page tables, U-HIPE intercepts changes implied by the copy-on-write mechanism and transparently adapts its protection like in the *page remapped* case described above.

5.4 Multi-core support

We also considered the cases of Symmetric Multi-Processor (SMP) systems. Actually, the only thing that is influenced by running our security component on such a system is the way the EPT structures are managed. There are two strategies:

- *Centralized EPT*: there is a single set of EPT, which are shared among all VMCSs assigned to the virtual CPUs given to a VM. This way, protection settings are consistent among all virtual CPUs. Though, in order this method to be effective, concurrent changes on the EPT entries coming from the multiple CPUs must be synchronized. This could be easily done in practice using a sort of spin-lock.
- *Distributed EPT*: there is a different set of EPT for each different VMCS (implicitly, for each virtual CPU). The difficulty with this method is that all EPTs must consistently reflect the same memory contents and page protection settings, which also requires a sort of synchronization between the virtual CPUs.

We decided for the centralized strategy, while it is simpler to implement and also benefits from memory saving. A sort of CPUs consistency maintenance is still needed in this case, too. This is because each CPU has its own TLBs. When protection settings inside an EPT entry are changed by one processor, it invalidates its corresponding TLB entry. That processor sends (after applying the changes) the other processors an IPI (i.e. inter-processor interrupt) message to also make them aware of the changes. In response, they invalidate the corresponding entries in their own TLBs. Such invalidation could be made, for instance, on x86 architecture by using the INVEPT CPU instruction.

6 Introspection of swapped-out user-process pages

We saw in Sect. 5 that U-HIPE needs to introspect and protect pages inside both the guest kernel and user spaces. There would be, though, a problem, if U-HIPE tries to introspect a swapped-out page.

The solution we propose is to leverage the existent *page-fault* (PF) handling mechanism of the guest OS. Any time U-HIPE needs accessing a swapped-out guest page, it will *inject a page-fault* in the guest VM. This way, U-HIPE makes the guest OS thinks that some process inside the guest VM

