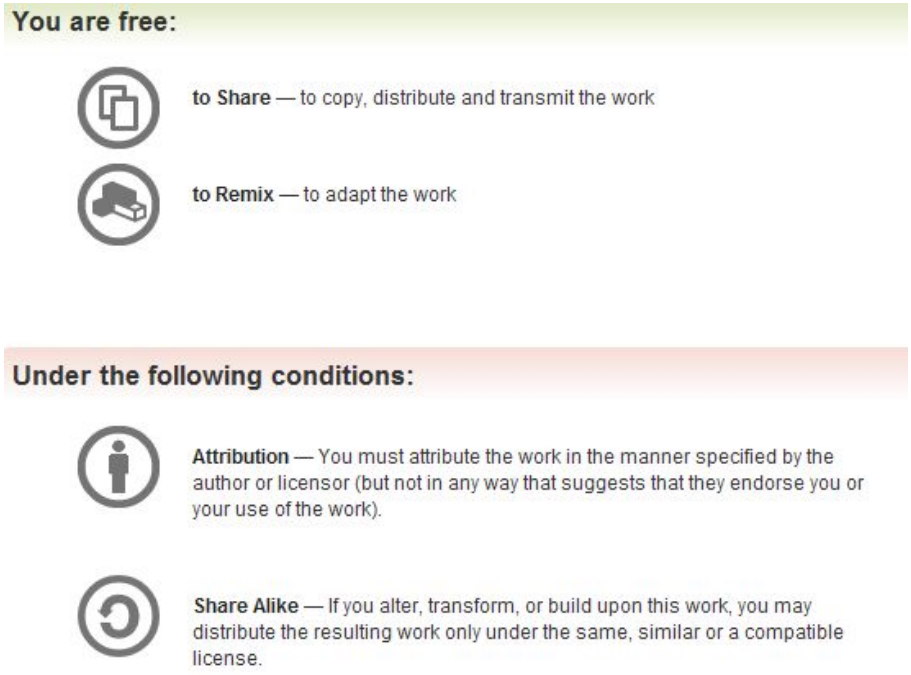
# Virtual Machine Introspection



# #whoami

Tamas K Lengyel - Co-Founder @ Zentific

- tamas@tklengyel.com, @tklengyel

Jacob Torrey - Advising Research Engineer @ Assured Information Security, Inc

- jacob@jacobtorrey.com, @JacobTorrey

# Goals

Understand modern hardware architectures to develop new security tools with

Introduce participants to existing open-source platforms and tools to build on

Draw attention to limitations and potential pitfalls

Look ahead to what’s next

# Agenda

1. Introduction to Virtualization and Xen
2. Virtual Memory and Paging — 2 hours | 10 minute break
3. Trappable events
   a. Two-stage paging
   b. x86 registers — 1 hour | 1 hour lunch break
4. LibVMI
   a. Four exercises
   b. Debugging and reversing — 4 hours | breaks as needed
5. There be dragons
   a. Issues and corner-cases
6. Intel #VE — 1 hour (if we get to them)

# Questions

Ask questions as soon as you got them!

- If you get lost you will stay lost, so fire them ASAP!

No such thing as a stupid question ;)

Browsing the web and/or checking email during class is a great way to get lost

# Introduction

Virtualization is supported on most of our modern hardware

Virtualization is built into most of our modern operating systems (both Windows and Linux)

Lots of virtualization platforms available for free

What security tools do you know of that actively use virtualization? Or the opposite, what malware?

# Terminology

| | |
|---|---|
| **VMX** | Virtual Machine Extensions |
| **VMM** | Virtual Machine Monitor, Hypervisor |
| **VMX Root** | CPU mode of VMM |
| **dom0** | Management VM |
| **domU** | VMX Non-root, guest VM |

# Virtualization is Resource Abstraction, yo!

"The process of hiding the underlying physical hardware in a way that makes it transparently usable and shareable by multiple operating systems." [IBM]

- Most hardware can be virtualized to the point that a guest doesn't know/care
- Underlying physical hardware supporting VM may not be dedicated to it

[IBM] http://www.ibm.com/developerworks/linux/library/l-hypervisor/

# Xen Full Virtualization Architecture

With the para-virtualized drivers

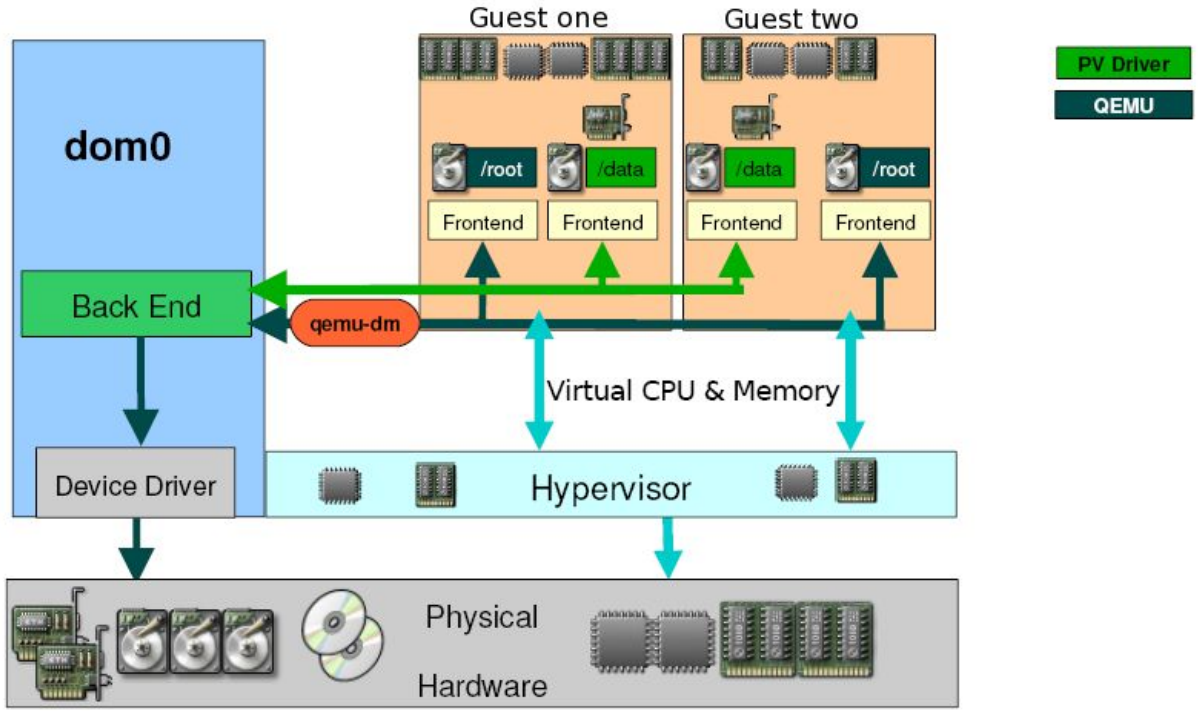



Image credit: https://docs.fedoraproject.org/en-US/Fedora/12/html/Virtualization_Guide

# CPU privilege level

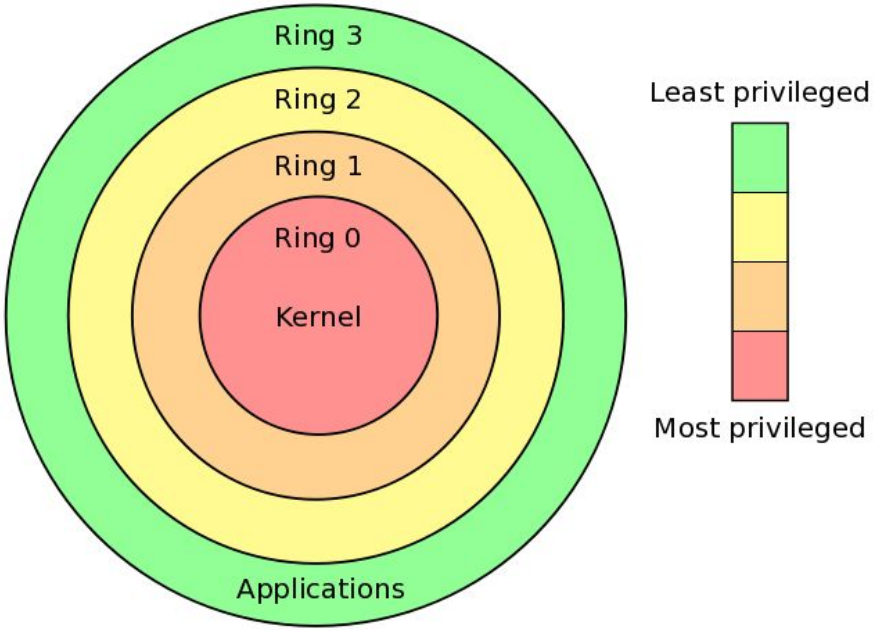
Has nothing to do with Operating System users (admin, root, etc.)

