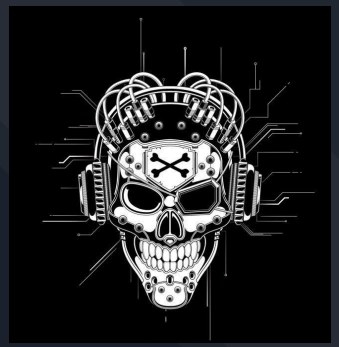
# Building a {Flexible} Hypervisor-Level Debugger



Insomni'Hack 2019

Mathieu Tarral

# Whoami

- ex F-Secure fellow
- Building stealth, hypervisor-based sandboxes
- Virtual Machine Introspection

https://github.com/**KVM-VMI/kvm-vmi**

@mtarral

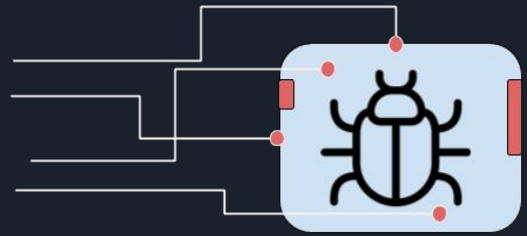
# Agenda

- Why ?
- History
- Challenges
- Demo
- HyperBreakpoints
- Future

# Why ?

# Debuggers observer effect

- Advanced malware analysis
- No stealth
  - changes in exposed structures
  - altered syscall behavior
  - visible breakpoints
- No robustness
  - how to protect the debugger against code modifications ?
  - runs at the same **privilege** level as the malware (malicious driver)

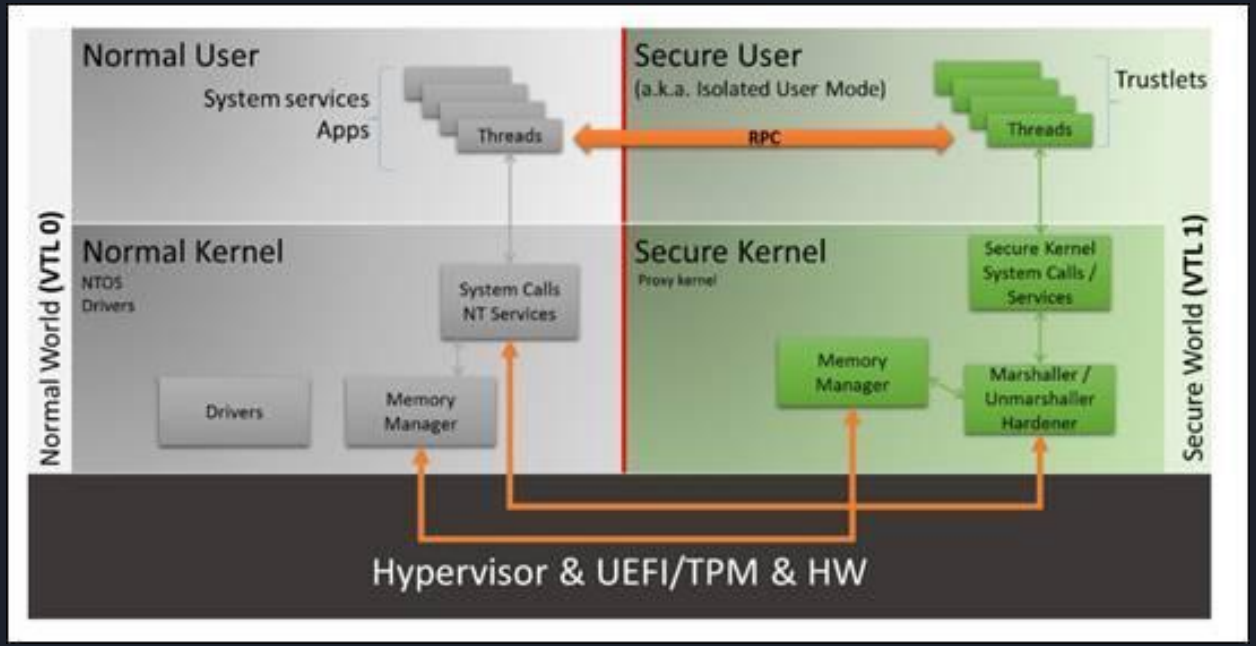


# Protected OS features

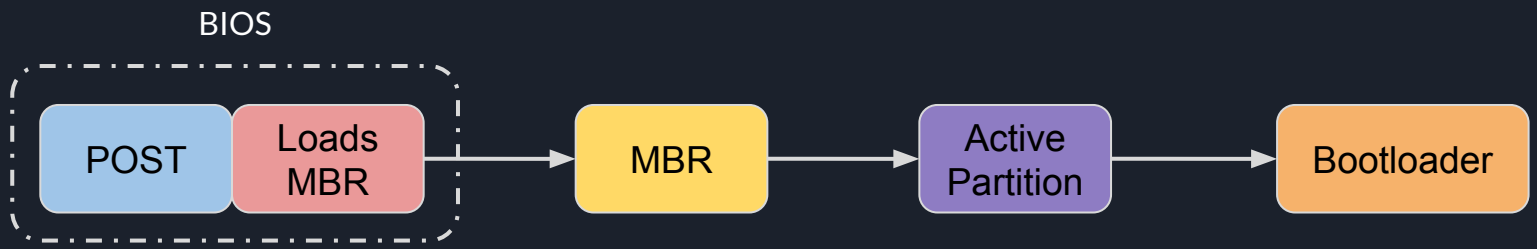
- The observer effect might sometimes be intentional
  - protect intellectual property from reverse-engineering
  - Protected Media Path (used to enforce DRM)
- Modern OS security mechanisms are interfering
  - PatchGuard
  - otherwise BSOD !

# Incomplete system view

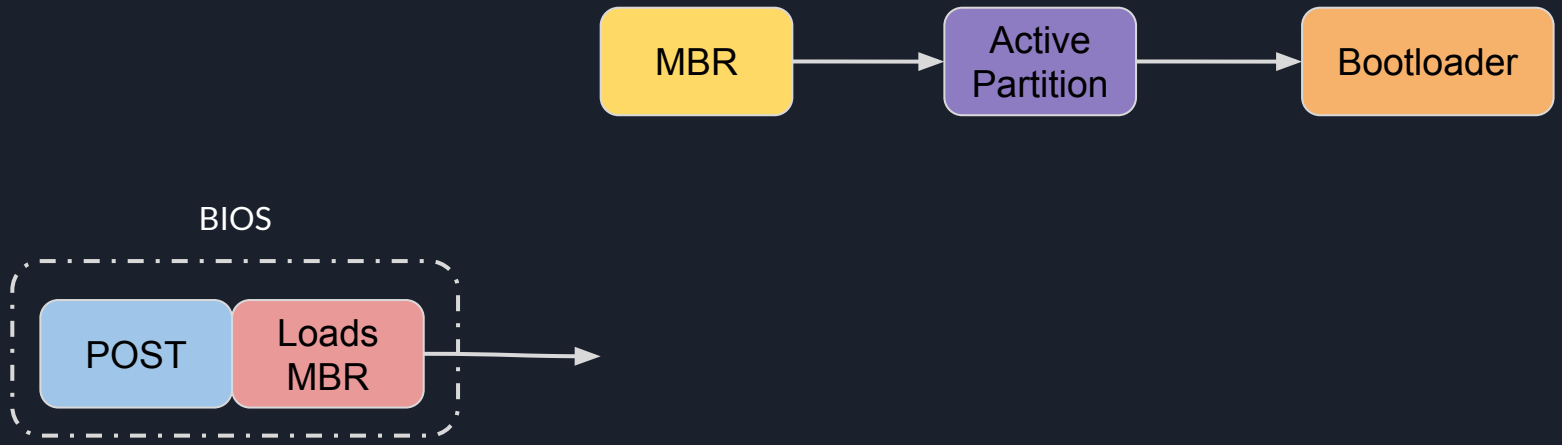
Debuggers are fighting against new OS security features