Author's personal copy

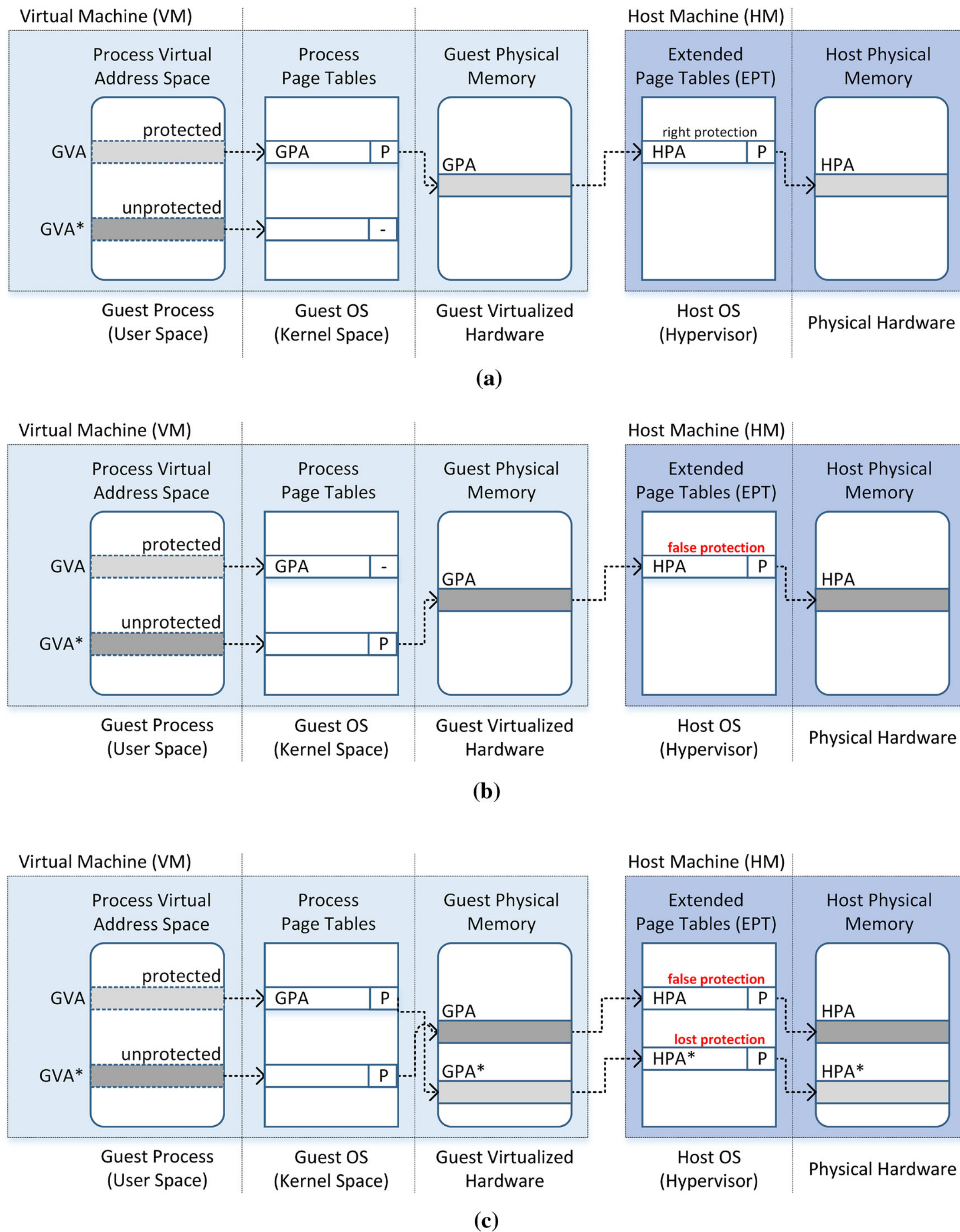

Fig. 2 Page protection anomalies. **a** The state of entries in process page tables (controlled by the guest OS) and EPT (controlled by the hypervisor) to protect a certain virtual page at the GVA guest virtual address. The protected GVA page is loaded (present) in the physical memory. **b** *False protection* due to wrongly keeping protection settings in the EPT entry, even if the corresponding guest/host physical memory frame was loaded with the contents of another process virtual page, due to the page replacement policy in the guest OS, i.e. the protected GVA page was swapped out, while an unprotected GVA* page was swapped in. **c** *Lost protection* happens when a protected process virtual page is swapped out from a physical memory frame and then swapped back in another frame, due to page replacement policies in guest OS, yet the EPT entry keeps protection on the first frame. *Wrong protection* is like false protection, but not having the protected virtual page being swapped out and in, only moved (and consequently, remapped) in the physical memory.

Author's personal copy

tried accessing a swapped-out page, and swaps it back into the physical memory, using the normal internal swapping mechanism.

6.1 Page-fault exception

The PF exception is a hardware exception triggered whenever the MMU cannot access a virtual address. A PF is generated when: an instruction accesses a non-present page, an user-space instruction accesses kernel memory or write inside a read-only page, CPU tries to fetch instructions from a non-executable page, etc. OS handles the exception, deciding what to do. If the page was swapped-out, it will swap it back in memory.

6.2 Page-fault injection and handling

When a page-fault (PF) is generated, the CPU communicates the OS the reason for the page-fault and the faulty address. This information is provided on Intel processors via the *page-fault error code* (PFEC) for the fault reason and the CR2 register for the faulty address. U-HIPE loads the corresponding fields in the VMCS with expected values (e.g. CR2 with the GVA of the needed swapped-out page), when injecting a PF.

We noted that most OSes do not check the faulty instruction. Because of this, we do not have to worry at all about the instruction the IP register points to when we inject the page-fault.

The next thing U-HIPE needs to solve is to determine the moment the swapped-out page is back in memory. For this, it must intercept the corresponding *swap-in event*. The way to do this is based on protecting for "no-write" the guest OS page table and was described in Sect. 5.3.

PF injection could also face the situation when a page table itself is not present in memory. In such a case, intercepting the needed swap-in event functions in two steps: intercepts the swap-in of the page table containing the PTE of the needed page, then intercept the swap-in of the page itself. This multi-step interception is based on "no-write" protection of the higher-level paging structures and was also described in Sect. 5.3.

When the waited swap-in event is intercepted, U-HIPE can read the needed page from the GPA it will be mapped. When this actually happens depends on the OS and its scheduler. Usually, the injected PF is processed immediately, but if a clock or inter-processor interrupt is delivered to the handling CPU, the actual processing of the PF will be postponed.

Another aspect U-HIPE must deal with is the PF *injection moment*. For a user-space page, a PF should normally be injected when the IP register points inside a user-space code segment. For most user-space pages U-HIPE must protect, it intercepts user-mode events (e.g. heap creation, module loading), when the IP is in user-space, so the injection could be done immediately. Other intercepted events (e.g. thread creation) correspond, however, to kernel-mode. In this case U-HIPE protects the PTE of the needed swapped-out page and postpones the PF injection until the next user-mode event interception. If in the meantime the guest OS swaps in the needed page, U-HIPE will intercept the corresponding swap-in event and introspect immediately the page, needing no more a PF injection.