Nothing to do with “rings” either

Pre-virtualization world: 4 rings with only 2 being used by most systems*

What ring are you in?

- Code Segment register (CS) holds Current Privilege Level (CPL) in bits 0:1




* System management mode will be ignored for now

# VMX on x86

VMX: Introduce a parallel set of rings

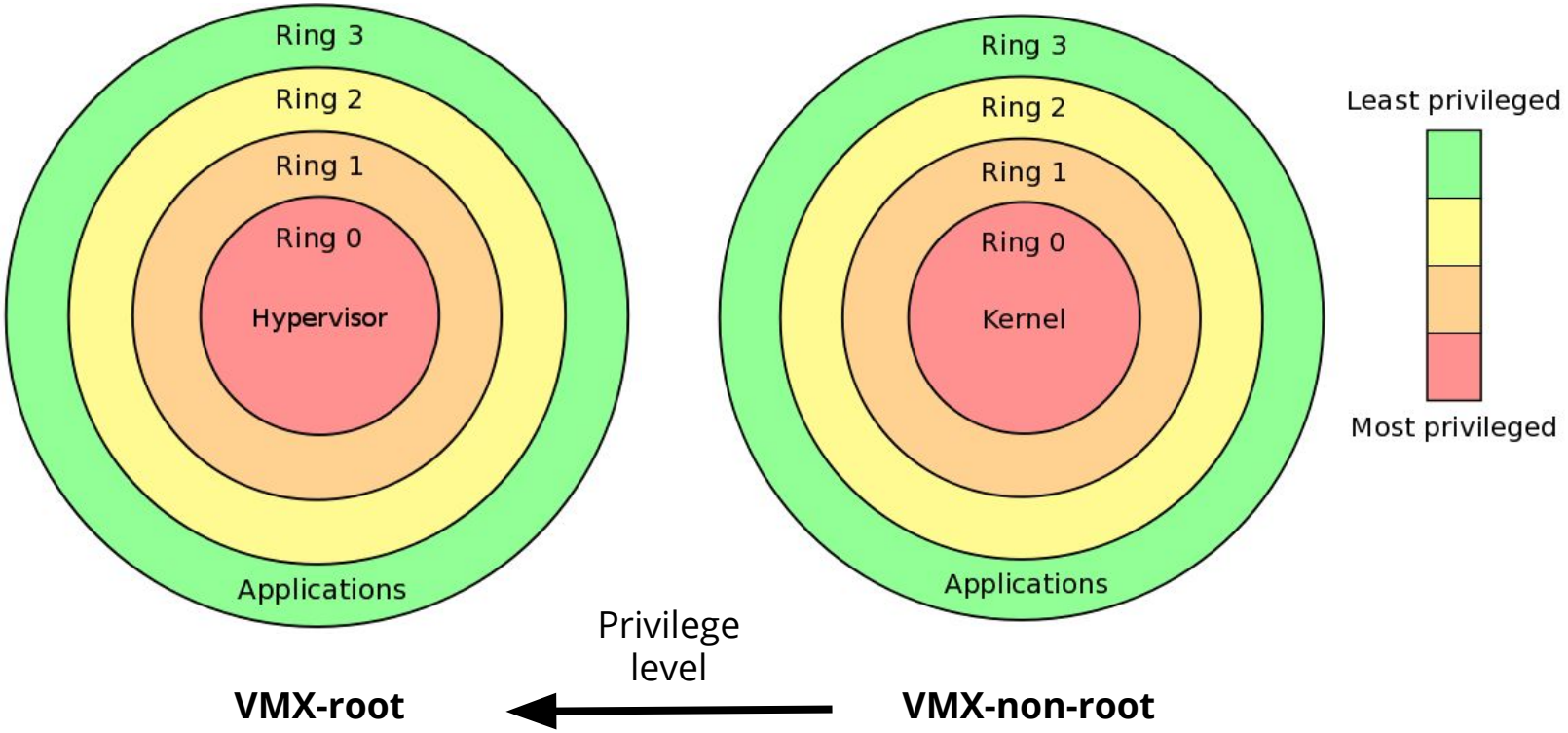
1st set of rings: VMX-root

- Hypervisor in VMX-root ring0
  - Can be just a kernel driver in a larger Operating System
- If you run things in ring3 here that’s what’s called type-2 VMM
  - KVM, VirtualBox, VMWare Player, etc.

2nd set of rings: VMX-non-root

- Guest VM
- Can be used however the guest sees fit (with some restrictictions)

# VMX rings


Ring 3
Ring 2
Ring 1
Ring 0
Hypervisor
Applications
Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Applications
Least privileged
Most privileged
Privilege level
VMX-root
VMX-non-root

# VMX on ARM

Only 2-rings were available to start with: User and Kernel

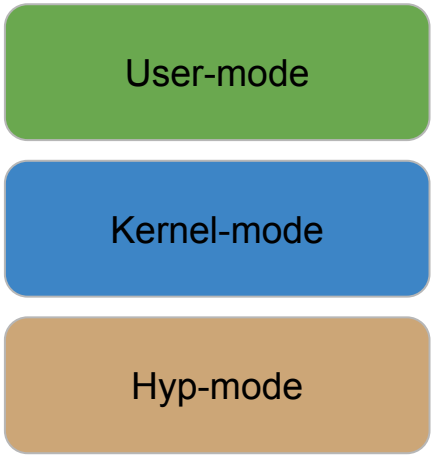
- That's how most systems used x86 anyway so not much difference

VMX introduces a third ring: Hyp

- Not a full parallel set of rings as on x86
- Makes type-2 hypervisors more "exotic"
- Perfect architecture for Xen and type-1

There *is* a parallel mode on ARM too though!

- ARM TrustZone
- We will ignore this for now



# Popek and Goldberg Virtualization Criterion

POPEK, G. J., GOLDBERG, R. P., “Formal requirements for virtualizable third generation architectures,” ACM Communications, July 1974

- Equivalence / Fidelity
  A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
- Efficiency / Performance
  A statistically dominant fraction of machine instructions must be executed without VMM intervention.
- **Resource control / Safety**
  **The VMM must be in complete control of the virtualized resources.**

# Guest has access only to a subset of instructions

Required to assure safety criterion

- Instructions controlling the virtualization extensions ALWAYS trap (VMXOFF, VMLAUNCH, VMCLEAR, etc..)
- Hypervisor may define additional traps as well

Guest runs until

- It does something that has been registered in data structures (i.e., VMCS) to exit out to VMM,
- It explicitly calls the VMCALL instruction

VMM can preempt guest regularly with a timer or when the guest is in a PAUSE-loop

# Virtual Machine Control Structure (VMCS)

Transition to and from guest has to preserve CPU state of previous mode

Virtual Memory Control Structure (VMCS) defined for each vCPU to store that state

The structure is in the memory space of the VMM but should only be read/written by special instructions (VMREAD/VMWRITE)

