Ramooflax

das pre-boot übervisor

Stéphane Duverger

EADS

Innovation Works
Suresnes, France

PAC SEC

Tokyo, Nov. 2011

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

We wanted a tool . . .

- to have control over complex systems (bios, kernel, . . .)
- running on a **physical** machine (x86 32 and 64 bits)
- without any software dependencies

We wanted a tool . . .

- to have control over complex systems (bios, kernel, . . .)
- running on a **physical** machine (x86 32 and 64 bits)
- without any software dependencies

The idea

- a hypervisor (VMM) with a dedicated virtual machine (VM)
- remotely controlled
- type 1 (*bare metal*)
  - simple isolation
  - control visible hardware
  - software independenza !
  - require startup before the VM

Overview of available hypervisors

Common solutions

- VirtualBox, KVM: misfit, type 2 (*hosted*)
- Xen: too complex to adapt/deploy

Overview of available hypervisors

Common solutions

- VirtualBox, KVM: misfit, type 2 (*hosted*)
- Xen: too complex to adapt/deploy

Minimalistic solutions

- bluepill, vitriol, virtdbg, hyperdbg ...
- too intrusive, *in vivo* virtualization
- OS dependent

Overview of available hypervisors

Common solutions

- VirtualBox, KVM: misfit, type 2 (*hosted*)
- Xen: too complex to adapt/deploy

Minimalistic solutions

- bluepill, vitriol, virtdbg, hyperdbg . . .
- too intrusive, *in vivo* virtualization
- OS dependent

restart from scratch !

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

A standalone minimalistic hypervisor

Desired specifications

- simple, lightweight, fast and reliable
- small impact on native performances
- based on Intel-VT (vmx) and AMD-V (svm)
- take benefit of existing stuff (BIOS)
- keep simple design/mechanisms into complex software pieces (VMM)
- delegate operational complexity to userland layer remotely controlled (client)

A standalone minimalistic hypervisor

Desired specifications

- simple, lightweight, fast and reliable
- small impact on native performances
- based on Intel-VT (vmx) and AMD-V (svm)
- take benefit of existing stuff (BIOS)
- keep simple design/mechanisms into complex software pieces (VMM)
- delegate operational complexity to userland layer remotely controlled (client)

Targeting *cutting edge* CPUs

- depend upon *recent* hardware virtualization extensions
- especially Intel EPT[a] and AMD RVI
    - simpler code
    - faster vmm
    - reduced attack surface

---

[a] Actually it also depends on Unrestricted Guest feature.

Introduction

Concept
  Specifications
  Architecture

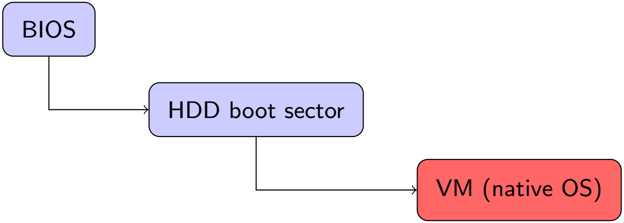
Hardware virtualization
  Overview
  Limitations

Ramooflax internals
  Execution flow
  Filtering
  Emulation
  Communication
  Interaction

