Xenpwn – Breaking Paravirtualized Devices

Felix Wilhelm
#whoami

- Security Researcher @ ERNW Research
- Application and Virtualization Security
- Recent Research
  - Security Appliances (Palo Alto, FireEye)
  - Hypervisors
- @_fel1x on twitter
Agenda

- Device Virtualization & Paravirtualized Devices
- Double Fetch Vulnerabilities
- Xenpwn: Architecture and Design
- Results
- Case Study: Exploiting xen-pciback
Device Virtualization

- Virtualized systems need access to virtual devices
  - Disk, Network, Serial, ...
- Traditionally: Device emulation
  - Emulate old and well supported hardware devices
  - Guest OS does not need special drivers
  - Installation with standard installation sources supported
Paravirtualized Devices

- Most important downsides of emulated devices:
  - Hard to implement securely and correctly
  - Slow performance
  - No support for advanced features

- Solution: Paravirtualized Devices
  - Specialized device drivers for use in virtualized systems
  - Idea: Emulated devices are only used as fallback mechanism
  - Used by all major hypervisors
  - Not the same as Xen paravirtualized domains!
Paravirtualized Devices

- Split Driver Model
  - Frontend runs in Guest system
  - Backend in Host/Management domain
- Terminology differs between hypervisors
  - VSC / VSP in Hyper-V
  - Virtio devices and drivers
- Implementations are quite similar
Paravirtualized Devices

- PV devices are implemented on top of shared memory
  - Great Performance
  - Easy to implement
  - Zero copy algorithms possible
- Message protocols implemented on top
  - Xen, Hyper-V and KVM all use ring buffers
- Shared memory mappings can be constant or created on demand
Security of PV Devices

- Backend runs in privileged context ➔ Communication between frontend and backend is trust boundary
- Low level code + Protocol parsing ➔ Bugs
- Examples
  - Heap based buffer overflow in KVM disk backend (CVE-2011-1750)
  - Unspecified BO in Hyper-V storage backend (CVE-2015-2361)
- Not as scrutinized as emulated devices
  - Device and hypervisor specific protocols
  - Harder to fuzz
Very Interesting Target

- Device **emulation** often done in user space ⇔ PV backend often in kernel for higher performance
  - Compromise of kernel backend is instant win 😊
- PV devices are becoming more important
  - More device types (USB, PCI pass-through, touch screens, 3D acceleration)
  - More features, optimizations
- Future development: Removal of emulated devices
  - see Hyper-V Gen2 VMs
Research Goal

- "Efficient vulnerability discovery in Paravirtualized Devices"
- Core Idea: No published research on the use of shared memory in the context of PV devices
- Bug class that only affect shared memory? ➔ Double fetches!
Double Fetch Vulnerabilities

- Special type of TOCTTOU bug affecting shared memory.
- Simple definition: Same memory address is accessed multiple times with validation of the accessed data missing on at least one access.
- Can introduce all kinds of vulnerabilities
  - Arbitrary Write/Read
  - Buffer overflows
  - Direct RIP control 😊
Double Fetch Vulnerabilities

- Term “double fetch” was coined by Fermin J. Serna in 2008
  - But bug class was well known before that
- Some interesting research published in 2007/2008
  - CCC 2007: “From RING 0 to UID 0” and Phrack #64 file 6 – twiz, sgrakkyu
- First example I could find is sendmsg() linux bug reported in 2005
  - Happy to hear about more 😊
int cmsghdr_from_user_compat_to_kern(...) {
    ...
    while(ucmsg != NULL) {
        if(get_user(ucmlen, &ucmsg->msg_len))
            return -EFAULT;
        ...
        tmp = (((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
                CMSG_ALIGN(sizeof(struct cmsghdr))));
        kcm_len += tmp;
        ...
    }
    ...
    if(kcmlen > stackbuf_size)
        kmsg_base = kmsg = kmalloc(kcmlen, GFP_KERNEL);
    while(ucmsg != NULL) {
        _get_user(ucmlen, &ucmsg->msg_len);
        
        if(copy_from_user(CMSG_DATA(kmsg),
                          CMSG_COMPAT_DATA(ucmsg),
                          (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))))
        ...
    }
}
Bochspwn