A PF injection could also be performed when the IP points inside the guest kernel, for either a kernel or a user page. However, several other validations to avoid possible race conditions might be needed. They are though very OS-dependent and we do not deal with them here.

7 U-HIPE implementation on Windows

We implemented and tested a prototype of U-HIPE on different MS Windows OSes for both x86 and x64 processors, starting with Windows Vista and up to the latest Windows 8.1. Certain information is, however, version dependent, such as various offsets inside the kernel data structures we used, mainly because these structures are opaque and not officially documented. Therefore, the current description of the implementation will be strongly dependent on this type of OS. The steps described in Sect. 5 will be detailed in this section specific to Windows OSes.

Another important aspect that must be mentioned is the fact that while we access user-mode memory, we never do any assumption with regard to the accessed memory: if the required memory is present, we will simply access it, but if it is not, we will use the mechanisms described in Sect. 6 in order to access swapped-out memory. Although most of the times all these structures will be mapped inside the physical memory, sometimes, especially when the memory pressure is high, these structures may be swapped out of the physical memory.

7.1 Intercepting process creation and termination

Process creation is intercepted using the internal kernel function `PspInsertProcess()` [30], which basically inserts a newly created process inside the system' process list. Process termination is intercepted using the internal kernel function `PspCleanupProcessAddressSpace()`. Both functions are identified using binary patterns matching on the functions' contents (signature).

Another method would be to write-protect pages containing nodes of the list of all currently active processes, pointed to by `PsActiveProcessHead`. By intercepting all write operations performed on this list, U-HIPE basically intercepts process creation and termination events.

Author's personal copy

The main drawback of the first method is the fact that function signatures must be extracted for each supported OS, and the main drawback of the second method is the significant performance impact, since the virtual memory pages that contain the list nodes could contain various other structures that may be written very often. The main advantage of the first method is its performance, while the main advantage of the second one is its simplicity.

In our implementation, we opted for the first approach, which offers the best performance.

The structure that is used by U-HIPE is the `EPROCESS` structure. This structure describes the process at the kernel level. Inside this structure we access two important fields: the `DirectoryTableBase` field inside the `Pcb` sub-field of the `EPROCESS` structure, which is the PDBR (Page Directory Base Register, i.e. CR3) of the newly created process, and the `Peb` field, which will point to the user-mode `PEB` (Process Environment Block) structure that identifies the user-mode part of the process. If the process is a 32-bit process started on a x64 version of Windows, a new field will be used: `Wow64Process`, which contains a pointer to the `PEB` structure that identifies the 32-bit subsystem of the newly created process, while the initial `Peb` field contains a pointer to the 64-bit `PEB` of the process.

7.2 Intercepting thread creation and termination

After a protected process was started and the initial protection activated on it, thread creation is intercepted. U-HIPE intercepts the function `PspInsertThread()` that inserts a newly created thread inside the list of active threads, function `PspExitThread()` for the case a thread terminates itself, and function `PspTerminateThreadByPointer()` for the case a thread is terminated forcefully by another thread.

Alternatively, we could intercept the *ThreadListHead* field inside the EPROCESS structure of the associated process, which contains the list of all the process' threads. The advantages and disadvantages of these methods are identical with the ones described for the process creation and termination interception.

The used structure was the `ETHREAD` structure, which describes a thread. The most important field used from this structure is `Teb` inside the `Tcb` subfield of the `ETHREAD` structure, which points inside user-mode to a `TIB` (Thread Information Block) structure, which contains the base address of the threads stack: this base address will be used in order to activate stack "no-execution" protection. Once a thread has terminated, the protection on the underlying stack will be removed.

7.3 Intercepting heap creation and termination

Inside the `TEB` structure, the field `ProcessHeaps` points to an array containing the addresses of each process heap. Field `MaximumNumberOfHeaps` identifies the maximum number of heaps that can be stored inside the `ProcessHeaps` array and `NumberOfHeaps` the current number of heaps. A write interception inside the `ProcessHeaps` enables us to identify heap creation and termination. When a heap is created, "no-execution" protection will be activated on it. When a heap is destroyed, the protection will be terminated. The size of the heap will be determined from the heap descriptor itself. In order to detect when the process has created more than its maximum number of heaps, we intercept the `ProcessHeaps` field inside the `PEB`. When the OS writes this field, we will simply move interception on the new array, which will usually be larger than the previous one. This is basically a "realloc" operation for the heaps array.