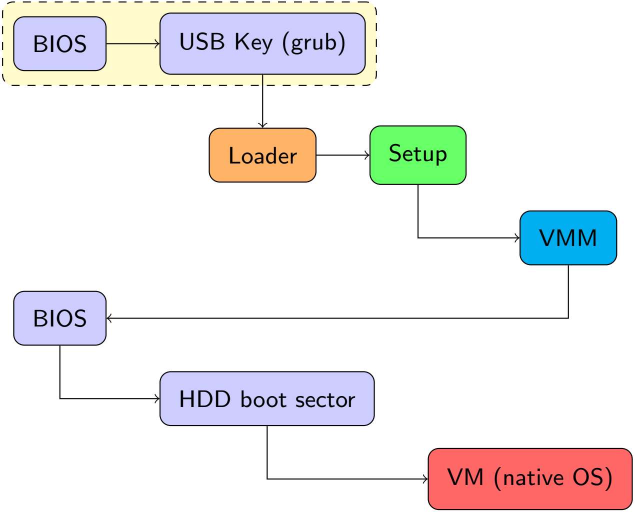
Remote client

Conclusion

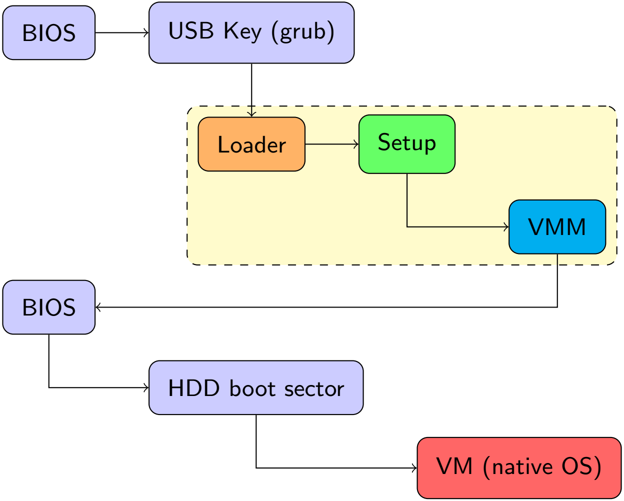
Classical boot sequence



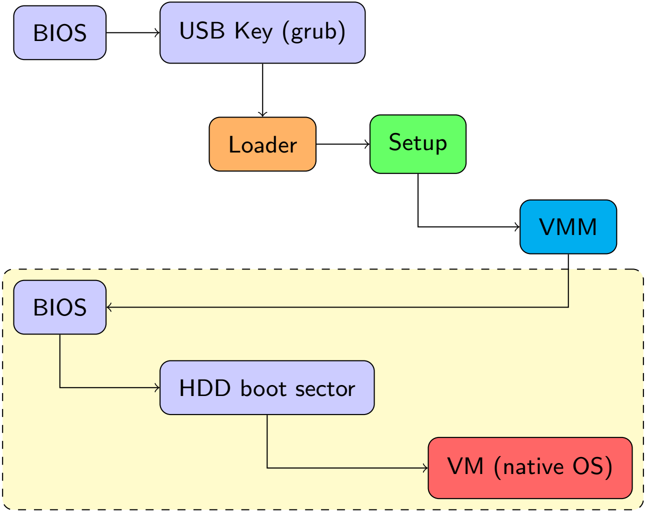
Ramooflax boot sequence



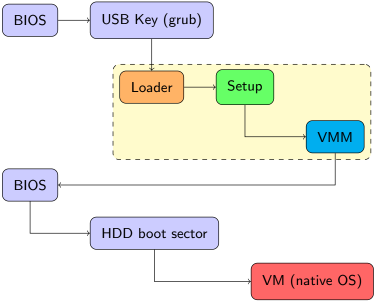
Ramooflax boot sequence



Ramooflax boot sequence



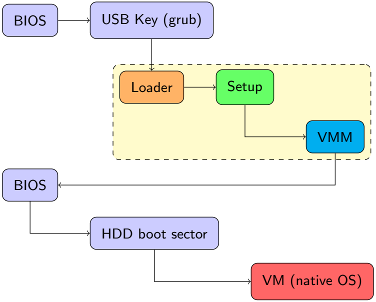
Ramooflax building blocks



Loader

- boots in 32 bits protected mode (multiboot standard)
- enters *longmode* (64 bits) then loads Setup

