Overview

- This Lecture
  - Virtualization and security
  - Source: A comparison of software and hardware techniques for x86 virtualization, Backtracking intrusions
  - VM hypervisor on Raspberry Pi
    - http://www.raspberrypi.org/forums/viewtopic.php?p=357354#p357354

Virtualization

- What is virtualization?
  - Accurate simulation of a computer on VMM
- Virtual Machine Monitor
  - VMM is used to facilitate the simulation
  - Three essential characteristics
    - Fidelity: software on VMM executes identically to its execution on hardware except the timing effect
    - Performance: most instructions are executed directly on HW
    - Safety: VMM manages all hardware resources
  - Guest OS/virtual machine

Why use VMM?

- One computer, multiple operating systems
- Server consolidation
- Better security and fault isolation
- Simplify management of big machines
- Debugging for OS development (QEMU)
- Current VMM
  - VMware, Xen, Virtual PC, VirtualBox, AMD SVM, Intel VT, UML, etc.
  - Note we don't talk about in-kernel VMM

Approaches to virtualization

- Trap-and-emulate (classical)
  - Emulate privilege instructions using trap
- Para-virtualization
  - Modify OS to fit the VMM interface in order to improve performance, e.g. Xen
- Binary translation
  - Convert special instructions into a sequence of simulation instructions, e.g. VMware
- Hardware VMM
  - Hardware implementation supporting VMM

x86 virtualization

- Primary and shadow structures
  - VMM uses shadow structures to mirror the primary structures in a virtual machine
  - On-CPU privileged state such as page table pointer CR3 can be easily handled, since VMM can maintain an image of the registers for the guest OS and traps are used to maintain the consistency between the shadow and the primary
  - Off-CPU privileged structures such as page table are more troublesome to maintain

Memory traces

- Memory traces are used to maintain the coherency of shadow off-CPU structures
  - For example, guest page table is write protected.
  - When guest modifies the page table, there is page fault which brings control to VMM
  - VMM can simulate the faulting guest instruction to change the guest page table as well as the shadow page table in VMM, after some validation check of the modification
  - How to make the changes more efficient?
  - How to balance among trace cost, hidden faults and context switch?

X86 obstacles

- Visibility of privileged state
  - The guest can observe that it has been deprivileged when it reads its code segment selector %CS since the CPL is stored in the low two bits of %CS
- Lack of traps when privileged instructions executed at user mode
  - For example, popf (load ALU and system flags from stack) can change system flags like IF (for control of interrupt delivery) in kernel mode, but in user mode attempts to modify IF are simply suppressed. No traps to use for VMM.

Solutions

- Binary interpretation
  - Run guest OS on an interpreter
  - The interpreter can prevent leakage of the privileged state such as CPL from the physical CPU to the guest
  - Very slow
- Binary translation (by VMware)
  - Convert the instructions into a set of mostly user-mode instructions
  - Some instructions like popf can be replaced with INT and then emulated by the translator

Advantages of BT

- Full virtualization instead of paravirtualization
  - No assumption about the guest code
  - Input is full x86 instruction set, output is a safe subset (mostly user-mode instructions)
- Dynamic
  - The translation happens at runtime, interleaved with execution of the converted code
- On-demand
  - Code is translated only when it is about to execute, which need not tell code and data apart in advance
- Adaptive
  - Translated code can be adjusted according to guest behavior in order to improve overall efficiency

How BT works?

- Read guest's memory at the address indicated by the guest PC (i.e. IP in x86) register
- Classify the bytes as prefix, opcode, operands, and produce intermediate representation (IR) objects
  - Each IR object represents one guest instruction
- Organize IR objects into translation units (TU)
  - Stop at 12 instructions or a termination instruction such as jump instructions for changing control flow
- Translate most IR objects except the termination instruction identically into x86 instructions
  - The termination instruction is turned into invocation of a translator continuation.
- Translation Cache (TC) is used for translated code for code reuse

IsPrime C code

```
int isPrime(int a) {
    for (int i = 2; i < a; i++) {
        if (a % i == 0) return 0;
    }
return 1;
}
```

IsPrime assembly

- Compile the C code into 64-bit binary:
- isPrime: mov %ecx, %edi ; %ecx = %edi (a)
- mov %esi, $2 ; i = 2
- cmp %esi, %ecx ; is i >= a?
- jge prime ; jump if yes
- nexti: mov %eax, %ecx ; set %eax = a
- cdq ; sign-extend
- idiv %esi ; a % i
- test %edx, %edx ; is remainder zero?
- jz notPrime ; jump if yes
- inc %esi ; i++
- cmp %esi, %ecx ; is i >= a?
- jl nexti ; jump if no
- prime: mov %eax, $1 ; return value in %eax
- ret
- notPrime: xor %eax, %eax ; %eax = 0
- ret