7.4 Intercepting modules loading

Interception of libraries loading is also based on `PEB`. Field `Ldr` points to a `_PEB_LDR_DATA` structure, which contains a linked list of user-mode libraries that are currently loaded inside the process. In order to intercept module loading, U-HIPE protection as "no-write" the loaded modules list. The mechanism is identical with the interception of the processes or threads lists described earlier.

Whenever a new module is loaded, its associated structure `LDR_MODULE` is parsed. This structure contains all the information about the given module: the path, the base address and the size. Based on the path, U-HIPE decides whether to activate hook protection on the given module or not. The hook protection will apply to every non-writable section of the module, including the *Import Address Table* (IAT) and the *Export Address Table* (EAT). Special treatment is provided for the main module, which is the first module that gets loaded (before Windows 8) or the second one, after `ntdll` (starting with Windows 8). It is protected against unpacking code, in order to detect possible infections with file-infectors.

7.5 Implementation and debugging details

Both U-HIPE and our hypervisor have been written in C. The code size of U-HIPE is roughly 100K LOC, but also including features not described here as being outside the scope of this paper.

The debugging was performed using the serial port: the hypervisor was designed to both send log messages and accept control commands on the serial port. Therefore, both logging facilities and command interpretation were available inside both U-HIPE and the hypervisor at any moment. The guest OS has been debugged using the native Windows

Author's personal copy

debugger *WinDbg*, via a IEEE 1394 (Firewire) connection. This was needed in order to check and analyze certain guest OS's structures and functions used by U-HIPE. Examples include the hooked internal functions and the internal OS structures, such as the EPROCESS, ETHREAD, TEB, PEB and so on.

7.6 Avoiding Windows' internal protection

It is known that the 64-bit editions of MS Windows have an internal kernel protection mechanism against kernel function hooking. The protection mechanism is called *Kernel Patch Protection (KPP)* or *PatchGuard* [30]. Normally, PatchGuard would be able to detect the fact that our security solution would have changed (i.e. hook) the addresses of some kernel functions, like the ones for process/thread creation.

We developed two methods to avoid such a protection:

1. Apply the hooking before the PatchGuard observes the initial kernel function addresses. This way PachGuard takes the U-HIPE's hooked functions as the original Windows' ones. The difficulty of this method is the way the appropriate hooking moment is determined. In the Windows versions we implemented U-HIPE for, we found a moment during the kernel initialization before which PatchGuard is not yet active, particularly when MSRs such as the syscall MSR is being initialized. The problem of this solution is that in future Windows versions PatchGuard may initialize earlier and thus render our method useless.
2. Apply the hooking mechanism any time, but restrict the access to the pages containing the changed function addresses (e.g. configure "'no-read" access in the corresponding page table entries in EPT), such that any attempt to read the hooked function addresses will trigger an access violation exception in the hypervisor, giving it the possibility to return the original content as the result of the read operations. Otherwise, code execution inside these pages will be carried out normally, without faults.

The main advantage of the first technique is its performance. On the other hand, it is not totally transparent, letting the PatchGuard see other function addresses than the original ones. On the contrary, the second method provides perfect transparency, but at the cost of a performance penalty, while each read access to the page containing the function addresses will generate an access violation exception. Fortunately, usually only PatchGuard reads these pages and it does so only rarely (once every several minutes or so), such that the performance penalty is not a real problem in practice.

Regarding the other event interception method we used, i.e. setting in EPT write restrictions on the pages containing data structures supposed to be changed in relation to the monitored events, it could not be even detected from inside the guest OS, so the interception mechanism is totally transparent for the PatchGuard in this case. Actually, this technique is independent on the guest OS, so it has no particularities for Windows.

8 Security analysis

8.1 Security properties

U-HIPE is immune to direct attacks from the guest VM due to its isolation imposed by the hardware. Its memory cannot be accessed, since the CPU forbids this due to the SLAT mechanism. DMA attacks can also be avoided if the hypervisor uses an IOMMU. U-HIPE cannot also be disabled or malfunctioned by malware, since we assumed that the boot process is measured using trusted execution technology.

Though, since U-HIPE depends on information extracted from the guest, if it would contain vulnerabilities, it could be exploited by an attacker capable of providing it specially crafted data able to trigger the vulnerability. However, due to its small size and simplicity it could tested and verified more easier than a normal OS.

Another way of attacking U-HIPE is the denial of service (DoS), but this would be of little use for the attacker, since a DoS attack on U-HIPE would result in a DoS on the guest OS itself, and thus rendering the malware inert.

