# Hijacking (Xen) Virtual Machine for Fun and Profits

## Bellua Security Conference 2007

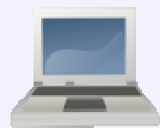

NGUYEN Anh Quynh

<aquynh –at- gmail com>

National Insitute of Advanced Industrial Science and Technology, Japan

# Who am I?

- Nguyen Anh Quynh, postdoctoral researcher of National Institute of Advanced Industrial Science and Technology (AIST), Japan. Member of VnSecurity group.
- Interests: Network/Computer Security, Data forensic, Trusted Computing, Operating system, Virtualization

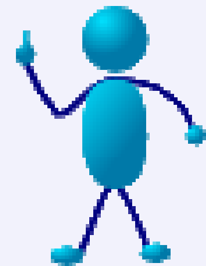


# Motivation

- What can we do if we can take over a host Virtual Machine (VM) with multiple VMs running?
- Explore techniques to dynamically inject code into any running VM to hijack its execution, in order to inspect and capture sensitive data.
  - Focus on Xen Virtual Machine case.
  - Can be done quietly and secretly without awareness of VM's owner.
  - Require absolutely no modification to hijacked VM or underlying hypervisor.
  - Implementation done in user-space.
  - OS independence.

# Agenda

- Background on Xen Virtual Machine and Xen debugging facility.
- Hijacking VM execution techniques.
- Performance evaluation.
- 2 demos.
- Cat & Mouse game.
- Related Works.
- Conclusions.
- Q & A.

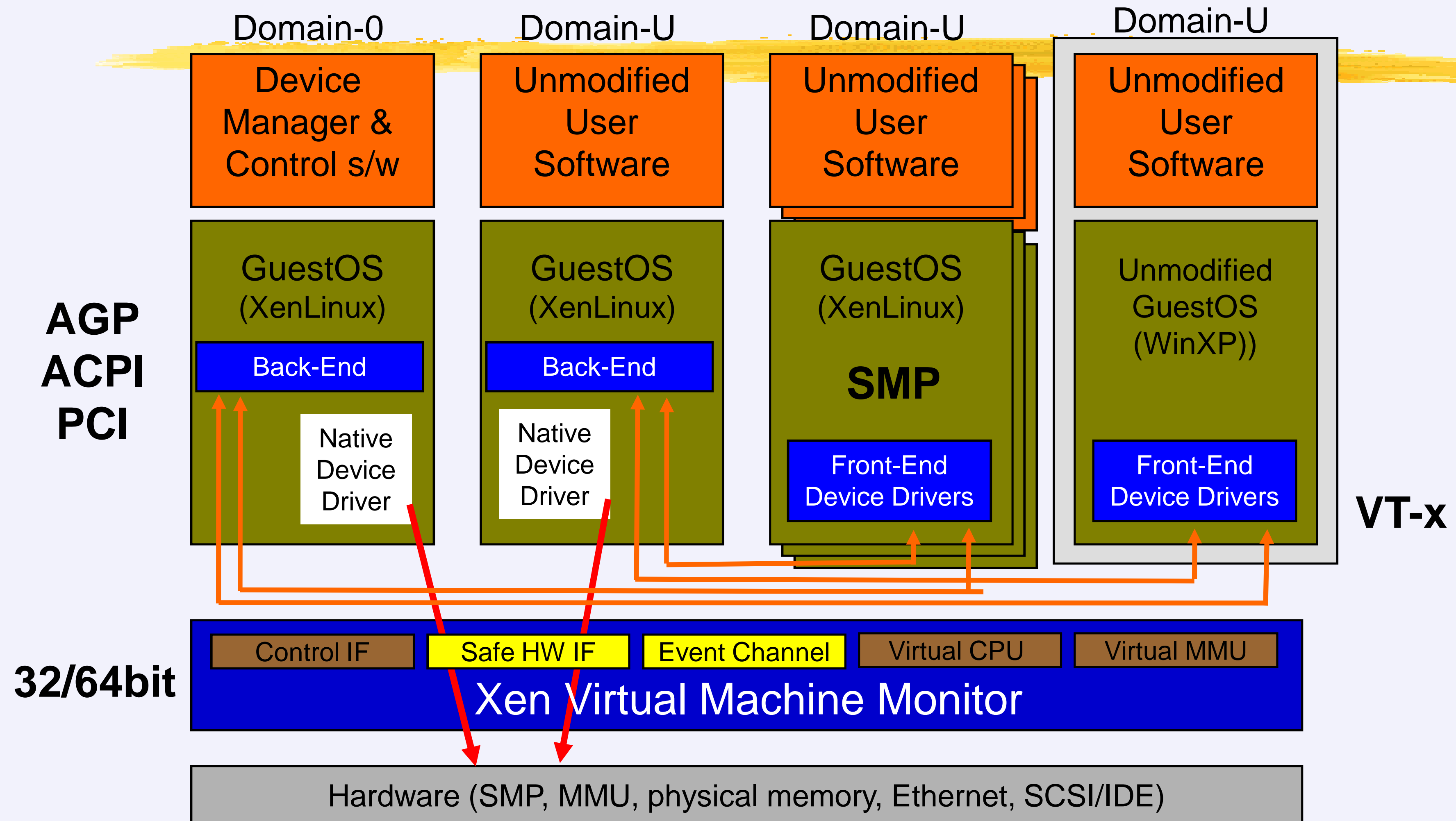


# Part I

## Background

- Xen Virtual Machine Architecture
- Xen debugging architecture

# Xen 3 Architecture



# Xen's Future: Bright

- Xen 3.0 was realeased at the end of 2005
- Object: to be gradually merged into Linux kernel in 2007
  - In mainline kernel from 2.6.23? (October 2007)
- Already adopted by ISPs, Data centers, E-commerce, banks,...
- Will be widely used in the future

# Xen Virtualization

- Para-virtualization
  - Make VM aware of virtualization
    - CPU virtualization on special platform “xen”
    - Special IO drivers for better performance
    - Require OS kernel customized and recompiled
- Full-virtualization
  - Virtualize OS without any modification to OS kernel
    - Need hardware support (Intel VT, AMD SVM)

# Debugging Support on Intel

- INT1 & INT3 on Intel architecture for debugging
  - INT1: Debug interrupt
    - Single-step trace mode
    - Turn ON TF flag in EFLAGS
  - INT3: Breakpoint interrupt
    - 0xCC Instruction

# Debug handling in Xen

Handling INT1/INT3 in Xen

- When a breakpoint (0xCC) is hit
  - Exception #BP raised (INT3)
  - Hyperswitch to INT3 handler in hypervisor
    - Xen hypervisor intercepts interrupt rather than let above VM do that
  - Hypervisor checks to see if VM is in user mode?
    - Yes, return control back to VM
  - No (→in kernel mode), pause VM for debugger (gdb?) to come to inspect
- Handled similarly with INT1 case (triggered by TF flag).