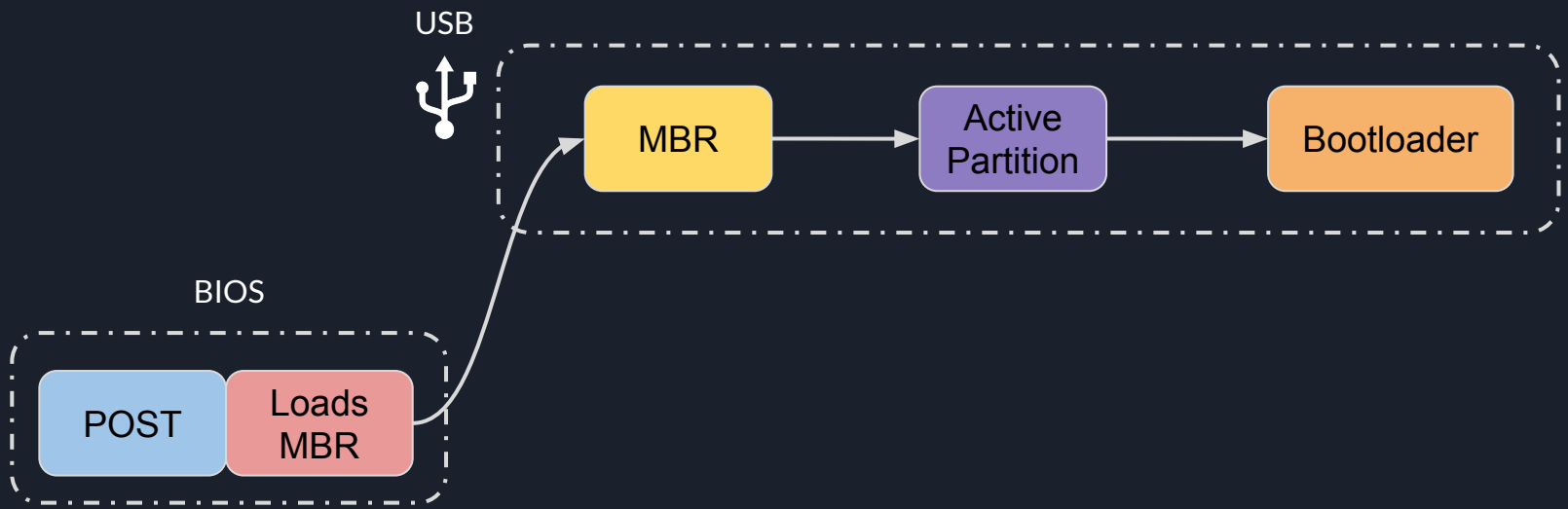
# Boot sequence analysis



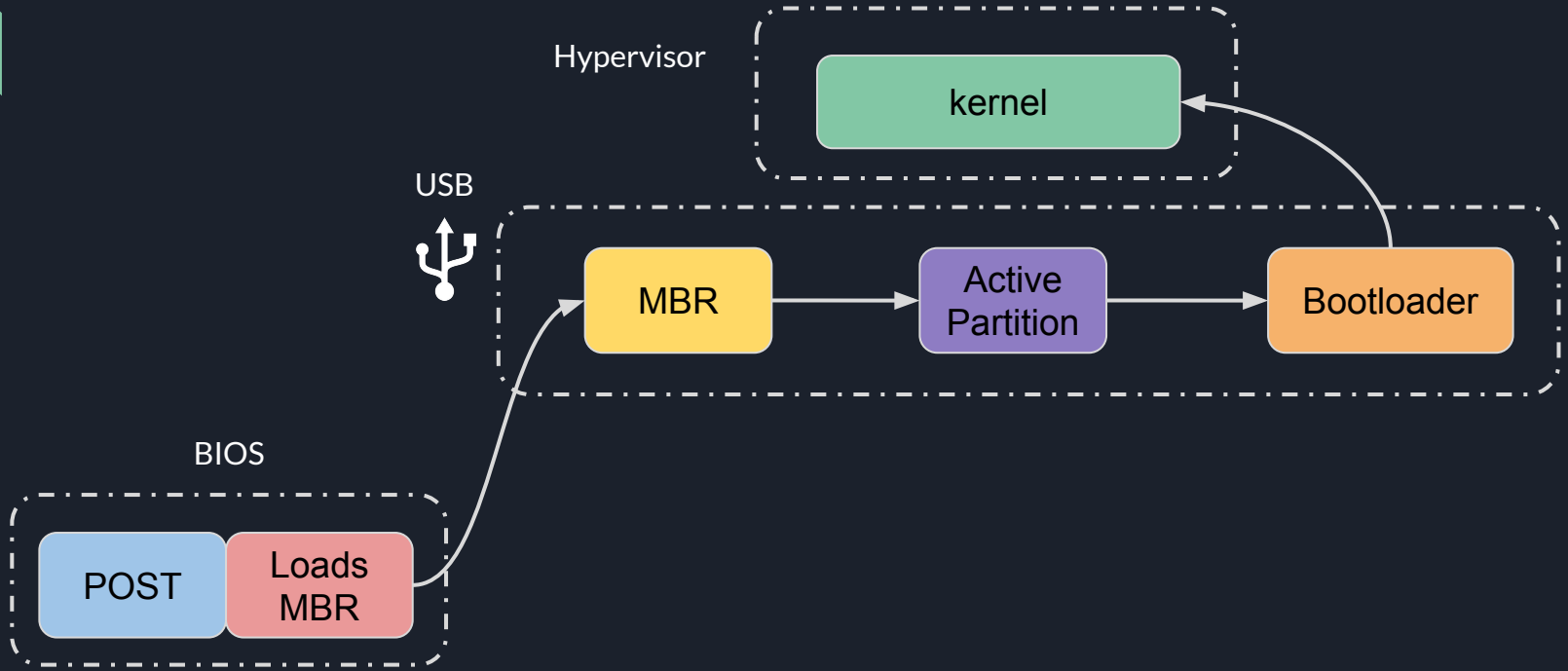
# Boot sequence analysis



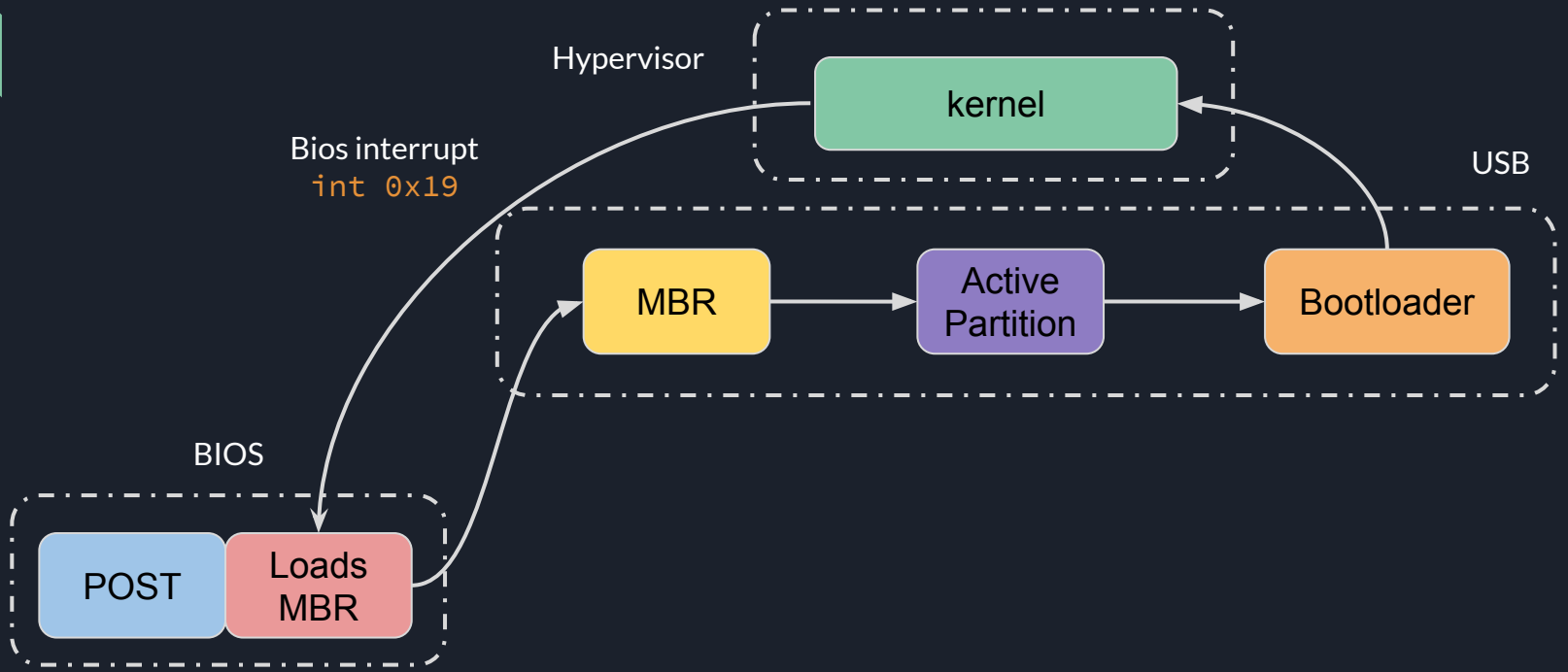
# Boot sequence analysis



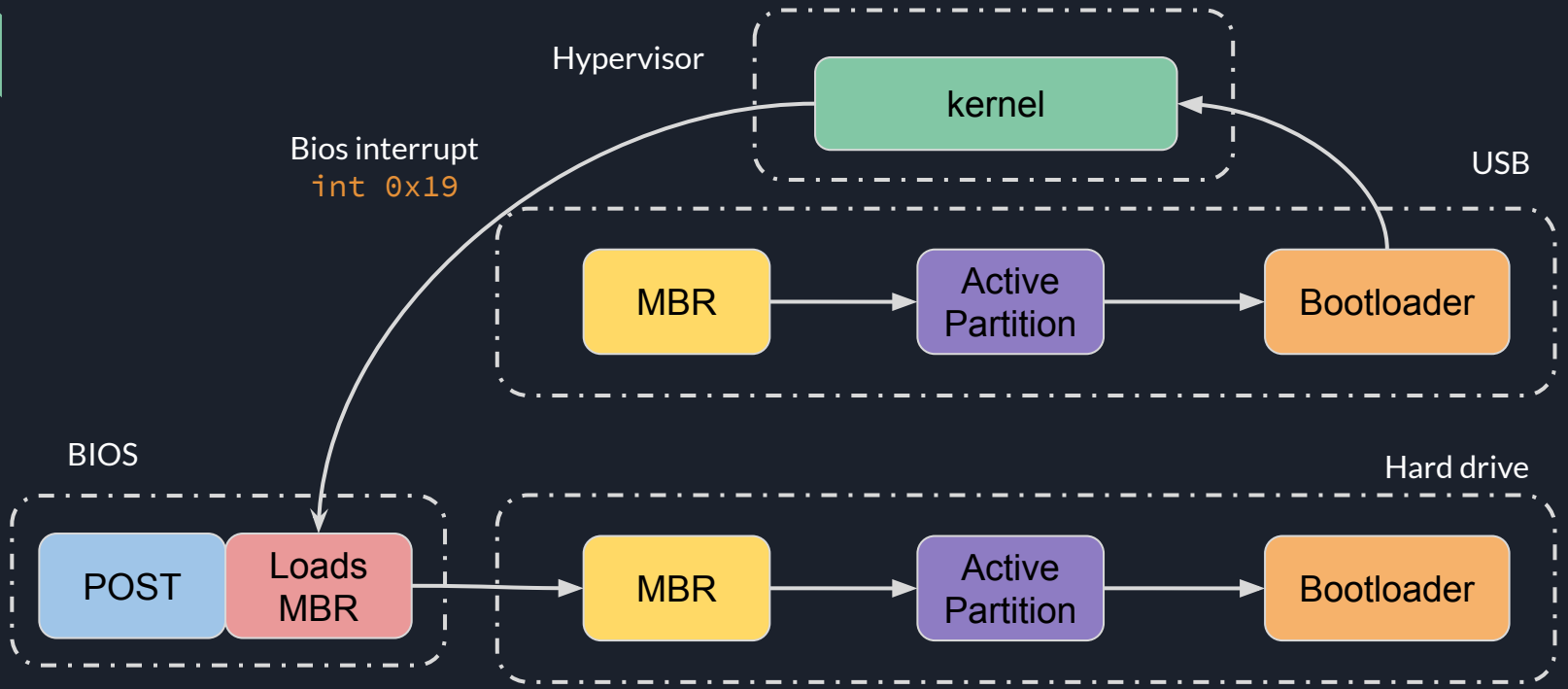
# Boot sequence analysis



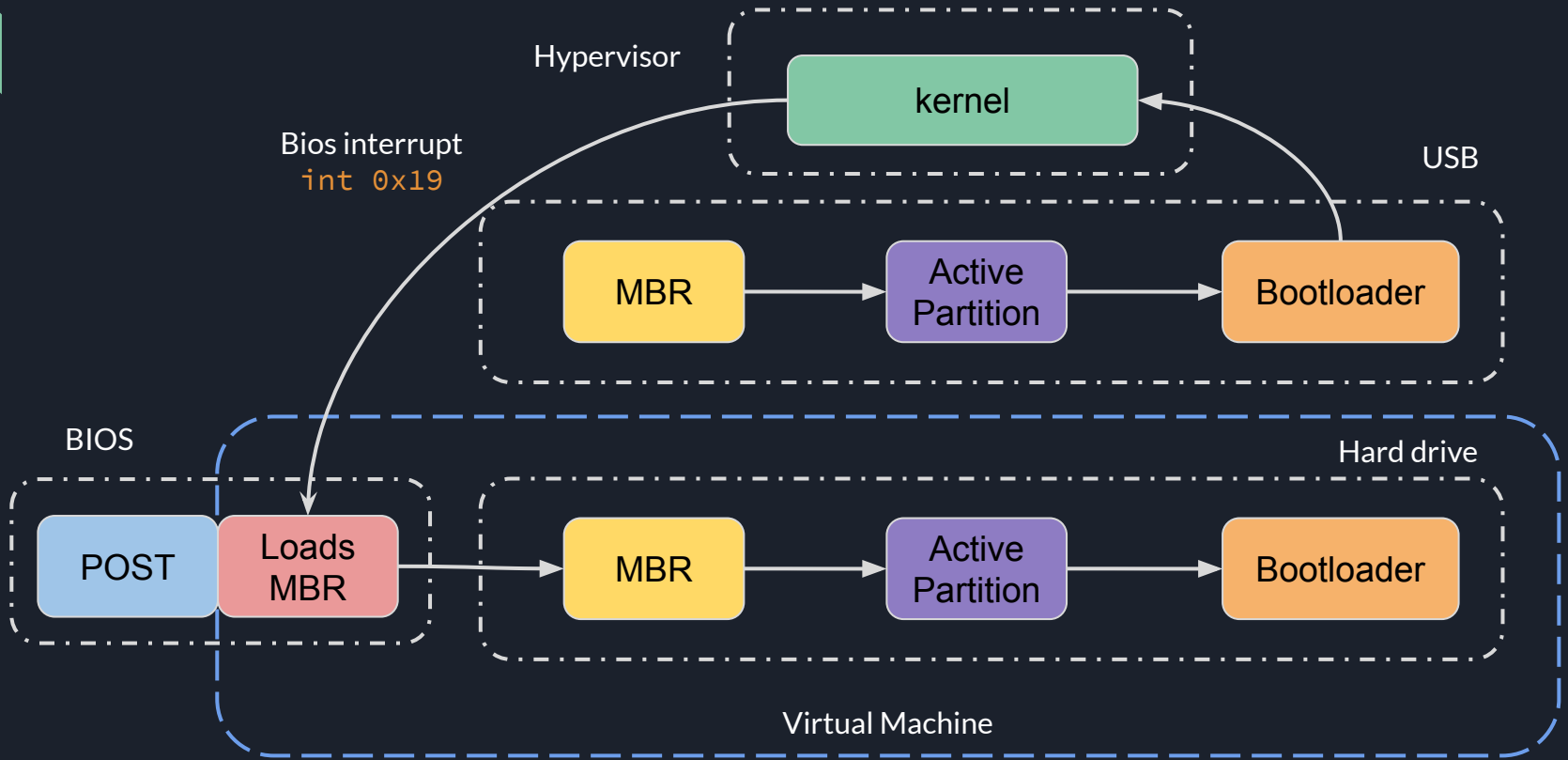
# Boot sequence analysis



# Boot sequence analysis

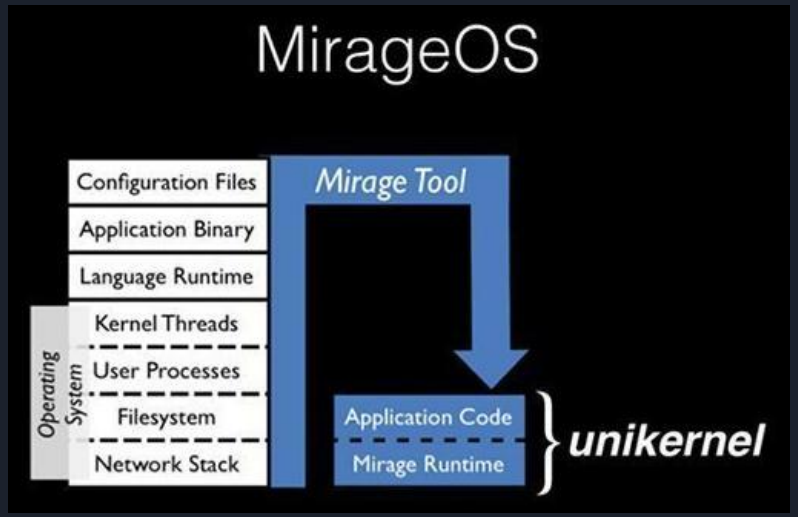


# Boot sequence analysis



# Unikernel debugging

- Specialized embedded system images
- Kernel/app runs in ring0
- One process, one address space
- No debug stub...




Y **Hacker News** new | past | comments | ask | show | jobs | submit

▲ cdoxsey 9 months ago | parent | favorite | on: A Rust-Based Unikernel: First Version of a Rust-Ba...

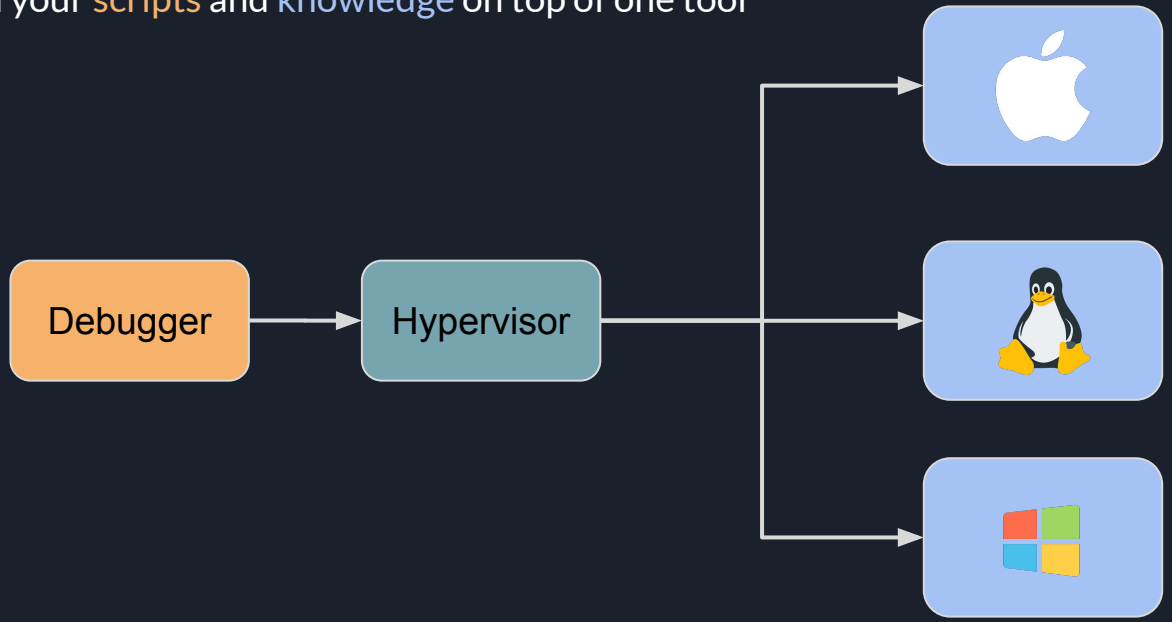
Unikernels can be difficult to debug. All the tools that come with your operating system are missing.

# Unmodified guests

- No remote debug agent/stub
- No custom VM setup
  - hardware
    - network card
    - serial cable
  - software
    - install Windows SDK
  - configuration
    - bcdedit /set debug on
    - bcdedit /dbgsettings serial debugport:1 baudrate:115200
- On-the-fly debugging

# Cross-platform debugging

- Unified debug framework for Windows/Linux/MacOS
- Build your scripts and knowledge on top of one tool



# Follow malware sandbox trend

- Nowadays, malware sandboxing solutions are **hypervisor-based**
  - VMRay, Joe Security, DRAKVUF
- Cross-platform
- Agentless
- Stealth breakpoints
- Process hijacking and code injection
  - start guest binary execution via hypervisor

# Why: Recap

1. Advanced malware analysis
2. OS debug API untrusted
3. Boot sequence and unikernel analysis
4. Untouched guest VMs
5. Unified tool for OS debugging

-> Leverage the hypervisor as a new debugging platform

# History

# Timeline (2003 - 2019)

- 2003: QEMU GDB stub
- 2007: VMware GDB stub
- 2010: HyperDBG
- 2011:
  - virtdbg
  - ramooflax
- 2012: VirtualBox GDB stub
- 2014: vmidbg
- 2016: Winbagility
- 2017:
  - PulseDBG
  - PyREBox
  - rVMI
- 2018:
  - Sandbagility
  - r2vmi
- 2019:
  - pyvmidbg
  - XenDBG

# Categories

- Built-in hypervisor debug stubs
- Bare metal debuggers
  - hyperjacking
  - USB-boot
- Virtual Machine (virtual hardware) debuggers
  - emulated
  - full-virtualization

# Built-in hypervisor debug stubs

- QEMU/VirtualBox/VMware GDB stub
- no real stealth
  - VMware: `debugStub.hideBreakpoints = “TRUE”`
  - Uses debug registers (4 breakpoints...)
- no guest awareness
  - debug the kernel, that’s it
- no flexibility
  - GDB-only
  - one hypervisor

# Bare-metal debuggers (hyperjacking)

- HyperDBG (2010)
  - *“I want to take full control of a production system”*
  - driver is installed on the host (Windows 7)
- Virtdbg (2011)
  - *“I want to debug PatchGuard”*
  - driver is injected via DMA attack

- driver development
- not OS-agnostic

# Bare-metal debuggers (USB boot)

- Ramooflax (2011)
  - “*Debugging modern operating systems and real BIOS on physical machine*”
  - python remote control API
- PulseDBG (2017)
  - “*I want a better WinDBG UI*”
  - hypervisor is contained in an EFI bootloader (*bootx64.efi*)
  - SDK to interact with stub

# Virtual machine debuggers (emulation)

- PyREBox - CISCO Talos (2017)
  - “*I want a scriptable dynamic instrumentation system*”
  - Instrumentation of QEMU (emulator)
  - Volatility for semantic gap
  - IPython shell
  - Fine grained callbacks

# Virtual machine debuggers (full-virt)

- Unmaintained (💀)
  - vmidbg (2014)
    - GDB stub on top of LibVMI
  - rVMI - FireEye (2017)
    - KVM instrumentation
    - Rekall as introspection layer / debugger interface

# Virtual machine debuggers (full-virt)

- Unmaintained (💀)
  - vmidbg (2014)
    - GDB stub on top of LibVMI
  - rVMI - FireEye (2017)
    - KVM instrumentation
    - Rekall as introspection layer / debugger interface
- Winbagility (2016)
  - VirtualBox instrumentation
  - KD stub (WinDBG)
- Sandbagility (2018)
  - malware analysis framework
  - based on Winbagility

# Virtual machine debuggers (full-virt)

- Unmaintained (💀)
  - vmidbg (2014)
    - GDB stub on top of LibVMI
  - rVMI - FireEye (2017)
    - KVM instrumentation
    - Rekall as introspection layer / debugger interface
- Winbagility (2016)
  - VirtualBox instrumentation
  - KD stub (WinDBG)
- Sandbagility (2018)
  - malware analysis framework
  - based on Winbagility
- Xendbg - NCC Group (2019)
  - GDB stub on top of Xen
  - debug unikernels

# Objectives

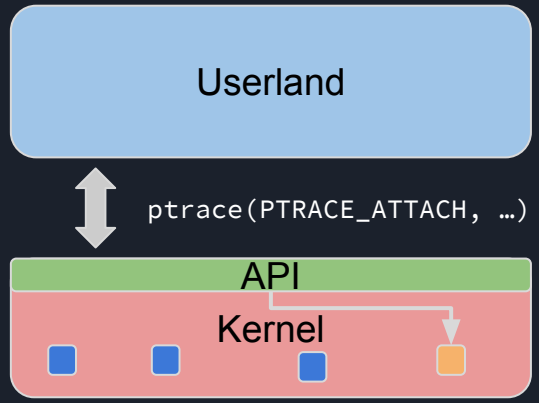
- Emulation, bare-metal: not interested
- I want a {flexible} hypervisor-level debugger to debug VMs
  - r2vmi (2018)
    - based on LibVMI to be hypervisor-agnostic
    - radare2 for introspection/analysis and shell interactivity
  - pyvmidbg (2019)
    - GDB stub ➡️ LibVMI ➡️ hypervisor
    - compatibility over pure performance
    - guest aware debug stub

# Challenges

# Rebuilding the Debugger API

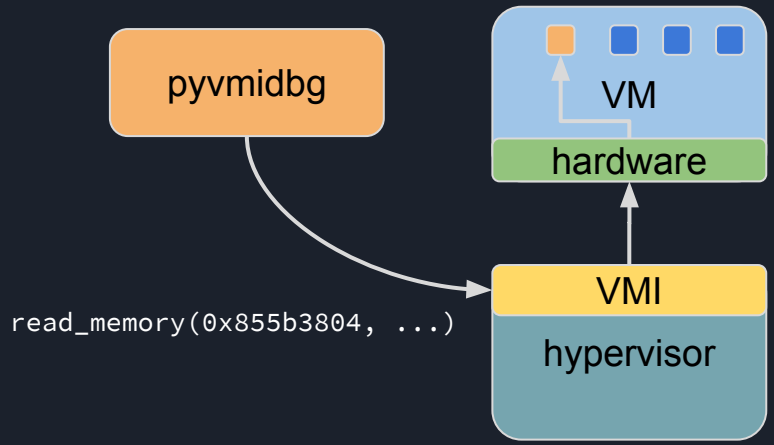
`gdb attach <PID>`

- Kernel knows how to query process states
- Interface is exposed via defined API