  - by j00ru and Gynvael Coldwind
- Uses extended version of Bochs CPU emulator to trace all memory access from kernel to user space.
Bochspwn

- Resulted in significant number of Windows bugs (and a well deserved Pwnie)
  - but not much published follow-up research
- Whitepaper contains detailed analysis on exploitability of double fetches
  - On multi core system even extremely short races are exploitable
- Main inspiration for this research.
Example:
Bochspwn

```assembly
1. mov ecx, [edi+18h]
2. ;[...]
3. push 4
4. push eax
5. push ecx
6. call _ProbeForWrite
7. push dword ptr [esi+20h]
8. push dword ptr [esi+24h]
9. push dword ptr [edi+18h]
10. call _memcpy
```
Xenpwn

- Adapt memory access tracing approach used by Bochspwn for analyzing PV device communication.
- Why not simply use Bochspwn?
  - Extremely slow
  - Passive overhead (no targeted tracing)
  - Compatibility issues
  - Dumping traces to text files does not scale
- Idea: Implement memory access tracing on top of hardware assisted virtualization
Xenpwn Architecture

- Nested virtualization
  - Target hypervisor (L1) runs on top of base hypervisor (L0)
- Analysis components run in user space of L1 management domain.
  - No modification to hypervisor required
  - Bugs in these components do not crash whole system
- L0 hypervisor is Xen
libVMI

- Great library for virtual machine introspection (VMI)
  - Hypervisor agnostic (Xen and KVM)
  - User-space wrapper around hypervisor APIs
- Allows access to and manipulation of guest state (memory, CPU registers)
- Xen version supports memory events
libVMI Memory Events

- Trap on access to a guest physical address
- Implemented on top of Extended Page Tables (EPT)
  - Disallow access to GPA
  - Access triggers EPT violation and VM exit
  - VM exit is forwarded to libvmi handler

```c
auto event = new vmi_event_t();
event->type = VMI_EVENT_MEMORY;
event->mem_event.physical_address = paddr;
event->mem_event.npages = 1;
event->mem_event.granularity = granularity;
event->mem_event.in_access = access;
event->callback = callback;

if (vmi_register_event(s->vmi, event) != VMI_SUCCESS)
    { /*... */}
```
Memory Access Tracing with libvmi

1. Find shared memory pages
2. Register memory event handlers
3. Analyze memory event, extract needed information and store in trace storage.
4. Run analysis algorithms (can happen much later)
Trace Collector

- Use libvmi to inspect memory and identify shared memory pages
  - Target specific code.
  - Identify data structures used by PV frontend/backend and addresses of shared pages
- Registers memory event handlers
- Main work is done in callback handler
  - Disassemble instructions using Capstone
Callback handler

1. Extract domain ID
   - If privileged domain?
     - Yes: Extract RIP, CR3
     - No: Create empty trace entry
2. New instruction?
   - Yes: Disassemble instruction
     - Add to cache/instruction stream
   - No: Fetch from cache
     - Create trace entry
3. Single step and continue
   - Add to tracing stream
Trace Storage

- Storage needs to be fast and persistent
  - Minimize tracing overhead
  - Allow for offline analysis
- Nice to have: Efficient compression
  - Allows for very long traces
- Tool that fulfills all these requirements: Simutrace
  - simutrace.org
Simutrace

- Open source project by the Operation System Group at the Karlsruhe Institute of Technology
- Designed for full system memory tracing
  - All memory accesses including their content
- C++ daemon + client library
  - Highly efficient communication over shared memory pages
- Uses specialized compression algorithm optimized for memory traces
  - High compression rate + high speed
- Highly recommended!
Trace Entries

For every memory access:
- cycle count
- instruction pointer
- memory address
- data + size

32 bytes

For every unique instruction:
- rip
- cr3
- offset
- instruction bytes
- module_name

96 bytes
Double Fetch Algorithm
Simplified version

1. Read next entry from trace
2. Empty entry?
   - Yes: Analyze map, clear map
   - No: Read access type
3. Read access?
   - Yes: Add instruction pointer to map[address]
   - No: Clear map[address]
Advantages & Limitations