The events the U-HIPE protection is based on will always be triggered correctly (e.g. an EPT violation) because the attacker cannot reconfigure the hardware memory protection imposed by U-HIPE. Thus, as long as a process exists, all its associated structures are protected. The only apparently possible problem would be an attack on a protected process' memory structures, launched from within a compromised guest OS kernel. Since the kernel is normally more privileged than the user mode, a kernel rootkit could maliciously change and manipulate the process's page tables to bypass the process' protection. However, even this type of attack is limited, since the process' page tables are protected and any page table change is intercepted by U-HIPE. Thus simply remapping or reallocating process' memory would be of no use for the attacker, as the protection would simply be moved to the new pages.

Regarding the page-fault injection, it could be a security issue if an attacker could somehow gain control of the guest OS's PF handler functionality. This way, the attacker could provide U-HIPE with fake GVA and false information, and thus avoid protection. However, as previously mentioned in Sect. 3, we assume that the kernel code is also secured, so we consider the guest's PF handler to be trusted.

Author's personal copy

8.2 Prevented attacks

Polymorphic/Packed viruses [33] infecting a user-process executes decoding/unpacking code that writes sequences of bytes, i.e. decrypted/unpacked code, into process' own memory and tries to execute them later, an anomaly described as "*execute-after-write*". U-HIPE could block such attacks by applying "no-execution" protection on previously written pages.

Hook attacks in a user-process try to redirect functions in the loaded libraries by changing IAT/EAT entries or the function starting code.

Each process contains a main-module, which is the actual application's code that gets executed, and a number of shared dynamic libraries. Some of these libraries contain OS code, including APIs that will get called by the main module. If a malware gets access to a certain process, it can interfere with the normal execution by placing hooks on certain functions inside these libraries. Types of hooking include function-table hooking (modifications inside IAT of a certain module or inside the EAT of the attacked library) and in-line function hooking, that involves replacing the first few bytes of a function with a branch instruction to a piece of malicious code. In both cases, the introspection engine will protect as "no-write" (i.e. "read-only") the virtual memory pages that contain the IAT, the EAT and the code and write attempts inside these pages will be forbidden. Basically, whenever someone tries to hook a function (either via IAT/EAT or in-line), an EPT violation will be triggered. U-HIPE will analyze the attempt, and if it decides it is malicious, it will block it by skipping the faulting instruction and thus blocking any modification to the contents of the protected memory page.

U-HIPE can also block attacks that exploit vulnerabilities (like buffer overflow) allowing *malicious code* to be *injected* and then *executed* from the stack or heaps of the vulnerable process. U-HIPE protection for this type of attacks is based on the "no-execution" restriction on the process' heaps and stacks.

Although the heaps and the stacks of a process (also the written code pages mentioned above) should also be flagged by the guest OS as being "non-executable", a malicious payload can easily bypass this protection [29] by removing the "non-execute" flag on the subsequent heap or stack. Though, if the "no-execution" protection is activated and controlled from the hypervisor, it cannot be removed, and the attacks could be blocked.

Unfortunately, as it is known, the "no-execution" protection cannot fight against return-oriented-programming (ROP) attacks, which are not based on malware-injected code, but on small code sequences found in standard loaded system libraries. Such attacks, which do not need the heap or the stack hosting any injected code, cannot be detected by U-HIPE.

9 Tests and results

We used a system with Intel Core i7-2600 CPU running at 3,400 MHz, 8 logical threads (4 cores with HT) and 16 GB of DDR3, running Windows 7×64 with SP1. Tests were performed on both real and synthetic user applications. Same tests were run more times with and without U-HIPE activated and the average results calculated.

9.1 Performance

The performance impact introduced by U-HIPE is given by both the virtualization mechanisms it uses and the actual processing it does.

9.1.1 Benching application startup time

The startup time penalty for a protected process varies with the number of objects that process creates during its initialization and correspondingly, U-HIPE needs to introspect and protect. A process that creates many heaps and threads will start slightly slower than another one creating not as many. PassMark AppTimer [32] was used to measure the penalty induced by U-HIPE on different applications startup time.

Figure 3 and Table 1 illustrate the results for several applications run with their default configuration. For some applications, e.g. Chrome, Opera, and Internet Explorer, the performance impact is higher (startup time increased by about an order of magnitude, compared to the "no-protection" case), because they spawn other processes, which being also protected by U-HIPE increases the performance penalty. Simple processes, such as the Calculator, have a much smaller performance impact, i.e. about 22 %. We also measured the internal processing overhead induced by U-HIPE to introspect and protect user-process structures for each application. This is illustrated by the third bar in Fig. 3 and last column of Table 1. It is much lower than the overall overhead detected from the user space in guest VM. This is because U-HIPE was implemented inside an experimental in-lab hypervisor. We admit the hypervisor overhead is implied by our protection needs, still we estimate that further optimizations of the hypervisor would significantly decrease the overall performance penalty.