BT example

- isPrime:
  - mov %ecx, %edi
  - mov %esi, $2
  - cmp %esi, %ecx
  - jge prime
- isPrime':
  - mov %ecx, %edi ; IDENT
  - mov %esi, $2
  - cmp %esi, %ecx
  - jge [takenAddr] ; JCC
  - jmp [fallthrAddr]

Optimization

- isPrime': *mov %ecx, %edi ; IDENT
- mov %esi, $2
- cmp %esi, %ecx
- jge [takenAddr] ; JCC
- ; fall-thru into next CCF
- nexti': *mov %eax, %ecx ; IDENT
- cdq
- idiv %esi
- test %edx, %edx
- jz notPrime' ; JCC
- ; fall-thru into next CCF
- *inc %esi ; IDENT
- cmp %esi, %ecx
- jl nexti' ; JCC
- jmp [fallthrAddr3]
- notPrime': *xor %eax, %eax ; IDENT
- pop %r11 ; RET
- mov %gs:0xff39eb8(%rip), %rcx ; spill %rcx
- movzx %ecx, %r11b
- jmp %gs:0xfc7dde0(8*%rcx)

Non-IDENT instructions

- PC-relative addressing
  - The translated code resides at a different address from the original code
- Direct control flow
  - Control flow must be reconnected in TC since code layout has been changed by the translation
- Indirect control flow (jmp %cs:off, call, ret)
  - Since the target is not fixed, translation-time binding is not possible
- Privileged instructions
  - Use TC sequence to replace simple operations
  - E.g. cli (clear interrupts) takes longer time than vcpu.flags.IF := 0

Further optimizations

- Use direct execution for guest user code
  - Switching guest execution between BT mode and direct execution when the guest switches between kernel- and user-mode
  - Caveat: there is no protection for the guest kernel from the user code
- Adaptive binary translation
  - BT can avoid privileged instruction traps which are expensive for modern CPU
  - rdtsc: trap-and-emulate (2030 cycles), callout-and-emulate (1254 cycles), in-TC emulation (216 cycles)
  - Some other traps remain: non-privileged instructions accessing sensitive data like page table
  - Adaptive BT
    - Start in the innocent state until proven guilty
    - For code with frequent traps, retranslate the non-IDENT to avoid the trap, or patch the original IDENT translation with a forwarding jump to the new translation

Hardware assisted VMM

- VMCB is used for VM state
- A guest mode is used, in contrast of host mode
- When VMM schedules a guest to run,
  - It fills in a VMCB with the current guest state
  - Call vmrun
  - Hardware loads guest state from VMCB and continues execution in guest mode until some condition expressed by VMM in VMCB
  - When a condition occurs, hardware performs an exit operation, which is the inverse of vmrun.
  - On exit, the hardware saves guest state to the VMCB, loads VMM-supplied state into the hardware, resumes the host mode and executes the VMM.

Security

- Source: Backtracking intrusions

TOCTTOU

- Time-Of-Check To Time-Of-Use (TOCTTOU) bug
  - Example: cleaning a /tmp directory
  - root runs: rm /tmp/*/* (which is really something like find /tmp -not-accessed-recently | xargs rm)
  - two phases: expand list of files, then unlink them
  - attacker: mkdir /tmp/a; echo >/tmp/a/passwd
  - root's rm: finds /tmp/a/passwd
  - attacker: rm /tmp/a/passwd; rmdir /tmp/a; ln -s /etc /tmp/a
  - root's rm: unlink("/tmp/a/passwd"), unlinks passwd from /tmp/a==/etc
- Fix?

Unix/Linux security model

- Uid/gid are used for access control
- Uid 0 (root) is treated specially
- Uid/gid are used for the following checks
  - Inode access (3-bit permissions for U, G, O)
  - Root (uid 0) can change uid (with setuid() syscall), create device inodes, chown, reboot, etc
  - However, file descriptor access is via pid
- Login
  - Run as root, check user name, password against /etc/shadow, setuid(user), run user's shell
- How to get back to root? How su is implemented?
  - Hint: setuid bit

Limitations of Unix security

- Can't pass privileges to other processes
- Can't have multiple privileges at once
- Not a very general mechanism
  - cannot create a user unless root
- Subjective policy
  - Can have unexpected results
  - Easy to escape from chroot with root
    - fd=open("/"), chroot("/tmp"), fchdir(fd), chroot("./../../../..")
  - Can create hard link to any file
  - Buggy code remains setuid even if root rm's it
  - Users can keep a copy of sensitive data once they obtain root access
- Mandatory access control vs discretional access control
- KEYKOS, every object needs a key (capability) to access