Also controls the optional trap configuration for the VM

# VMX flow

VMM will take these actions

1. Initially enter VMX mode using VMXON
2. Clear guest's VMCS using VMCLEAR
3. Load guest pointer using VMPTRLD
4. Write VMCS parameters using VMWRITE
5. Launch guest using VMLAUNCH
6. Guest exit (VMCALL or instruction, ...)
7. Read guest-exit info using VMREAD from VMCS
8. Maybe reenter guest using VMRESUME
9. Eventually leave VMX mode using VMXOFF

# Xen commands bootstrap

| | |
|---|---|
| xen-detect | Check if and what version of Xen you are running with |
| xl list | View the list of running VMs |
| xl create <cfg> | Start a new virtual machine with the configuration file specified |
| xl destroy <vm name or ID> | Destroy (without shutting down) the guest VM |
| xl save <vm name or ID> <filename> | Save the state of the running VM into a file |
| xl restore <filename> | Restore the VM from the file |
| vncviewer <ip:port> | Connect to the VNC interface of the VM |

# Virtual Memory and Paging

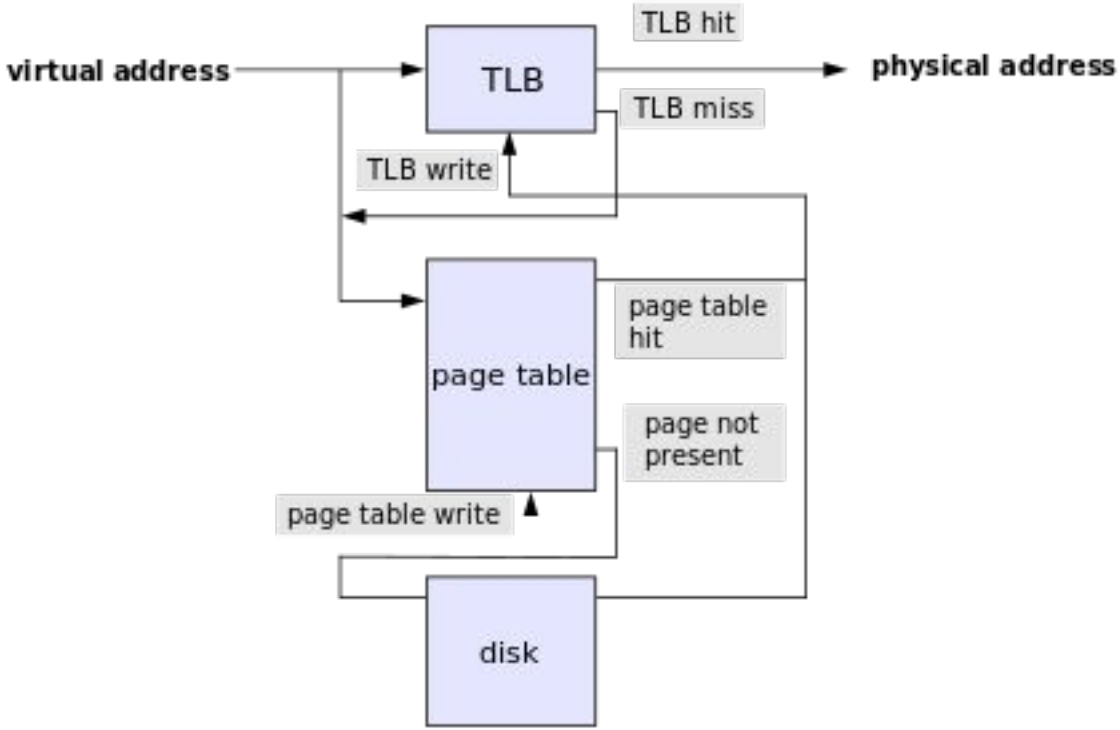
Has been introduced in Intel 286 Protected Mode

- Don’t allow programs to corrupt each others’ memory space
- Flat empty memory space for each process
- Kernel usually mapped into the upper regions for performance reasons
- On 32-bit paging kernel normally mapped above 0x80000000
- On 64-bit paging mapped above 0xffffffff80000000

Virtual address needs to be translated to physical address

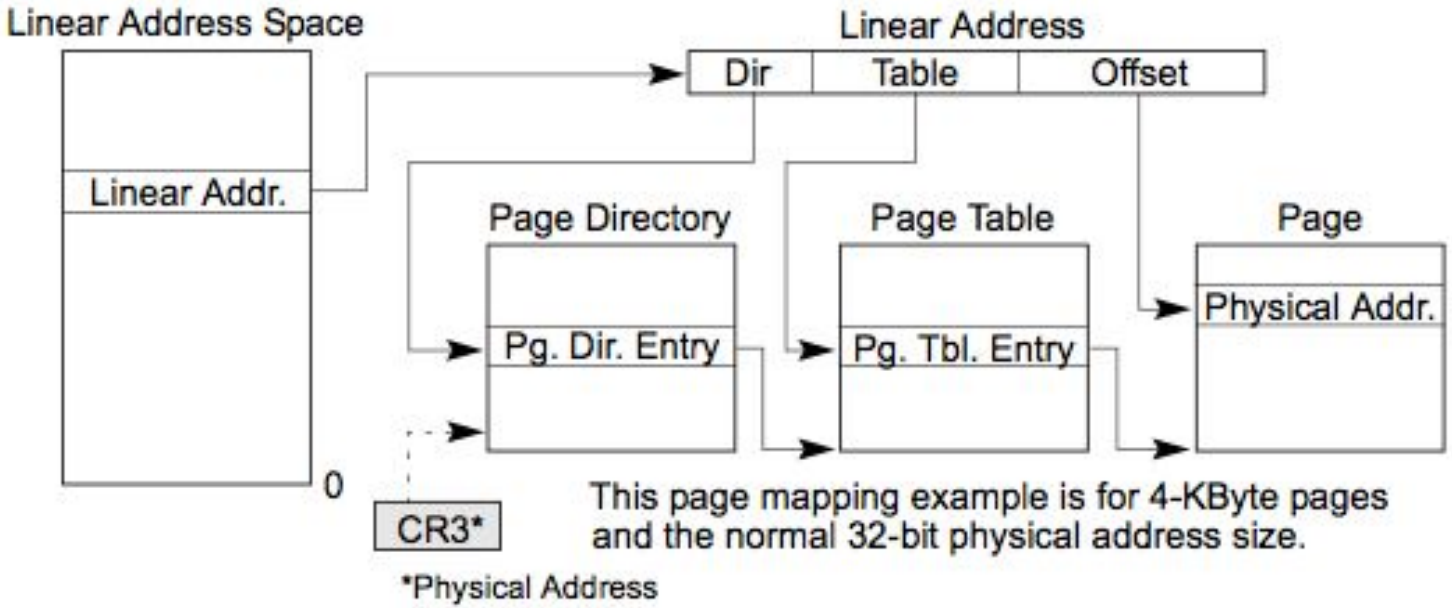
- Paging structures setup by the kernel to be used for lookup
- Hardware automatically traverses the tables pointed to by a register
- Translation result is cached in the Translation Lookaside Buffer (TLB)

