devastating capability, revolutionary advantage

Cluck Cluck: On Intel's Broken Promises*

Mr. Jacob Torrey (@JacobTorrey)

8/5/2014
BSidesLV 2014

*Not just Intel, all PCIe-compliant systems

Thesis/Disclaimer

devastating capability, revolutionary advantage

- **Thesis: The PCIe specification allows software with sufficient privileges to break-out of virtual memory**

- **Disclaimer: Please jump in at any time! Opinions and rude comments my own, not those of my employer**

About Me

devastating capability, revolutionary advantage

- **Sr. Research Engineer at Assured Information Security in Denver, CO**
  - Shameless plug: we're hiring!
- **Leads low-level computer architectures team**
  - Built custom hypervisor for DARPA
  - Built BIOS
  - Likes C & assembly
- **Lover of the outdoors**
  - Aconcagua here I come!

Outline

- **Background Concepts**
- **The Problem**
  - Goal
  - A catch-22
- **The Solution**
  - PCIe Enhanced Configuration Access Mechanism (ECAM)
  - Take advantage of the poor CPU/MCH communication channels
  - Some math to map
  - Caveats
- **Why this happened**
- **Future Work/Importance**
  - But wait! There's more!

Terms

- **Virtual Memory**
  - Allows for a process to think it is running in it's own 4GB address space (32-bit). Prevents processes from interfering with OS and others
- **MCH – Memory Controller Hub**
  - Provides the CPU with a simplified view of system memory and memory-mapped IO space
- **Memory Mapped IO**
  - Accessing hardware devices as if they were physical memory regions – Newer and faster
- **Port IO**
  - Accessing hardware devices through the CPU instructions IN and OUT – Older and slower

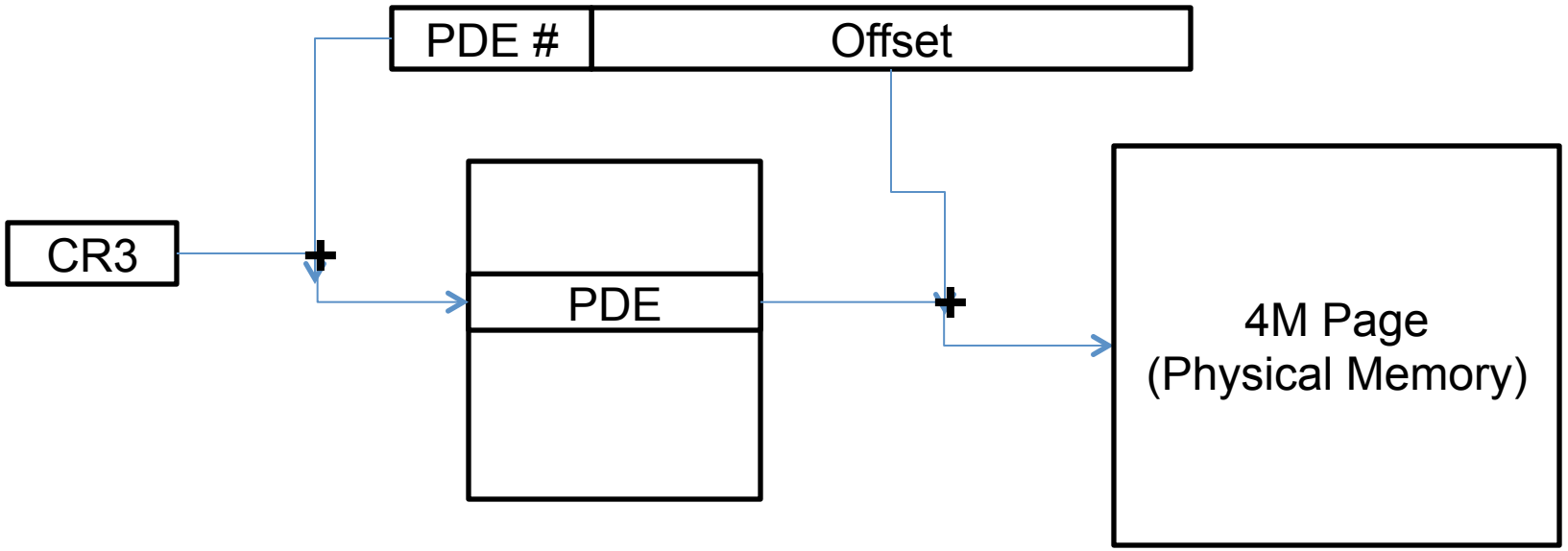
Terms II

- **TLB - Translation Look-aside Buffer**
  - A small CPU cache that stores recently used virtual memory translations
- **PAE – Physical Address Extensions**
  - Provides 32-bit operating systems the ability to support more than 4G of memory through an ugly hack that expands addresses to 36-bits
- **PCI Configuration Space**
  - A portion of the PCI card's memory used to store configuration information
- **PCIe ECAM**
  - Extended configuration space for PCIe devices
- **CR3 Register**
  - Tells the CPU where to look for the page directory when converting virtual addresses to physical

Virtual Memory

devastating capability, revolutionary advantage

- **(Non-PAE 32-bit for pedantic viewers)**



• Uses the CR3 register in concert with tables in memory to convert virtual addresses to physical

• Alternatively you can add an extra level of indirection through the use of an intermediary page table.

