Sevigator: Network Confinement of Malware Applications and Untrusted Operating Systems

Denis Efremov and Nikolay Pakulin
Institute for System Programming, Alexander Solzhenitsyn st. 25, Moscow, Russian Federation

Keywords: Virtualization-based Security, Network Access Control, Hypervisor, Virtual Machine Monitor, Virtualization, Security, Privacy Protection.

Abstract: This project is an attempt to combine the advantages of software flexibility and security of hardware firewalls. It aims at the implementation of these advantages in the hypervisor source code for the purpose of creating user data confidentiality protection against its leakage from the personal computer through the network. The hypervisor implementation is based on the hardware virtualization extensions of both processors and motherboards. This constitutes a key feature, which enables hypervisor to combine the following advantages: the advantages of access to the OS environment and hardware protection against various intruders' methods of compromise, including those capable of exploiting OS kernel resources for performing the malicious actions.

1 INTRODUCTION

Traditional firewalls provide only partial protection from viruses and alikes, since they do not know exact context for IP packets — whether the sender is a trusted process or a malware, whether the contents of the packet is valid or it is altered by a malicious software. This statement holds for both bump-in-the-wire firewalls that run on physically separate computer or router, and firewalls running on the same network node with the malware. The reason is that modern monolithic OS grant low-level software virtually unbounded rights. If a malware managed to compromise OS and installed a driver deep into the kernel, it can either mimic a system process or even inject malware code into running trusted application.

This paper presents Sevigator – a toolkit for network confinement when only trusted application gain access to local network while other application and even OS kernel have no networking at all. Sevigator is based on hardware virtualization support: a custom hypervisor hides network interface card from the OS kernel and delegates network-related system calls of trusted applications to a dedicated service virtual machine. To prevent code injection or data alteration by a malicious kernel code or driver the hypervisor maintains integrity of the trusted applications binaries, shared objects and in-memory data.

2 ARCHITECTURE

In Sevigator the hypervisor ensures simultaneous execution of two virtual machines (VM), primary and service, completely isolated from each other. The primary VM contains secured applications; it is the VM that potential users have access to. The service VM is designed to process network input/output for trusted application of the primary VM. (Ta-Min et al., 2006); (LeVasseur et al., 2004)

A user works in the primary VM, which controls all devices except for the network adapter, through which the information leakage might occur. The primary VM contains sensitive data, and software handling this data (both trusted and untrusted). The data are stored in a plaintext (unencrypted) form, and protection system does not limit read access of processes (both system and user level) to these data making its processing possible by any programs including untrusted ones.

When primary VM starts the hypervisor blocks its access to the network interface. An OS running in it believes a network adapter to be physically absent. Therefore any attempt to establish a network connection from within this VM and transfer data to the remote computer will unavoidably lead to an error. Hence the malicious code running within the primary VM with any level of hardware processor privileges will not be able to transfer data, even though it manages to get read access to it.

Efremov D. and Pakulin N..
Sevigator: Network Confinement of Malware Applications and Untrusted Operating Systems.
DOI: 10.5220/0004070103950398
In *Proceedings of the International Conference on Security and Cryptography* (SECRYPT-2012), pages 395-398
ISBN: 978-989-8565-24-2

Among all user programs included in the primary VM, a number of trusted programs is distinguished. The service VM is used to provide network access to them. This VM runs in background mode. Any application within this VM can interact with remote network computers.

The support of network communication for the trusted applications is implemented by means of remote servicing of the required set of system calls from the primary VM to the service VM. The only data communicated outside the primary VM are explicitly specified by the arguments of a trusted process system call. Transfer of data outside this VM is performed by the trusted code - the hypervisor.

The trusted processes are executed under an untrusted OS. In the absence of proper control by the security system, the malicious code in the OS kernel can inject code into the address space of the trusted process, pass control to it, and the trusted process in its own name may send sensive data to a remote computer controlled by intruder. To avoid this threat, security system protects the context of the trusted application from an unauthorized modification by any code in the primary VM, including privileged code.

3 PROTECTION OF THE TRUSTED APPLICATIONS CONTEXT

Malicious software in OS kernel has capabilities to alter application's executable file or modify the shared libraries that the application uses. As a countermeasure we perform transparent validation of the application's executable file and its shared libraries when a trusted application is being launched. Authenticity is validated by means of secure control sums of the pages of the memory-mapped file of the application or a library. Checksums are 160-bit SHA-1 hashes, one for each page of static data and code in a binary file. Checksum failure for at least one hash causes loss of the trusted status of the application. The procedure of checksum verification is implemented in the hypervisor. System administrator has ability to dynamically load them into the hypervisor during the system operation.

