An approach to on the fly activation and deactivation of virtualization-based security systems

D. Yefremov
Moscow State University
Moscow, Russian Federation
Email: yefremov.denis@gmail.com

Scientific Advisor:
P. Iakovenko
Institute for System Programming RAS
Moscow, Russian Federation
Email: yak@ispras.ru

***Abstract*—We report on work in progress which allows on the fly activation and deactivation of hardware virtualization based security systems intended for protecting applications running under the control of untrusted operating system. We present an approach for reserving hardware resources from the operating system for the hypervisor and additional virtual machines that may be required by the security system. We also consider that hypervisor is launched from the untrusted environment that may try to fool the user during the startup and shutdown of the hypervisor.**

***Keywords*-VMM; hypervisor; virtual machine monitor; virtualization; secure activation; on the fly load.**

I. Introduction

Hardware virtualization technology is widely used for consolidating hardware resources, reducing power consumption, simplifying datacenter administration and improving system's reliability. Recently it has got spread into the computer security area. Among the problems that are being solved with it are malware analysis [1], reliable host-based intrusion detection systems [2], securing applications in the untrusted operating systems [3] and others.

Virtualization technology provides the means for executing unmodified operating system (OS) inside a hardware virtual machine (VM) under the control of a relative small system program — virtual machine monitor (hypervisor) [4]. Depending on the requirements hypervisor may assign a device to a particular VM or share a device among virtual machines multiplexing their accesses to it. In the former case VM has exclusive access to the device and no other virtual machines may use it. In the later case the amount of virtual machines that may access the device is not limited but hypervisor must virtualize it providing each VM with the device software model. Such models may constitute significant amount of overall hypervisor code size and require having device driver inside the hypervisor or priviledged control virtual machine [5].

The common property of the existing virtualization-based systems that secure applications inside the untrusted operating system is the absence of necessity for such systems to stay activated all the time the computer is up and running. They are needed only for that periods of time when user executes *trusted applications*. The rest of time the execution of OS inside the VM may introduce restrictions into the user workflow. In particular the performance of virtualized system is lower than of real one and some features of specific devices may become unavailable.

Overshadow [3] secures applications by keeping their executable and data files encrypted in the file system. When user launches such applications Overshadow dynamically decrypts their content in the memory. Overshadow is idle for the time when only untrusted applications are running. The same case applies to Proxos [6] which runs trusted applications in the dedicated virtual machines. All other applications execute in one and the same untrusted VM. In [7] hypervisor prevents data leakage through the network connection by running OS in the VM that doesn't have network adapter at all. Then it provides network access for trusted applications by delegating their socket-related system calls to other network-enabled VM. The same level of isolation may be ensured without the hypervisor if network cable is disconnected from the computer.

Obviously when user activities are restricted due to system is virtualized he may switch between the bare hardware and virtualized configurations by rebooting the computer. During the boot time he selects the desired configuration: bare system with high performance or virtualized system with ability to run trusted applications. However such workflow may be inconvenient for the user and lead to declining using particular virtualization-based security solution.

Therefore we deem that that the problem of dynamic activation and deactivation of virtualization-based security system is topical. The solution should allow on the fly launch of the hypervisor which moves the up and running OS into the virtual machine environment and if necessary launches additional virtual machines. Then some time later user should be able to bring the system to the original state without rebooting the computer. Meaning that hypervisor stops all additional virtual machines, brings back OS to the bare environment and finally terminates itself. It is also important to include robust attestation procedure into such solution since hypervisor is launched from the untrusted environment.

Dynamic load of the hypervisor is described in [8]. Authors discuss the consequences of using hardware assisted virtualization for implementing malware and present BluePill system that loads on the fly into the memory putting operating system into the virtual machine. Authors claim that malware hypervisor may fool OS inside the VM that it runs on bare hardware. Since hypervisor has unrestricted access to VM resources it may bypass built-on OS security mechanisms and perform intended malicious actions. Our work follows the same approach for on the fly hypervisor loading however we propose to use our approach for virtualization-based security systems that normally do not need to hide from the OS. This gives more options for reserving hardware resources from the OS for the hypervisor own usage (for example, legally disabling OS access to some devices).