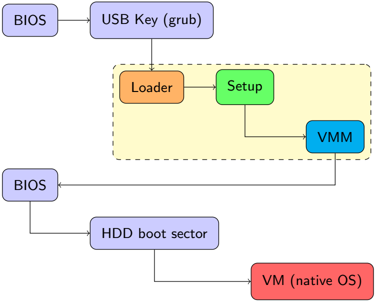
Ramooflax building blocks



Setup

- initializes virtualization structures, drivers, memory
- retrieves RAM size and computes VMM needed space
- relocates vmm to `size(RAM) - size(vmm)`
- reduces RAM size (craft special VM SMAPs)
- installs `int 0x19` into conventional memory
- invokes vmm

Ramooflax building blocks



VMM *resident*

- PIE binary (variable RAM size)
- starts its dedicated VM in real mode on `int 0x19`
- tells the BIOS (virtualized) to start native OS

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

Common points between Intel-VT (vmx) and AMD-V (svm)

Interest

- simplify hypervisor development
- reduced instruction set (~ 10)
- vm-entry/vm-exit paradigm
    - vm-entry loads VM and saves VMM
    - vm-exit loads VMM and saves VM

Relies upon data structures configuration

- AMD VMCB, Intel VMCS (asynchronous `vmread, vmwrite`)
- system registers setup (`cr, dr, gdtr, idtr, ...`)
- events injection (interrupts, exceptions)
- interception bitmaps setup
    - events
    - sensitive instructions
    - I/O, MSRs . . . accesses

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

Many limitations

- Compatibility fail between Intel/AMD
- different features among CPU models
- hard to obtain CPU skills before buying it ! `http://cpuid.intel.com` ?

Many limitations

- Compatibility fail between Intel/AMD
- different features among CPU models
- hard to obtain CPU skills before buying it ! `http://cpuid.intel.com ?`
- lack of information after vm-exit
- need to embed an emulation/disassembly engine
- hardware interrupts interception is on/off . . . no vector granularity
- Intel does not provide software interrupts interception
- AMD keeps hardware interrupts *pending*
- SMIs headache (CPU bugs, BIOS bugs, SMM virtualization needed, . . .)

Real mode management disaster under Intel
painfull for *real-life* BIOS virtualization !

BIOS virtualization

BIOS and real mode

- 16 bits default CPU mode
- 20 bits (1MB) memory addressing, no protection
- massively used by the BIOS

Real mode virtualization *the merovingian way*

- harware assisted virtualization exists since 80386: v8086 mode
- real mode mechanisms emulation (interrupts, far call, . . .)
- redirect/intercept I/O, interrupts

BIOS virtualization

BIOS and real mode

- 16 bits default CPU mode
- 20 bits (1MB) memory addressing, no protection
- massively used by the BIOS

Real mode virtualization *the merovingian way*

- harware assisted virtualization exists since 80386: v8086 mode
- real mode mechanisms emulation (interrupts, far call, . . .)
- redirect/intercept I/O, interrupts

Real mode virtualization *the vmx/svm way*

- AMD provides a new *paged real mode* (`CR0.PE=0 && CR0.PG=1`)

BIOS virtualization

BIOS and real mode

- 16 bits default CPU mode
- 20 bits (1MB) memory addressing, no protection
- massively used by the BIOS

Real mode virtualization *the merovingian way*

- harware assisted virtualization exists since 80386: v8086 mode
- real mode mechanisms emulation (interrupts, far call, . . .)
- redirect/intercept I/O, interrupts

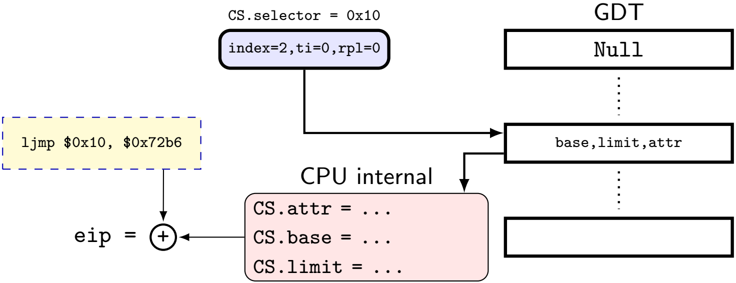
Real mode virtualization *the vmx/svm way*