`pyvmidbg attach <VM> <PID>`

- We don't know the operating system
- Read and interpret raw memory



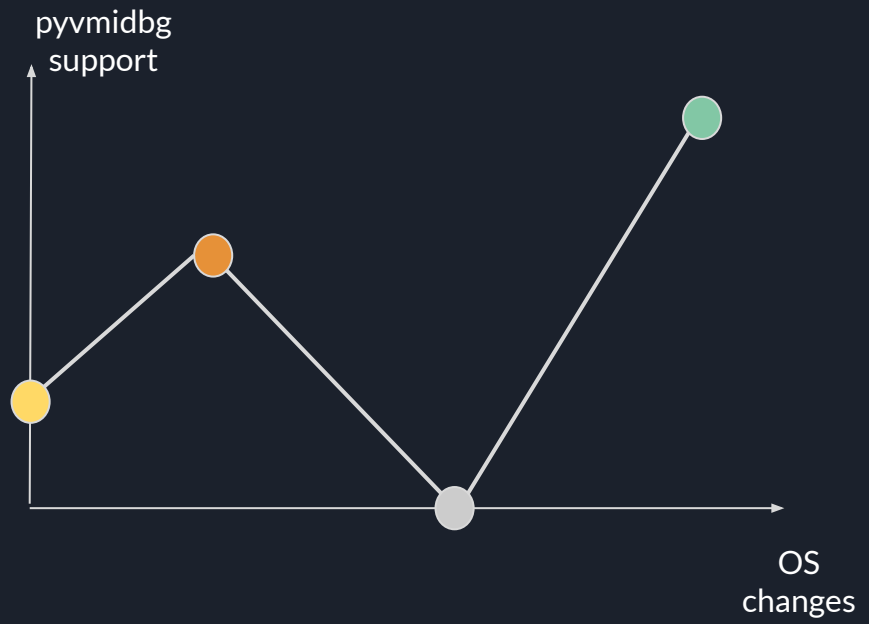
# Filling the Semantic Gap

- We need an intimate knowledge of the guest
- Handle various debug formats (PDBs, DWARFs, etc...)
- Make sense of raw memory
  - forensic tools
- Rekall profiles to the rescue !
  - JSON format
    - $CONSTANTS
    - $ENUMS
    - $FUNCTIONS
    - $STRUCTS
- Offsets from kernel base address


```
{
 "$CONSTANTS": {
  "BiosBegin": 2012200,
  "str:A_driver_has_added_a_device_obje": 1817744
 },
 "$ENUMS": {
  "BUS_QUERY_ID_TYPE": {
   "0": "BusQueryDeviceID",
   "1": "BusQueryHardwareIDs",
   "2": "BusQueryCompatibleIDs"
  }
 },
 "$FUNCTIONS": {
  "CmDeleteKey": 1434156,
  "IoGetAttachedDevice": 99966,
  "NtCreateProcess": 1024640
 },
 "$STRUCTS": {
  "LIST_ENTRY32": [8, {
   "Blink": [4, ["unsigned long", {}]],
   "Flink": [0, ["unsigned long", {}]]
  }],
  "_EPROCESS": [608, {
   "ActiveProcessLinks": [136, ["_LIST_ENTRY", {}]],
   "ActiveThreads": [416, ["unsigned long", {}]],
   "AddressCreationLock": [240, ["_FAST_MUTEX", {}]]
  }
}
```

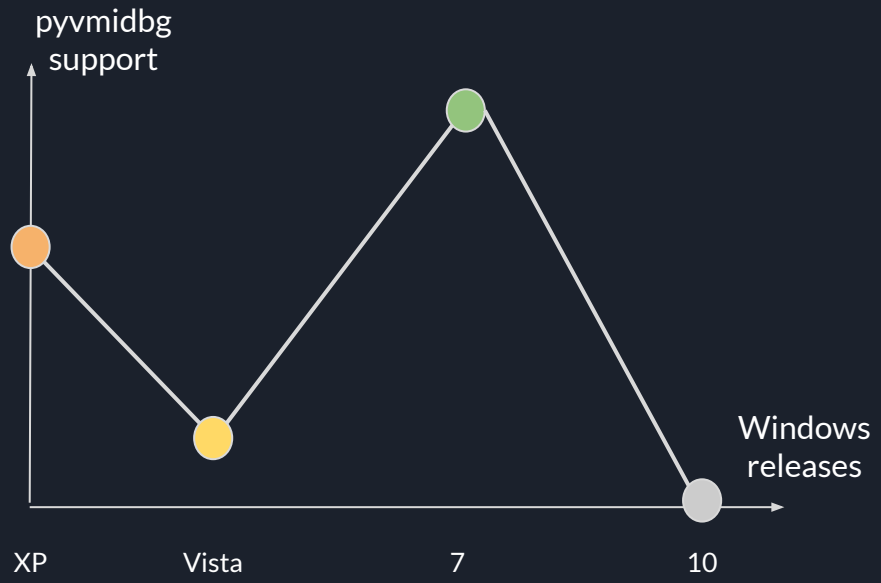
# Following OS changes

- pure semantic layer is not enough
- the debugger needs a bit of logic to adapt to operating system



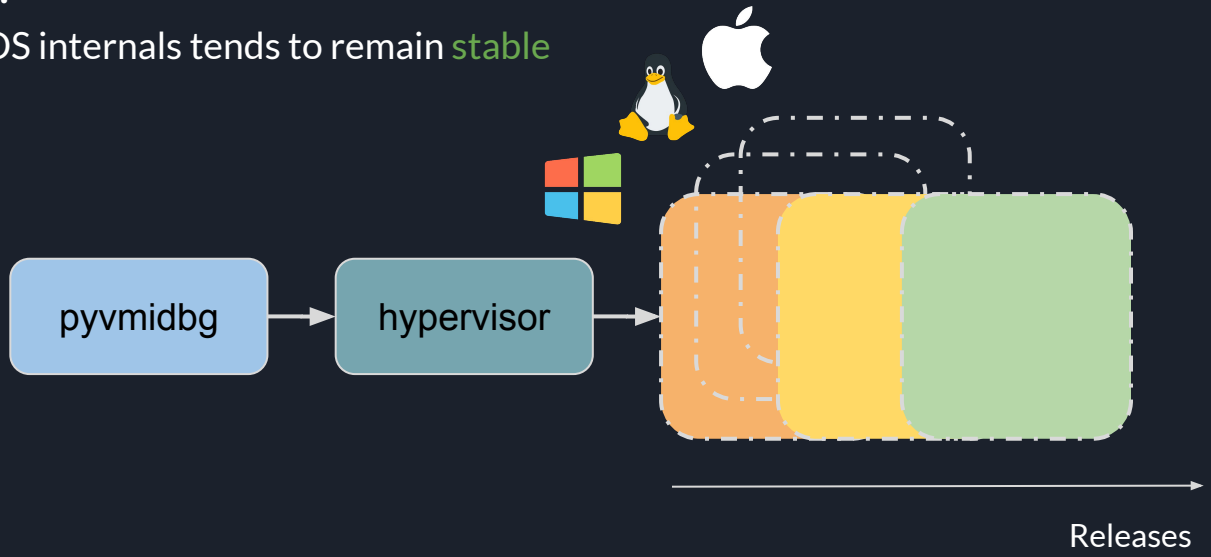
# Following OS changes

- pure semantic layer is not enough
- the debugger needs a bit of logic to adapt to operating system
- eg: Windows
  - **WinXP**: IDLE process outside of main process list
  - **Windows Vista**: big changes in the kernel
  - **Windows 10**: VSM, Secure kernel



# Following OS changes

- Problem is extended for every OS that you want to support
- Cost of building a hypervisor-based {flexible} debugger
- However:
  - core OS internals tends to remain stable



# Hypervisor-agnostic: LibVMI

- VMI Abstraction layer
- Offers basic introspection
  - rekall profiles support
- Standard for VMI applications

```
winxp {
    ostype = "Windows";
    rekall_profile = "/etc/libvmi/winxp-profile.json";
}
```

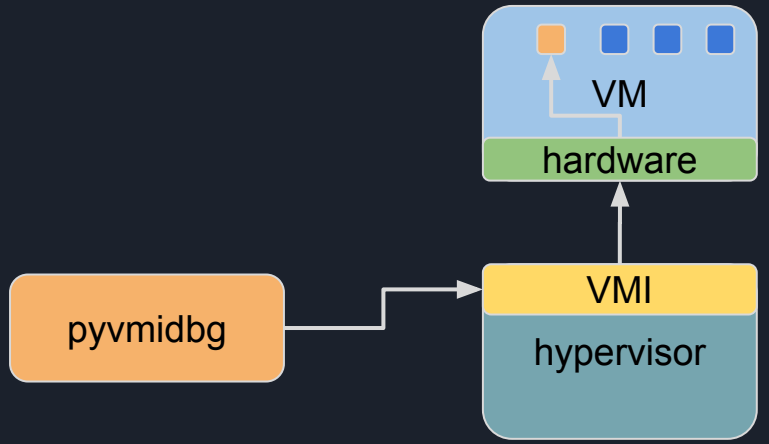
| | VCPU Registers | Physical memory | Hardware events |
|---|---|---|---|
| Xen | ✔ | ✔ | ✔ |
| KVM | ✔ | ✔ | ❌ |

https://github.com/**libvmi/libvmi**

# {Flexibility}

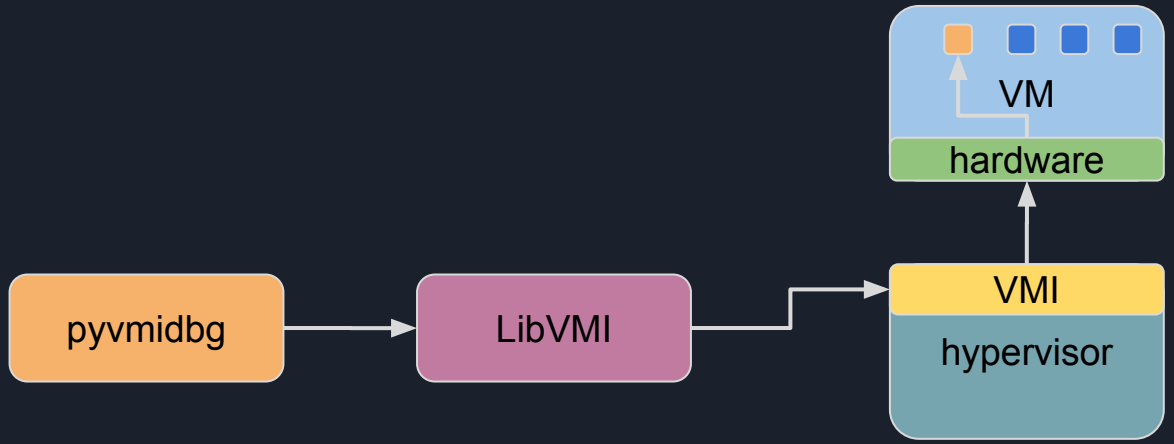
# {Flexibility}

1. Virtual Machine Introspection



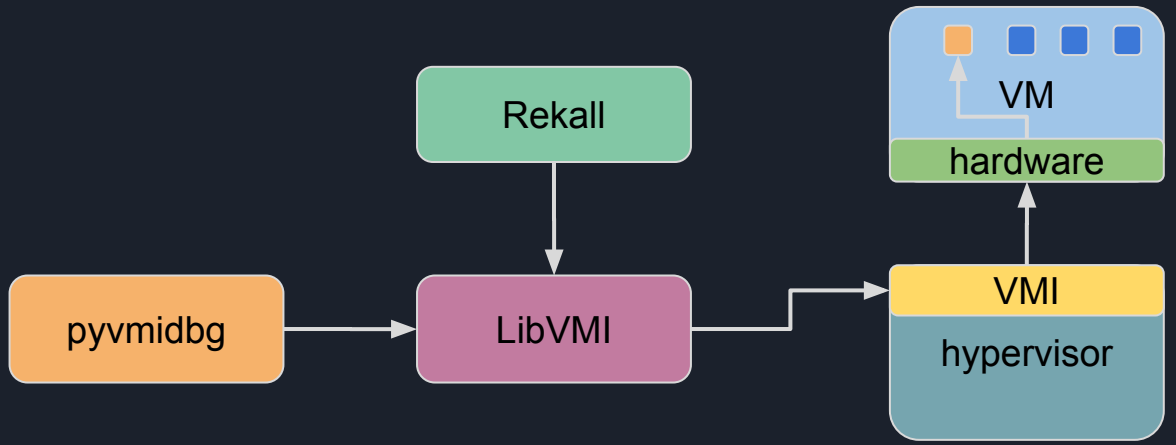
# {Flexibility}

1. Virtual Machine Introspection
2. LibVMI



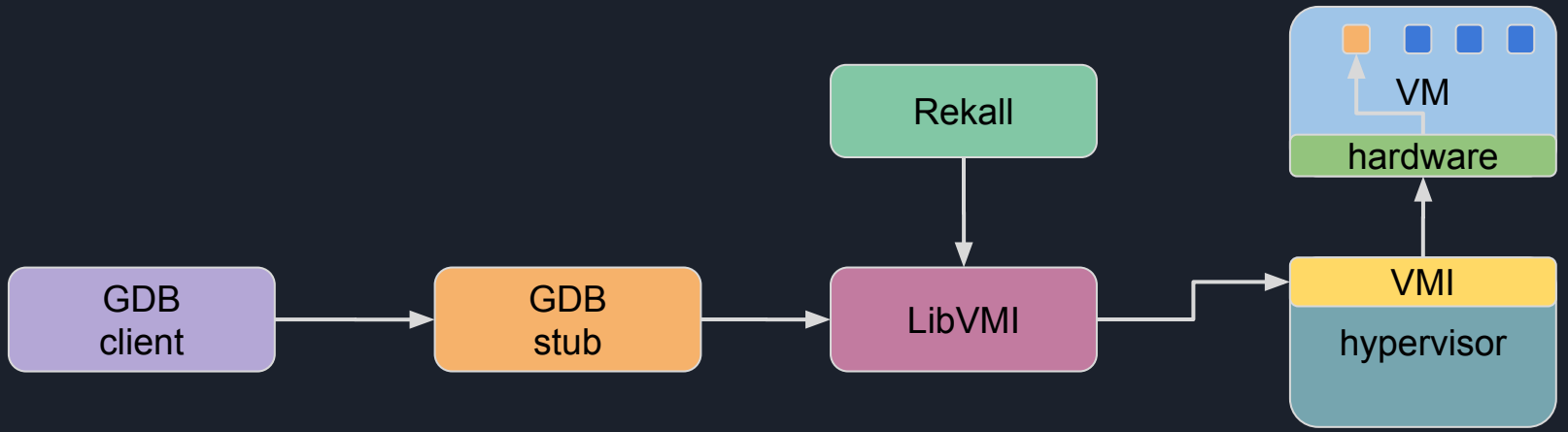
# {Flexibility}

1. Virtual Machine Introspection
2. LibVMI
3. Rekall profiles



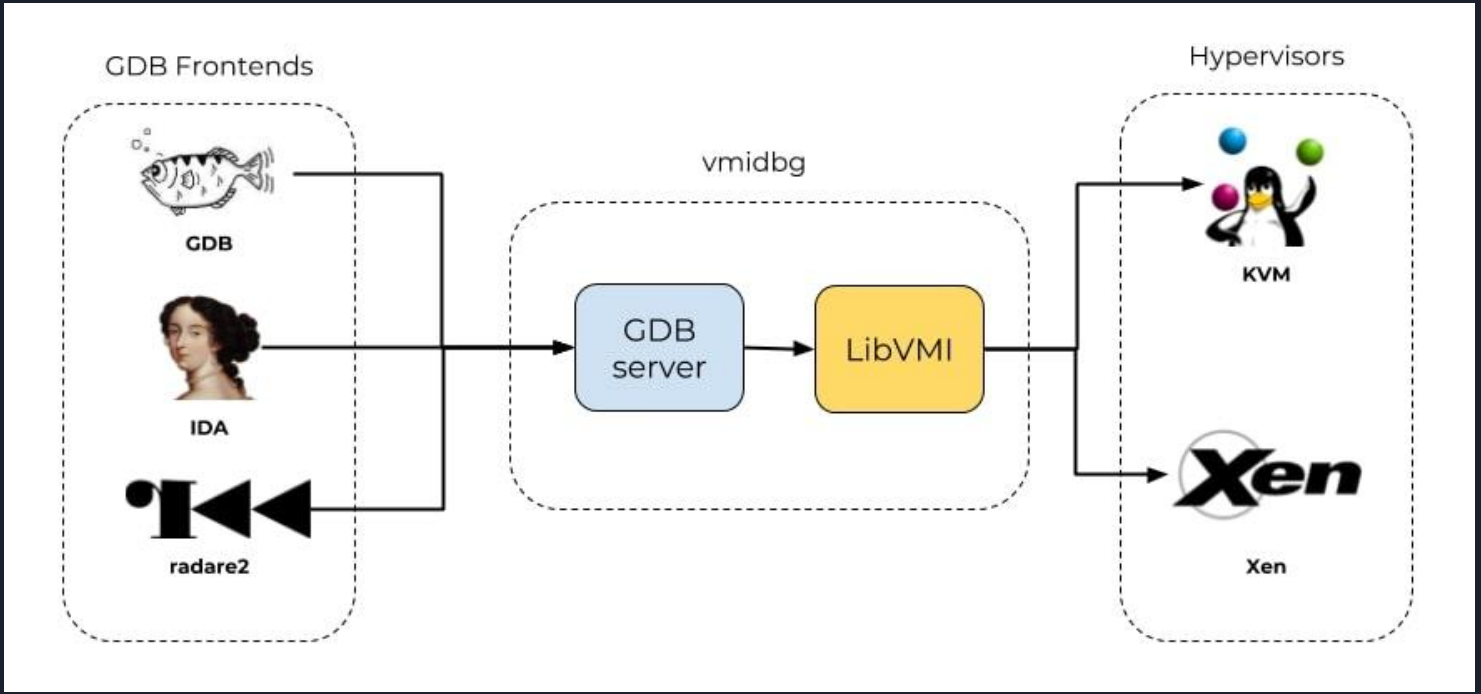
# {Flexibility}

1. Virtual Machine Introspection
2. LibVMI
3. Rekall profiles
4. GDB protocol



# Demo

# pyvmidbg

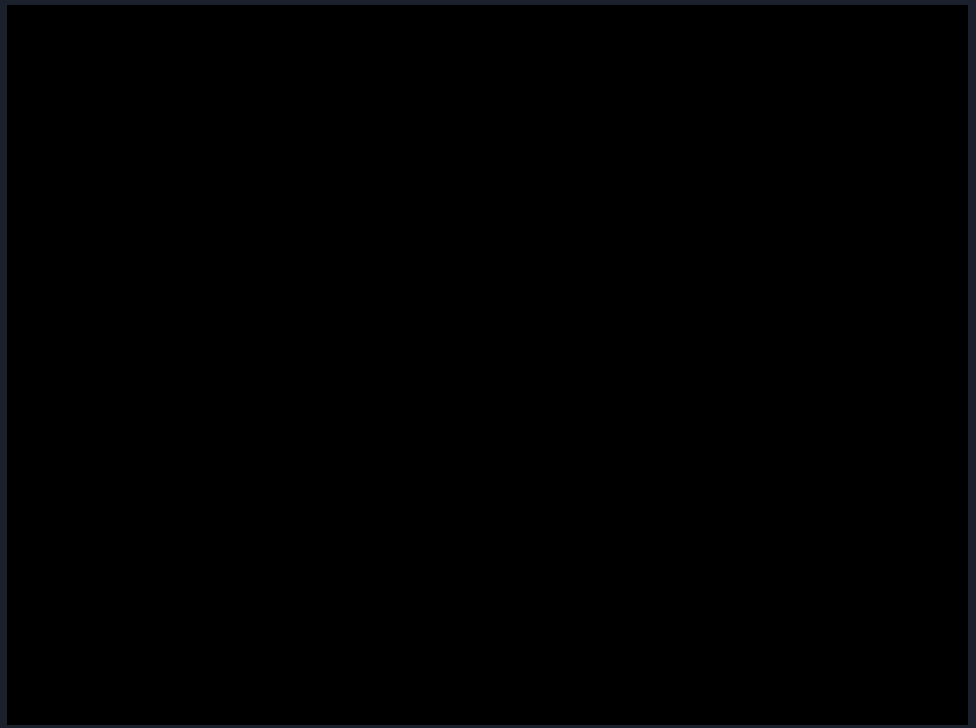

GDB Frontends
GDB
IDA
radare2
vmidbg
GDB server
LibVMI
Hypervisors
KVM
Xen
Xen

# pyvmidbg

- `pyvmidbg.py <port> <vm_name> [<process>]`
- 2 modes
  - raw
    - 1 process -> system
    - thread -> VCPU
    - attach -> pause VM
    - basically your QEMU GDB stub
  - guest-aware
    - Windows/Linux debug context
    - locate the targeted process descriptor
    - enumerate threads and read execution context
- basic software breakpoints
- singlestep
- breakin