Backtracking intrusions

- Purpose
  - Provide information for system admin to understand how an attack happened.
- Idea
  - Log objects and events at OS level
  - Objects: files, file names, processes
  - Events: system calls
  - Establish a dependency relationship between them
- Kernel level logging vs application level logging
  - Application-level: semantically rich but easily disabled
  - Kernel-level: difficult to disable but semantically poor

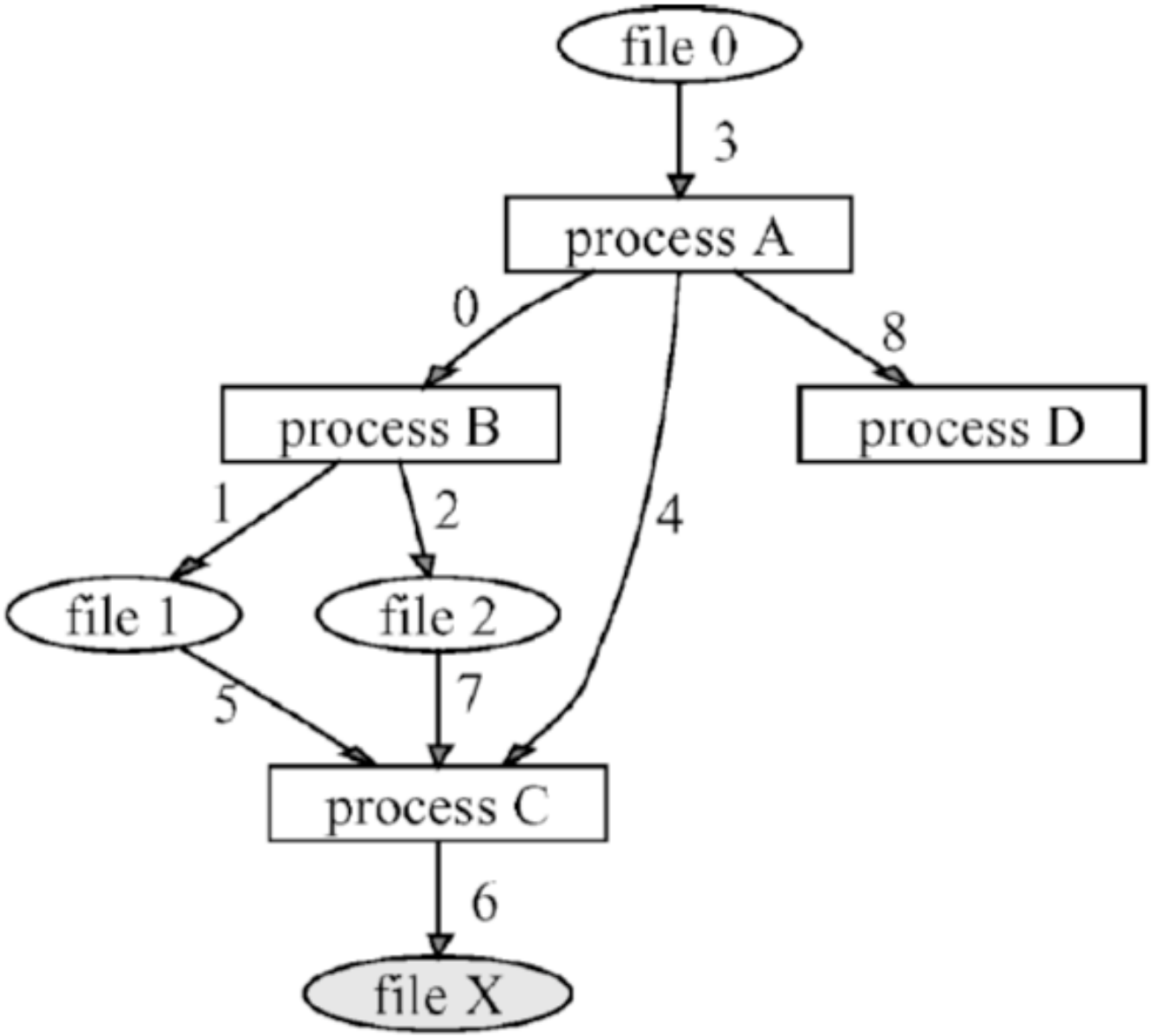
Dependency

- Process/process dependency
  - Fork(), clone(), P->C
- Process/file dependency
  - Modify file: process->file
  - Read file: file->process
  - Or both
- Process/filename dependency
  - Any system call including a file name (open, creat, link, unlink, mkdir, rename, rmdir,stat, chmod) causes a filename->process
  - Any successful system call that modify a filename argument causes a process->filename. E.g. creat, link, unlink, rename, mkdir, rmdir, mount