Virtual Memory Security

- **Paging/virtual memory is a protective feature/ promise**
  - First code in will be able to control system – usually OS
- **Unless you can access the pages tables, you are locked out (until now)**
  - Can’t add mappings to page tables unless you have a mapping to the page table
- **Protects against certain classes of attack**

Cluck Cluck Goal

- **Goal: Map in arbitrary physical memory**
  - Requires modifying page tables – need to know where they are in virtual memory
- **Can be kernel shell-code, live memory forensics, etc.**
- **Have ring-0 access, but confined to OS-controlled mappings**
  - Cannot access MMIO devices for example
- **OS independent**

Problem

- **Only know where in physical memory (CR3) the page tables are**
- **Cannot map in the page tables without having the page tables mapped in already**
  - The OS usually has a hard-coded value (0xC0000000 in many Windows systems)
  - OS-specific attacks are lame, let's exploit the architecture!
- **You do not know where your code is executing since you cannot access the page tables**

PCIe ECAM

- **Need control over just 32-bits of memory at a known physical address**
  - This is the crux
  - Can bootstrap a recursive mapping
- **Enhanced Configuration Access Mechanism**
  - PCIe has more configuration space per device
  - Port I/O is slow
  - Need a way to access it faster
- **ECAM shadows device configuration space into physical memory**
  - Base address is stored in PCIEXPBAR register

Solution

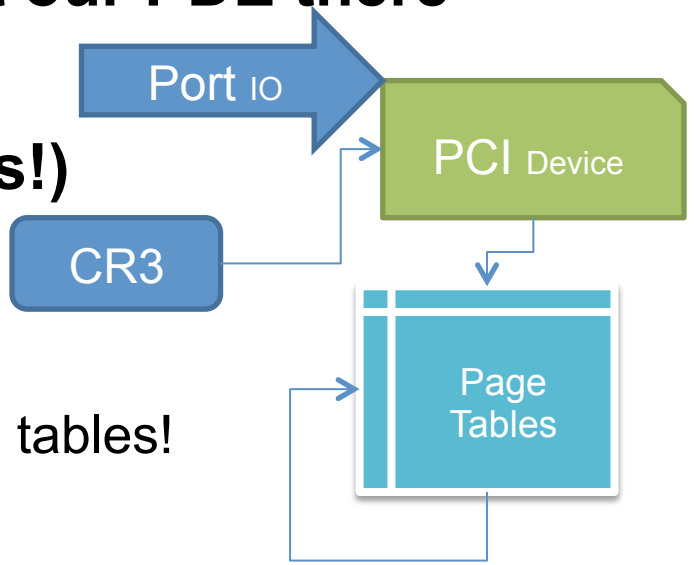
- **Construct a PDE that maps in the page directory (recursive entry)**
  - Use the CR3 physical address and mark it as present/RW/PS
- **Utilize Port IO to insert new PDE into PCI configuration space**
  - We have just modified what the CPU thinks is physical memory through port IO!
- **Determine physical location**
  - MCH stores the PCI base address in a configuration register (port IO again!)

Solution II

- **But where can our PDE go?**
  - Can't trash random registers or system may crash!
- **Thank you Intel for the SCRATCHPAD DATA register**
  - "This register is for software use, it has no functionality"
  - 32-bits of beautiful storage right in the MCH (D0:F0)
  - Port I/O access to physical memory, write that PDE!
- **Determine physical location**
  - MCH stores the PCI base address (PCIEXBAR) in a configuration register (port IO again!)

Solution III

- **Change CR3 to point to PCI configuration space**
  - Kernel code is marked as Global, thus the TLB will cache the code segment, so the box won't crash
  - The CPU doesn't know that it's doing anything wrong (using PCI config like this is wrong) and the MCH doesn't know how the CPU is using the memory!!!
- **Scan the 'real' page directory (we know where it is now) for an empty entry and put our PDE there**
- **Switch CR3 back (yes this works!)**
- **Profit! All in a few lines of ASM**
  - You have a virtual pointer to the page tables!