# Part II

## Hijacking VM Execution Techniques

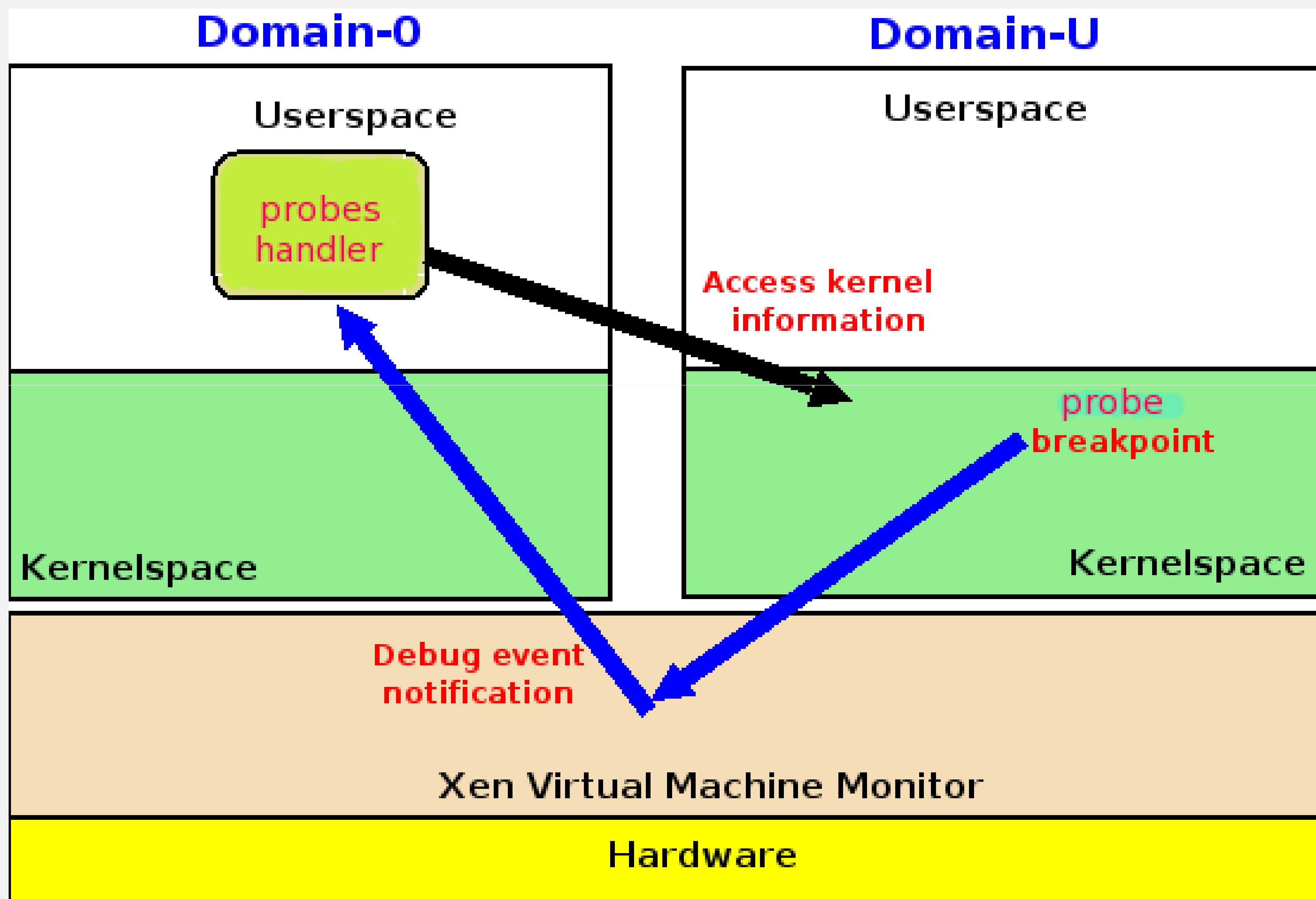
- General technique
- Technical issues
  - Performance penalty
  - Injecting breakpoint place
  - Handling breakpoint event

# Hijacking VM Execution

Employ the Xen debugging infrastructure to hijack VM execution at run-time

- Define our breakpoint handlers and associate them with breakpoints → a probe
  - Probe point: Inject breakpoint instruction here
  - Probe handler: Handle breakpoint event
- Run handlers in user-space of Dom0
- Let them handle corresponding breakpoint events

# Xenprobes Architecture



# Injecting Breakpoint

- Insert software breakpoints at the right place into VM
  - Breakpoint put into VM kernel at run-time
  - Associate breakpoint with handler
    - Handler defined by us, and run in user-space of Dom0

# Handling Breakpoint Event

- Find corresponding handler of this breakpoint event
  - EIP (which indicates breakpoint address) is available
- Execute the handler in Dom0's user-space
  - Inspect/capture/manipulate VM
- Resume VM (paused at that time)

# Implementation Issues

- Pick up breakpoint event ?
- Where to inject the breakpoint ?
- Handling breakpoint event ?