- **Good:**
  - Low passive overhead
  - Largely target independent
    - only Trace collector requires adaption
  - Easy to extend and develop

- **Bad:**
  - High active overhead
    - VM exits are expensive
  - **Reliance on nested virtualization**
Nested Virtualization on Xen

- Xen Doku: Nested HVM on Intel CPUs, as of Xen 4.4, is considered "tech preview". For many common cases, it should work reliably and with low overhead

- Reality:
  - Xen on Xen works
  - KVM on Xen works (most of the time)
  - Hyper-V on Xen does not work 😞
Results

- KVM: no security critical double fetches
  - Main reason seems to be endian independent memory wrappers
  - .. but discovered other interesting issues while reading the virtio code ;)
- bhyve: one very interesting result
  - Ongoing disclosure process
- Xen: Three interesting double fetches
QEMU xen_disk

Normally not exploitable thanks to compiler optimizations

```c
void blkif_get_x86_64_req(blkif_request_t *dst, blkif_x86_64_request_t *src)
{
    int i, n = BLKIF_MAX_SEGMENTS_PER_REQUEST;
    dst->operation = src->operation;
    dst->nr_segments = src->nr_segments;
    // ...
    if (src->operation == BLKIF_OP_DISCARD) {
        //...
    }
    if (n > src->nr_segments)
        n = src->nr_segments;
    for (i = 0; i < n; i++)
        dst->seg[i] = src->seg[i];
} 
```
for (n = 0, i = 0; n < nseg; n++) {
    //...
    i = n % SEGS_PER_INDIRECT_FRAME;
    seg[n].nsec = segments[i].last_sector -
    segments[i].first_sector + 1;
    seg[n].offset = (segments[i].first_sector << 9);

    if ((segments[i].last_sector >= (PAGE_SIZE >> 9)) ||
        (segments[i].last_sector < segments[i].first_sector)) {
        rc = -EINVAL;
        goto unmap;
    }
    //...
}
xen-pciback: xen_pcibk_do_op

```c
switch (op->cmd) {
    case XEN_PCI_OP_conf_read:
        op->err = xen_pcibk_config_read(dev,
                                      op->offset, op->size, &op->value);
        break;
    case XEN_PCI_OP_conf_write:
        //...
    case XEN_PCI_OP_enable_msi:
        //...
    case XEN_PCI_OP_disable_msi:
        //...
    case XEN_PCI_OP_enable_msi:
        //...
    case XEN_PCI_OP_disable_msi:
        //...
    default:
        op->err = XEN_PCI_ERR_not_implemented;
        break;
}
```

1. `cmp    DWORD PTR [r13+0x4], 0x5`
2. `mov    DWORD PTR [rbp-0x4c], eax`
3. `ja     0x3358 <xen_pcibk_do_op+952>`
4. `mov    eax, DWORD PTR [r13+0x4]`
5. `jmp    QWORD PTR [rax*8+off_77D0]`
xen-pciback

- switch statement is compiled into jump table
- op->cmd == $r13+0x4
- Points into shared memory
- Range check and jump use two different memory accesses
- Valid compiler optimization
  - op is not marked as volatile
Exploiting pciback

- Race is very small: 2 Instructions
  - But can be reliably won if guest VM has multiple cores
- Lost race does not have any negative side effects
  - Infinite retries possible
- Simple to trigger
  - Send PCI requests while flipping value using XOR

```
cmp DWORD PTR [r13+0x4],0x5
mov DWORD PTR [rbp-0x4c],eax
ja 0x3358 <xen_pcibk_do_op+952>
mov eax,DWORD PTR [r13+0x4]
jmp QWORD PTR [rax*8+off_77D0]
```

```
"loop_header_%e=:\n"
"inc rcx\n"
"xor dword ptr [rax], 25\n"
"cmp rcx, 5000\n"
"jnz loop_header_%e=\n"
```
Exploiting pciback

- Indirect jump ➔ No immediate RIP control
  - Need to find reliable offset to function pointer
- Load address of xen-pciback.ko is random
- Virtual address of backend mapping also not known
- A lot of similarities to a remote kernel exploit
- Chosen approach: Trigger type confusion to get write primitiv
Type Confusion