Caveats

- **Alignment – PDE and CR3s are not aligned, requires some bitwise operations**
- **Needs PCI registers that are OK to be trashed (like the MCH's scratchpad register)**
  - There are plenty of options on modern systems
- **Requires Ring-0 and global pages (more on this later!)**

Design Flaws

devastating capability, revolutionary advantage

- **Classic case of feature creep**
- **PCIe ECAM is for higher performance**
- **Violates assumptions**
- **This has happened before**
  - SMM caching bug
  - Virtual Machine side-channels
  - Etc…

Intel Responds!

- **Spoke at length with Branco Rodrigo from Intel**
  - Super smart guy, try to bump into him this week
- **His thoughts:**
  - Impressive attack
  - Requires ring-0, thus you are already pwned
  - Should provide education to OS developers, but not a critical concern
- **Already possible for target OSes**
  - Cluck Cluck is NOT target specific!

Improving Future Designs

- Codify invariants and platform guarantees
- Review when new feature is added
- Modeling software such as Alloy is powerful and can find stuff you might miss
- Maintain an “adversarial mindset” whenever building/designing

Use Cases

devastating capability, revolutionary advantage

- **Live forensics**
  - In an environment where you cannot know or trust the OS API
  - Need full memory access
  - Need memory mapped IO to export data
- **Hypervisors**
  - Provides a method to OS-independently map in memory
- **Kernel Shell-code**
  - You know, for… reasons ☺
  - Want to pivot to access full system memory and MMIO devices

Future Work

- **All theoretically possible**
- **Extend to work on PAE/64-bit**
  - Will need to support more levels of indirection
  - Larger scratch pad registers to be found
- **Remove global page requirements!**
  - Then can execute anywhere with kernel privileges
- **Remove Ring-0 requirements (still requires IOPL)**
  - Create Ring-3 -> Ring-0 code!

Ring 3 -> Ring 0?

Or: don't tell me my attack is pointless

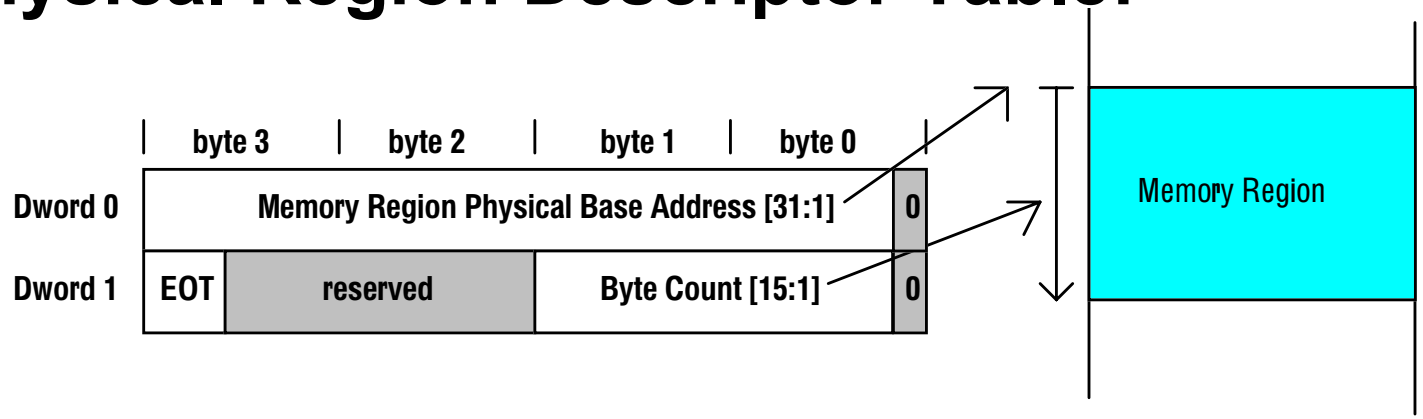
- **Let's see if we can abuse this even more**
  - Disclaimer: This hasn't been implemented/tested, hence Future Work
- **Background:**
  - DMA – Direct memory access
    - Needs port IO
    - Needs physical memory table of blocks -> addresses
  - ATA – Disk drive mode used by legacy BIOSes and older drives
- **Goal: Ring 3 code with IOPL -> Ring 0**
  - Who cares? Perhaps on a BSD system with securelevel?

R3 -> 0 Overview

- Use ATA DMA to overwrite kernel code or IVT:
- Set up 8-byte table in ECAM pointing to target memory address
- Write payload to disk at known block
- Use Port IO to tell DMA controller to read from block (can have it read from memory first to patch) and write to target address

DMA

- **Physical Region Descriptor Table:**



- **Needs to be at known physical address**
  - If only there was a way to write to physical memory using port IO…

- **Store target memory address**

DMA II

- **Use port IO to have DMA controller read/write**
  - Command byte: R/W and Start/Stop bits
  - Status byte: Who cares…
  - PRDT Address: Point to table we set up previously