# Pick up Breakpoint Event

Exploit of a speacial feature of Xen debugging technique

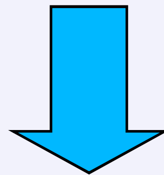
- Xen always sends an event to Dom0 to notify potential debugger
- Put VM in debugging mode
  - Xen hypercall with XEN_DOMCTL_setdebugging command
- Bind our handler to virtual interrupt VIRQ_DEBUGGER
- Poll for this interrupt to detect breakpoint event

# Challange on Injecting Breakpoint

- Where to put the breakpoints?
- Look at the source code to get the idea where is the appropriate place to put breakpoints
  - Kernel compiled with debugging information → kernel binary with debugged data
    - Retrieve information from DWARF data format
  - Disassemble kernel binary to verify the correctness
  - Kernel symbol file accompanied kernel binary
    - /boot/System.map file

# Handling Breakpoint Issues

- Original instruction at breakpoint address must be saved
- Original instruction must be executed after running the handler



2 schemes to handle original instructions

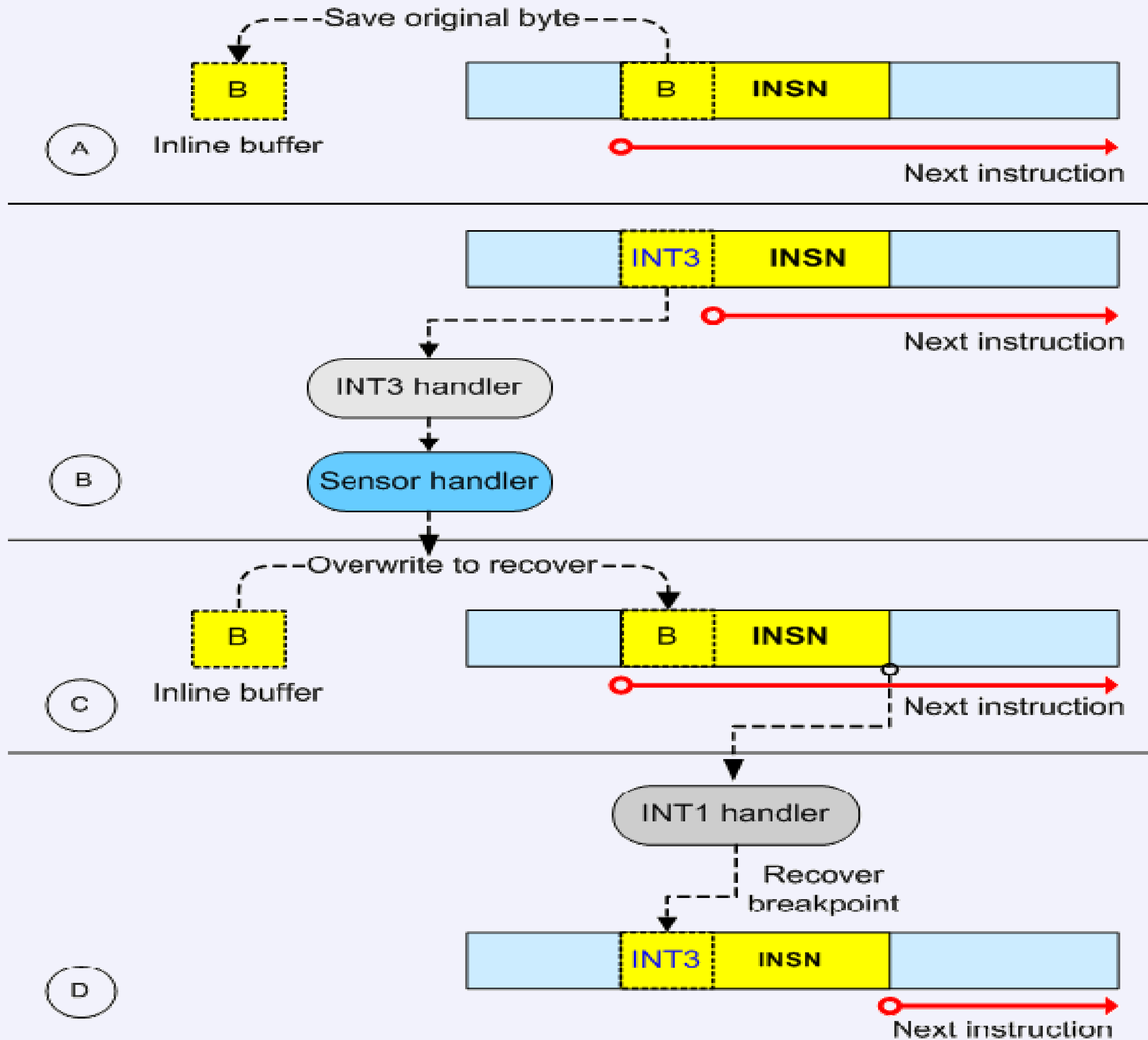
- Inline-Execution scheme
- Outline-Execution scheme

# Inline-Execution Scheme (1)

Execute original instruction at the original place

- Preparation
  - Save the original byte overwritten by the breakpoint
- Handling breakpoint-event
  - Overwrite the breakpoint instruction with the original byte
  - Decrease EIP by 1, then put VM into single-step mode, and resume VM
  - In the single-step event, recover the breakpoint and disable single-step mode, then resume VM