We also noted that the performance overhead increases linearly with the number of objects (stacks, heaps, modules) U-HIPE must protect for a process. In real applications the number of such objects is about few tens during the entire lifetime of a process and with just few of them being created/terminated at one moment. Thus the overall impact is relatively smaller in practical cases. Moreover, this impact is mostly the startup penalty. This is revealed by Fig. 4, which illustrates a normal usage of a custom application that iterated through all the files of a hard disk, read their entire contents

Author's personal copy

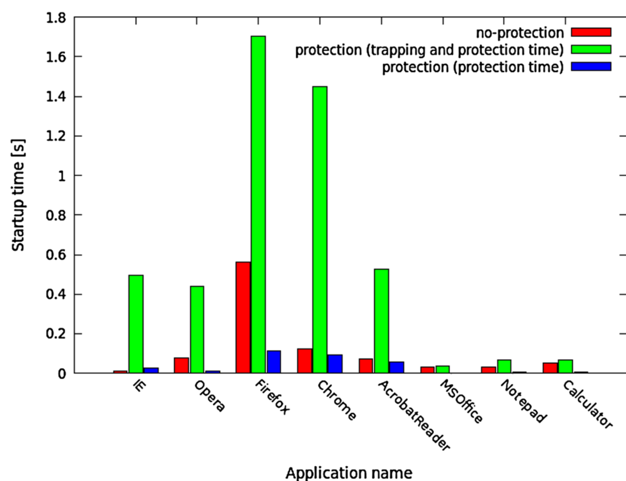

Fig. 3 Cumulated overhead over the start-up time of different processes

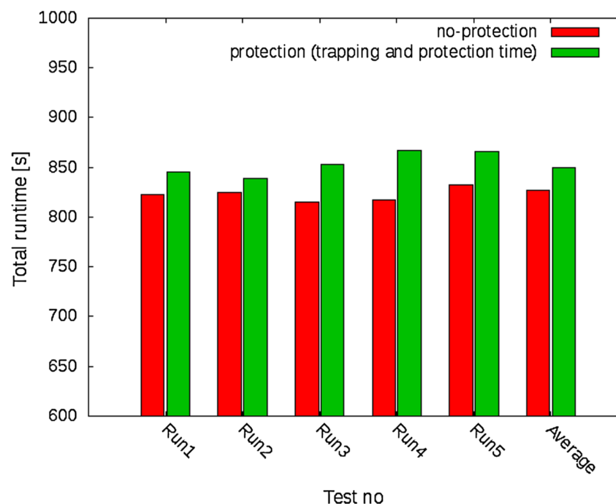

Fig. 4 Cumulated overhead over the entire runtime of a process

in memory and encrypted them. The tests revealed an overall impact of less than 4 %.

9.1.2 Page-fault injection penalty

The page-fault injection usually had a minimal impact on the page replacement policy of the guest OS. On average, we injected less than ten for the pages containing the needed data structures. Page-fault injection timings vary due to OS thread scheduling and disk load. We ran 1,000 tests measuring the time took from the moment of the page-fault injection and up until the moment the page was swapped in and U-HIPE was notified. We got an average time of approximately 0.0149 ms for one such operation.

Another factor regarding the performance overhead introduced by U-HIPE would be the effect of page-fault injections on the TLB configuration, especially in cases when TLB would be full. In such a situation, injecting a page-fault would lead to a TLB entry replacement, just to make place for the mapping of the page requested by U-HIPE. Even if we did not performed such specific tests (it would have been difficult to create and precisely evaluate in real applications the described scenario), we consider that they were covered by the tests we did. Moreover, as we already mentioned, there was a need for only few tens of page-fault injections during the entire lifetime of any tested process, which surely did not have a significant influence on the overall performance.

On the other hand, there are some aspects that make our hypervisor-based introspection perform even better that a similar tool integrated in the guest OS. One such aspect is related to the way TLBs are used. Basically, there are two types of TLB entries maintained when EPT are used: one for virtual to physical guest memory mappings and one for guest to host physical memory mappings. Running the introspection code outside the VM will not impose extra pressure on the TLB entries mapping guest addresses, as it would an in-guest introspection. The only possible performance penalty would remain that implied by the U-HIPE requests for additional swapped-out pages (i.e. injected page-faults). Though, in such cases the performance penalty incurred by the TLB

Table 1 Measurements of different applications' startup time in cases protection was activated or not

