Jeffrey S. Dwoskin, Mahadevan Gomathisankaran, David Champagne, and Ruby B. Lee.
SP Reference Manual Addendum -- Secure Stacks for TSMs and Emulation of SP Interrupt Protection.
Princeton University Department of Electrical Engineering Technical Report CE-L2009-006,
August 2009.

SP Reference Manual Addendum –

Secure Stacks for TSMs and Emulation of SP Interrupt Protection

Jeffrey Dwoskin, Mahadevan Gomathisankaran, David Champagne, Ruby Lee
Princeton University

Version 0.1, 8/7/2009

Princeton University Department of Electrical Engineering Technical Report CE-L2009-006

1. Introduction

Recent work on the SP architecture has involved implementations of authority-mode SP [3] on the VMware virtualization platform [1,2] and the design and implementation of a testing framework for security architectures [1], with an emphasis on testing attacks on TSMs that run on the emulated SP platform. This technical report addresses two architectural issues that have arisen during the implementation of SP on the virtualization platform.

Secure Stacks

First, we have known that a TSM must use SP's secure memory to protect its stack. The new Secure Area mechanism for secure memory simplifies the process of designing secure TSM code, at least for the emulation implementation of SP, by eliminating the need to modify the compiler and other build tools to support the use of secure memory. Instead of using special instructions for each access to secure memory, a secure area region of memory is defined which is always accessed as secure memory when in CEM. Secure Areas also simplify the architectural changes needed on a CISC architecture, such as the x86 platform used in VMware, by eliminating the need to duplicate all instructions that access memory.

Using secure areas, we have implemented macros that can be inserted into a TSM immediately after Begin_CEM and immediately before End_CEM which change the stack pointer to use a secure area as the stack for a TSM. However, x86 introduces another complication in that on an interrupt, the hardware directly uses the stack pointer to push an exception frame onto the application's stack. The OS then reads this frame to handle the interrupt and later resume the application. This works fine for normal code, however for a TSM with its stack as part of a secure area, this overwrites part of the secure area region. Since the region is encrypted and hash checked as the first step in the interrupt (before the exception frame is written), any modifications, even to parts of the stack that are unused, will cause verification to fail when the TSM is resumed.

To solve this problem, we have added new instructions, defined below, which make the SP hardware aware of the secure stack. When an interrupt occurs during CEM, it can then swap back the unprotected stack pointer, which the hardware can safely use to write the exception frame. The secure stack is then swapped back in when the TSM resumes.

Emulations of SP Interrupt Protection in VMware

For the VMware implementation of the SP architecture, we implement the new SP instructions and new SP hardware behavior inside VMware's VMM. Since the VMM uses a combination of direct execution of

guest VM code and binary translation of code, it is sometimes complicated to implement certain new features. To implement SP interrupt protection, we must detect and intercede upon any interrupt (hardware IRQ, software interrupt, or fault/ exception) that occurs while in CEM, and we must detect when the guest OS reschedules the TSM process to resume CEM.

Detecting interrupts is fairly simple, since all cause a trap to an interrupt handler in the kernel, which is automatically directed first to the VMM before being forwarded to the guest OS kernel. We insert a call to the SP Suspend_CEM routine in the VMM and are able to ensure that the TSM's general registers are encrypted and hashed, that all secure areas are encrypted and hashed, and that the CEM state is transitioned to suspended.

Detecting the return from interrupt is more difficult in the VMM. There are a number of ways that the guest OS kernel switches privilege levels and returns to a user process. Real SP hardware can watch the resulting instruction pointer address of all return/jump instructions that return to user code to trigger the Resume_CEM process when the target is the suspended CEM thread. As far as we can tell thus far, the VMM does not intercede in all of these cases, permitting us to insert our emulated Resume_CEM routine.

In most cases, all guest OS kernel code should be executed using binary translation in the VMM. Therefore it might be possible to instrument a check in the translated code for each instruction that might return to user code, checking addresses and other parameters (e.g., process ID) against the SP interrupt registers. At a minimum, this would require significant effort to find all such instructions and instrument them individually. If any return paths use direct execution, we would run into difficulty.

Instead, we borrow a technique from Overshadow [4], creating a piece of jump code outside of the TSM, which the VMM can use to detect all returns. A small piece of code is statically compiled into the TSM, at a known address, which contains at a minimum code for a hypercall to the SP emulation module in the VMM. The address of this code block is provided to the emulation module as part of any Begin_CEM call, and will be saved in the VMM. During the interrupt suspend process, the real return address in the TSM will be saved in the emulation module and replaced with the address of the jump code in the exception frame returned to the guest OS. When the guest OS returns, it will return to the jump code instead of the real TSM. The jump code then triggers the hypercall, so the VMM knows to intercede and trigger the Resume_CEM operation. Resume_CEM will then redirect the processor's instruction pointer to the real TSM return address while restoring the registers and secure areas and resume active CEM.