# Inline-Execution Scheme (2)



# Outline-Execution Scheme (1)

Execute original instruction in separate area

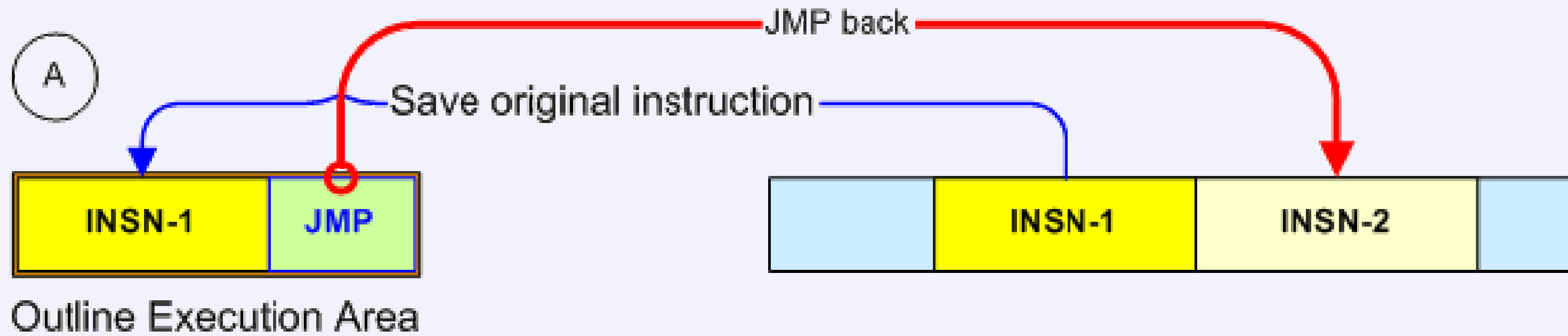
- Preparation
  - Copy original instruction to a separate area, called Outline-Execution-Area (OEA)
    - OEA must be big enough for an instruction and a JMP instruction
    - JMP instruction jumps to the instruction next to the original instruction

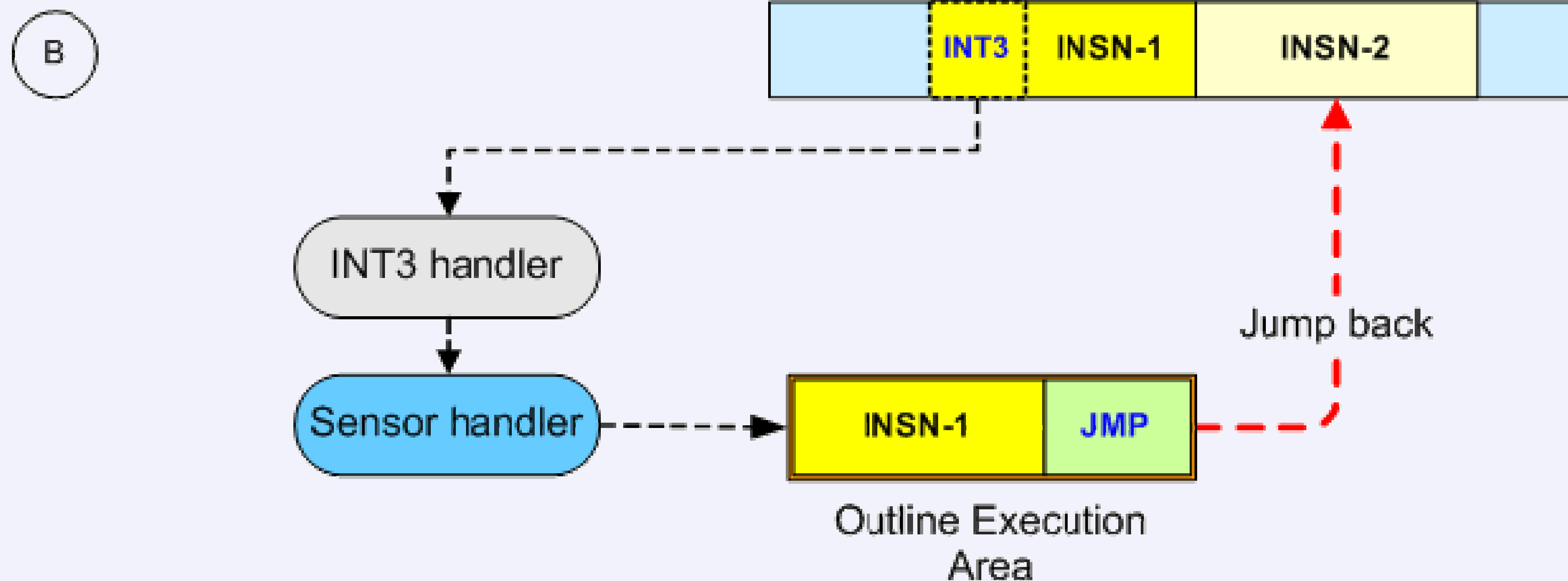
# Outline-Execution Scheme (2)

Execute original instruction in a separate area

- Handling breakpoint event
  - Point EIP to the corresponding OEA area
    - Execute original instruction and jump to the instruction right after it
  - Resume hijacked VM

# Outline-Execution Scheme (3)



# OEA Allocation

OEA: execution buffer of original instruction

- Where to get the OEA memory?
  - Must stay inside hijacked VM rather than in Dom0
- Allocation ourself
  - Pre-allocate OEA memory
      - xenprobesU kernel module
  - Split it into number of chunks, and allocate one for each probe
  - OEA address and size transmitted to Dom0
    - Employ Xenstore to send information

# IE versus OE (1)

- IE features?
  - Good
    - No need cooperation from hijacked VM
      - Flexible and easy to deploy
  - Bad
    - Flaw design with SMP machine/preemptive kernel
      - What happen if kernel is preemptive and breakpoint-place is hit when the orginal instruction has been recovered?
        - Breakpoint missed
    - Slow because always requires single-step mode
      - Suffer 2 hyperswitches each time when a breakpoint is hit

# IE versus OE (2)

- OE features?
  - Good
    - Work well with SMP/preemptive kernel
    - Higher performance than IE scheme
      - Twice faster because no need single-step mode in most cases
        - It is best to insert breakpoint at "boostable" instruction
  - Bad
    - Need cooperation from hijacked VM
      - Not easy to evade VM's owner