Application	Startup time (no protection) [s]	Startup time (protection) [s]	Introspection processing time [s]	Overall overhead [%]	Introspection overhead [%]
IE	0.0101	0.49515	0.02725	4802.48	269.80
Opera	0.0769	0.43932	0.01124	471.28	14.62
Firefox	0.5604	1.70275	0.11489	203.85	20.50
Chrome	0.12549	1.44739	0.09273	1053.39	73.89
Acrobat Reader	0.07258	0.52359	0.05852	621.40	80.63
MS Office	0.03241	0.03841	0.00331	18.51	10.21
Notepad	0.03347	0.06518	0.00461	94.74	13.77
Calculator	0.05352	0.06532	0.00589	22.05	11.01

Author's personal copy

misses is in practice irrelevant, compared with that implied by the disk operations (i.e. few orders of magnitude smaller).

9.2 Protection

U-HIPE was successfully tested against a variety of malware.

Tests with malicious applications were performed on a freshly installed Windows OS running on a host system that was disconnected from the network. The hypervisor and U-HIPE started before the guest OS and activate protection as soon as the OS starts to initialize. The malware samples were manually run on the target system and alerts generated by U-HIPE were observed.

Examples of malicious actions attempted by the tested malware included *execution of code injected* on the stack or heap of a currently monitored process or inside another processes, *polymorphic code* execution and *hook insertion*. These samples cover a wide class of attacks that U-HIPE prevents.

In some cases, actions performed by rootkits were clearly blocked on the target system, as was the case of Bagle [26] or Agony that were prevented to hide their infected processes and/or files. In some other cases, like the ZBot/Zeus trojan [4], the protection effect was not directly visible because the attempts to inject code inside the protected processes have been completely blocked.

The stack smashing attacks were successfully detected and prevented by running the exploits for some known vulnerabilities [35] in MS Office Word 2007.

The polymorphic code and hook insertion detectors were also tested successfully on viruses that not only infect a target program, but once such a target program gets executed, they also attempt to hook certain APIs inside the host process. Hooking prevention was tested on different types of hooking technique: (1) hooks placed inside system DLLs (usually user32.dll, ws2_32.dll and wininet.dll) as performed by file infectors such as Virtob [3], Sality [2] and by malware in the ZBot family, (2) hooks placed inside the IRP/MJ table of the disk or atapi driver object, performed by TDL rootkit, and (3) in-line hooks placed inside some kernel functions or SSDT, as performed by Bagle or Agony.

Similar tests were performed on the Qhost [1] family and various other password stealer and key-logger families.

10 Conclusions

We presented U-HIPE's strategy for providing user-mode processes protection from below the guest OS and its implementation on MS Windows. Its main contributions are: (1) memory protection types on certain process components; (2) maintenance of memory protection on swappable pages; (3) page-fault injection mechanism to get access to swapped-out pages.

U-HIPE uses an OS-dependent introspection strategy, yet this is realistically applicable on production systems needing precision. The performance impact is relatively high, especially on the startup time, compared with the no-protection case. Though, it is acceptable as a compromise between security and performance for normal user applications (e.g. Internet browsers) usually targeted by malicious attacks. Besides an optimized hypervisor could significantly reduce this impact, as long as internal actions taken by U-HIPE do not account too much on the overall user-space perceived overhead.

Our main future research interests will mainly concentrate on:

1. Proving the support for other operating system families, such as Linux;
2. Improving the performance of U-HIPE, especially for new process and thread launching operation.

Acknowledgments A. Coleșa's work on this paper was supported by the Post-Doctoral Programme POSDRU/159/1.5/S/137516, project co-funded from European Social Fund through the Human Resources Sectorial Operational Program 2007–2013.

References

1. Bitdefender: Qhost Virus Description. http://www.bitdefender.com/free-virus-removal/#Trojan.Qhost.WU. Accessed 28 Jan 2015
2. Bitdefender: Sality Virus Description. http://www.bitdefender.com/free-virus-removal/#Win32.Sality.OG. Accessed 28 Jan 2015
3. Bitdefender: Virtob Virus Description. http://www.bitdefender.com/free-virus-removal/#Win32.Virtob.Gen. Accessed 28 Jan 2015
4. Bitdefender: Zbot Virus Description. http://www.bitdefender.com/free-virus-removal/#Trojan.Spy.ZBot.EHE. Accessed 28 Jan 2015
5. Chen, P.M., Noble, B.D.: When virtual is better than real. In: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, HOTOS '01. IEEE Computer Society, Washington, DC (2001)
6. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. SIGOPS Oper. Syst. Rev. **42**(2), 2–13 (2008)
7. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08, pp. 51–62. ACM, New York (2008)
8. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: narrowing the semantic gap in virtual machine introspection. In: IEEE Symposium on Security and Privacy (SP), pp. 297–312. IEEE, New York (2011)
9. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. 36(SI), 211–224 (2002)