Besides the libraries specified in the executable, an application may dynamically load libraries. The hypervisor intercepts mapping of those files into memory and validate the libraries.

Every trusted process is executed in a separate security domain. It is essential to maintain statuis of control tables and ensure access interception to the desired memory pages. Security domain is nothing more than a dynamically changeable set of physical pages with specified read and write access rights. An attempt of an executable code to get access to the physical page outside its domain as well as an access to the page within its domain accompanied by the access violation is intercepted by the hypervisor.

The implementation of security domain technique is based on nested page tables for each domain (NPT) (AMD, 2008). Nested tables specify translation of nominal physical VM addresses to the actual absolute physical addresses. The hypervisor creates a new (empty) set of nested tables for a process when it enters the trusted mode for the first time. Every time the control is passed to the process, the hypervisor sets the nested tables for this process as active for the virtual machine. Upon the interruption of process execution and control pass to the OS, the hypervisor toggles active nested tables and sets the tables of an untrusted domain (domain of kernel and untrusted processes). (Chen et al., 2008); (Yang and Shin, 2008)

4 THE HYPERVISOR AND VM DATA COMMUNICATION METHODS

Remote servicing of the system calls is implemented by the hypervisor in association with the system components, working in both VMs, primary and service. During the system initialization modules are dynamically loaded into the kernel of the primary and service VM.

Each module allocates continuous physical memory space for ring buffer, registers several interrupt handlers which are used by the hypervisor to notify virtual machine of incoming events or processing, and communicates this information (buffer address and interrupt numbers) to the hypervisor through a hypercall (AMD, 2011). Synchronous nature of hypercall makes it possible to transfer hypercall parameters, much as the user process transfers parameters to the OS kernel during the system call execution: numerical parameters and memory area addresses are passed through the registers, the hypervisor reads virtual machine memory area at the specified addresses and fetches addition information from there (or records it there) when needed.

To notify VM about event the hypervisor uses a capability to throw interrupts and exceptions into the virtual machine by means of the corresponding fields in the VM control structure VMCB (AMD, 2011) provided by the virtualization hardware. Upon VM

resumption, the hardware ensures interrupt delivery immediately before the execution of the first VM instruction. In response to the interrupt OS passes control to the corresponding interrupt handler which was registered in the interrupt handlers table by the kernel module during the system initialization process.

System call parameters are passed through the ring buffer. If the buffer is overflowed, the request delivery is suspended until space in the buffer is freed up. The data structure representing a ring buffer element is common for all system calls and includes fields for all possible fixed-size parameters. Variable-size parameters are transferred through a separate buffer allocated in the storage (hypervisor memory). The coordinates of a variable-size parameter, an offset from the storage origin and a size, are specified in the data structure of the ring buffer. The hypervisor maintains a separate storage copy for every trusted process.

Upon the receipt of a request containing variable-size parameters, the VM code for which this request is intended performs a hypercall for the storage access sending a requested parameter coordinates and an address of the buffer in its own memory intended for holding the data from the storage. The hypervisor services the call and resumes VM execution.

5 REMOTE SERVICING OF THE NETWORK SUBSYSTEM SYSTEM CALLS

User applications use sockets for networking. One of the parameters specified during creation of a new socket is a protocol type. This parameter defines which handlers need to be called by the OS kernel for correct completion of a system call and creation of a new socket. In the subsequent data reading, writing, and other actions with the created socket the OS kernel handlers related to this protocol are called.

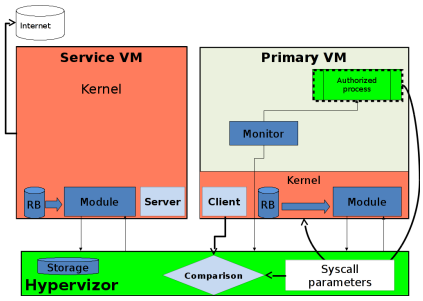

Figure 1: Configuration of the security system components in charge of remote execution of the system calls, and simplified chart of their interaction.

To ensure data confidentiality for trusted applications, untrusted applications and OS kernel in the primary VM are deprived of network access: when kernel enumerates PCI bus the hypervisor intercepts all replies from network card. As a result, the primary VM gets no knowledge about available network facilities and runs in network-less mode. For application it means that only loopback adapter is available, no ways to send or receive data from outside of VM.