2. Architecture

Secure Stacks

Hardware SP Instruction	SP Emulation Instruction/Hypercall	Description
N/A	CEM_SetSecureStackPtr (void *addr)	Saves the current Stack Pointer register to the CEM_StackPtr register and sets the Stack Pointer register to the value in *addr*. In practice the memory pointed to by secure_sp should be inside a Secure Area region, with sufficient

		space in the region to hold all data the TSM will push onto the stack. The hardware does not verify that the address is within a secure area. If a Suspend_CEM operation occurs due to an interrupt during CEM, the SP hardware will swap the current Stack Pointer register (containing the secure stack location) with the value in the CEM_StackPtr register (containing the original app's unprotected stack pointer). The CEM_StackPtr register now contains the pointer to the secure stack, and the hardware interrupt handling will use the unprotected stack to write the exception frame. (This is true on x86. Other architectures may not use the stack pointer in hardware.) When a Resume_CEM operation occurs, to re-enter CEM after an interrupt, the current Stack Pointer register and the CEM_StackPtr register will be swapped back before returning control to the guest code, which should be signed TSM code. If the CEM_StackPtr register has the value zero, no swapping will occur. Swapping only occurs when suspending during active CEM with a non-zero CEM_StackPtr value, and when resuming to active CEM with a non-zero CEM_StackPtr value (and when other checks, such as the return interrupt address, match). *Available to TSM only.*
N/A	CEM_RestoreStackPtr()	If the CEM_StackPtr register contains a non-zero value, overwrites the current Stack Pointer register with the value stored in CEM_StackPtr and zeroes the CEM_StackPtr register. Note: If the TSM wishes to use the old SP address, it should save a copy of the SP before calling CEM_RestoreStackPtr(). *Available to TSM only.*

Interrupt Protection

Hardware SP Instruction	SP Emulation Instruction/Hypercall	Description

Begin_CEM	Begin_CEM (void *return_ptr)	Same function as original Begin_CEM() but also copies the value of *return_ptr* to the CEM_InterruptResumePtr register. When an interrupt occurs during active CEM, in addition to copying the return address to the CEM_InterruptAddress register, the SP hardware will now also replace the return address given to the OS with the value in the CEM_InterruptResumePtr register if it is non-zero. The SP emulation module will also implement a hypercall to trigger the Resume_CEM routine to return to active CEM from interrupt. Instead of checking the CEM_InterruptAddress value against the new instruction pointer, SP hardware will instead jump to the value in the CEM_InterruptResumePtr register, if non-zero, and then complete the Resume_CEM process as usual. Note: Any other checks performed by Resume_CEM (e.g., checking process ID or compartment ID, if implemented) will still be done before triggering the jump and resume to active CEM.
End_CEM	End_CEM()	Same function as original End_CEM() but also clears the value of the CEM_InterruptResumePtr register. *Available to TSM only.*

Note that the CEM_InterruptResumePtr must be set atomically with the Begin_CEM instruction. Each block of TSM code, or more importantly TSM code in each user process/application, will have its own code block for the jump-to-resume-hypercall routine. Therefore, if application code sets the register before calling Begin_CEM, it may be interrupted in between the two operations. Then another application may execute, overwrite the register with its own value, and return control to the first application; this would cause the first application's TSM to execute using an invalid resume pointer in its address space.

As the Begin_CEM instruction is not CIC checked in SP, the Interrupt Resume Pointer is set by untrusted code. Also, the jump-to-resume-hypercall routine itself will be untrusted, unsigned code. This can allow denial of service attacks on the TSM. By making the pointer or routine invalid, or preventing the routine from triggering the resume hypercall to the VMM, untrusted code can prevent the resume CEM operation. However, it cannot cause a breach of confidentiality or integrity of CEM data, nor can it change the return address into the TSM.

If a resume hypercall is triggered at an incorrect time, the VMM will attempt to return to active CEM. It will decrypt the registers and secure areas and enter CEM for the next instruction. If there is a suspended CEM thread, there is no Interrupt Resume Pointer set, and the CEM Interrupt Return Address is correct, it will execute the next instruction in CEM with CIC checking. As in the existing SP model, if

this location does not contain correct, signed TSM code, CIC checking will fail before any instructions are executed in CEM, causing an exception that will re-protect the CEM data. If there is an Interrupt Resume Pointer set and a suspended CEM thread, the processor will attempt to jump to that address as it resumes CEM. If in the correct process context, this will actually resume the TSM, regardless of whether or not it was intended to be scheduled. If not in the correct process context, CIC checking will again fail and exit CEM.

New SP Registers

To implement the above new features, two additional SP registers are added to the SP CPU registers:

- CEM_StackPtr
  - 64-bits (or same size as the processor word size used for the existing stack pointer)
  - Available only to SP hardware only
  - Set/cleared through the CEM_SetSecureStackPtr and CEM_RestoreStackPtr instructions and during suspend/resume SP hardware operations for interrupt handling.
- CEM_InterruptResumePtr
  - 64-bits (or same size as the processor word size used for the existing instruction pointer)
  - Available to SP hardware only
  - Currently only used or planned for implementation in SP emulation and not for real hardware.
  - Set as a parameter to Begin_CEM and cleared on End_CEM.

References

[1] Jeffrey Dwoskin, Mahadevan Gomathisankaran, Ruby Lee. “Framework for Design Validation of Security Architectures,” Princeton University Department of Electrical Engineering Technical Report CE-L2008-013, November 2008.

[2] Jeffrey Dwoskin, Ganesha Bhaskara, Thuy D. Nguyen, Ruby Lee, “SecureCore Prototype/Demo Manual,” Version 1.0. Princeton University Department of Electrical Engineering Technical Report CE-L2008-009, 8/11/2008.

[3] Jeffrey Dwoskin and Ruby B. Lee, “SP Processor Architecture Reference Manual,” Princeton University Department of Electrical Engineering Technical Report CE-L2008-008, 8/11/2008. (Previous version: CE-L2007-009. Version 0.7, 11/21/2007)

[4] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, D. R. K. Ports, “Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems,” Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2008.