# Translation process

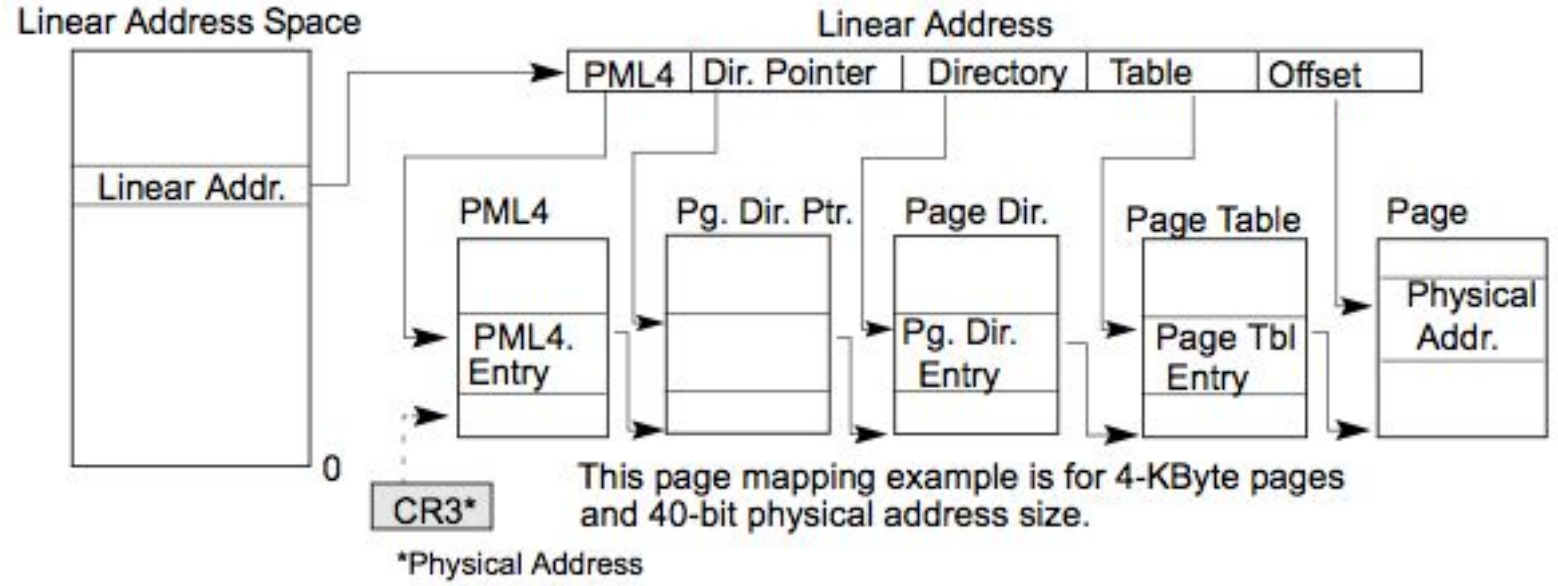



https://en.wikipedia.org/wiki/Page_table#The_translation_process

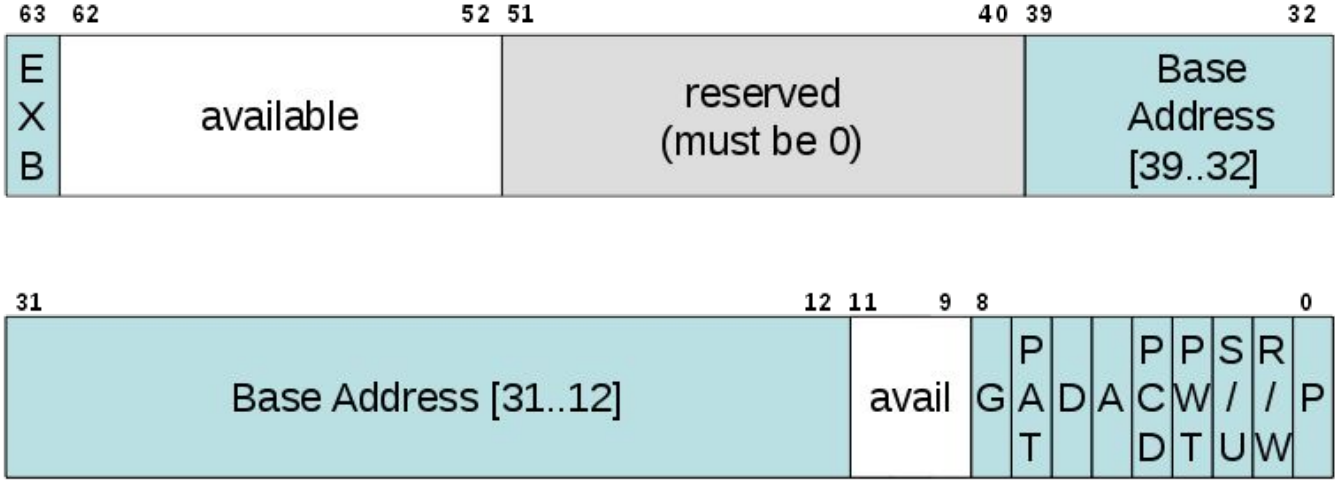
# 32-bit pagetables



# 64-bit pagetables (IA32e)



# Page Table Entry (PTE)




| | |
|---|---|
| P = present (0=no, 1=yes)<br>PWT = Page Write-Through (0=no, 1=yes)<br>R/W (0=read-only, 1=writable)<br>PCD = Page Caching Disable (0=no, 1=yes)<br>S/U (0=supervisor-only, 1=user) | PAT = Page-Attribute Table-Index<br>A = accessed (0=no, 1=yes)<br>G = Global page (1=yes, 0=no)<br>D = dirty (0=no, 1=yes)<br>EXB = Execute Disable |

# Translation as seen in C

```
for (level = pt_levels; level > 0; level--) {
        paddr += ((virt & mask) >> (xc_ffs64(mask) - 1)) * size;
        map = xc_map_foreign_range(xch, dom, PAGE_SIZE, PROT_READ, paddr >>PAGE_SHIFT);
        if (!map) return 0;
        memcpy(&pte, map + (paddr & (PAGE_SIZE - 1)), size);
        munmap(map, PAGE_SIZE);
        if (!(pte & 1)) return 0; // Note: This may actually break some valid lookups on Windows
        paddr = pte & 0x000ffffffffff000ull;
        if (level == 2 && (pte & PTE_PSE)) {
                mask = ((mask ^ ~-mask) >> 1); /* All bits below first set bit */
                return ((paddr & ~mask) | (virt & mask)) >> PAGE_SHIFT;
        }
        mask >>= (pt_levels == 2 ? 10 : 9);
}
return paddr >> PAGE_SHIFT;
```

http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=tools/libxc/xc_pagetab.c

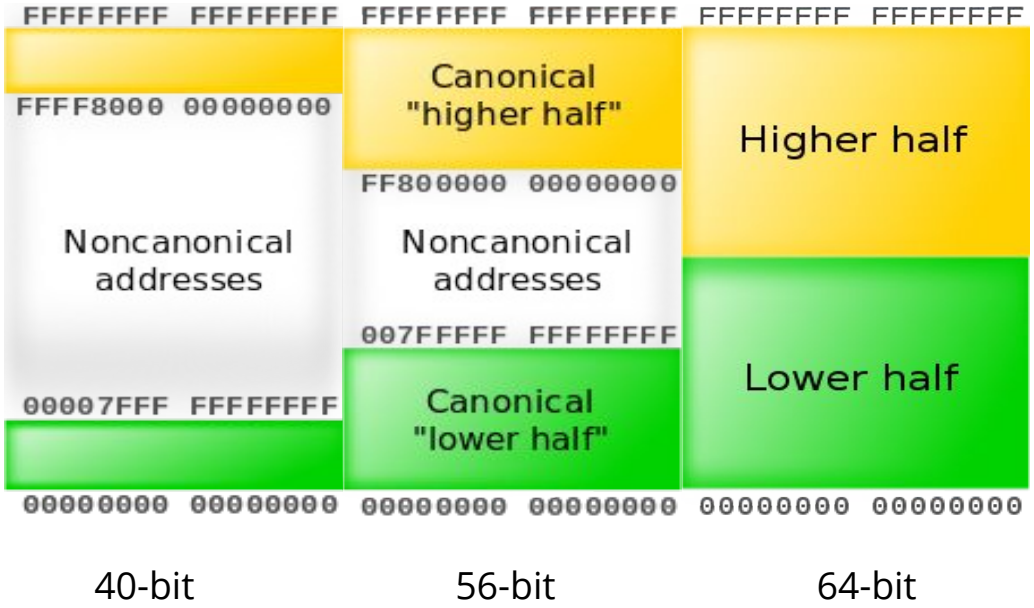
# Canonical 64-bit virtual address

All current CPUs use 40-bit VAs

The CPU checks for each VA whether VA[48:63] == VA[47]

For the translation itself VA[48:63] has no relevant information and we can just ignore it

It is not uncommon for some forensics tools (for example Rekall) to only show you the first 48-bits of all pointers



# Tricks with the translation

The translation process allows for some nifty tricks

- Memory paging to disk
  Offload memory not actively used to free up space (swap, pagefile, etc.)
  Done slightly differently in Windows and Linux, where Windows keeps some pages ready to be paged-out “in transition” using optional PTE bits
- Memory sharing
  The same physical page can back multiple virtual pages!
  No need to load the same library separately for multiple process’
  Works for code-pages and static (read-only) data pages
  Or for efficient IPC

# Virtual-virtual memory

A guest doesn’t see the real physical memory directly as that would violate our safety criterion

What can we do?