- **Use port IO to communicate with ATA controller**
  - Use traditional port IO ATA/ATAPI spec
  - Command bytes:
    - 0xC8 Read DMA (28 bit LBA)
    - 0x25 Read DMA (48 bit LBA)
    - 0xCA Write DMA (28 bit LBA)
    - 0x35 Write DMA (48 bit LBA)

Ring 3 -> Ring 0

- **You now (theoretically) have full read and write access to the entire memory space from ring 3 (with IOPL)!**
- **What you do with that power is left as an exercise for the reader ☺**
- **Caveats:**
  - Some newer drives are in AHCI mode and will not respond to ATA commands
  - Intel VT-d can block DMA to sensitive regions if present

Conclusions

- **Nifty trick, not really a security hole in itself**
  - x86 is full of weird machines!
- **New architectural feature creates broken invariant**
- **Intel is actually really cool, hope they can take some ribbing**
- **Read more in PoC||GTFO 0x05!**
  - Coming to a printer near you soon!

Questions

devastating capability, revolutionary advantage

Thanks!

Backup

devastating capability, revolutionary advantage

Modify Physical Mem

```
59    ULONG MMIORange = 0;
60    __asm
61    {
62    pushad
63
64        // Utilize the scratch pad register as our mini-PDE
65        mov ebx, cr3
66        and ebx, 0xFFC00000      // This is going to hold our new PDE (The bits in CR3 with the least significant stuff removed)
67        or ebx, 0x83             // P | RW | PS
68
69        mov dx, 0x0cf8
70        mov eax, 0x800000DC     // Offset 0x37 (0xDC / 4)
71        out dx, eax
72
73        mov dx, 0x0CFC
74        mov eax, ebx
75        out dx, eax // Write our PDE
76
77        // Determine where in physical memory we can find the PDE
78        mov dx, 0x0cf8
79        mov eax, 0x80000060
80        out dx, eax
81
82        mov dx, 0x0CFC
83        in eax, dx
84        mov MMIORange, eax // Save our value and BAM!
85
86    popad
87    }
88
89    if(VDEBUG)
90        DbgPrint("MMIO Base Address: %x", MMIORange);
91
92    return MMIORange;
```

Insert PDE I

```
136     __asm {
137         cli
138
139         pushad
140
141         // Save the current CR3, seems like overkill, but it makes sense
142         mov ebx, cr3 // A copy to use to construct our virtual address
143         mov ecx, cr3 // Save a copy so we don't fuckz0r things up too much
144
145         mov edx, MMIORange // Our new CR3 val
146
147         // Setup our virtual address
148         and ebx, 0x003FFFFF      // Gets us our offset into stuff
149         or ebx,  0x0DC00000       // Reference the PDE offset of (0x37 << 22)
150         // EBX should now have our virtual address :)
151
152     // Tests to see if the PDE is free for use
153     test_pde:
154
155         add ebx, 0x4 // Offset to unused PDE
156
157         // Keep the offset var up to date (but uint32 aligned, not uint8)
158         mov eax, PDEoffset
159         add eax, 0x1
160         mov PDEoffset, eax
161
162         //*************** BEGIN CRITICAL SECTION ************************************************
163         mov cr3, edx  // Inject our new CR3
164
165         mov eax, [ebx]  // Add our mirthful PDE entry which should map in the PD
166         invlpg [ebx] // Invalidates the virtual address we used just incase it could cause later problems
167
168         mov cr3, ecx   // Restore everything nicely
169         //*************** END CRITICAL SECTION **************************************************
170         cmp eax, 0 // Can we use this entry?
171         je inject_pde // Try the next one
172         jmp test_pde // Found an empty one, w00t!
173
```

Insert PDE II

```
174     // Injects our recursive PDE into the PDT
175     inject_pde:
176         // Setup our recursive PDE (again)
177         mov eax, cr3 // A copy to modify for our new recursive PDE
178         and eax, 0xFFC00000 // Only the most significant bits stay for 4M pages
179         or eax, 0x93 // P | RW | PS | PCD
180         // EAX now holds the same PDE to put into the 'real' PDT
181         //*************** BEGIN CRITICAL SECTION ***************************************************
182         mov cr3, edx  // Inject our new CR3
183
184         mov [ebx], eax  // Add our mirthful PDE entry which should map in the PD
185         invlpg [ebx] // Invalidates the virtual address we used just incase it could cause later problems
186
187         mov cr3, ecx   // Restore everything nicely
188         //*************** END CRITICAL SECTION *****************************************************
189
190         // Determine the virtual address of the base of the PDT (remembering the differences in alignment)
191         mov eax, cr3 // A copy to modify for our new recursive PDE
192         and eax, 0x003FFFFF // Only the most significant bits stay for 4M pages
193         mov ebx, PDEoffset
194         shl ebx, 22 // Offset into the PDT
195         or eax, ebx
196         mov PDEoffset, eax
197
198         popad
199
200         sti
201     }
202
```