The attestation of a software stack launched from the untrusted environment is discussed in [9]. Authors present an approach and describe attestation protocol that allows safe usage of public computer by providing secure initialization of trusted software stack. The protocol uses Trusted Platform Module (TPM) chip that must be installed on the machine and provides convincing evidences to the user that actually initialized software stack is exactly the same as the expected by the user one. Our approach to attestation of the dynamically loaded hypervisor is also based on the TPM chip however unlike the system in [9] we do not need rebooting the computer which in turn allows achieving simpler attestation procedure.

We will describe our approach with regards to the security system described in [7]. However, we believe that suggested approach may be used for other virtualization-based security systems that 1) assign hardware devices to virtual machines for exclusive usage 2) allow their late launch (i.e. after OS has been booted) without sacrificing security requirements.

We assume that computer has CPU supporting AMD-SVM technology [10] including memory virtualization (Nested Page Tables) [11]. Computer is also equiped with Trusted Platform Module chip supporting TPM specification version 1.2 [12] whose public certificate is known to the user. However our choice of using AMD virtualization technology is merely based on the hardware available to us and the approach described in this article may be applied to the computers equiped with Intel processors supporting Intel-VT [13] virtualization technology.

We will briefly describe the security architecture presented in [7]. Hypervisor executes two virtual machines: primary (called *private* VM) and service (called *public* VM). User works in the primary VM which controls all hardware devices except network adapter. Hypervisor runs primary VM without network adapter since OS in the primary VM is not trusted and may use network connection to leak sensitive data from the VM. Service VM may run in the background since the only purpose of this VM is to serve socket-related system calls executed by the trusted processes in the primary VM. System call requests are delivered by the hypervisor to the service VM through the tamper-proof inter-VM channel. It should be noted that 1) primary VM must run on one CPU (one CPU core) and 2) the only device that is controlled by the service VM is network adapter while primary VM must not have access to it at all.

Service VM is required only for providing network access for the trusted applications. During the time intervals when these applications are not executing the OS in the primary VM may run directly on the bare hardware and even may control network adapter as long as the user ensures that network cable is *disconnected* from the computer. Additionally primary and service virtual machines do not share any peripheral devices.

This article is organized in the following way. In section 2 we describe the steps taken during the on the fly security system activation. In section 3 we discuss the attestation of activated system. In section 4 we describe the steps taken to deactivate the security system in a fool-proof way. In section 5 we conclude presented approach.

II. Security system activation process

Initially user boots operating system installed on the computer (hereafter we call it primary OS) ensuring that network cable is disconnected from it, i.e. computer is physically isolated from the network. Hypervisor is not started during the boot process however it is possible that there is a malware hypervisor (or any other kind of malware) running on the computer.

The file system contains hypervisor image, image for OS in the service VM (hereafter we call it service OS) and a secure loader. Secure loader is a special piece of software required for SKINIT instruction. OSLO secure loader may be used for it [14]. It is worth mentioning that all these files are not required to be encrypted however user knows their checksums (hash codes) and these checksums were calculated on a trusted machine.

There is a driver preloaded into the operating system kernel which exposes an interface to the user space applications via a device file in the file system. The driver's task is to reserve specific hardware resources from the operating system using built-in OS kernel interfaces, load images from the file system into the memory and start secure system initialization. Driver is also responsible for printing informational messages coming from hypervisor on the display. Hypervisor notifies driver about pending messages by injecting interrupt into the virtual machine which passes control to the handler installed by the driver.