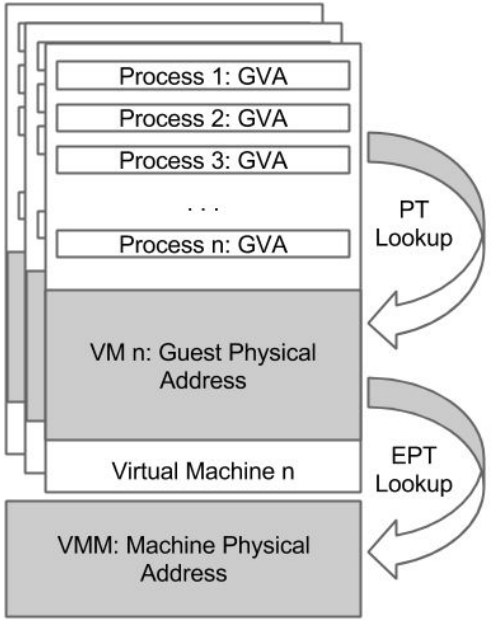
- Shadow page-tables: trap every page-table update made by the guest and maintain a copy with different mappings
- OS sets up V->P, VMM sets up V->M
- Traps all CR3 update to overload with shadow pointer
- Heavy performance overhead

# Two-stage paging

EPT on Intel, RVI on AMD. On ARM it has doesn't have a separate name, it's just part of the virtualization extensions.

Introduce another hardware-assisted lookup for guest-physical to machine-physical

- Hardware first translates V->P using tables in the guest
- Then P->M (p2m) using tables in the VMM
- All done by the hardware, no need to involve the VMM
- Better performance



# Extended Page Tables (EPT)

EPT similar to the 64-bit page-table layout discussed previously

EPT pointer (EPTP) is stored in the VMCS

A violation of EPT PTE permissions trap to the VMM with a VMEXIT EPT_VIOLATION code

Most significant difference is that you can mark a page *execute only* in the EPT PTE!

Allows for trapping when the guest reads from the page

# EPT PTE as defined by Xen

```
typedef union {
        struct {
                u64 r   :   1,  /* bit 0 - Read permission */
                w       :   1,  /* bit 1 - Write permission */
                x       :   1,  /* bit 2 - Execute permission */
                emt     :   3,  /* bits 5:3 - EPT Memory type */
                ipat    :   1,  /* bit 6 - Ignore PAT memory type */
                sp      :   1,  /* bit 7 - Is this a superpage? */
                a       :   1,  /* bit 8 - Access bit */
                d       :   1,  /* bit 9 - Dirty bit */
                recalc  :   1,  /* bit 10 - Software available 1 */
                snp     :   1,  /* bit 11 - VT-d snoop control in shared
                                    EPT/VT-d usage */
                mfn     :  40, /* bits 51:12 - Machine physical frame number */
                sa_p2mt :   6,  /* bits 57:52 - Software available 2 */
                access :   4,  /* bits 61:58 - p2m_access_t */
                tm      :   1,  /* bit 62 - VT-d transient-mapping hint in
                                    shared EPT/VT-d usage */
                suppress_ve :   1;  /* bit 63 - suppress #VE */
        };
        u64 epte;
} ept_entry_t;
```

http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/include/asm-x86/hvm/vmx/vmx.h

# ARM LPAE as defined by Xen

```
typedef struct __packed {
        /* These are used in all kinds of entry. */
        unsigned long valid:1;      /* Valid mapping */
        unsigned long table:1;      /* == 1 in 4k map
                                       entries too */

        unsigned long mattr:4;      /* Memory Attributes */
        unsigned long read:1;   /* Read access */
        unsigned long write:1;  /* Write access */
        unsigned long sh:2;         /* Shareability */
        unsigned long af:1;         /* Access Flag */
        unsigned long sbz4:1;
        /* The base address must be appropriately
           aligned for Block entries */
        unsigned long long base:36; /* Base address of
                                       block or next
                                       table */
        unsigned long sbz3:4;

        /* These seven bits are only used in Block entries
         *and are ignored  in Table entries. */
        unsigned long contig:1;     /* In a block of 16
                                       contiguous entries */
        unsigned long sbz2:1;
        unsigned long xn:1;   /* eXecute-Never */
        unsigned long type:4;       /* Ignore by hardware.
                                       Used to store p2m
                                       types */

        unsigned long sbz1:5;
} lpae_p2m_t;
```

http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/include/asm-arm/page.h

# Xen-access demo

http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=tools/tests/xen-access/xen-access.c

# ./xen-access 22 write

PAGE ACCESS: rw- for GFN 3f46 (offset 00000c) gla 0000000083d4200c (valid: y; fault in gpt: n; fault with gla: y) (vcpu 0, altp2m view 0)

Got event from Xen

PAGE ACCESS: rw- for GFN 3f47 (offset 000004) gla 0000000083d43004 (valid: y; fault in gpt: n; fault with gla: y) (vcpu 0, altp2m view 0)

Got event from Xen

PAGE ACCESS: rw- for GFN 3f48 (offset 000018) gla 0000000083d44018 (valid: y; fault in gpt: n; fault with gla: y) (vcpu 0, altp2m view 0)

# How does it work?

Xen needs to know which EPT violations were deliberate and which ones are erroneous

EPT PTE bits 61:58 holds information about the EPT permission mask that was deliberately set and need to be forwarded

Access permissions need to be cleared to let the VM continue execution, otherwise it will keep faulting on the same memory access

Without some additional steps we will only be able to trace the first access on every page

# How does it work on ARM?

Xen needs to know which LPAE violations were deliberate and which ones are erroneous

Not enough bits in the PTE to store the custom permission mask directly

Binary-lookup tree defined for each domain to store the permissions

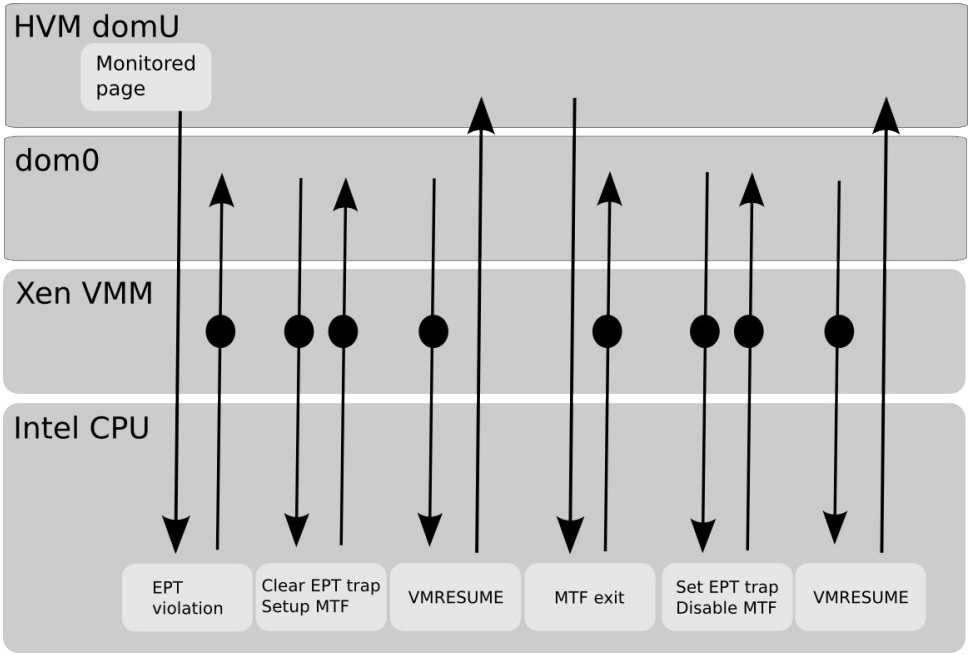
On a violation the lookup tree is queried to see if a corresponding entry is present