# Part III

## Performance Evaluation

**Native versus IE versus OE**

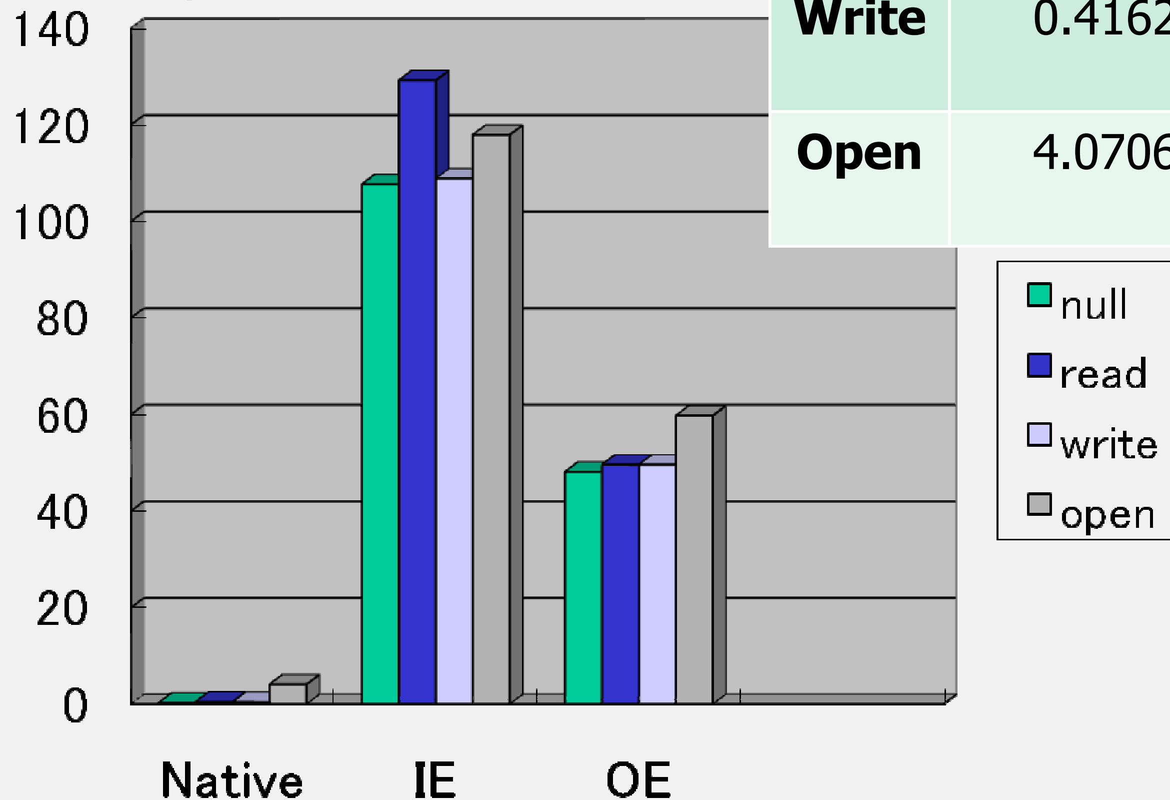
# System Configuration

- (guest) Linux VM, Ubuntu Drake Drapper 6.10, kernel 2.6.18
- Xen 3.0.4
- Thinkpad x60, memory 2GB, SATA HDD
- Dom0
  - Memory: 600MB
- DomU
  - Memory: 400MB
  - Partition: Tap IO, file-based file-system
  - Main partion: 2GB
  - Swap partition: 1GB

# Microbenchmark

- lmbench to measure latency
  - null/read/write/open benchmarks
- Native vs IE vs OE
  - getppid/read/write/open system-calls

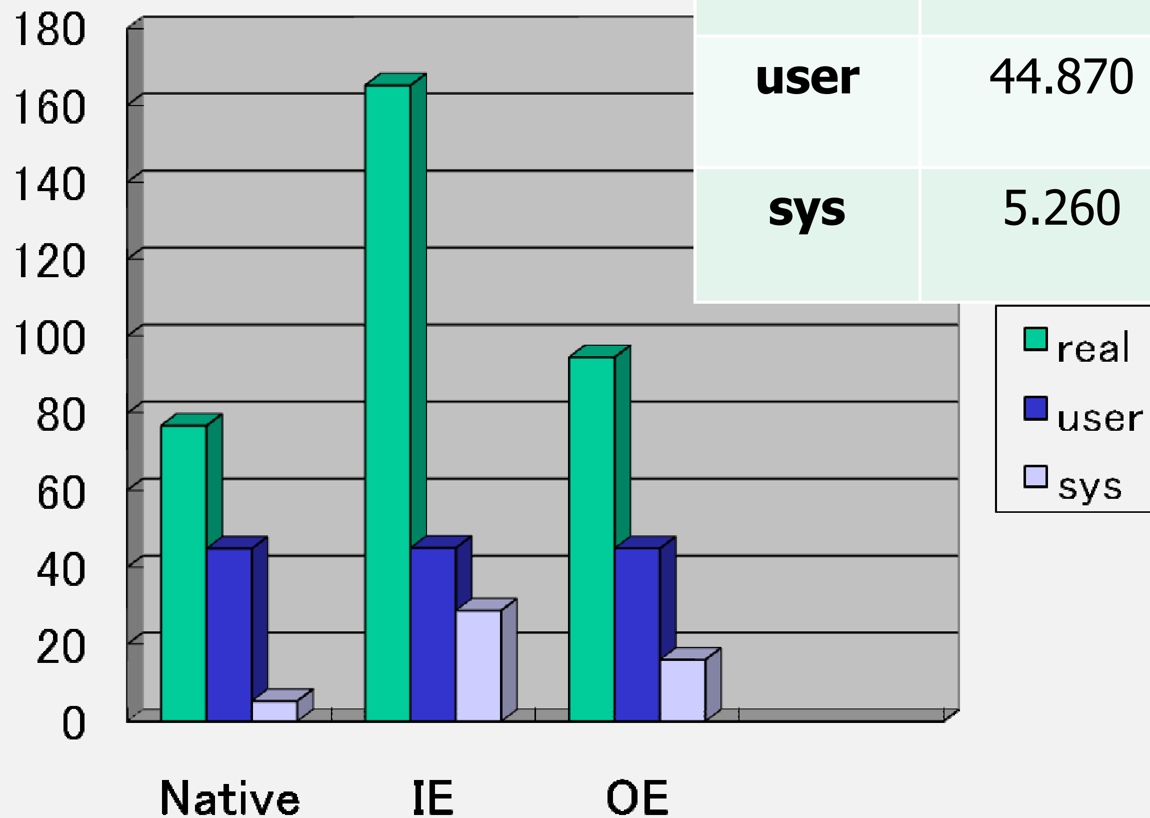
| | Native | IE | OE |
|---|---|---|---|
| **Null** | 0.2664 | 107.6732 (404.17) | 48.109 (180.55) |
| **Read** | 0.4732 | 129.1951 (273.02) | 49.6081 (104.83) |
| **Write** | 0.4162 | 108.8627 (261.56) | 49.6027 (119.19) |
| **Open** | 4.0706 | 117.8936 (28.96) | 59.7527 (14.67) |