To start security system activation user launches an application that sends "activate" command to the driver using *ioctl* system call passing the file names of the images. The driver then:

1) Unplugs all CPU cores, except the boot strap core (BSP), using the public Linux API for hot plugging CPUs.
2) Reserves required amount of physical memory. This may be done in two ways. The preferred way is to unplug memory banks using the public Linux API for hot plugging memory modules. Alternatively memory may be reserved using `get_free_pages()` function. Unplugging memory banks gives less fragmented memory areas (in ideal case not fragmented at all). The advantage for having contiguous memory is described below.

3) Loads hypervisor image, service OS image and the secure loader into the reserved memory.
4) Shutdowns network interface, unloads network driver and installs “pci-stub” driver, that is available in Linux, instead of it. This stub driver reports to OS that it controls network adapter without actually initializing the device.
5) Executes SKINIT instruction which securely transfers control to the secure loader which in turn passes control to the hypervisor.

Upon receiving the control hypervisor prepares VMCB structure filling it with the current hardware state, creates nested page tables [11] for the VM and starts VM execution. From the user perspective nothing has changed in the environment except the decrease of active CPU cores and memory amount. Since network cable was originally disconnected completely removing network adapter from the system does not harm user space applications.

Before hypervisor starts VM execution it re-initializes released CPU cores and starts idle loop on them while executing inside virtual machine primary OS cannot regain control over released resources. Attempt to access memory outside its guest physical address space, re-initialize unplugged CPU cores or reload original network adapter driver will lead at worst to the VM crash without getting access to the hardware resources.

One of the released CPU cores is used for running service VM. Hypervisor virtualizes local APIC so it presents this core to the service OS as a BSP core of a single core processor. Service OS completely runs inside memory. The OS kernel is preconfigured by the administrator in such a way that it does not require access to any peripheral devices except network adapter.

Both virtual machines execute on separate CPU cores: primary VM on the BSP core (zero core), service VM on the other core. Such approach does not require having VM scheduler in the hypervisor that distributes CPU time between virtual machines. Hypervisor itself does not require dedicated CPU cores for execution since it gains control only on virtual machine exit. We assume that CPU has at least two cores which is common for most of modern CPUs.

Mapping of the guest (virtual machine) physical address space into the host address space (machine memory) is resolved using nested page tables (NPT) provided by the hardware. These page tables add additional layer to the hardware address translation so every memory access made from inside the virtual machine is translated with NPT to evaluate machine address. It is desirable to reserve physical memory from the primary OS using memory hotplug feature available in Linux since having large contiguous memory blocks makes the structure of NPT rather simple. However success of memory unplug operation depends on proper kernel configuration, kernel boot options (“movablecore” option) and kernel runtime state.

With nested page tables the hypervisor can control accesses to the machine memory made from CPU context. However if OS in VM owns DMA capable device then it may use it to modify arbitrary physical memory areas. This constitutes potential threat to the security system since malware may use DMA operations to subvert hypervisor [15]. Physical address space visible to a DMA capable device may be limited by the IOMMU device [16]. IOMMU sits between PCI bus and system memory and allows specifying page tables that are used to translate every memory access originated from the devices connected to PCI bus. Whenever an address does not have valid translation in IOMMU page tables the device receives master abort and memory access is rejected.

Currently hardware supporting IOMMU is not wide spread so we also consider possibility to use Device Exclusion Vector (DEV) feature of AMD CPUs. The major difference between IOMMU and DEV is that DEV does not support address translation. It simply allows to specify bit mask (one bit per each physical page) that marks DMA write-protected pages. Since DEV may not perform address translation it may be used for one VM only whose guest physical memory is one-to-one mapped to the machine memory starting from zero address. That is the case for the primary VM. DEV bit mask for that VM write-protects all physical pages outside primary VM memory.

The use of DEV for the service VM is not that straightforward. However service OS kernel is configured by the administrator and he may include paravirtualized driver for the network adapter into it. Such driver asks hypervisor to translate address for every DMA operation. So DEV may still be used for the service VM with the bit mask write-protecting all machine memory outside service VM address space.