Tracing more than the first access of a page is not yet possible with Xen on ARM

# Continuous memory access monitoring on Xen

Monitor Trap Flag (MTF) allows stealthily single-stepping the vCPU

Only available for Intel processors



# x86 control registers

Only a handful or registers are trappable to the hypervisor

- CR0: CPU behavior control, like enabling protected mode (bit 0) and paging (bit 31), etc
- CR3: Holds the physical memory address of the pagetable of the current process
- CR4: More CPU behavior control, like SMEP, SMAP, PAE, PGE (TLB behavior), etc
- XCR0: Yet more CPU behavior control, FPU and SSE options

Technically both reads (MOV FROM) and writes (MOV TO) are trappable but Xen only supports trapping write events at the moment.

# x86 Model Specific Registers (MSR)

Registers that may not be present in all CPUs, all trappable by the VMM.

- EFER: CPU control register for system call extension (SCE, bit 0) and other features
  - SCE required for SYSCALL/SYSENTER, otherwise legacy interrupt based system calls are used (int 0x2e on Windows and int 0x80 on Linux)
  - SYSENTER/SYSEXIT is Intel, SYSCALL/SYSRET is AMD.
  - SYSCALL/SYSRET is also supported on Intel* so most OS's just use that.
- SYSENTER_EIP/ESP/CS: Kernel system call handler for SYSENTER
- STAR/LSTAR: Kernel system call handler for SYSCALL

*Except when they implement it slightly differently, leading to all sorts of security issues. See https://blog.xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation

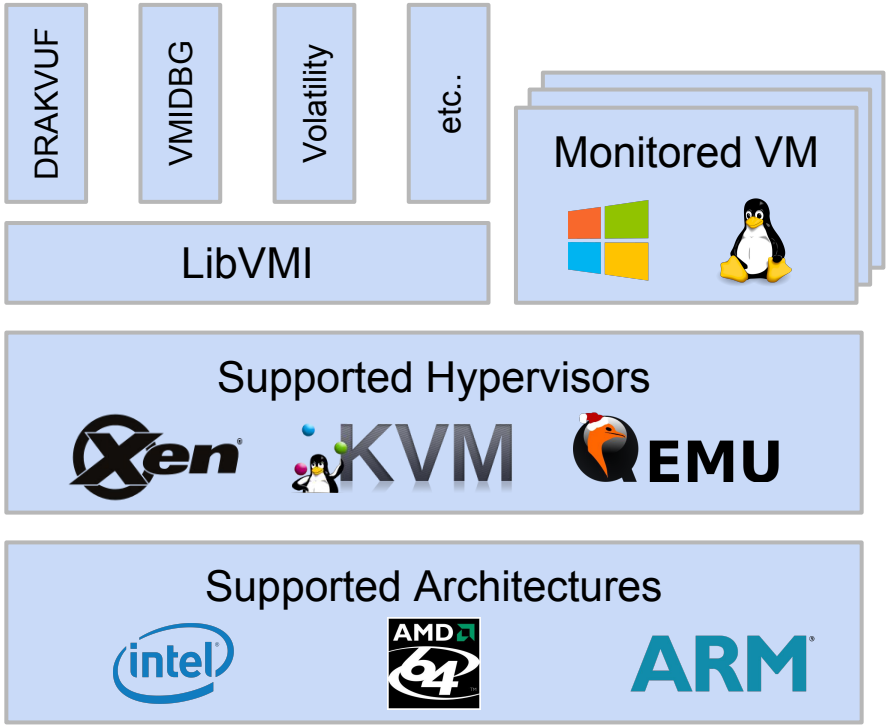
# LibVMI

Simplify many common tasks when inspecting a VM from the outside

- Translate guest virtual memory to guest physical (PAE, IA32e, ARM, etc.)
- Read/write guest memory with arbitrary buffer lengths
  - LibVMI will handle page-boundaries for you
- Cache frequently used values for better performance
  - vTLB, v2p, page handles, etc.
- Access/set guest vCPU registers
- Convert common string formats (UTF-8, UTF-16, etc)

It’s the swiss army knife for VMI
LGPL licensed, free to integrate into any types of projects

# LibVMI overview



# LibVMI basic lifecycle

```
...
vmi_instance_t vmi;
if (vmi_init(&vmi, VMI_AUTO | VMI_INIT_COMPLETE, “win7”) == VMI_SUCCESS) {
        // Do your VMI work here
        vmi_destroy(vmi);
}
...
```

Note: VMI_INIT_COMPLETE enables using vmi_*_va functions.
It requires more information about the guest kernel to be able to find the target process.

# LibVMI configuration file

Usually placed to /etc/libvmi.conf, contains information about the VM and the kernel running in the VM:

```
windows7-sp1-x64 {
        ostype = "Windows";
        sysmap = "/root/windows7-sp1-x64.rekall-profile.json";
}
troopers1-guest {
        ostype = "Linux";
        sysmap = "/root/System.map-3.16.0-4-amd64";
        linux_name = 0x4f0;
        linux_tasks = 0x280;
        linux_mm = 0x2d0;
        linux_pid = 0x334;
        linux_pgd = 0x40;
}
```

# LibVMI examples

libvmi/examples contains several tools that can be used as reference implementations:

- process listing (both Windows and Linux)
- kernel module listing (both Windows and Linux)
- various event examples: single-stepping, EPT, MSR, interrupt, CR3
- dumping memory to file
- listing memory pages mapped into a process (VA pages)

Use these to learn about LibVMI syntax and best practices!

# Anatomy of process listing

```
... // Init LibVMI

// Get addr of kernel symbol for process'
vmi_read_addr_ksym(vmi, "PsActiveProcessHead", &list_head);

// Loop circular linked list starting from list_head
do {
        // Read next list entry, process name, PID, etc and print
} while (next_list_entry != list_head);

... // Destroy LibVMI
```

https://github.com/libvmi/libvmi/blob/master/examples/process-list.c

# Pointers in space

Consider the following C snippet:

```
int *test = malloc(sizeof(int));
*test = 1;
```

What is going on here precisely?

- The pointer is on the *stack* at address &test
- The value is on the *heap* at address test
- The value is *test

For example: test is at 0x7ffc85d8c9b8 on the stack, it points to 0x1f06010 on the heap where the value is 1

# Pointers in space

How you would normally access pointers:

```
int *test = malloc(sizeof(int));
*test = 1;
int test_value = *test;
```

Now let's say we know where *test* is and want to read the value it points to:

```
int stack_address = 0x<some address>, heap_address = 0, test_value = 0;
vmi_read_addr_va(vmi, stack_address, PID, &heap_address); // The * op
vmi_read_32_va(vmi, heap_address, PID, &test_value);          // The = op
```

# Exercise 1: Read/write

There is a process running in your VM called “exercise1”.

It has an integer allocated on the heap, with its pointer on the stack. We know the address of the pointer on the stack.

1. Get the value stored on the heap using LibVMI and print it
2. Change the value of this integer on the heap

# Events with Xen

Currently available events you can subscribe to with LibVMI:

- CR register write-events (CR0, CR3, CR4)
- MSR register write-events
- EPT violations
- Singlestepping
- Software breakpoints (0xCC instruction)

# LibVMI event outline

```
vmi_event_t cr3_event;
vmi_init(&vmi, VMI_AUTO | VMI_INIT_COMPLETE | VMI_INIT_EVENTS, “win7”);
SETUP_REG_EVENT(&cr3_event, CR3, VMI_REGACCESS_W, 0, cr3_callback);
vmi_register_event(vmi, &cr3_event);
while(!interrupted) vmi_events_listen(vmi,500);
vmi_destroy(vmi);
```

# Exercise 2: Process scheduling

On x86 systems the CR3 register is used to store the virtual memory translation table's base address. Thus, the value in the CR3 uniquely identifies each process running on the system.