- AMD provides a new *paged real mode* (`CR0.PE=0 && CR0.PG=1`)
- Intel forbids `CR0.PG=0` and so `CR0.PE=0`
  - recommands the use of v8086 mode
  - vm-entry while in v8086 is very restrictive
  - especially with regard to segmentation

Segmentation reminder

Segment registers

- visible part (selector)
- hidden part managed by the CPU (base, limit, attributs)
- real mode: $\textit{base} = \textit{selector} * 16$, $\textit{limit} = 64K$
- protected mode: segment descriptors



BIOS virtualization

Unreal mode (*flat real, big real mode*)

- access more than 1MB of memory while in real mode
- protected to real mode transition keeping ie $base = 0$ and $limit = 4GB$
- used by the BIOS to access memory mapped devices, . . .

```
seg000:F7284          mov     bx, 20h
seg000:F7287          cli
seg000:F7288          mov     ax, cs
seg000:F728A          cmp     ax, 0F000h
seg000:F728D          jnz     short near ptr unk_7297
seg000:F728F          lgdt    fword ptr cs:byte_8163     (1)
seg000:F7295          jmp     short near ptr unk_729D
seg000:F7297          lgdt    fword ptr cs:byte_8169
seg000:F729D          mov     eax, cr0
seg000:F72A0          or      al, 1
seg000:F72A2          mov     cr0, eax                   (2)
seg000:F72A5          mov     ax, cs
seg000:F72A7          cmp     ax, 0F000h
seg000:F72AA          jnz     short near ptr unk_72B1
seg000:F72AC          jmp     far ptr 10h:72B6h          (3)
seg000:F72B1          jmp     far ptr 28h:72B6h
seg000:F72B6          mov     ds, bx                     (4)
seg000:F72B8          mov     es, bx
seg000:F72BA          mov     eax, cr0
seg000:F72BD          and     al, 0FEh
seg000:F72BF          mov     cr0, eax                   (5)
seg000:F72C2          mov     ax, cs
seg000:F72C4          cmp     ax, 10h                    (6)
seg000:F72C7          jnz     short near ptr unk_72CE
seg000:F72C9          jmp     far ptr 0F000h:72D3h
seg000:F72CE          jmp     far ptr 0E000h:72D3h
```

BIOS virtualization

Intel failure

- vm-entry in v8086 mode checks[1] $base = selector * 16$
- can not virtualize unreal mode using v8086

---

[1] Intel Volume 3B Section 23.3.1.2

BIOS virtualization

Intel failure

- vm-entry in v8086 mode checks[1] $base = selector * 16$
- can not virtualize unreal mode using v8086

With basic hardware virtualization extensions

- real mode emulation while in protected mode
- intercept segment registers accesses: *far call/jump*, *mov/pop seg*, *iret*
- double fail: Intel does not provide segment registers interception
- solution: force GDT and IDT limits to 0 and intercept raised `#GP`

---

[1] Intel Volume 3B Section 23.3.1.2

BIOS virtualization

Intel failure

- vm-entry in v8086 mode checks[1] $base = selector * 16$
- can not virtualize unreal mode using v8086

With basic hardware virtualization extensions

- real mode emulation while in protected mode
- intercept segment registers accesses: *far call/jump*, *mov/pop seg*, *iret*
- double fail: Intel does not provide segment registers interception
- solution: force GDT and IDT limits to 0 and intercept raised `#GP`

With newer CPUs (Westmer)