# pyvmidbg: raw

# pyvmidbg: guest-aware



# Demo

# Improvements

- info proc
  - process 1854
  - cmdline = ‘C:\WINDOWS\system32\cmd.exe /C dir’
  - cwd = ‘C:\Document and Settings\Vagrant’
  - exe = ‘C:\WINDOWS\system32\cmd.exe’
- info sharedlibrary
  - 0x7c800000 kernel32
  - 0x7c900000 ntdll
  - 0x7c9c0000 SHELL32
- callstacks

# pyvmidbg: dev-friendly

- Strong dev requirements
  - running (modified) hypervisor
  - virtual machines
- Vagrant environment
  - Xen (packaged/from source)
  - libvmi
  - libvirt
  - vms
    - ubuntu 16.04/winxp
    - rekall profiles
- Nested virtualization
  - KVM > Xen


https://github.com/**Wenzel/vagrant-xen-pyvmidbg**

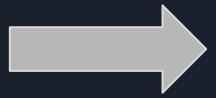
# HyperBreakpoints

# HyperBreakpoints

- *“HyperBreakpoints”* Winbagility article (SSTIC 2016)
- Breakpoints using virtualization layer capabilities
- We are not using the best breakpoints available in our debuggers

# Basic SoftHyperBreakpoint

- write `0xCC`
- listen on `int3` events
  - reinject in guest (*if needed*)
  - restore opcode
  - singlestep
  - restore breakpoint



- ✔️ fast
- ❌ stealth
- ❌ safe

Avoid multi-VCPU race conditions ?

--> pause all VCPUs at each event

- ✔️ fast-
- ❌ stealth
- ✔️ safe

# HardHyperBreakpoint

- breakpoints in DR0-DR3
- listen on `int1` events
  - reinject in guest (*if needed*)
  - disable breakpoint in DR7
  - singlestep
  - enable breakpoint in DR7

- ✔️ fastest
- ❌ stealth
- ✔️ safe

❌ 4 breakpoints limit...

To gain stealth:

- listen on MOV-TO-DR events
  - restore guest's DRx
  - singlestep
  - restore our DRx

- ✔️ fastest
- ✔️ stealth
- ✔️ safe

# PageHyperBreakpoint

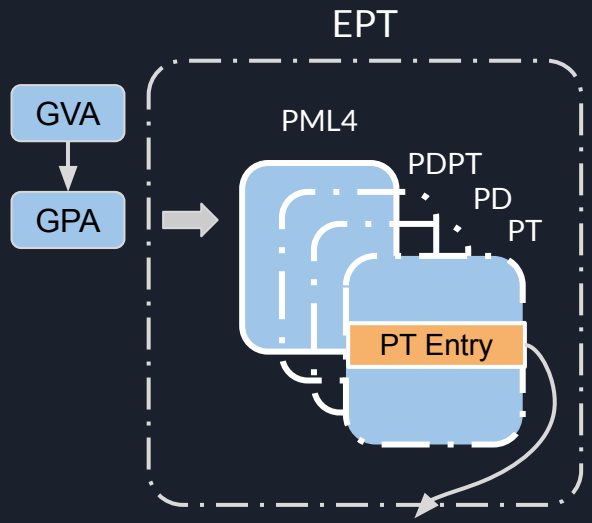
- GVA ➡️ GPA ➡️HPA
- Extended Page Tables
- From Guest Virtual Address (GVA)
  - find Guest Physical Page (GPP)
  - translate via EPT
  - find Host Physical Page
  - change permissions
  - generate *#EPTViolation*
- fine grained breakpoints ( R/W/X )

- ❌ fast
- ✔️ stealth
- ❌ safe

singlestep or emulate ?

- emulators are incomplete
- emulation opens vulnerabilities
- heavy emulation can lead to instabilities

# SoftHyperBreakpoint

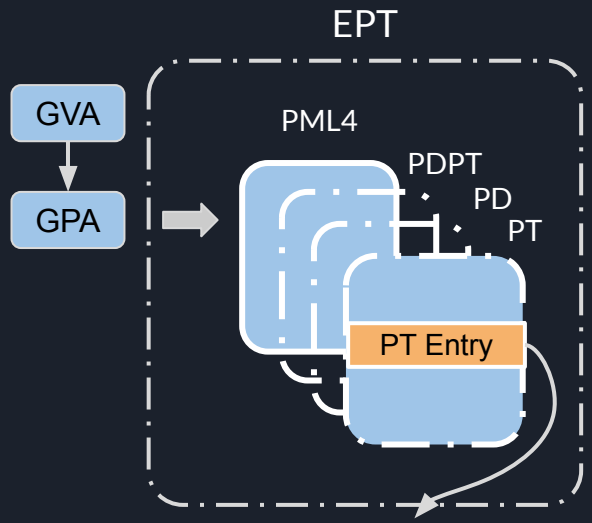



```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page

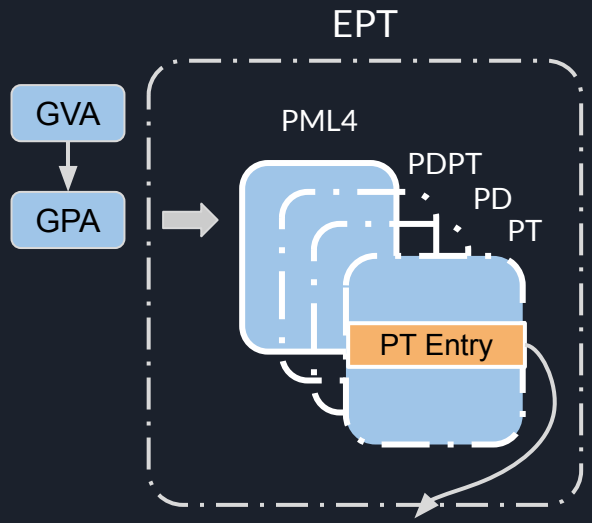



```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page



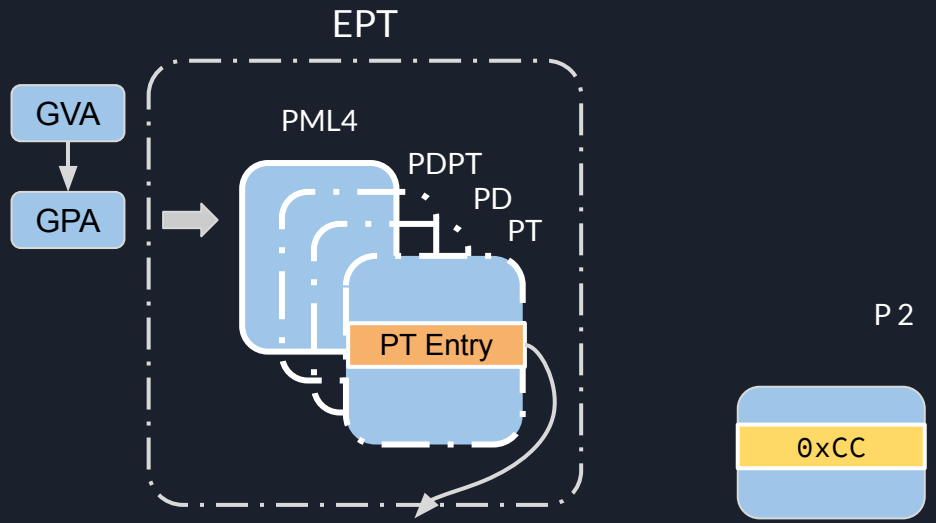


```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page
2. insert breakpoint in P2

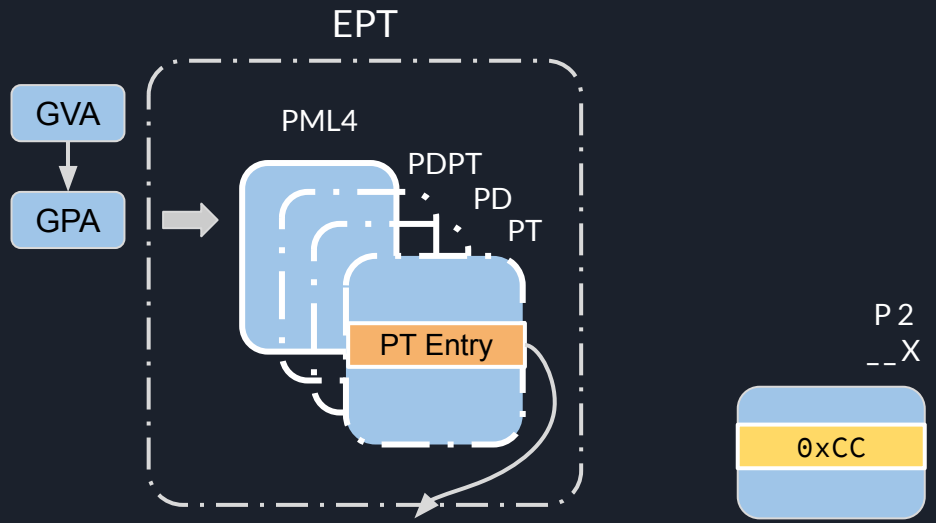



```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page
2. insert breakpoint in P2
3. set P2 as execute-only

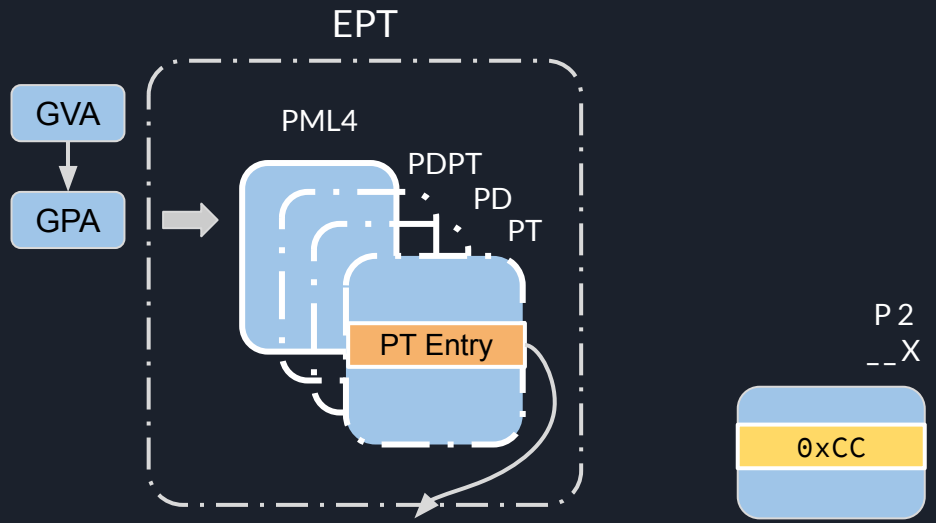



```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page
2. insert breakpoint in P2
3. set P2 as execute-only
4. set P1 as read/write

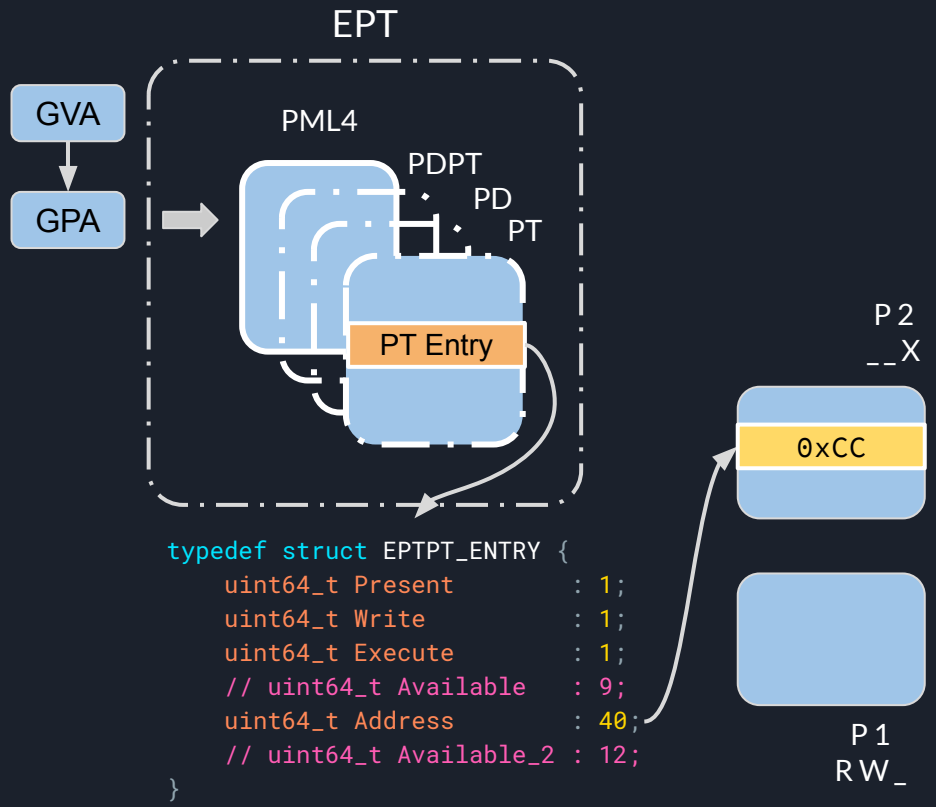



```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page
2. insert breakpoint in P2
3. set P2 as execute-only
4. set P1 as read/write
5. point GPA to P2



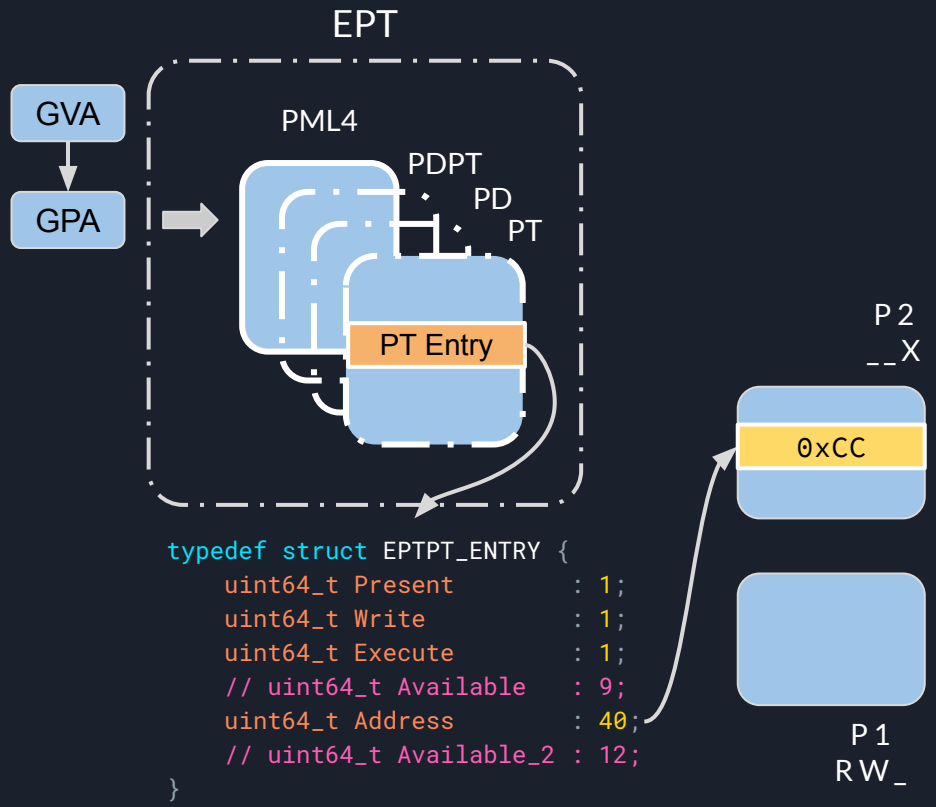
# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page
2. insert breakpoint in P2
3. set P2 as execute-only
4. set P1 as read/write
5. point GPA to P2

*#INT3*

- handle breakpoint



# SoftHyperBreakpoint

Setup:

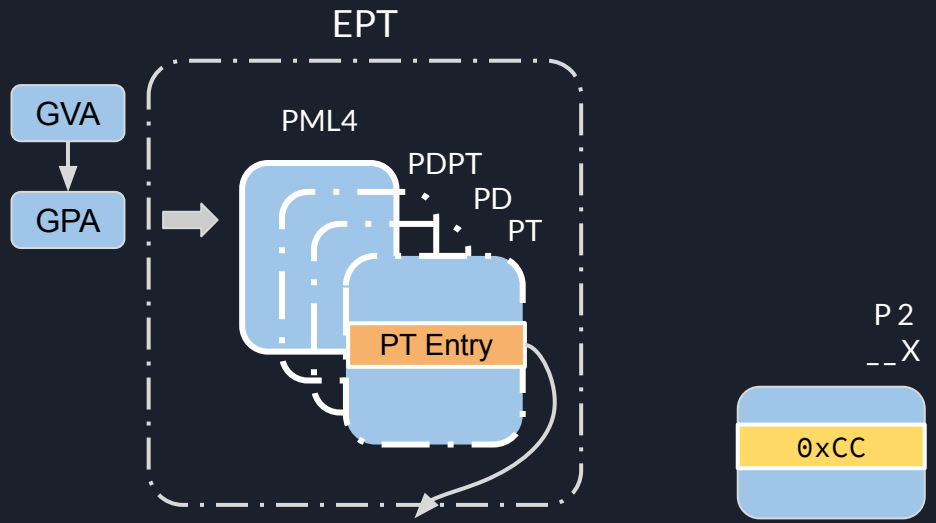
1. duplicate Host Physical Page
2. insert breakpoint in P2
3. set P2 as execute-only
4. set P1 as read/write
5. point GPA to P2

*#INT3*

- handle breakpoint

*#EPTViolation* (Read/Write)

- switch on P1




```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# SoftHyperBreakpoint

Setup:

1. duplicate Host Physical Page
2. insert breakpoint in P2
3. set P2 as execute-only
4. set P1 as read/write
5. point GPA to P2

*#INT3*

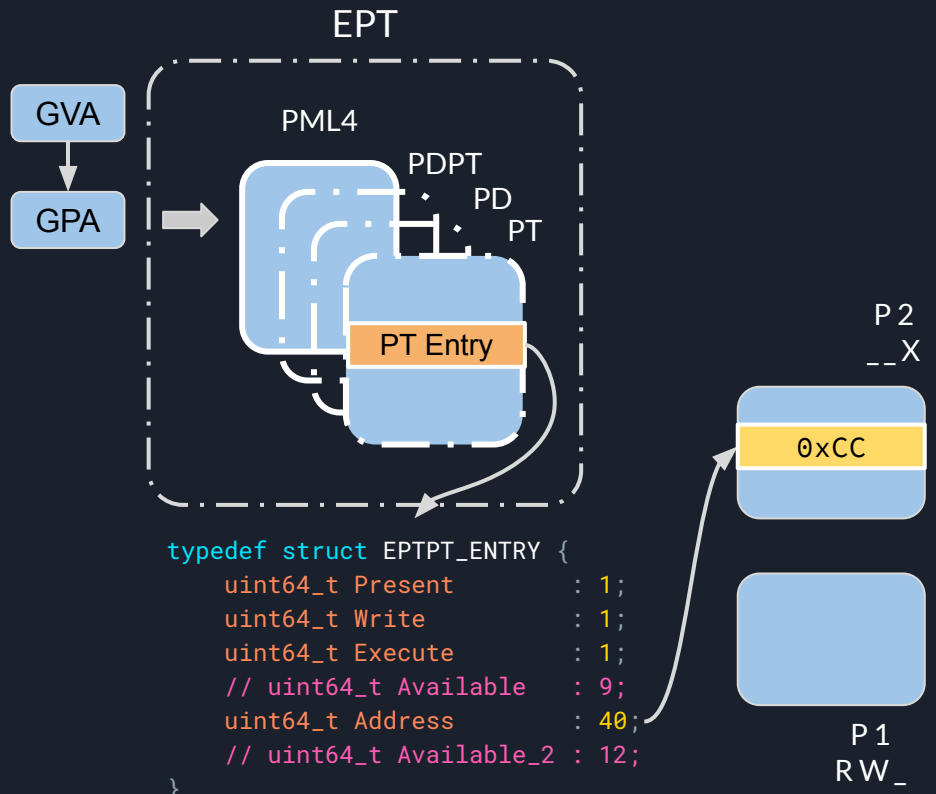
- handle breakpoint

*#EPTViolation* (Read/Write)

- switch on P1

*#EPTViolation* (Execute)

- switch on P2




```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

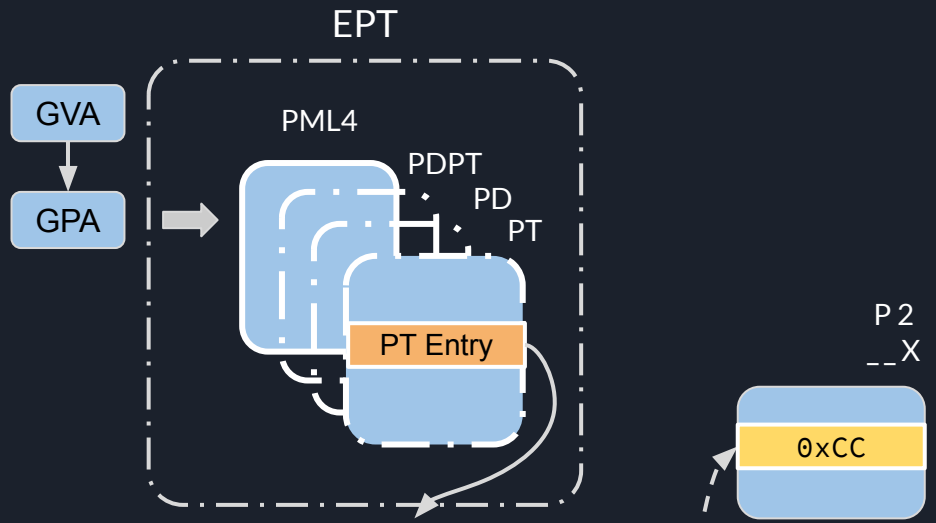
# SoftHyperBreakpoint

*#INT3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch on P1

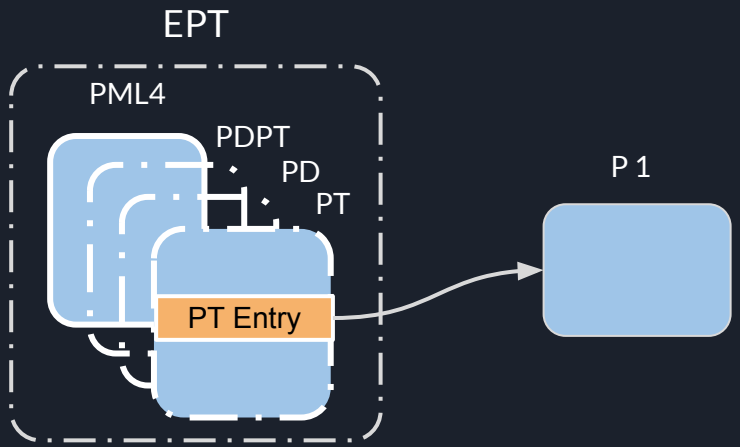
*#EPTViolation* (Execute)
- switch on P2

- ✔ fast
- ✔ stealth
- ❌ safe



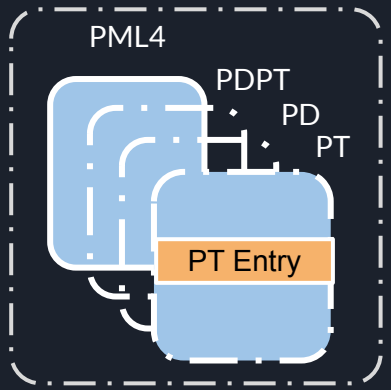

```
typedef struct EPTPT_ENTRY {
    uint64_t Present         : 1;
    uint64_t Write           : 1;
    uint64_t Execute         : 1;
    // uint64_t Available    : 9;
    uint64_t Address         : 40;
    // uint64_t Available_2  : 12;
}
```

# Advanced SoftHyperBreakpoint



# Advanced SoftHyperBreakpoint

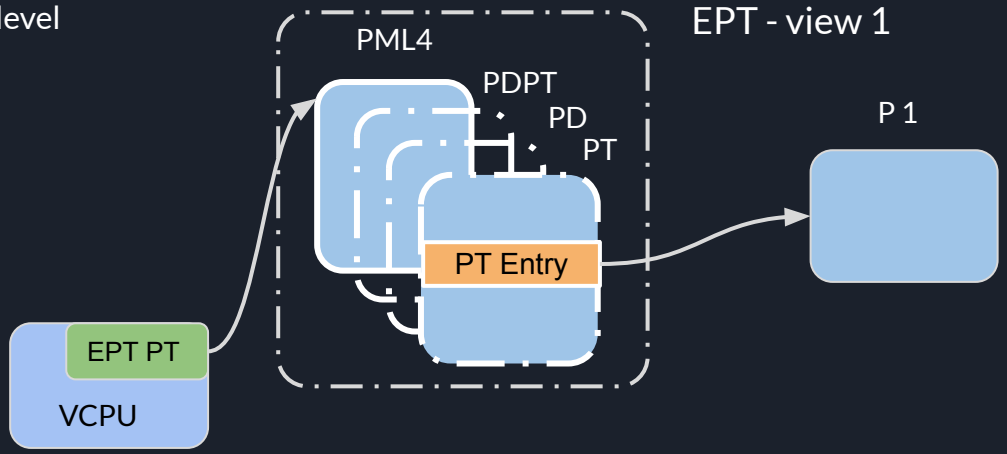
- Switch memory views at VCPU level




EPT - view 1

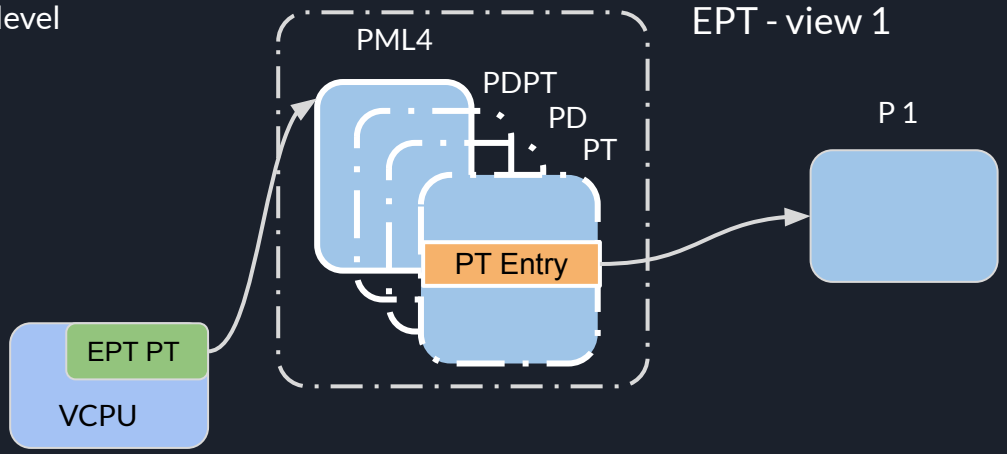
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level



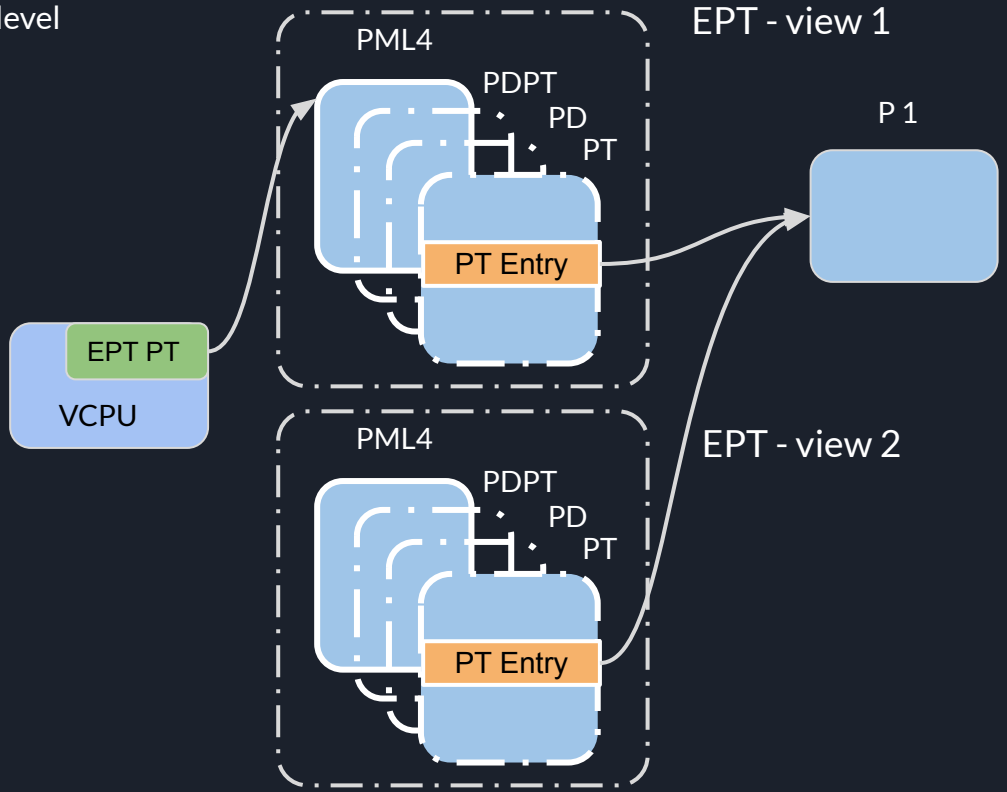
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT



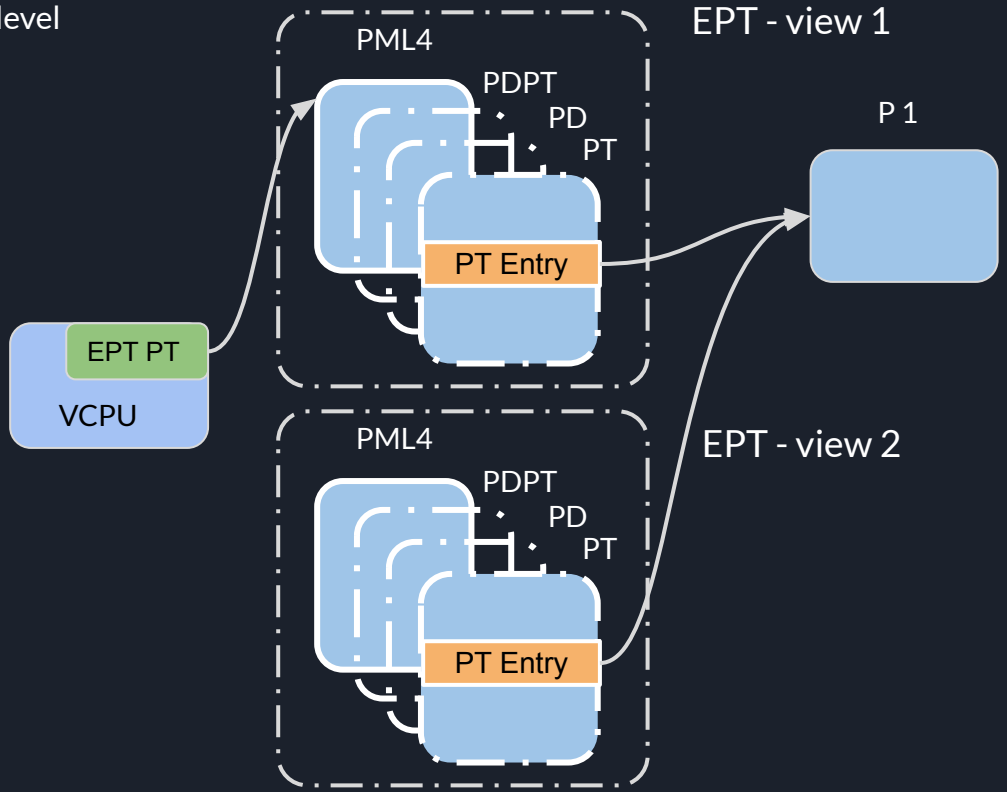
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT



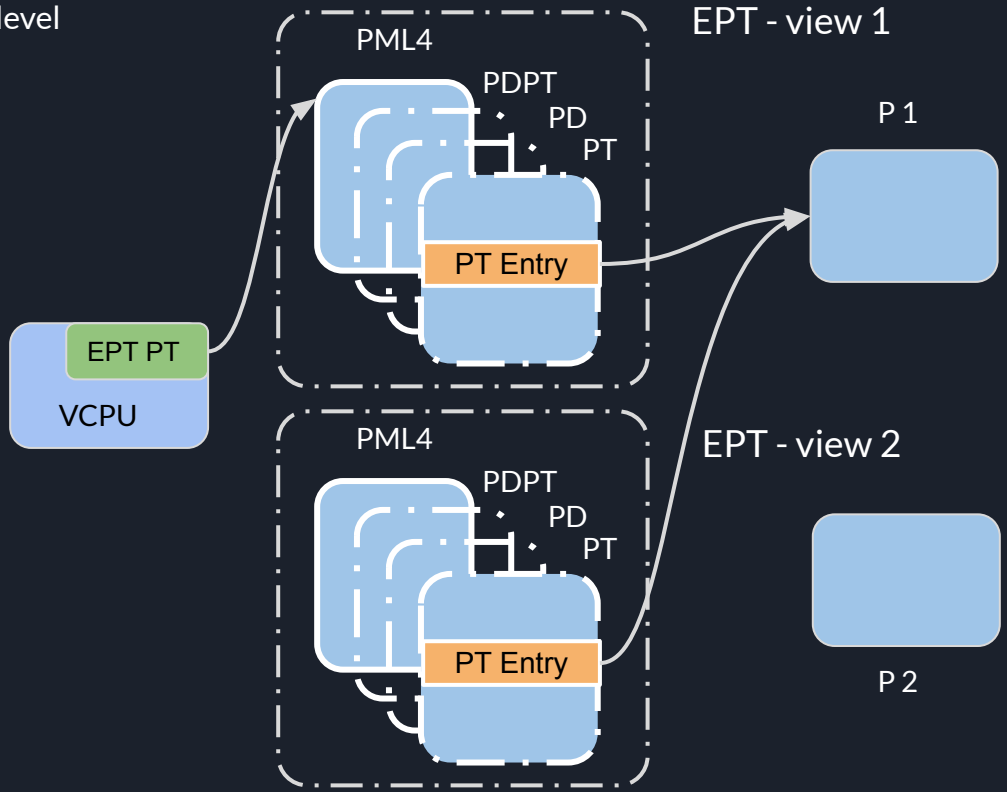
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1