Author's personal copy

10. Fu, Y., Lin, Z.: Space traveling across VM: automatically bridging the semantic gap in virtual machine introspection via online Kernel data redirection. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, pp. 586–600. IEEE Computer Society, Washington, DC (2012)
11. Fu, Y., Lin, Z.: Bridging the semantic gap in virtual machine introspection via online Kernel data redirection. ACM Trans. Inf. Syst. Secur. **16**(2) (2013)
12. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of Network and Distributed Systems Security Symposium, pp. 191–206 (2003)
13. Gavitt, B.D., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: mining memory accesses for introspection. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communication Security, CCS '13, pp. 839–850. ACM, New York (2013)
14. Hizver, J., Chiueh, T.c.: Real-time deep virtual machine introspection and its applications. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, pp. 3–14. ACM, New York (2014)
15. Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E.: InkTag: secure applications on an untrusted operating system. SIGPLAN Not. **48**(4), 265–278 (2013). doi:10.1145/2499368.2451146
16. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. 325462–050US (2014). http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf. Accessed 02 Feb 2015
17. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. ACM Trans. Inf. Syst. Secur. **13**(2) (2010)
18. Jones, S.T., Arpaci Dusseau, A.C., Arpaci Dusseau, R.H.: Antfarm: tracking processes in a virtual machine environment. In: Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference, ATEC '06, pp. 1–14. USENIX Association, Berkeley (2006)
19. Jones, S.T., Arpaci Dusseau, A.C., Arpaci Dusseau, R.H.: Geiger: Monitoring the buffer cache in a virtual machine environment. SIGARCH Comput. Archit. News **34**(5), 14–24 (2006)
20. Jones, S.T., Arpaci Dusseau, A.C., Arpaci Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '08, pp. 91–100. ACM, New York (2008)
21. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting past and present intrusions through vulnerability-specific predicates. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05, pp. 91–104. ACM, New York (2005)
22. Lange, J.R., Dinda, P.: SymCall: Symbiotic virtualization through VMM-to-guest upcalls. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '11, vol. 46, pp. 193–204. ACM, New York (2011)
23. Laureano, M., Maziero, C., Jamhour, E.: Intrusion detection in virtual machine environments. In: Proceedings of the 30th EUROMICRO Conference, EUROMICRO '04, pp. 520–525. IEEE Computer Society, Washington, DC (2004)
24. Litty, L., Cavilla, A.L., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: Proceedings of the 17th Conference on Security Symposium, SS'08, pp. 243–258. USENIX Association, Berkeley (2008)
25. Martignoni, L., Fattori, A., Paleari, R., Cavallaro, L.: Live and trustworthy forensic analysis of commodity production systems. In: Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection, RAID'10, pp. 297–316. Springer, Berlin (2010)
26. Microsoft: Win32/Bagle. http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?name=Win32%2fBagle. Accessed 19 Nov 2014
27. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP '08, pp. 233–247. IEEE Computer Society, Washington, DC (2008)
28. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection, RAID '08, vol. 5230, pp. 1–20. Springer, Berlin (2008)
29. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1) (2012)
30. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows Internals, 6th edn. Microsoft Press, USA (2012)
31. Seshadri, A., Luk, M., Qu, N., Perrig, A.: 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, vol. 41, pp. 335–350. ACM, New York (2007)
32. Software, P.: AppTimer. Application Startup Timer. http://www.passmark.com/products/apptimer.htm. Accessed 28 Mar 2014
33. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Boston (2005)
34. Vogl, S., Eckert, C.: Using hardware performance events for instruction-level monitoring on the x86 architecture. In: Proceedings of the 2012 European Workshop on System Security (EuroSec'12) (2012)
35. Vulnerabilities, C., Exposures: CVE-2010-3333. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3333. Accessed 07 Apr 2014
36. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, pp. 545–554. ACM, New York (2009)
37. Wojtczuk, R., Rutkowska, J.: Following the White Rabbit: Software Attacks Against Intel VT-d Technology (2011). http://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf. Accessed 02 Feb 2015
38. Yan, L.K., Jayachandra, M., Zhang, M., Yin, H.: V2E: combining hardware virtualization and software emulation for transparent and extensible malware analysis. SIGPLAN Not. **47**(7), 227–238 (2012)
39. Yang, J., Shin, K.G.: Using hypervisor to provide data secrecy for user applications on a per-page basis. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '08, pp. 71–80. ACM, New York (2008)