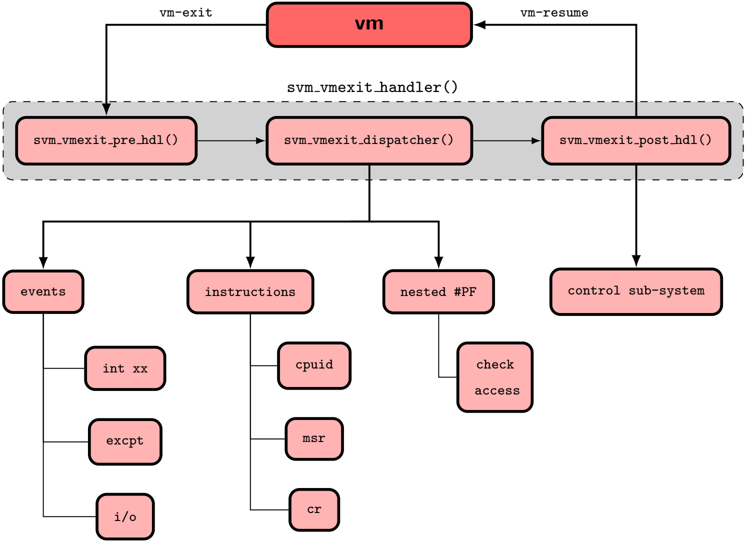
- *Unrestricted Guest* mode (allow `CR0.PE=0 && CR0.PG=0`)
- need Intel EPT to protect over VMM memory

---

[1] Intel Volume 3B Section 23.3.1.2

- Introduction
- Concept
  - Specifications
  - Architecture
- Hardware virtualization
  - Overview
  - Limitations
- Ramooflax internals
  - Execution flow
  - Filtering
  - Emulation
  - Communication
  - Interaction
- Remote client
- Conclusion

Execution flow (AMD one)



Introduction

Concept
  Specifications
  Architecture

Hardware virtualization
  Overview
  Limitations

Ramooflax internals
  Execution flow
  Filtering
  Emulation
  Communication
  Interaction

Remote client

Conclusion

System registers filtering

Control Registers

- cr0 for mode transitions, cache consistency and memory mappings
- cr3 for remote control (*more on this later*)
- as a remote client feature

reading MSRs and CPUID

- native execution or backed VMCS/VMCB reading
- postprocessing to hide specific features

writing MSR

- emulate `wrmsr` if backed to VMCS/VMCB
- else native execution

Events filtering

Exceptions

- fine grain interception of `#DB` and `#BP` mainly for control sub-system
- filter `#GP` under Intel for specific software interrupts interception

Software interrupts

- only in real mode
- filter SMAPs accesses (`int 0x15`)

Hardware interrupts

- not intercepted
- . . . but you can do it

- Introduction
- Concept
  - Specifications
  - Architecture
- Hardware virtualization
  - Overview
  - Limitations
- Ramooflax internals
  - Execution flow
  - Filtering
  - Emulation
  - Communication
  - Interaction
- Remote client
- Conclusion

Emulation

Instructions

- disassembly/emulation needed to properly handle vm-exit
- Ramooflax embeds udis86 .... *overkill*
- emulated instructions are simple
- take care of execution context

Devices

- partial emulation/interception of UART, PIC, KBD and PS2 System Controller
- mainly to control *reboot* bits

Introduction

Concept
  Specifications
  Architecture

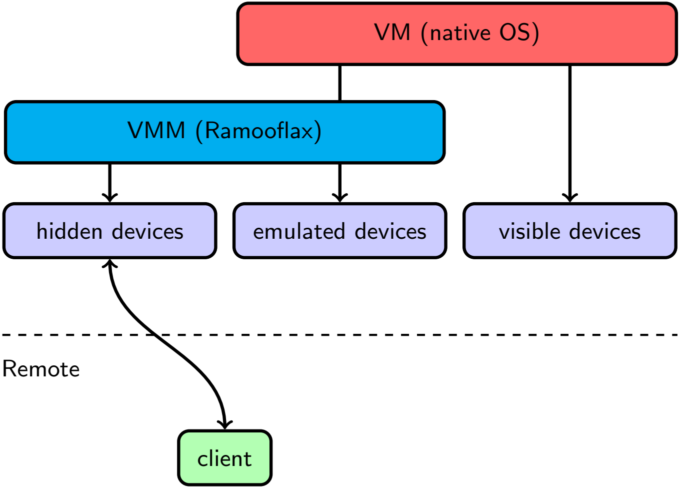
Hardware virtualization
  Overview
  Limitations