It is worth mentioning that DEV does not provide ability to read-protect pages so every VM may read data from any machine address. This may violate security requirements if malware in service VM would use DMA operations to read sensitive data from the primary VM memory. Therefore the use of DEV for DMA protection requires service OS to be *trusted*. Service OS is merely used to execute socket-related system calls so a microkernel OS (e.g. Minix 3) may be used in service VM instead of Linux.

III. Security system attestation

We use Trusted Platform Module (TPM) [12] to perform attestation of the activated security system. TPM is a chip normally attached to the motherboard that provides set of security primitives. TPM has several platform configuration registers (PCR) that are intended for accumulating SHA-1 hash-codes. These hash-codes (measurements) represent hardware installed or software running on the machine. The contents of a PCR register may be updated by executing `TPM_Extend` operation only (there is no way to write to PCR directly). `TPM_Extend` operation concatenates current value of the PCR register with the provided data, hashes the result and updates PCR with it. CPU is integrated with the TPM to perform secure late launch of the software using SKINIT instruction. Upon executing SKINIT CPU asks TPM to reset PCR 17 register to zero value. This is the only way how PCR 17 may receive zero value.

Contents of the PCR register may be cryptographically signed with Attestation Identity Key (AIK) using `TPM_Quote` operation. Private part of the AIK is sealed inside the TPM while public part is exposed to the user. Signed PCR value gives creditable and believable proof to the user that PCR value originates from the legal TPM. A random number (nonce) may be provided to the `TPM_Quote` operation as a parameter in order to protect signed PCR value from the replay attacks.

Activation of security system starts from executing SKINIT instruction passing address of secure loader (SL) to it [14]. CPU disables interrupts, blocks DMA writes to the memory area occupied by the SL, asks TPM to measure (hash) SL and finally transfers control to the SL. On a multi processor (multi core processor) SKINIT must be executed on the BSP core with all other cores put into the idle state. SL in turn performs the same operations with regards to the hypervisor image and then transfers control to the hypervisor code. Hypervisor measures service OS image and asks TPM to sign the PCR 17 contents using the `TPM_Quote` operation to which it passes user-provided nonce. The signed PCR 17 value is delivered to the user (e.g. displayed on the screen). The necessity of having SL in this chain is caused by the hardware size limitations for the SL. It should be at most 64K which may be insufficient to fit all the hypervisor code.

The resulting PCR 17 value contains measurements of the SL, hypervisor, service OS image and the nonce. User knows hash-codes of these components so he may repeat all calculations performed by TPM on a separate trusted device (e.g. mobile phone). By comparing hash-code displayed on the screen against the hash-code calculated on the trusted device he checks whether all security system components have been activated in the proper order. And by validating TPM signature using the public part of AIK (known to the user) he gets assurance that displayed measurement represents the actual state of the system and is not a fake measurement generated by the malware that has intercepted activation process and tries to fool the user. Only after validating displayed measurement the user is safe to connect network cable to the computer.

We consider that execution of SKINIT instruction may be trapped by the malware hypervisor running on the machine. This malware may perform all above-mentioned measurements without actually passing control to the security system or it may start security system inside the virtual machine hence maintaining control over it in a hidden way. However `TPM_Quote` operation may be executed only against PCR register which in turn may be updated using the `TPM_Extend` operation only. Therefore malware must use TPM to perform all measurements. However since PCR 17 is not reset to zero (this may be done by executing SKINIT instruction only) the resulting PCR value will not match the one calculated by the user on the trusted device.

If malware executes SKINIT instruction to reset PCR 17 to zero value then hardware securely passes control to the SL which in turn securely transfers it to the hypervisor thus not allowing malware to interpose on the control transfer. Malware can modify SL or hypervisor image to regain control after security system has been activated but then these modifications will be caught by the user upon validating the final measurement displayed on the screen.