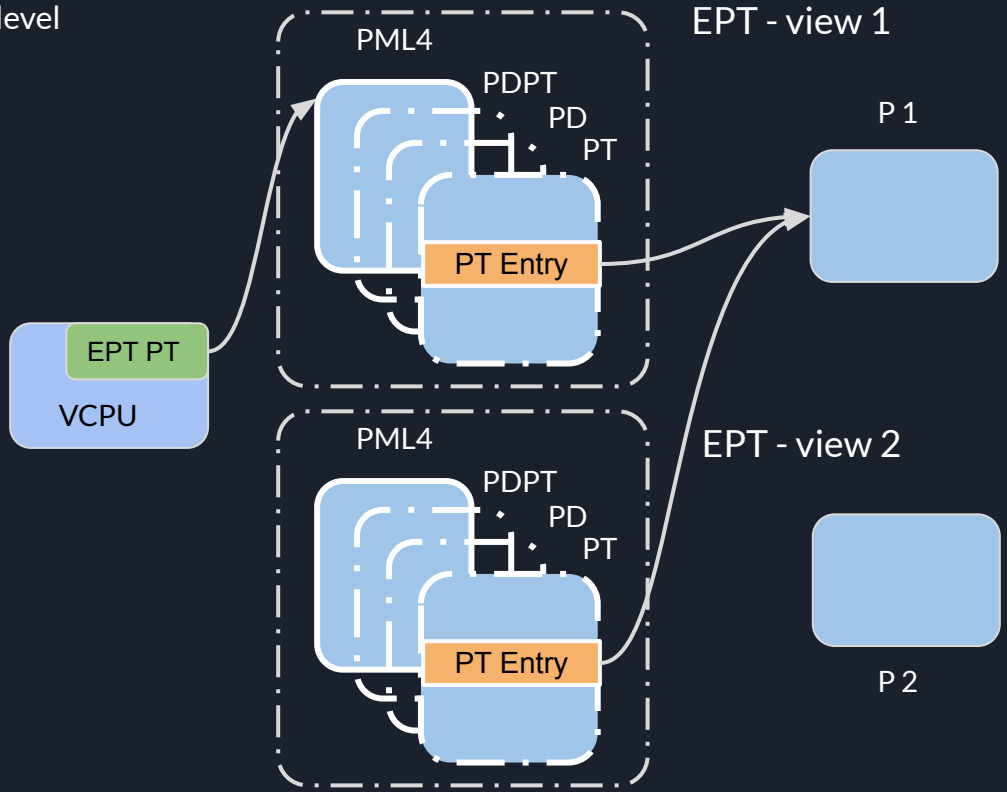
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1



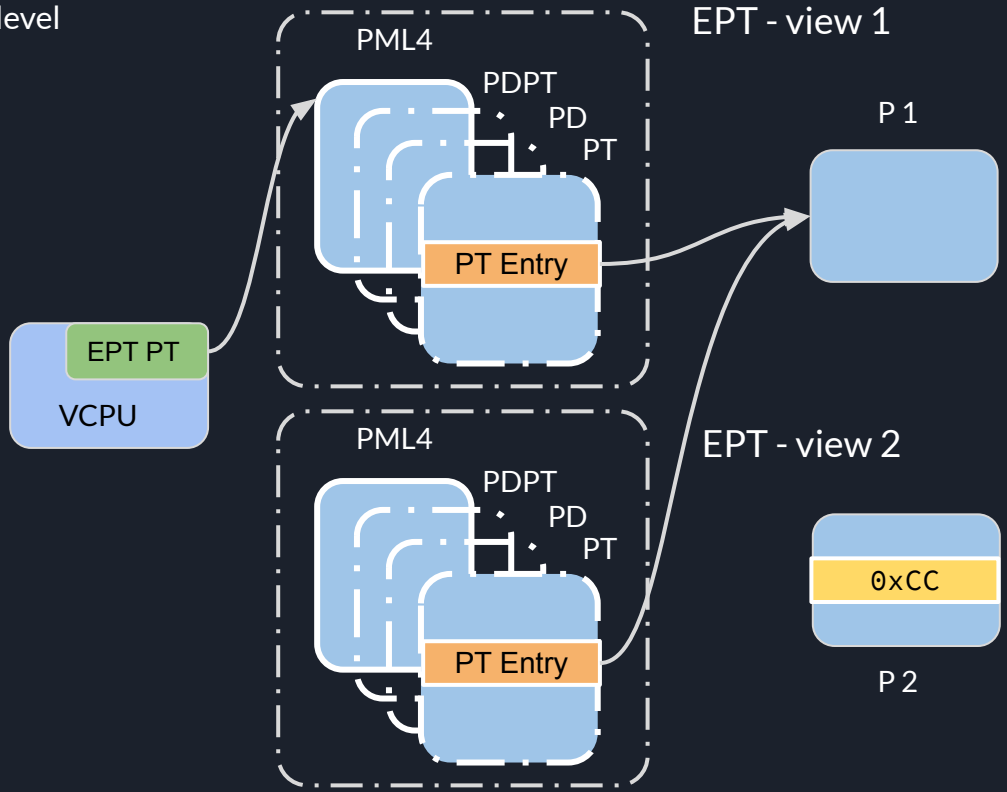
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2



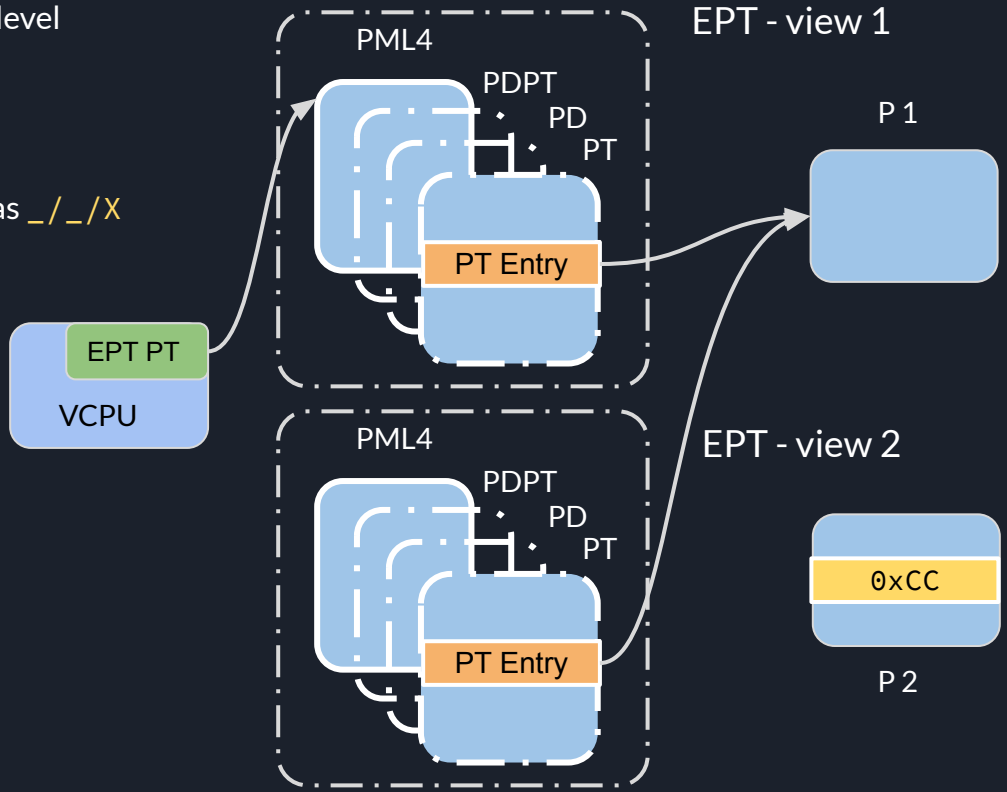
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2



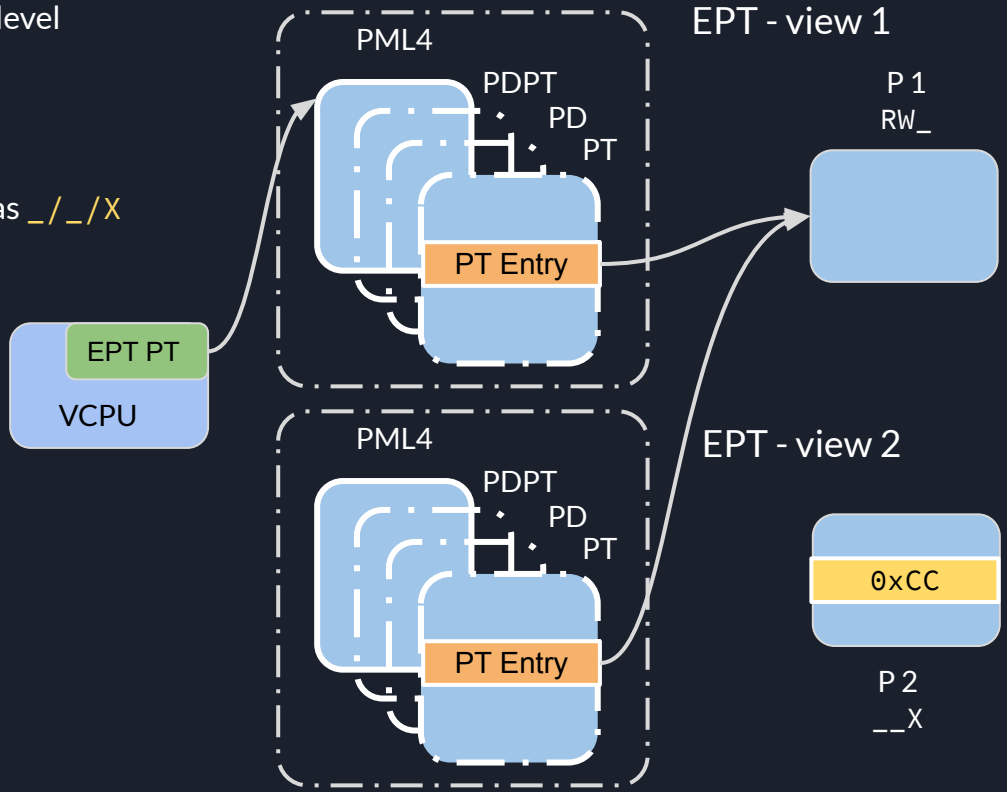
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`



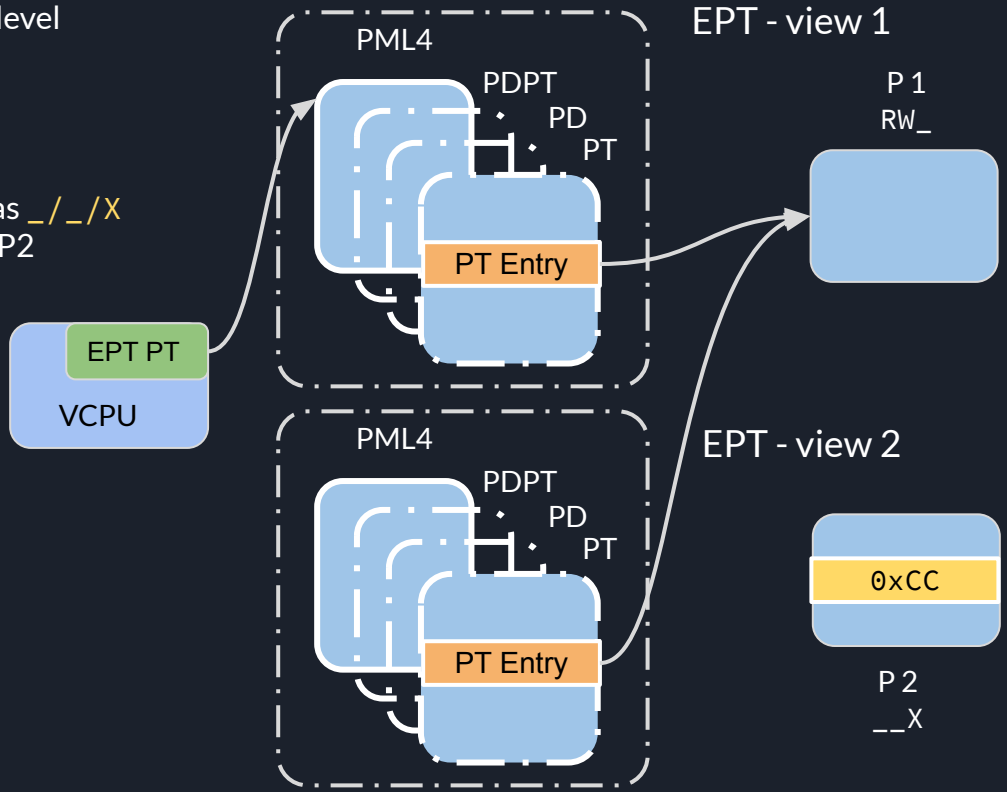
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`



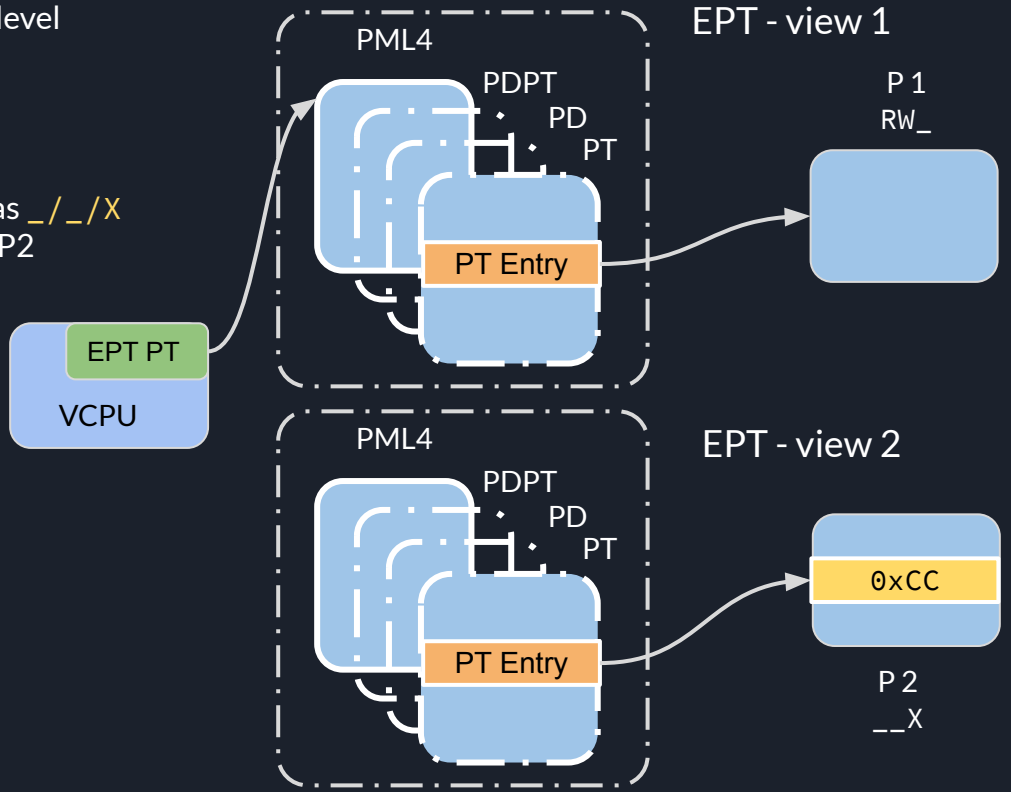
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2



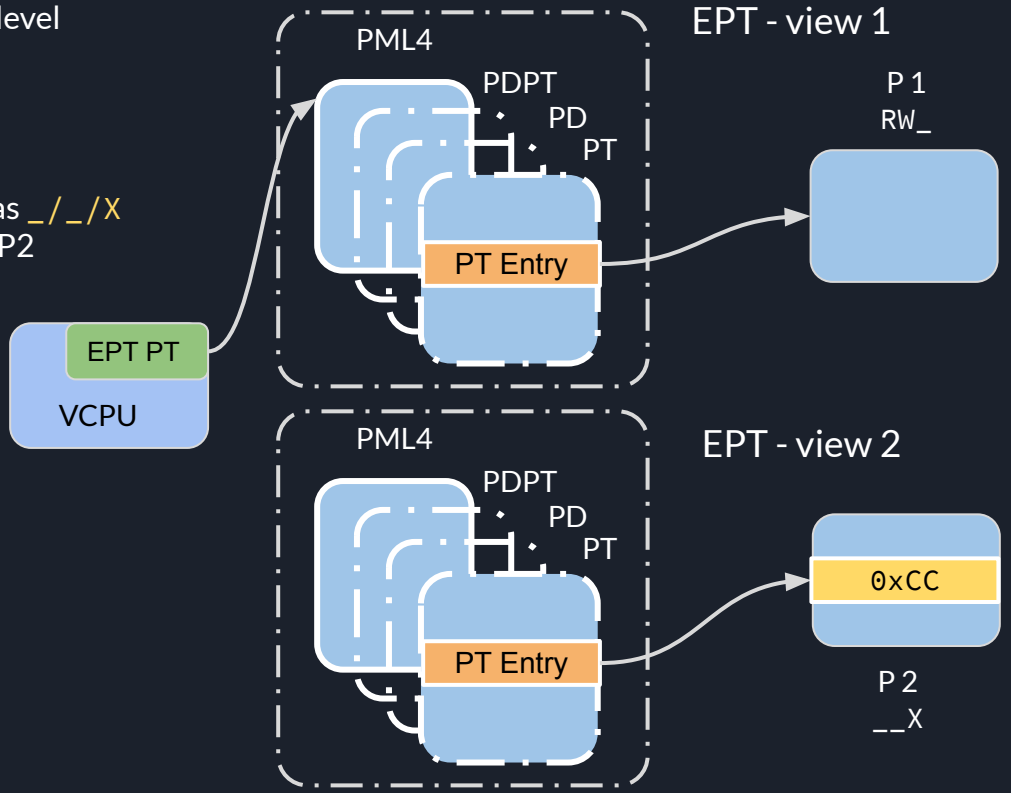
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2