- Second jump table generated for xen-pciback
  - Almost directly behind the jump table generated for vulnerable function
- XenbusStateInitialized uses value of r13 register as first argument
  - Should be a pointer to a xen_pcibk_device structure
  - Is a pointer to the start of the shared memory page 😊
Getting a write primitive

- `xen_pcibk_attach` first tries to lock the `dev_lock` mutex of referenced structure.
- Gives us the possibility to call `mutex_lock` with a fake mutex structure.
- `mutex_lock`
  - Fastpath: Switch lock count from 1 -> 0
  - Slowpath: Triggered when lock count != 1

```c
struct xen_pcibk_device {
    void *pci_dev_data;
    struct mutex dev_lock;
    struct xenbus_device *xdev;
    struct xenbus_watch be_watch;
    u8 be_watching;
    int evtchn_irq;
    struct xen_pci_sharedinfo *sh_info;
    unsigned long flags;
    struct work_struct op_work;
    struct xen_pci_op op;
};

void __sched mutex_lock(struct mutex *lock) {
    might_sleep();
    /* The locking fastpath is the 1->0 transition from
     * 'unlocked' into 'locked' state. */
    __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
    mutex_set_owner(lock);
}
```
Getting a write primitive: mutex_lock slowpath

1. mutex_optimistic_spin needs to fail.
   - Can be achieved by setting lock->owner to a readable zero page
2. If lock count still not 1, mutex_waiter structure is created and stored on stack
3. mutex_waiter structure is added to lock->wait_list and kernel thread goes to sleep till wake up.
   ➔ Pointer to waiter is written to attacker controlled location.

```c
/* add waiting tasks to the end of the waitqueue (FIFO): */
list_add_tail(&waiter.list, &lock->wait_list);
waiter.task = task;

wait_list->prev = new;
waiter->next = wait_list;
waiter->prev = WRITE_TARGET;
WRITE_TARGET->next = new;
```
Write Primitive

- write-where but not write-what
  - Pointer to pointer to attacker controlled data
  - Can’t simply overwrite function pointers
- One shot
  - pciback is locked due to xen_pcibk_do_op never returning
- Idea: Add faked entries to a global linked list.
  - Requires known kernel version + no KASLR or infoleak

```c
struct list_head {
    struct list_head *next, *prev;
};
```
Overwrite target

- Global data structure
  - Need to know address of list_head
- No new elements should be attached during run time
  - list_head.prev is not changed, new entry might be added directly behind list_head
- Needs to survive one “junk” entry
  - No full control over waiter structure / stack frame
/*
 * linux/fs/exec.c
 *
 * Copyright (C) 1991, 1992 Linus Torvalds
 */

/*
 * Demand-loading implemented 01.12.91 - no need to read anything but
 * the header into memory. The inode of the executable is put into
 * "current->executable", and page faults do the actual loading. Clean.
 *
 * Once more I can proudly say that linux stood up to being changed: it
 * was less than 2 hours work to get demand-loading completely implemented.
 *
 * Demand loading changed July 1993 by Eric Youngdale. Use mmap instead,
 * current->executable is only used by the procfs. This allows a dispatch
 * table to check for several different types of binary formats. We keep
 * trying until we recognize the file or we run out of supported binary
 * formats.
 */
fs/exec.c: formats

- **formats** linked list contains entries for different file formats supported by exec
  - ELF
  - `#!` shell scripts
  - a.out format
- Walked every time exec* syscall is called to load input file.
- waiter entry is skipped because try_module_get function fails
Getting Code Execution

- Set address of load_binary pointer to stack pivot
- ROP chain to allocate executable memory and copy shellcode
  - vmalloc_exec + memcpy
- Restore original formats list
- $shellcode
- Return to user space
Demo
Open Source

- Xenpwn open source release:
  - https://github.com/felixwilhelm/xenpwn
- Whitepaper contains a lot more technical details
  - Implementation details
  - Performance evaluation
  - ...
Future Work

- Use Xenpwn against Hyper-V and VMWare
  - Requires improved support for nested virtualization
- Identify and analyze other shared memory trust boundaries
  - Sandboxes?
- What types of bugs can we find with full memory traces?
Thanks for your Attention!

Q&A