IV. Security system deactivation process

The deactivation of security system implies removing hypervisor from the memory, bringing back primary OS to the bare hardware and returning all reserved resources to it. So this procedure may start only as long as hypervisor has a proof that deactivation command originates from the user that has disconnected network cable from the computer. We propose the following protocol for the deactivation procedure:

1) User disconnects network cable and executes a program that requests hypervisor to deactivate security system.
2) Hypervisor asks TPM to generate nonce using `TPM_GetRandom` operation which is then passed to the user.
3) User signs this nonce on a trusted device using his own private key and returns signed nonce to the hypervisor.
4) Hypervisor validates users signature with the public user key known to him, compares original nonce with the one received from the user and as long as they are equal deactivates security system.

While completing deactivation procedure hypervisor simply stops service VM. There is no information in this VM that must be persistently stored. Then hypervisor upon next VM exit event of the primary VM rewrites CPU registers with the VM state stored in its VMCB structure and resumes primary OS execution without entering virtual machine mode. Finally driver in the service OS returns previously reserved hardware resources back to the primary OS by hot plugging CPU, memory banks and reloading original network adapter driver.

V. Conclusion

In this article we have presented an approach to on-the-fly activation and deactivation of virtualization based security systems. We target this approach on the systems that allow discrete functioning without violating stated security properties. Among such systems are hypervisor-based solutions that secure user applications running under the control of untrusted operating system. Both activation and deactivation procedures consider the presence of malware on the computer and provide means for attesting security system being activated and for performing safe deactivation correspondingly.

In our approach the hypervisor is started on the fly from the up and running operating system upon the user request. Operating system and all running user applications are transparently moved inside the hardware virtual machine where they continue executing but from now under the control of the launched hypervisor and security system implemented inside it. If necessary additional virtual machines may be started at this moment. The security system shutdown procedure is performed upon the user request too with hypervisor being unloaded from the memory and operating system brought back to the bare hardware. Both activation and deactivation procedure do not require rebooting the computer.

Currently we are working on implementing proposed approach for the security system described in [7].

References

[1] R. Riley, X. Jiang, and D. Xu, “Multi-aspect profiling of kernel rootkit behavior,” in *EuroSys ’09: Proceedings of the 4th ACM European conference on Computer systems*. New York, NY, USA: ACM, 2009, pp. 47–60.

[2] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[3] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, “Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems,” in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008, pp. 2–13.

[4] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 2–13.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[6] R. Ta-Min, L. Litty, and D. Lie, “Splitting interfaces: making trust between applications and operating systems configurable,” in *OSDI ’06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 279–292.

[7] I. Burdonov, A. Kosachev, and P. Iakovenko, “Virtualization-based separation of privilege: working with sensitive data in untrusted environment,” in *VDTS ’09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*. New York, NY, USA: ACM, 2009, pp. 1–6.

[8] J. Rutkowska, “Subverting vista^{TM}kernel for fun and profit.” BlackHat, 2006. [Online]. Available: http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Rutkowska.pdf

[9] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang, “Trustworthy and personalized computing on public kiosks,” in *MobiSys ’08: Proceeding of the 6th international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2008, pp. 199–210.

[10] *AMD Architecture Programmer’s Manual Volume2: System Programming*, Advanced Micro Devices Inc., November 2009.

[11] *AMD-V^{TM}Nested Paging*, White paper, Advanced Micro Devices Inc., July 2008.

[12] “Trusted platform module,” Trusted Computing Group. [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[13] *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide*, Intel Corporation, March 2010.

[14] B. Kauer, “Oslo: improving the security of trusted computing,” in *SS’07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–9.

[15] R. Wojtczuk, “Subverting the xen hypervisor.” BlackHat, 2008. [Online]. Available: http://blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf

[16] *AMD I/O Virtualization Technology (IOMMU) Specification*, Advanced Micro Devices Inc., February 2009.