# Macrobenchmark

- Unzip Linux kernel source code
  - time tar xjvf linux-2.6.17.tar.bz2
- Native vs IE vs OE
  - mkdir/chmod/open/read/write system-calls

| | Native | IE | OE |
|---|---|---|---|
| **real** | 76.781 | 165.187 (**115.14%**) | 94.572 (**23.17%**) |
| **user** | 44.870 | 45.050 | 44.930 |
| **sys** | 5.260 | 28.800 (**449.04%**) | 16.000 (**204.18%**) |



# Part IV

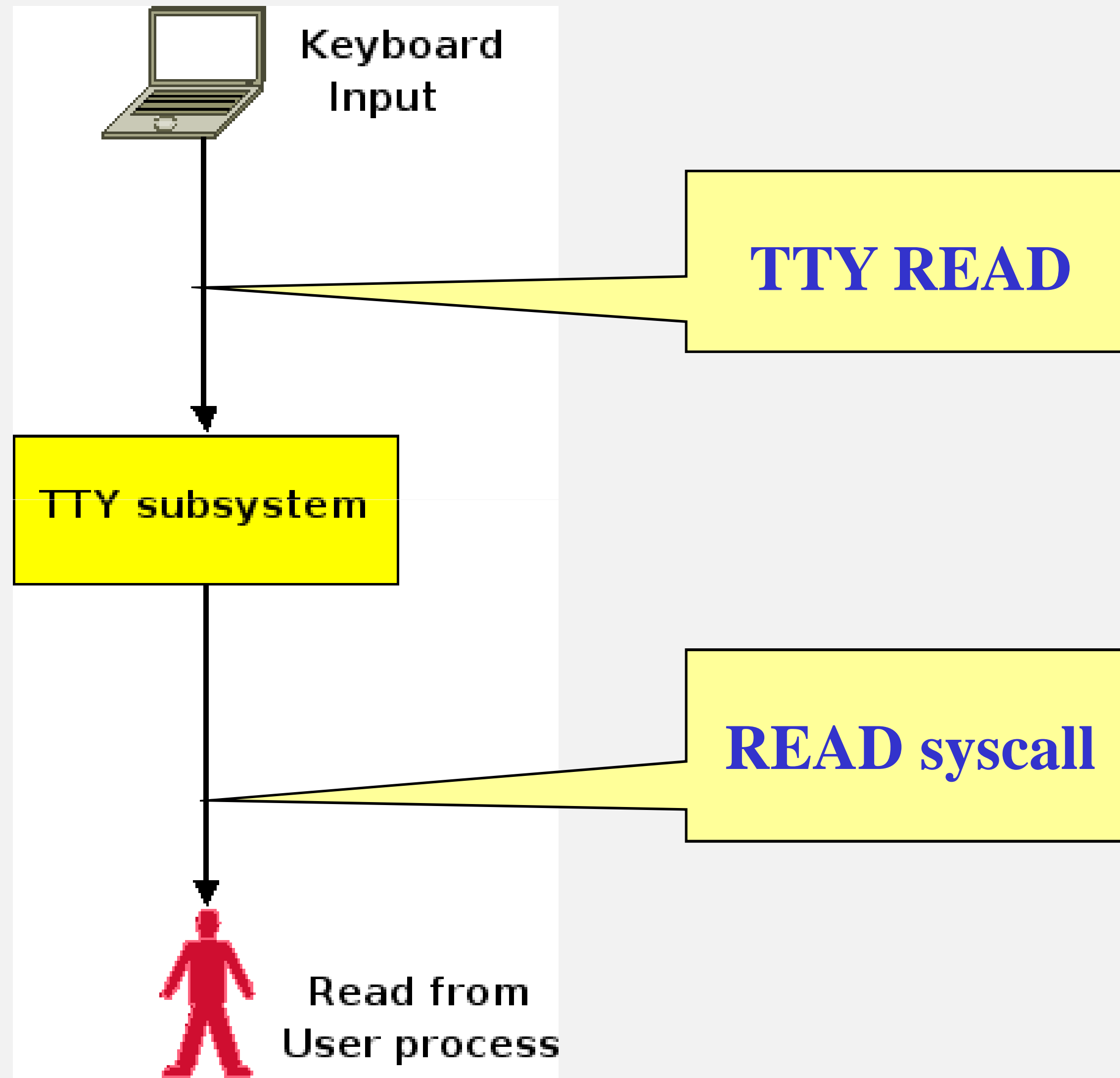
## Demos

1. **XenKamera**: Capture/replay system consoles activities
   - Keystrokes/output screen
2. **XenRIM**: Real-time file-system IDS
   - Verify IO activities against security policy