1. Trap CR3 register write events
2. Find the PID the process running

# Exercise 3: System call tracing

For system call tracing you have a variety of ways to obtain the addresses of system call handlers: read syscall table from memory or use kernel-specific debug information.

1. Using the list of syscall handler addresses to backup the first byte of the function, then overwrite it with 0xCC
2. Enable LibVMI breakpoint events
3. In the breakpoint handler, enable LibVMI singlestep event and replace 0xCC with the backup byte
4. In the singlestep handler place back 0xCC and turn off singlestepping

# Exercise 4: crackme!

Find the magic string to the crackme application running waiting for a magic string to be entered
FYI: magic string is different each time you run the application

1. Get the list of mapped memory pages of the process
2. Trap memory read events on the pages excluding the kernel
3. Print the contents of the memory at the location of the violation

# Debugging LibVMI apps

You can enable more verbose output of LibVMI in libvmi/debug.h:

```
VMI_DEBUG_MISC      = (1 << 0),
VMI_DEBUG_MEMCACHE = (1 << 1),
VMI_DEBUG_PIDCACHE = (1 << 2),
VMI_DEBUG_SYMCACHE = (1 << 3),
VMI_DEBUG_RVACACHE = (1 << 4),
VMI_DEBUG_V2PCACHE = (1 << 5),
VMI_DEBUG_V2MCACHE = (1 << 6),
VMI_DEBUG_PTLOOKUP = (1 << 7),
VMI_DEBUG_EVENTS   = (1 << 8),
VMI_DEBUG_XEN       = (1 << 9),
VMI_DEBUG_KVM       = (1 << 10),
VMI_DEBUG_FILE      = (1 << 11),
VMI_DEBUG_CORE      = (1 << 12),
VMI_DEBUG_READ      = (1 << 13),
VMI_DEBUG_WRITE     = (1 << 14),
VMI_DEBUG_DRIVER   = (1 << 15),
VMI_DEBUG_PEPARSE  = (1 << 16),
```

# Linux debugging/reversing

When building VMI applications your best option is using debug information for the kernel/application you are trying to monitor

For the kernel you can find many critical address in the *System.map* file:

- A for absolute
- B or b for uninitialized data section (called BSS)
- D or d for initialized data section
- G or g for initialized data section for small objects (global)
- i for sections specific to DLLs
- N for debugging symbol
- p for stack unwind section
- R or r for read only data section
- S or s for uninitialzed data section for small objects
- T or t for text (code) section
- U for undefined
- V or v for weak object
- W or w for weak objects which have not been tagged so
- - for stabs symbol in an a.out object file
- ? for "symbol type unknown"

https://en.wikipedia.org/wiki/System.map

# Linux debugging/reversing

Other debug information can be found in the debug file (if available)

Format of the debug file is DWARF: http://dwarfstd.org

On debian based system many DWARF related tools can be found in the *dwarves* package

- pahole
- addr2line
- pfunct

# pahole

C structures in memory may be different when loaded into memory

Compilers tend to *pad* structures to align structure sizes

- http://www.catb.org/esr/structure-packing

Pahole reveals the size of structure members, offset and the actual struct size:

```
struct mem_sharing_op_share {
    uint64_t        source_gfn;     /*     0     8 */
    uint64_t        source_handle;  /*     8     8 */
    uint64_t        client_gfn;     /*    16     8 */
    uint64_t        client_handle;  /*    24     8 */
    uint16_t        client_domain;  /*    32     2 */

    /* size: 40, cachelines: 1, members: 5 */
    /* padding: 6 */
    /* last cacheline: 40 bytes */
};
```

# pfunct

The pfunct tool shows the function aspects in the object code. It is capable of showing the number of goto labels used, number of parameters to the functions, the size of the functions etc.

- pfunct --symtab[=dynsym] <debug_file>: list symbol table information
- pfunct -P <debug_file>: function prototypes

Recommended reads:

- https://blogs.oracle.com/ali/entry/inside_elf_symbol_tables
- https://lwn.net/Articles/335942

# addr2line

When you have an instruction pointer (RIP) and want to determine what function it belongs to in a binary, addr2line can be your best friend.

- addr2line -f -p -e <debug_file> 0x<RIP>

This will display the name of the function and in what source file it is defined in:

```
# addr2line -f -p -e xen-syms-4.7-unstable 0xffff82d080203338
p2m_change_type_range at /home/tklengyel/workspace/xen/xen/arch/x86/mm/p2m.c:848
```

# Windows debugging/reversing

PE is the executable format and PDB is the debug format for binaries on Windows

PDB is a proprietary format which Microsoft only now started to officially document

PDB's are identified by a GUID in each PE executable

Microsoft provides an online repository to query for it's PDBs but you have to know both the filename AND the GUID

Executables in memory may not have this information loaded

# PEV: PE analysis toolkit

Binaries available on http://pev.sourceforge.net (or apt-get install pev), sourcecode at https://github.com/merces/pev

readpe: show information about the PE headers, include/export directories

pestr: search strings in PE files

ofs2rva: convert file offset to RVA

rva2ofs: convert RVA to file offset

# PDB GUID

Normally found as IMAGE_DIRECTORY_ENTRY_DEBUG in PE binaries

- For example 684da42a30cc450f81c535b4d18944b12

It is not a required field for the binary to work, so it can be stripped

How to get the debug information for these binaries from Microsoft?

There is also a *PE GUID*, that can be used to download the original executable

Executable files tend to still have this information

# pdbparse

https://github.com/moyix/pdbparse

Various utility tools to download and parse PDB files

examples/symchk.py: allows you to download both PE files and PDB files

examples/pdb_lookup.py: match RVA to symbol name

examples/pdb_tpi_vtypes.py: information about the defined structures

# Rekall PDB utilities

Some features of pev and pdbparse have been also implemented

Download PDB from Microsoft

- rekall fetch_pdb --pdb_filename <pdb_filename> --guid <guid>

Parse PDB information into JSON

- rekall parse_pdb <pdb_filename>

Get information from a PE binary

- rekall peinfo  <binary_filename>

# Live forensics

You may find yourself in a situation where you don’t actually know the state of the OS your and inspecting

Quickly poking around in memory may be very helpful to figure out odd issues

Both Volatility and Rekall have live “shells” that we can use to debug issues on live VMs through the VMIFS tool (part of LibVMI)

VMIFS mounts the memory of the VM as if it was a raw memory dump file

# Volatility shell

For Windows VMs:

- vol.py -f /mnt/mem --profile <some Windows profile> volshell

For Linux VMs:

- vol.py -f /mnt/mem --profile <some Linux profile> linux_volshell

Quick reference information in the shell: **hh()**

Change process context: cc(offset|pid|name)

Allows you to poke around (read/map memory)

# Detection and corner-cases

Most typical use cases for hypervisors and VMs are to consolidate IT infrastructure

Also used to introspect on applications to infer their intent (e.g., malware analysis)

Most VMMs are not designed to be covert, simply provide a high degree of similarity with system they virtualize

This section covers techniques to detect VMMs and other corner-cases outside the scope of typical IT VMM usage

# Detection and corner-cases

Low-hanging fruit

Timing-based detection (red pill) and counter-timing

Cache hierarchy and cache impact/detection of VMM

- Profiling and machine learning

Virtualization corner-cases (EPT R/W, etc..)