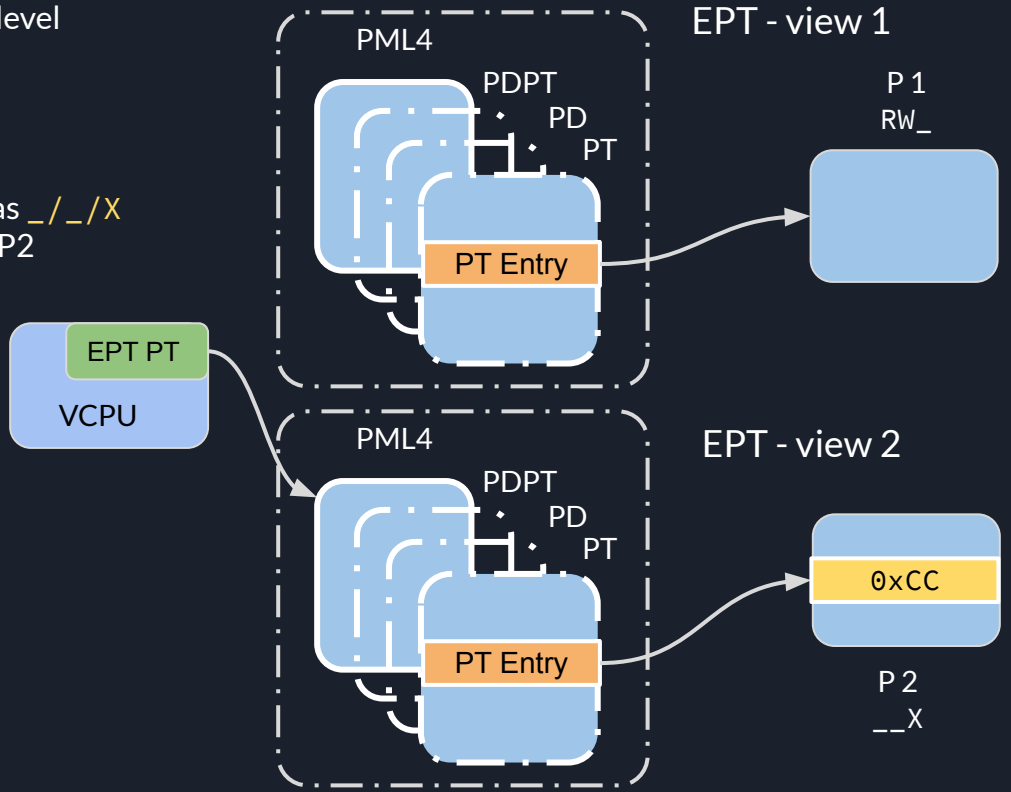
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2



# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

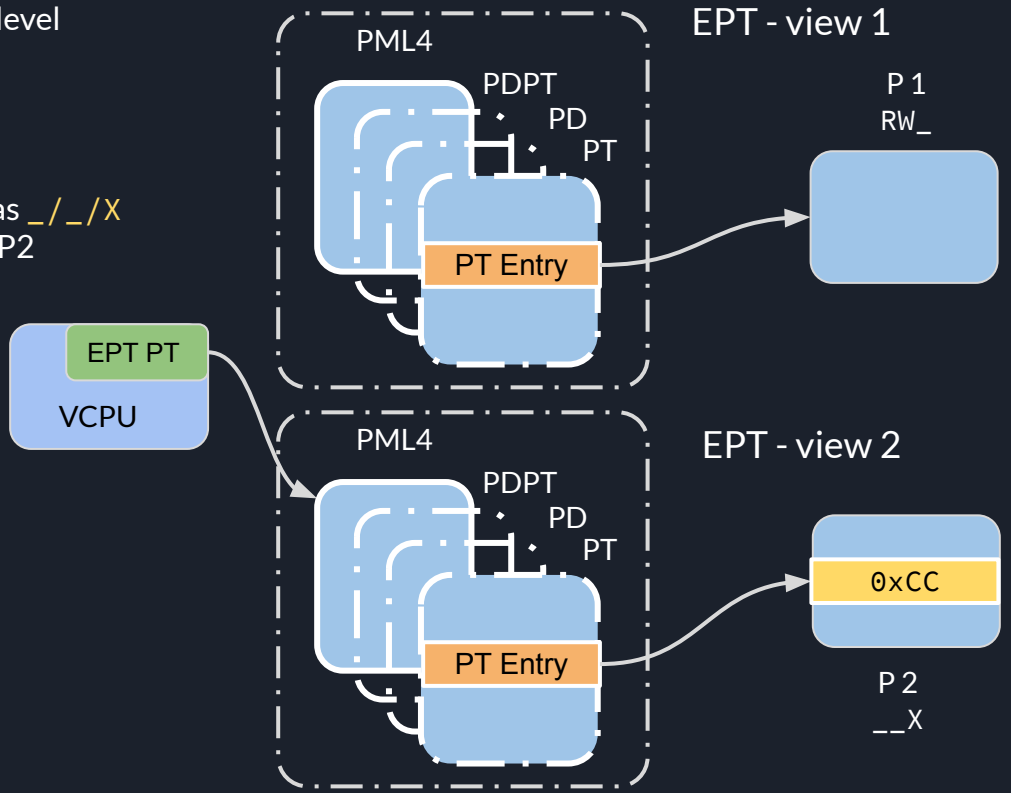


# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

*#int3*

- handle breakpoint

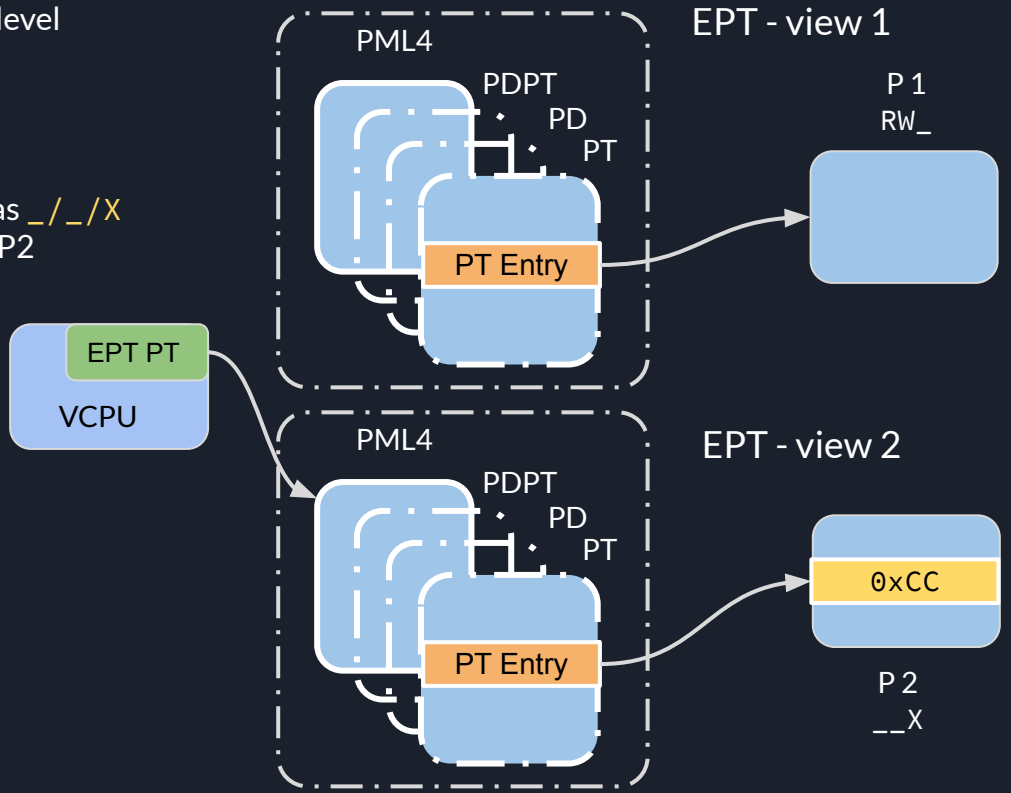


# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1

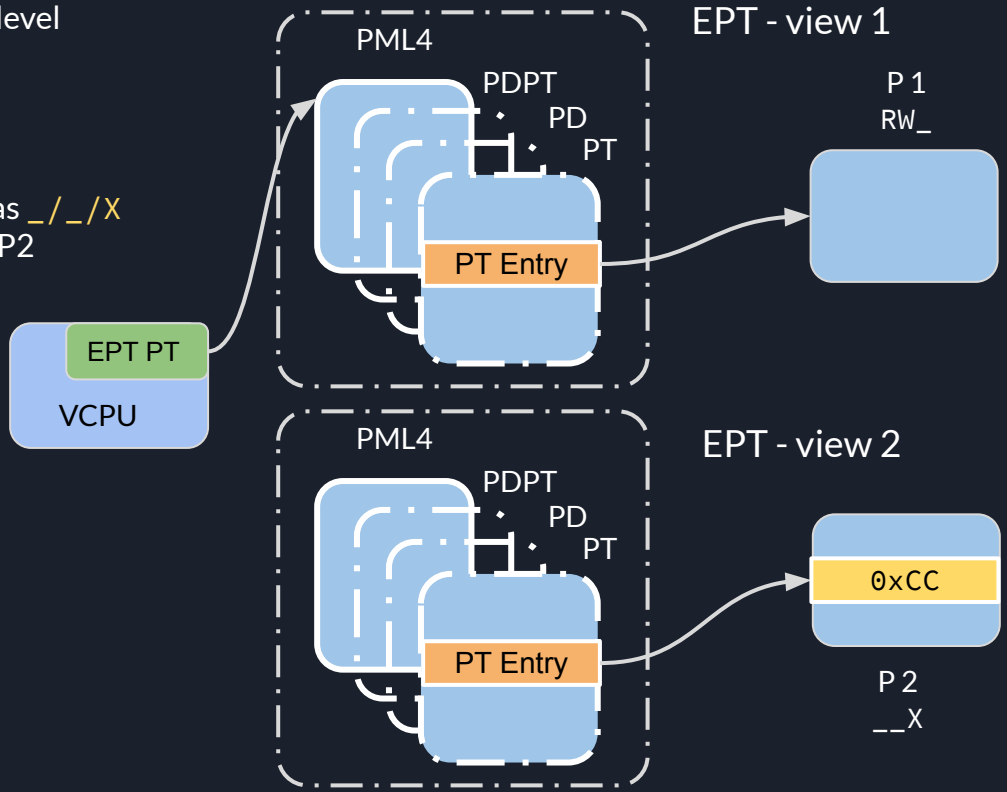


# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1



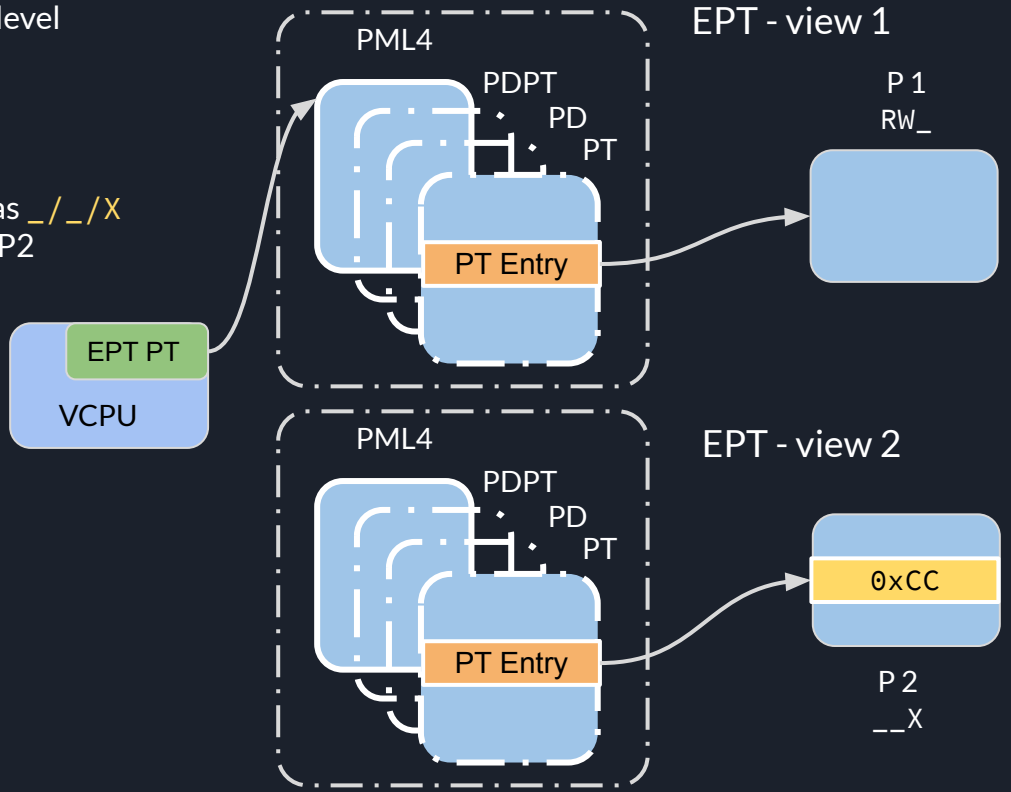
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1

*#EPTViolation* (Execute)
- switch VCPU to view 2



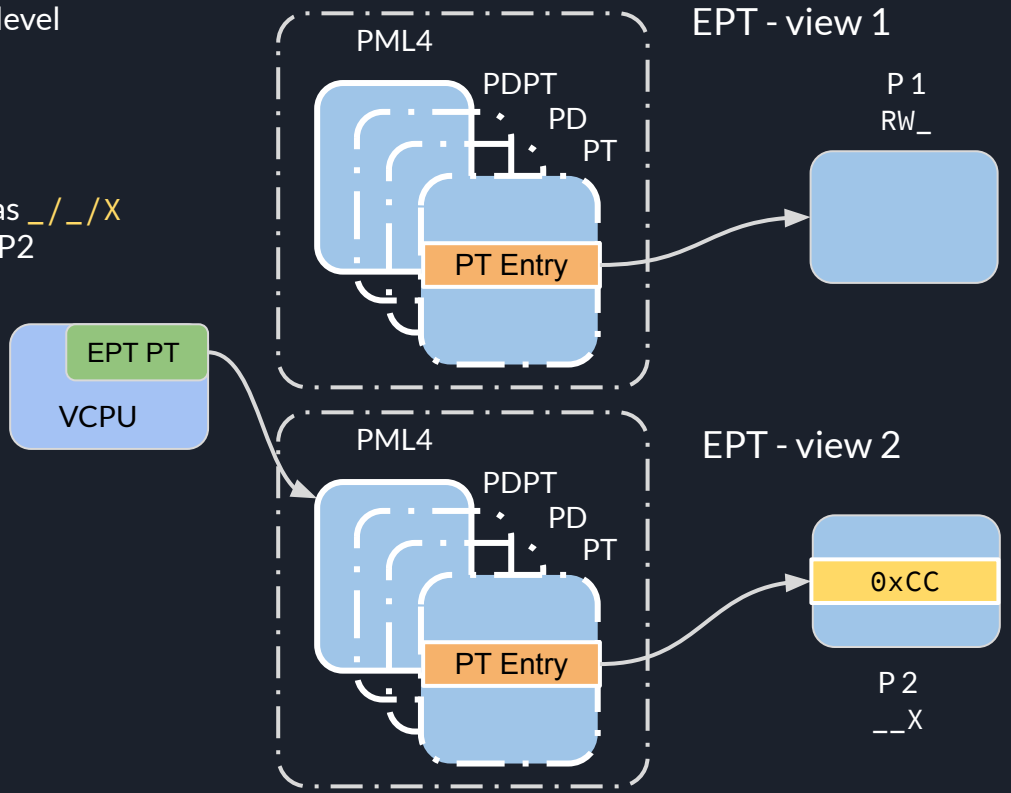
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1

*#EPTViolation* (Execute)
- switch VCPU to view 2



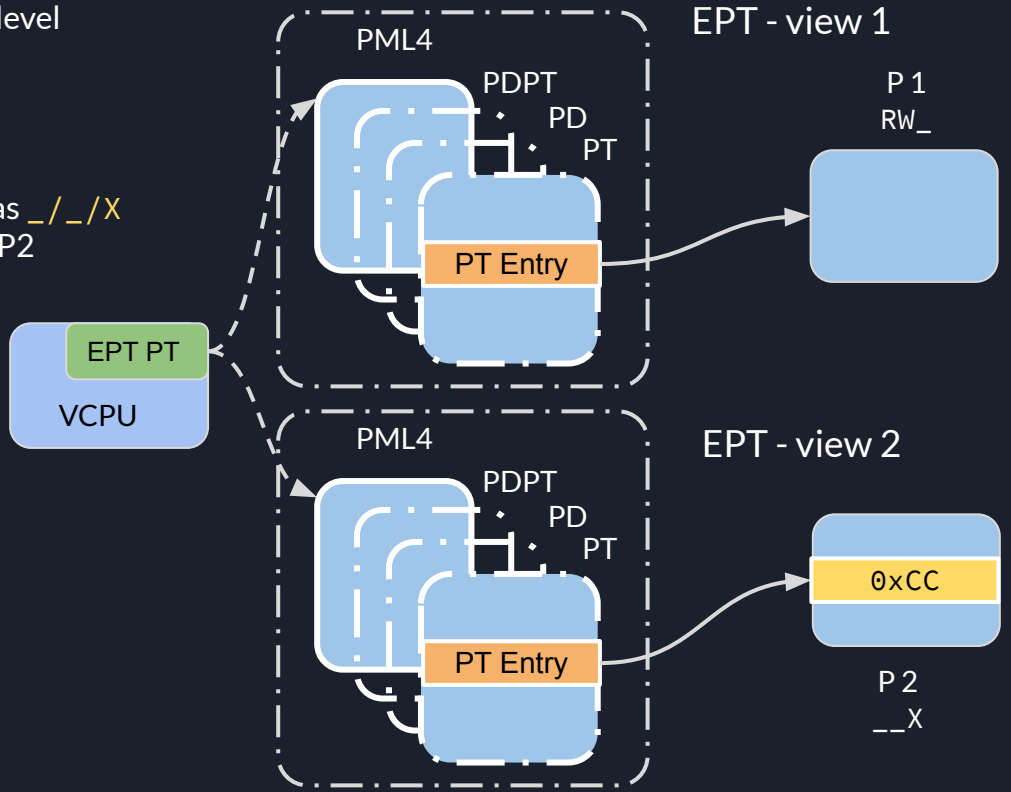
# Advanced SoftHyperBreakpoint

- Switch memory views at VCPU level
- Setup:
  a. duplicate EPT
  b. duplicate P1
  c. insert breakpoint in P2
  d. set P1 as `R/W/_` and P2 as `_/_/X`
  e. point view 2 PT Entry to P2
  f. switch VCPU to view 2