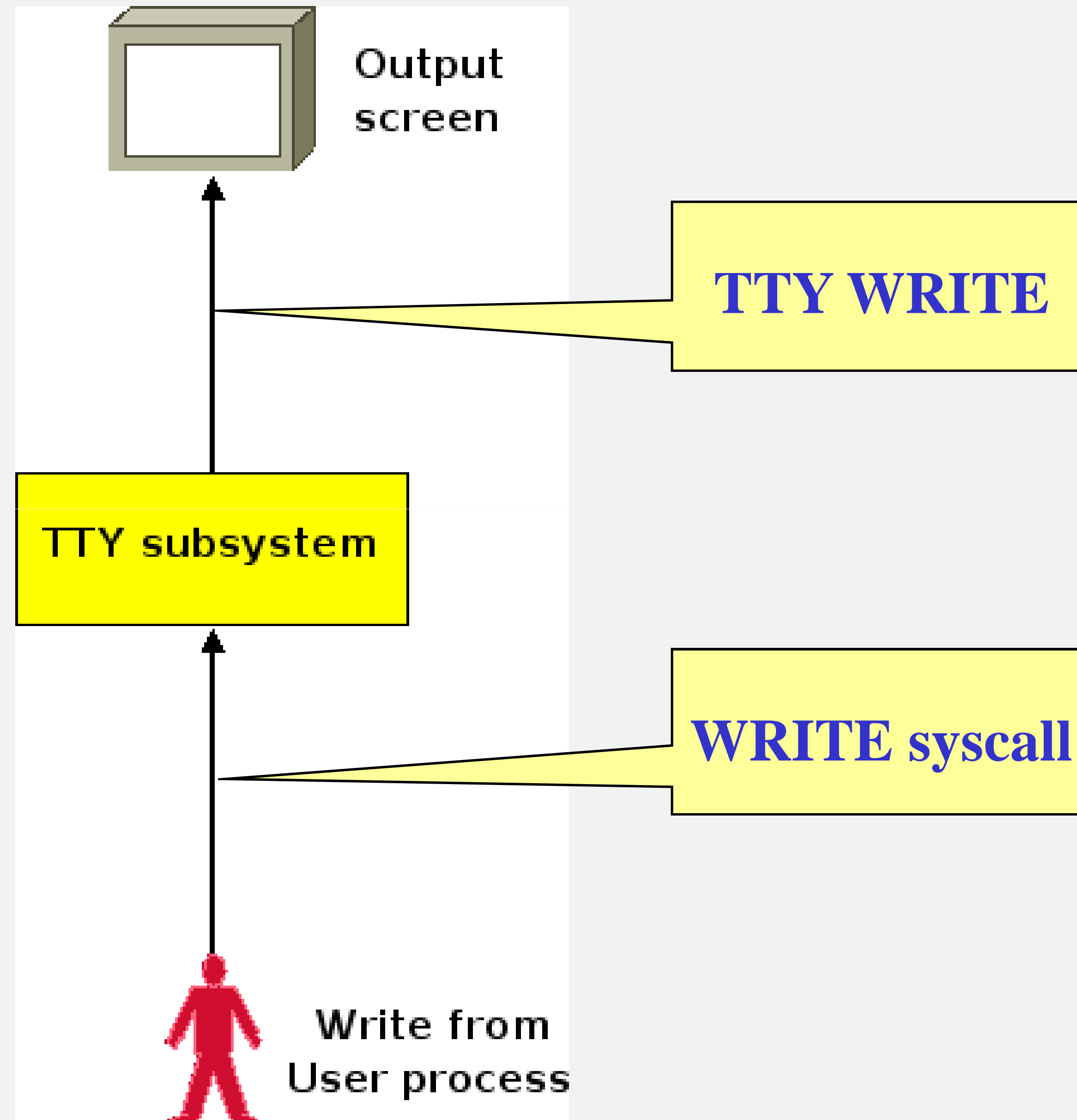
# Demo 1 – Blackhat Scenario

- XenKamera: Capturing and replaying keystrokes/output screen of VM's consoles
  - Hijack TTY subsystem to capture keystrokes/output screen
    - close/read/write to console devices
  - Replay captured data later