More events on ARM

# Detection: Low-hanging Fruit

Many VMMs are designed to execute benign applications, not ones attempting to avoid introspection

VMMs designed to provide similar operating environment with minimal performance impact

Some VMMs will return a CPUID string with the name and version of the VMM

Many VMMs designed to support typical OS workloads in protected mode (32- or 64-bit), will fail in bizarre ways when real-mode code is executed

# Detection of a hypervisor!

```
C:\Users\MrX\Downloads>pafish.exe
* Pafish (Paranoid fish) *

Some anti(debugger/VM/sandbox) tricks
used by malware for the general public.

[*] Windows version: 6.1 build 7601
[*] CPU: GenuineIntel         Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz

[-] Debuggers detection
[*] Using IsDebuggerPresent() ... OK

[-] CPU information based detections
[*] Checking the difference between CPU timestamp counters (rdtsc) ... OK
[*] Checking the difference between CPU timestamp counters (rdtsc) forcing VM exit ... traced!
[*] Checking hypervisor bit in cpuid feature bits ... traced!
[*] Checking cpuid vendor for known VM vendors ... OK

[-] Generic sandbox detection
[*] Using mouse activity ... OK
[*] Checking username ... OK
[*] Checking file path ... OK
[*] Checking common sample names in drives root ... OK
[*] Checking if disk size <= 60GB via DeviceIoControl() ... OK
[*] Checking if disk size <= 60GB via GetDiskFreeSpaceExA() ... traced!
[*] Checking if Sleep() is patched using GetTickCount() ... OK
[*] Checking if NumberOfProcessors is < 2 via raw access ... OK
[*] Checking if NumberOfProcessors is < 2 via GetSystemInfo() ... OK
[*] Checking if pysical memory is < 1Gb ... OK
[*] Checking operating system uptime using GetTickCount() ... OK
[*] Checking if operating system IsNativeVhdBoot() ... OK
```

# Detection: Cache Side-Channels

Cache Teller work showed that a user-space process on 1 core can monitor the L3 shared cache on another core to profile cache usage:

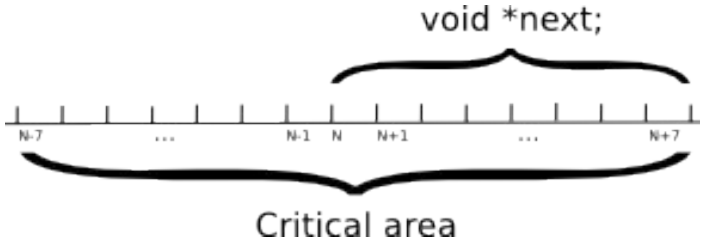
CPUID should impact the cache in a very minimal manner; VMM will exit and execute code to VM Resume, leaving cache impact

Can also be used to profile OS routines and SMM

Must be provisioned to learn what “normal” looks like using a binary classifier

# Corner Cases with EPT

No length of R/W operation reported during EPT violation.




CMPXCHG instruction does both a read and a write to memory

- EPT may only report is as WRITE violation

# TLB and VPID on x86

Split-TLB attacks no longer viable with sTLB

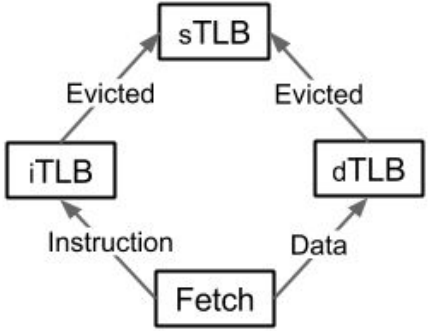
TLB is still problematic when VPID is enabled

LibVMI emulates V2P translation, thus relies solely on the pagetables

Guest still uses the TLB, so a malicious guest can forgo flushing the the TLB when it changes the pagetables

During EPT violation we still get both PA and VA, so we can validate the pagetables

**1) TLB Architecture**




**2) TLB Entry**

| | | | |
|---|---|---|---|
| Entry: VM | VPID: x | GVA | GPA |
| Entry: VMM | VPID: 0 | VA | PA |

# TLB and ASID on ARM

Split-TLB with no sTLB present

Has ASID (Address space ID, similar to VPID)

- Guest CAN perform split-TLB attacks

During two-stage paging violation we ONLY get the guest VA

ARM provides instructions to translate guest V to guest P, but it can only be performed as data-fetch access

Accurate translation of instruction-fetch accesses only available to the guest

- No way to validate the pagetables

# #VE, VMFUNC and altp2m

#VE and VMFUNC are Intel virtualization extensions starting in Skylake CPUs

#VE allows converting EPT violations to guest interrupts

- Remember, EPT-based monitoring is page-granular with lots of events
- With #VE you can have an in-guest filter on the events
- You can choose which EPT entry gets routed through #VE

#VE allows constructing Hybrid VMI applications

- Some components can run in the guest with hypervisor protection but better performance
- Some components run exclusively in the hypervisor

# VMFUNC

VMFUNC is an instruction on Intel allowing the guest to control some of the CPU's virtualization features without involving the hypervisor

- The notion breaks Popek and Goldberg's resource control requirement

Currently one function is supported starting in Skylake CPUs: EPTP switching

Remember, the VMCS allows 512 EPT pointers to be stored

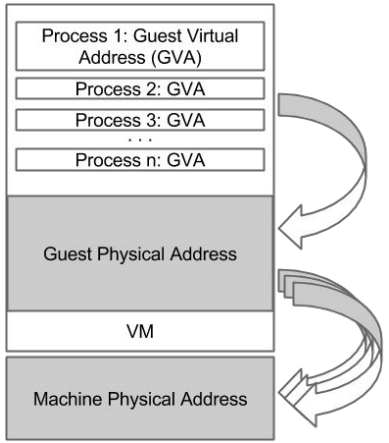
EPTP switching allows the guest kernel to specify which EPT pointer it wants to use (just by specifying the index into the EPTP array)

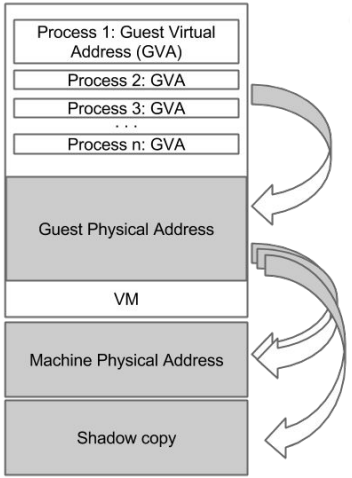
Most VMMs have only one EPTP in the VMCS so this is largely useless there

# Xen altp2m

Starting in Xen 4.6 there is support for multiple EPTs per guest with the altp2m system, and official support for VMFUNC as well

The different EPTs can have different permissions *and* contain different mappings!



# Xen altp2m + VMFUNC

Fast inter-guest communication

- No need to involve the hypervisor, thus no saving/restoring the context via the VMCS
- Safe zero-copy buffers, the two domains can alternate who has the buffer mapped through the EPT

Lightweight EPT based monitoring

- Avoid race-condition when EPT is restricted by switching to a non-restricted view instead of removing restrictions
- Allows tracing of multi-vCPU guests with less performance overhead and without the need of emulation

# ARM monitoring

Xen only supports two-stage paging based monitoring on ARM at the moment

No debug instruction trapping, no singlestepping through the hypervisor

No altp2m support but it can be implemented without any limitations as ARM has no concept of a VMCS

What instructions can be trapped to the hypervisor?

- Secure Monitor Call (SMC) that supposed to call TrustZone handlers
- How can we use it without singlestepping?

# ARM SMC based monitoring

On x86 instructions have variable lengths; on ARM CPUs each instruction is fixed length

- On x86 0xCC can only be injected into the beginning of instructions
- On ARM SMC can be written anywhere

With altp2m implemented for ARM we could do monitoring with SMC by having two views where two SMC's are written, one following the other ones

- View1: … [SMC] [……..] …
- VIew2: … [……..] [SMC] …

As we hit the SMCs, we can just swap the view so that no SMC needs to be removed while allowing the application to continue execution!

# Summary

Using Hypervisors can be very effective to monitor various low-level behavior of operating systems

LibVMI provides many utility functions to ease the implementation of introspecting applications

Various hardware and software limitations need be to taken into account to know where things can go wrong

Hardware features are constantly evolving and getting better, #VE was specifically designed for applications of this nature