Ramooflax internals
  Execution flow
  Filtering
  Emulation
  Communication
  Interaction

Remote client

Conclusion

VMM, VM, client interaction



VM (native OS)

VMM (Ramooflax)

hidden devices

emulated devices

visible devices

Remote

client

Remote communication

UART

- slow, unreliable
- only used for debug purpose

EHCI Debug Port

- USB 2.0 specification tells that a physical USB port can be used as a Debug Port
- found in most of EHCI host controllers
- reliable, standardized and fast
- as simple as an UART to drive

Ramooflax side implementation

- Debug Port driver
- EHCI host controller remains under VM control

Remote communication

EHCI Debug Port: client side

- USB specification: no direct data transfers between host controllers
- Debug Device needed
    - buy a specific device (ie Net20DC)
    - take benefit of USB On-The-Go controllers (*smartphones* . . .)

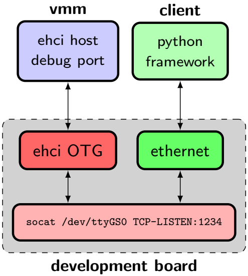
Remote communication

EHCI Debug Port: client side

- USB specification: no direct data transfers between host controllers
- Debug Device needed
  - buy a specific device (ie Net20DC)
  - take benefit of USB On-The-Go controllers (*smartphones* . . .)

Debug Device emulation under Linux

- Gadget API allows USB devices emulation (*mass storage* . . .)
- Debug Device gadget implementation exposing a serial interface (`ttyGS0`)



Introduction

Concept
  Specifications
  Architecture

Hardware virtualization
  Overview
  Limitations

Ramooflax internals
  Execution flow
  Filtering
  Emulation
  Communication
  Interaction

Remote client

Conclusion

Client interaction

Taking control

- VMM waits for vm-exit
- find the good trade off between client reactivity and VM performances
- ensure that VMM can get control over VM on client demand
- recently Intel introduced a *vmx_preemption_timer*, but not AMD

Via hardware interrupts ?

- no irq raised for Debug Port
- complexity, latency, . . .

Context switch

- modern OSes schedule processes
- intercept writes to cr3

Client interaction

GDB stub implementation

- read/write general purpose registers
- read/write memory
- add/remove software and hardware breakpoints
- single-stepping

Protocol limits

- designed for userspace applications debugging
- no ring 0 information (segmentation, paging, . . .)
- no virtual/physical memory distinction

Client interaction

Ramooflax specific extensions

- system registers access
    - cr0, cr2, cr3, cr4
    - dr0-dr3, dr6, dr7, dbgctl
    - cs, ss, ds, es, fs, gs base address
    - gdtr, idtr, ldtr and tr

Client interaction

Ramooflax specific extensions

- system registers access
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6, dr7, dbgctl
  - cs, ss, ds, es, fs, gs base address
  - gdtr, idtr, ldtr and tr
- memory access
  - now virtual/physical distinction
  - translation mechanism
  - fixed cr3 feature (force VMM to work with a specific cr3)

Client interaction

Ramooflax specific extensions

- system registers access
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6, dr7, dbgctl
  - cs, ss, ds, es, fs, gs base address
  - gdtr, idtr, ldtr and tr
- memory access
  - now virtual/physical distinction
  - translation mechanism
  - fixed cr3 feature (force VMM to work with a specific cr3)
- virtualization control
  - control registers intercept
  - exceptions intercept
  - ideally . . . full control over VMCS/VMCB