# TTY input scheme



# TTY output scheme

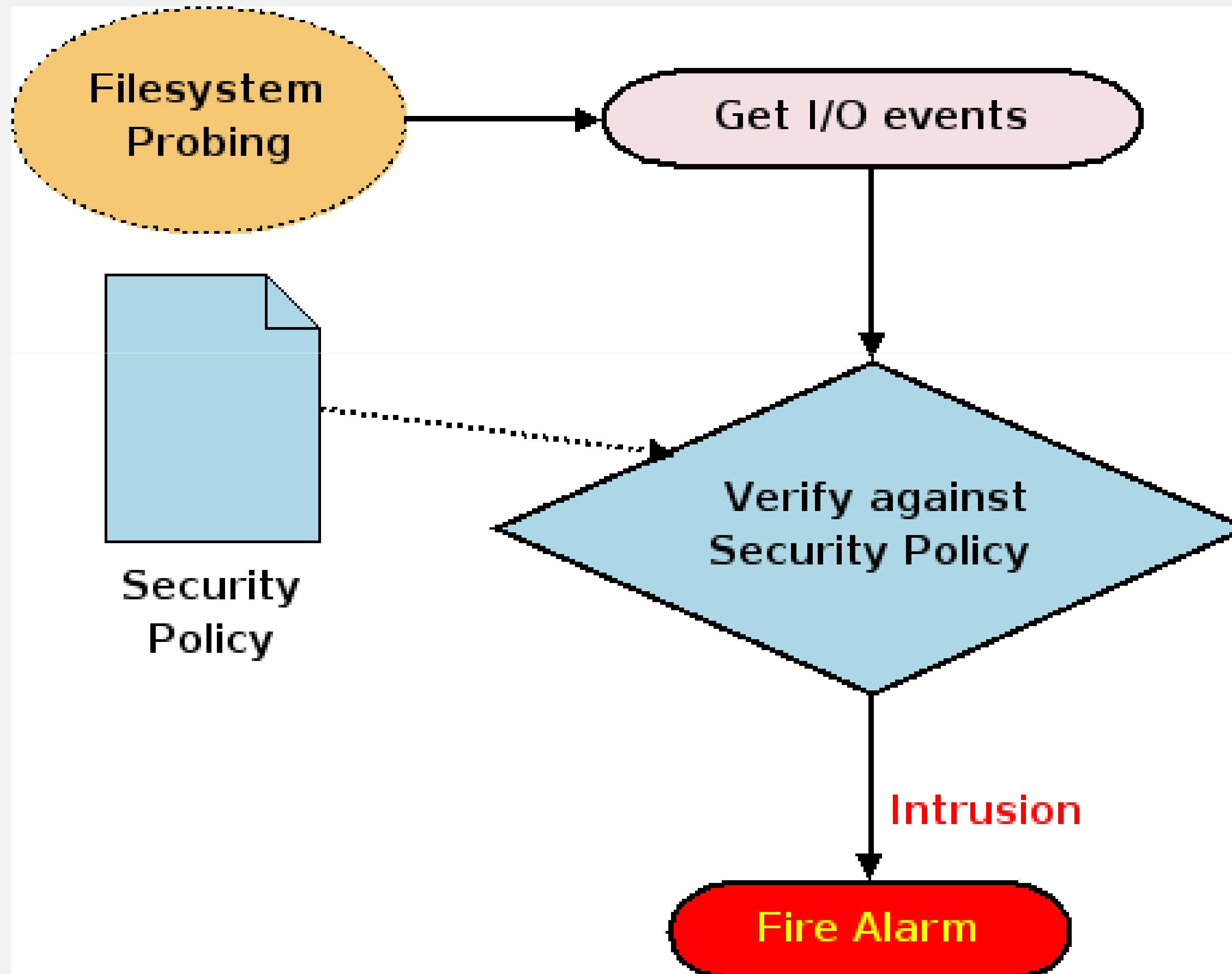


# Demo 2 – White-hat Scenario

- XenRIM: a real-time file-system based IDS
  - Hijack I/O file-system to capture file-system events
    - mkdir/rmdir/write/chmod/chown/hard-link/sym-link/unlink/rename
  - Verify against security policy to detect illegal access/modification
    - All files/dirs in {/etc, /boot, /sbin, /usr/sbin, /usr/bin} should not be modified.

# XenRIM Detect Intrusion

[2007-6-31 08:44:54] UNLINK /etc/issue 2679, uid 1 ➔ POLICY VIOLATION



# Simple policy

- No write to critical directories
- No change to any things in critical directories
- /bin: system binaries (for users)
- /sbin: system binaries (for root)
- /etc: system configuration files
- /boot: kernel binaries and boot loader

# Real-time IDS vs Tripwire

- Advantages
  - Real-time detection
    - Get notified immediately when incident happens
  - Zero-cost deployment
    - No need baseline database
  - Richer intrusion evidence
    - Information about environment available at incident time
  - More invisible to attacker
    - No code running in user-space
  - Attack resistance
    - No (IDS binary & security policy) staying inside VM.
- Disadvantages
  - Any? ☺

# Part V

## Cat & Mouse game

**Detect and Anti-detect VM hijacking**

# Detect Hijacking from Host VM

- Hijacked VM must be in Debugged Mode
  - Suspicious VM in Debugged Mode for no reason?
  - Which processes access /dev/xen/evtchn?
    - Can be tricked by rootkit in Dom0
      - Rootkit detection problem

# Detect Hijacking from Guest VM

- Has kernel access?
  - Scan for 0xCC at abnormal place
  - Look for suspicious kernel module of OEA
- No kernel access?
  - Microbenchmark to detect hight latency
    - But what to benchmark?
  - Look for suspicious kernel module of OEA
    - But kernel module can be hidden

# Anti-detect Techniques (1)

Blackhat view

- IE needs nothing to be loaded inside hijacked VM
  - But more vulnerable to latency benchmark
- OE cannot be used by attacker?
  - Yes, it is possible ☺
  - Exploit unused kernel code for OEA ☺
    - Unused system-calls (vm86old, olduname, oldolduname, oldfstat, oldfstat, oldstat)
      - Can be detected with crafted applications that call these system-calls
  - Exploit padding memory in kernel module ☺
    - Kernel module can be unloaded?
      - Not with critical kernel modules

# Anti-detect Techniques (2)

White-hat view

- Intruder can detect OE by looking for LKM?
  - LKM can be unloaded, as we only need to allocate memory ☺
    - Allocate memory for OEA
    - Inform Dom0 using XenBus/XenStore
    - Dom0 removes related XenStore nodes after picking up information
    - Unload LKM without deallocating memory

# Part VI

## Related Works

# Related Works (1)

- K.Asrigo et al, Virtual machine-based honeypot monitoring, Proceedings of Virtual Execution Environment 2006
  - Insert breakpoint handlers into hypervisor layer
    - Handler cannot be easily programmed and modified
    - Require modification to hijacked VM in source code
      - New hypercall to send security policy to hypervisor
    - Require modification to hypervisor
      - Accommodate handlers & new hypercall

# Related Works (2)

- Kprobes framework, Linux kernel
  - Probe handler must be in kernel code
    - Kernel-space programming is restricted and complicated
  - Not easy to transmit recorded information to out of probed VM
- A.Mavinakayanahalli et al, Probing the Guts of Kprobes, Proceedings of the Linux Symposium 2006
  - Present the architecture and implementation of Kprobes in Linux kernel

# Related Works (3)

- Nitin A.Kamble et al, Evolution in Kernel debugging using hardware virtualization with Xen, Proceedings of Linux Symposium 2006
  - Present the infrastructure supported for debugging Xen VMs
- N.A.Quynh et al, Xenprobes, a lightweight user-space framework for Xen Virtual Machine, Proceedings of Usenix Annual Technical Conference 2007
  - Develop OE scheme to be a framework → xenprobes
    - Aim for purpose of debugging/profiling VM
    - Available as a user-space library, ready to use
    - To be released under GPL license

# Xenprobes Framework

- Xenprobes code is available in a library
  - libxenprobes 0.2
    - C library
  - Going to be released soon under GPL license
    - **http://xenprobes.sf.net**
  - Works well with all Xen 3.x version
    - Xen 2.x is not supported because Xen 2 handles breakpoints differently
      - Possible but not desired
  - Samples available
  - I386 supported. X86_64 on the work

# Part VII

## Conclusions

# Conclusions

- **It is possible to hijack VM execution without awareness of VM's owner**
  - No need coopeartion from VM
  - OS independence and OS configuration independence
  - Done in user-space → easy to implement
- **As a customer, should we trust our rented VM**?
  - No, as everything happening in our VM can be monitored. The control stay in hosted VM instead of in our hand.
- **Lesson learned: Keep your host VM (Dom0 in Xen case) as secure as you can!!!**

# Hijacking (Xen) Virtual Machine for Fun and Profits

NGUYEN Anh Quynh
<aquynh –at- gmail com>

Question/Comment ?