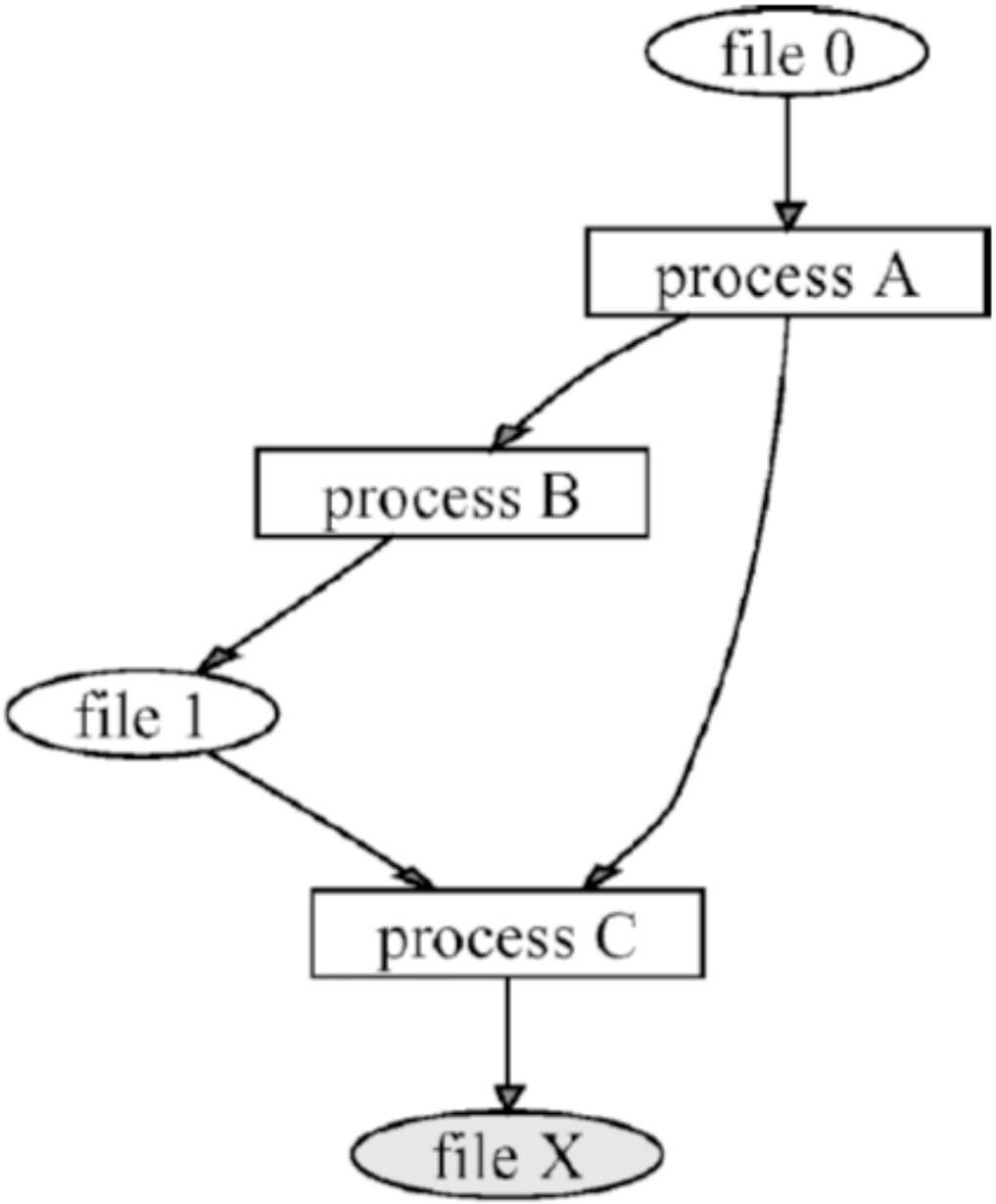
Dependency graph

- Create a dependency graph offline based on the log information
  - Start at a detection point when system admin found any suspicious behavior of the system, e.g a system file has been changed
  - Example
    - Time 0: process A creates process B
    - Time 1: process B writes file 1
    - Time 2: process B writes file 2
    - Time 3: process A reads file 0
    - Time 4: process A creates process C
    - Time 5: process C reads file 1
    - Time 6: process C writes file X
    - Time 7: process C reads file 2
    - Time 8: process A creates process D

Example graph



Simplified graph



Results

- Implemented with virtual machine
  - EventLogger is implemented in VMM
- 1.2G log file in 24 hours
- Three attacks are tracked
- GraphGen can take up to 3 hours for tracking
  - 26 seconds if irrelevant events are ignored

Limitations of BackTracker

- If an attacker can change guest kernel
  - E.g. load module
- If an attack can attain access to VMM
  - Very difficult
- If use hidden channel
  - Send password through internet
  - Login with the right password
- Hide actions in a huge dependency graph
- Hide actions by intermingling with innocent events
- Prolonged the attack period