Client interaction

Single-step management

- based on TF and exceptions intercepts
- many distinct modes under a VM
  - global (*implemented*)
  - kernel thread only
  - ring 3 process only (*implemented*)
  - ring 0/3 process only (follow system calls, . . .)
- no features related to the virtualized OS concepts (process termination)
- stealth/consistency (`pushf`,`popf`,`intN`,`iret` intercept)

Client interaction

Single-step management

- based on TF and exceptions intercepts
- many distinct modes under a VM
  - global (*implemented*)
  - kernel thread only
  - ring 3 process only (*implemented*)
  - ring 0/3 process only (follow system calls, . . .)
- no features related to the virtualized OS concepts (process termination)
- stealth/consistency (`pushf`,`popf`,`intN`,`iret` intercept)

Special case: `sysenter/sysexit`

- uninterceptable under AMD and Intel (!!!)
- do not mask TF when entering ring 0
- need to implement a fault based mechanism (as Intel software interrupts)

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

A python interface to the hypervisor

Framework components

- VM, high-level features
- CPU, registers, exception filtering . . .
- Breakpoints, soft/hard
- GDB, a GDB client with Ramooflax extensions
- Memory, control memory accesses
- Event, *vm-exit hooking* mechanism to implement your own python handlers

Framework components: VM

- run, stop, resume, singlestep, attach, detach

```
vm = VM(CPUFamily.AMD, "192.168.254.254:1234")
```

- interactive mode

```
vm.run(dict(globals(), **locals()))
```

- script mode

```
vm.attach()  # remote connection
vm.stop()    # stop it

# xxxx (breakpoints, filters, ...)

vm.resume()  # resume and wait for next vm-exit
vm.detach()  # disconnect, vm resumed
```

Framework components: CPU, Memory and Breakpoints

- breakpoints naming

```
# data write breakpoint
vm.cpu.breakpoints.add_data_w(vm.cpu.sr.tr+4, 4, filter, "esp0")

>>> vm.cpu.breakpoints
esp0 0xc1331f14 Write (4)
kernel_f1 0xc0001234 eXecute (1)
```

- cr3 tracking feature

```
# reading a virtual memory page
vm.cpu.set_active_cr3(my_cr3)
pg = vm.mem.vread(0x8048000, 4096)
```

Framework components: Event

- GDB conditional breakpoints syntax is . . . hmm
- allow the developer to execute a function after a vm-exit
- split architecture/OS specific mechanisms
- filter an exception, a write to cr3, a breakpoint, . . .

```
def handle_excp(vm):
   if vm.cpu.gpr.eip == 0x1234:
     return True
   return False

vm.cpu.filter_exception(CPUException.general_protection, handle_excp)

while not vm.resume():
   continue

vm.interact()
```

Introduction

Concept
- Specifications
- Architecture

Hardware virtualization
- Overview
- Limitations

Ramooflax internals
- Execution flow
- Filtering
- Emulation
- Communication
- Interaction

Remote client

Conclusion

Conclusion

Support

- AMD and Intel support
- successfully tested under
  - Windows XP/7 Pro 32 bits
  - Debian GNU/Linux 5.0 32/64 bits
- simpler OS should run (DOS, OpenBSD, . . .)

Conclusion

Support

- AMD and Intel support
- successfully tested under
    - Windows XP/7 Pro 32 bits
    - Debian GNU/Linux 5.0 32/64 bits
- simpler OS should run (DOS, OpenBSD, . . .)

Limitations

- Recent CPUs (*Phenom II*, *Westmer/Sandy bridge*)
- no SMP, multi-cores
    - tricky to setup
        - initialize all Cores and enable virtualization
        - intercept Cores initialization done by the VM
    - circumvent
        - BIOS settings
        - kernel parameters `/numproc`, `maxcpus`
- no *Nested Virtualization*

Thank you !

`https://github.com/sduverger/ramooflax`