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1

*#EPTViolation* (Execute)
- switch VCPU to view 2



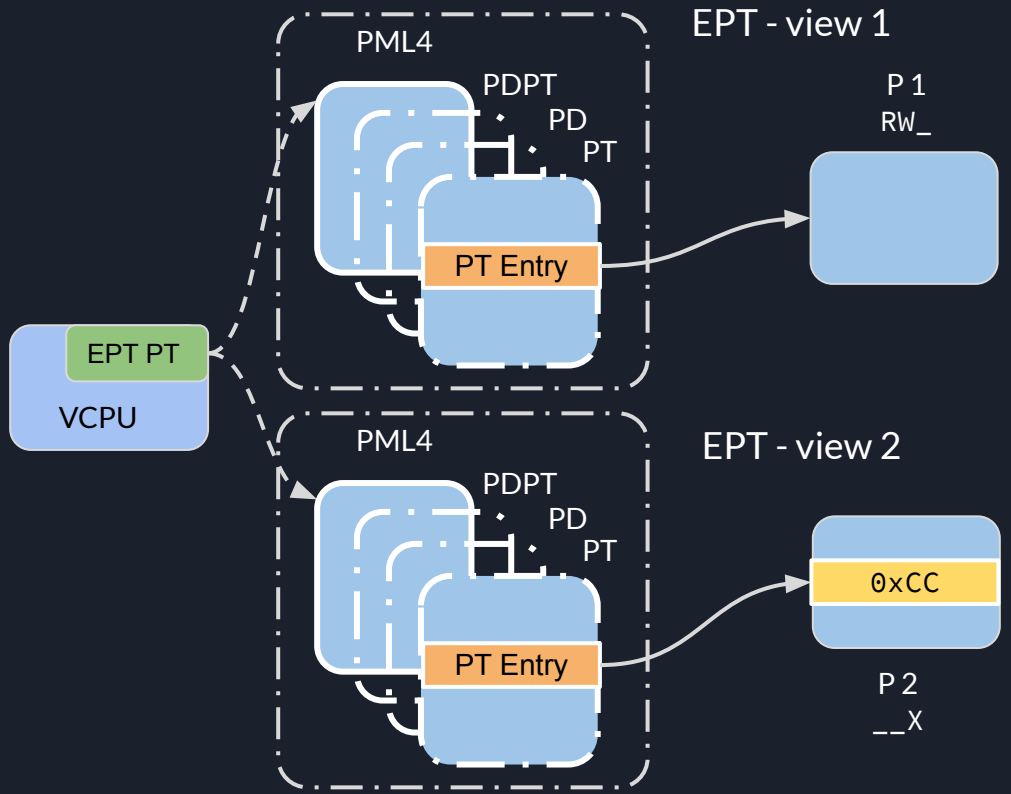
# Advanced SoftHyperBreakpoint

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1

*#EPTViolation* (Execute)
- switch VCPU to view 2

- ✔ fast
- ✔ stealth
- ✔ safe



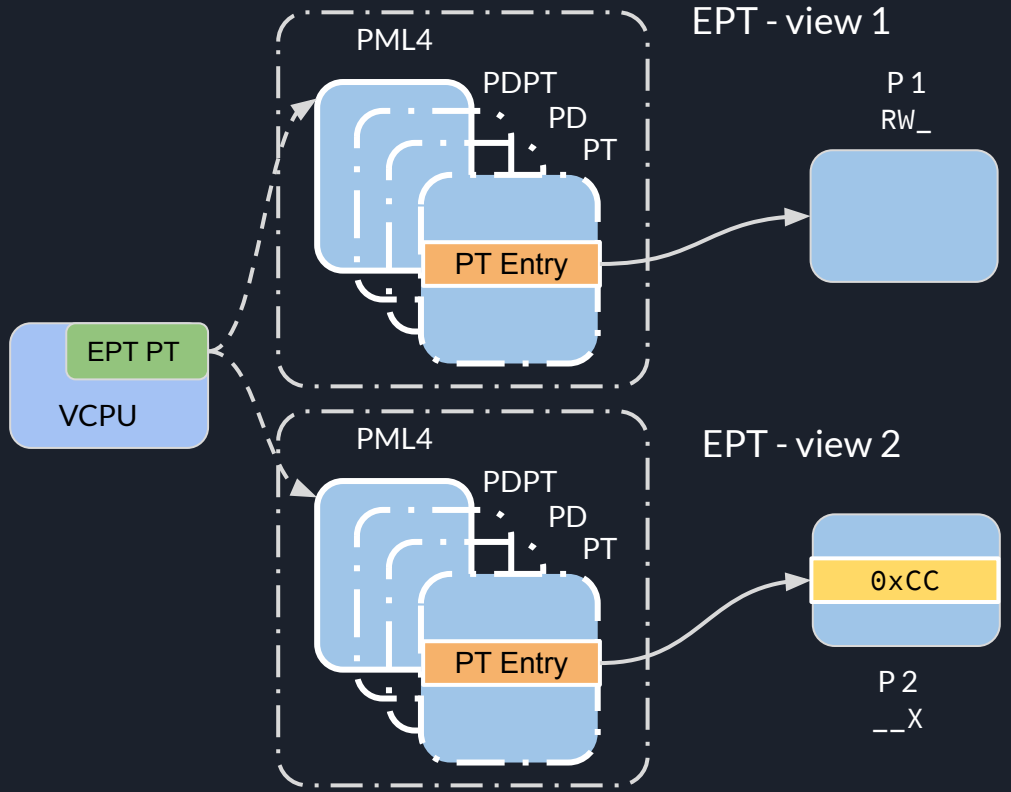
# Advanced SoftHyperBreakpoint

*#int3*
- handle breakpoint

*#EPTViolation* (Read/Write)
- switch VCPU to view 1

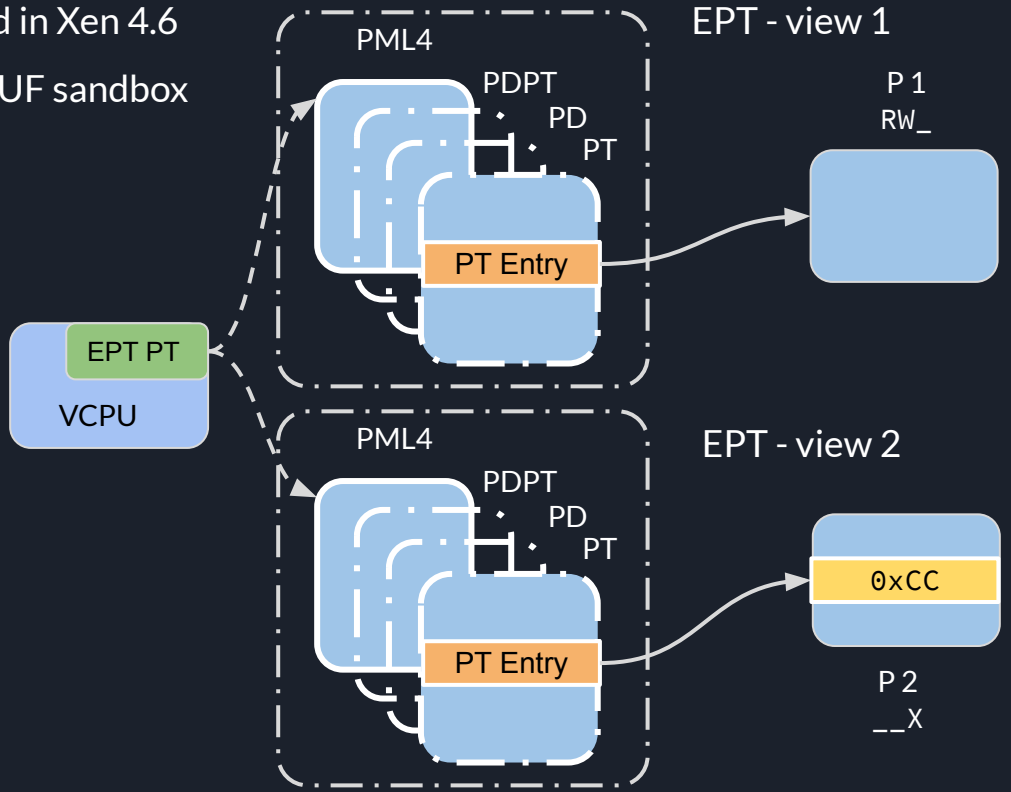
*#EPTViolation* (Execute)
- switch VCPU to view 2

- ✔ fast
- ✔ stealth
- ✔ safe



# Advanced SoftHyperBreakpoint

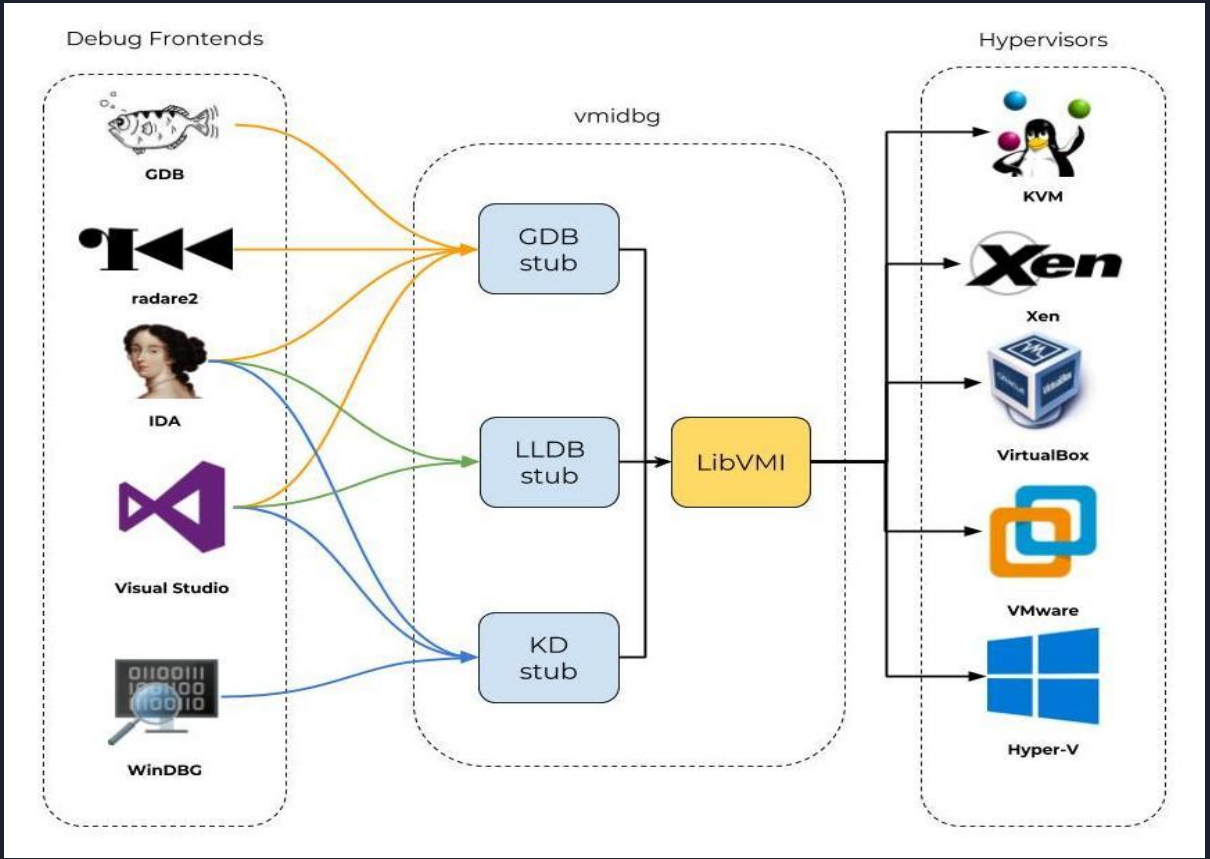
- 2015 - altp2m implemented in Xen 4.6
- 2016 - Available in DRAKVUF sandbox



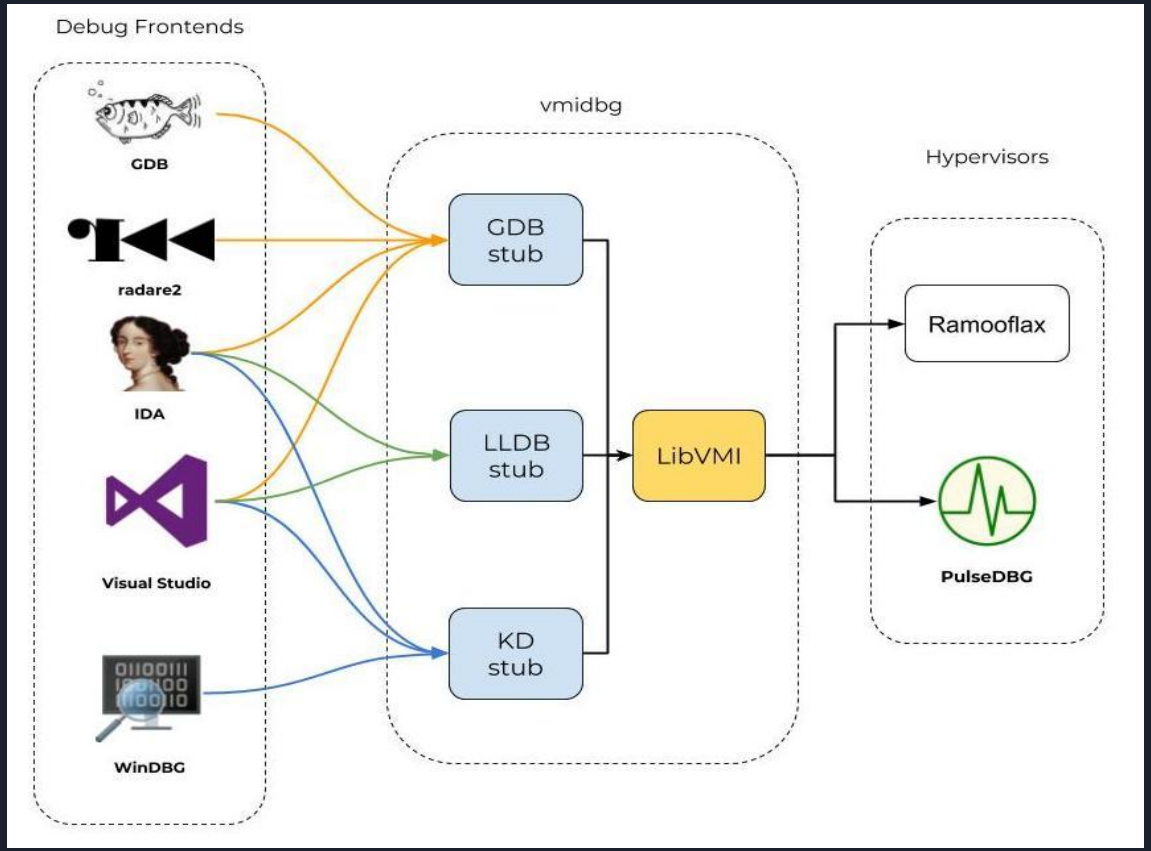

- ✔ fast
- ✔ stealth
- ✔ safe

# Future ?

# Flexible Full-system Debugging in 202X ?



# Flexible Boot-sequence Debugging



# LibVMI: design and future goals

- Switched to CMake recently
  - opens Windows drivers
- New ambitions
  - cross-platform
    - adapt to Windows/Linux/MacOS hypervisors
  - embedded use-case
    - ramooflax/PulseDBG
  - safety concerns
    - runs with very high-privileges (root for Xen APIs)
    - processes untrusted input, likely dangerous (malware)
- Long-term: reimplementation in Rust ?
  - to be discussed
  - call to rust developers !

# Getting Involved

- Hypervisor developers
  - open Virtual Machine Introspection API
  - VMware, Hyper-V, VirtualBox
- LibVMI
  - write a driver
  - improve introspection capabilities
  - performance study for Winbagility integration
  - long-term: rewrite in Rust
- Pyvmidbg
  - improve GDB stub
  - write LLDB/KD stubs
  - Windows/Linux internals (scheduling, process state)
  - join the Slack ! (ask for invites)
    - `https://vmidbg.slack.com`

# Conclusion

- For hypervisor developers
  - provides a unified debugger interface
  - leverage the full power out of your hypervisor
  - focus on the right abstractions, together
- For VM users
  - transform your vision of the Virtual Machines as another process tree
  - keep the same debug environment and tools for multiple OS
- For everyone
  - build hypervisor-level debugging framework as a commodity

# Thanks 🙏

- Saar Amar
- @J00ru
- Petr Beneš
- Artem Shishkin
- @aionescu
- @JmpCallPoo
- Tamas K Lengyel

# Questions

# Building a {flexible} hypervisor-level debugger

https://github.com/**Wenzel/pyvmidbg**