Network access to trusted applications is provided by means of a dedicated stub in the kernel that routs all network requests to the service VM through hypervisor. To make the data routing transparent to application the system call `socket` is intercepted by hypervisor: it substitutes protocol family parameter with the ID of the stub protocol.

Stub crotocol can be divided into client and server parts, with server part running in the service VM, and the client – in the primary VM (Figure 1).

When a trusted application invokes a `socket` system call, the execution of the primary VM is interrupted and control is passed to the hypervisor. The hypervisor substitutes the protocol identifier in the arguments of the system call by a special one corresponding to the stub protocol. The hypervisor also copies certain parameters of system call from the trusted application. Then the execution of the primary VM is resumed. System call is received by the OS kernel. It analyses the system call parameters and passes control to the registered handlers in the client part of the stub protocol.

The client part allocates required kernel resources, creates packet containing system call arguments, writes it to the ring buffer and notifies the hypervisor through the hypercall. The hypervisor partially compares previously stored system call arguments with those transferred through the ring buffer of the primary VM. Since the primary VM kernel is considered to be untrusted, such a comparison is required to avoid data leakage to forged addresses and ensure data integrity. In case of comparison error the received system call will not be executed in the service VM.

The hypervisor writes received data into the ring buffer of the service VM and throws an interrupt to it. Interrupt handler passes the control to the server part of the protocol. The data from the ring buffer are then unpacked. Then the standard handling of a system call with the received arguments is performed, as if this call was received from the user application running in the service OS without security system. The track presented above results in socket creation and IP packets been sent out by the service VM for a trusted application. Processing of inbound packets is performed in the reverse order.

The hypervisor guarantees that only parameters explicitly specified in the system call of trusted process address ranges may be modified. To accomplished it the stub driver allocates memory area of the required size and modifies the system call parameters, OS writes results in this memory area. The hypervisor copies the data into the corresponding memory area of the trusted process upon the return from the system call.

6 IMPLEMENTATION

Currently, the protection system is implemented as extension of KVM hypervisor. KVM runs in a host operating systems and uses x86 emulator QEMU for device emulation. In our work, we use KVM 88 version, QEMU 0.15 version and Linux kernel of 2.6.32 version.

6.1 Performance

In order to measure overhead of memory protection and system call servicing, we have performed several tests. In each measurement there was only one trusted process. The list below presents results of the tests for a few popular applications. Percent value is the extra time spent by the trusted application compared to the time spent by the same application with protection system disabled.

Table 1: Measurements.

Software	Description	Overhead
Apache	Flood test.	2%
Ttcp	Ordinary execution.	2%
SSH&SCP	SCP 4Gb file copy.	22%

7 CONCLUSIONS

This article presents an approach to protect application confidentiality from untrusted (potentially compromised) OS. The approach is implemented by selectively granting a network resource access to the specific trusted user applications. Confidentiality is achieved by preventing untrusted, malware or compromised OS components from communication channels to the outside world.

The implementation of the approach is based on the execution of the untrusted OS within a virtual machine and placement of the trusted part of the security system in the hypervisor body. It allows achieving full control of access of the processes executed under VM to the hardware resources. The security system therewith remains inaccessible for attacks by malicious software.

Facilities of system call intercepting, reading their parameters, and writing syscall results provided by virtualization makes it possible to protect access to any peripheral hardware resources.

The possibility to grant to certain processes a controlled access to the resources not accessible by the OS itself, provides a way to effectively solve particular issues of the user data confidentiality preservation problem and confines untrusted applications, including the OS kernel, within the network node, making malicious networking impossible.

At the moment the implementation of the hypervisor is based on KVM for Linux on AMD platform and it supports network confinement for Linux only. Future research goals are extension of the approach to Intel VT-x platform, support MS Windows family of operating systems, and “bare-metal” execution of the hypervisor.

REFERENCES

(2008). *AMD-V^{TM} Nested Paging*. Advanced Micro Devices Inc.

(2011). *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices Inc.

Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. (2008). 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*, pages 2–13, New York, NY, USA. ACM.

LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. (2004). Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 2–2, Berkeley, CA, USA. USENIX Association.

Ta-Min, R., Litty, L., and Lie, D. (2006). Splitting interfaces: making trust between applications and operating systems configurable. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292, Berkeley, CA, USA. USENIX Association.

Yang, J. and Shin, K. G. (2008). 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, pages 71–80, New York, NY, USA. ACM.