Dear Pastor Laphroaig,

Please consider the following submission to your church newsletter. I hope you think it worthy of your holy parishioners and readers.

Our friends at Intel are always providing Easter eggs for us to enjoy, and having stumbled across a new one for x86, the most neighborly option was naturally to share with all interested parties. This PoC is a weird quirk in which a newer x86 feature-set breaks invariants/security guarantees from older version. Specifically, the newer PCI Express configuration space access mechanism breaks virtual memory. Virtual memory is orchestrated by the CR3 register (storing the physical address of the page tables) and the page tables themselves. An issue with kernel shell-code and live memory forensics is that unless the virtual address of the page tables is known, it is impossible to map them (or any other physical address for that matter) into virtual memory, resulting in a chicken-and-egg problem. Luckily, most operating systems keep the page tables at a known virtual address (0xC0000000 on many Windows systems), but this Easter egg allows access to the page tables on any OS.

In kernel space, CR3 can be read, providing the physical address of the OS page tables; however, due to Intel’s virtual memory protections, there is no way to create a recursive virtual mapping to that physical address. All that is needed to do so, is a way to write an arbitrary 32-bits (which will become a PDE mapping in the page tables) to a known physical location. This is the crux of the issue, and the security of virtual memory depends on it. Luckily, with the advent of PCI Express, there is now the “Enhanced Configuration Access Mechanism” (ECAM), which shadows PCI configuration space registers into physical memory at an address kept in the PCIEXPBAR register (D0:F0 offset: 0x60). This is typically enabled on all the systems the author has come across, but your mileage may vary. With this ECAM, changes made to the configuration space via the legacy port I/O mechanism (0xCF8/0xCFC) will be reflected in physical memory. Now all that is needed is a register in configuration space that is at least 32-bits wide and can be changed to an arbitrary value without impacting the system. Again, Intel is looking out for our church, and through their grace, they provide a “Scratchpad Data” register (D0:F0 offset: 0xDC) that has no semantic meaning, just a location for software to store data. Now we have the function ModifyPM() for physical memory. (This is for Windows 32-bit without PAE, running as driver code.)

```
/**
  * Sets up the PDE to map in the real PDT using the MMIO ranges of PCI
  * Configuration space.
  *
  * @return The PCIEXPBAR for comparison.
  */

ULONG ModifyPM()
{
    ULONG MMIORange = 0;
    __asm
    {
        pushad
```
Once the scratchpad register is primed and ready, and the physical address of the ECAM is known, the next step is to treat the register as a PDE mapping in the OS page tables to add a recursive mapping at a known location.

```c
ULONG recurMap()
{
    ULONG MMIORange = 0;
    ULONG PDEBase = 0;
    ULONG PDEOffset = 0;

    // Sets up the (fake) PDE and
    MMIORange = ModifyPM();
    MMIORange &= 0xF0000000;

    if (VDEBUG)
        DbgPrint("Mapping PDT to itself ");

    return MMIORange;
}
```

Once the scratchpad register is primed and ready, and the physical address of the ECAM is known, the next step is to treat the register as a PDE mapping in the OS page tables to add a recursive mapping at a known location.
cli
pushad

// Save the current CR3, seems like overkill, but it makes sense
mov ebx, cr3 // A copy to use to construct our virtual address
mov ecx, cr3 // Save a copy so we don't mess up things up too much
mov edx, MMIORange // Our new CR3 val

// Setup our virtual address
and ebx, 0x003FFFFF // Gets us our offset into stuff
or ebx, 0x0DC00000 // Reference the PDE offset of (0x37 << 22)
// EBX should now have our virtual address :)

// Tests to see if the PDE is free for use
test_pde:
add ebx, 0x4 // Offset to unused PDE

// Keep the offset var up to date (but uint32 aligned, not uint8)
mov eax, PDEoffset
add eax, 0x1
mov PDEoffset, eax

/********************** BEGIN CRITICAL SECTION
mov cr3, edx // Inject our new CR3
mov eax, [ebx] // Add our mirthful PDE entry which should map in the PD
invlpg [ebx] // Invalidates the virtual address we used just in
// case it could cause later problems.

mov cr3, ecx // Restore everything nicely
/********************** END CRITICAL SECTION
cmp eax, 0 // Can we use this entry?
je inject_pde // Try the next one
jmp test_pde // Found an empty one, w00t!

// Injects our recursive PDE into the PDT
inject_pde:
// Setup our recursive PDE (again)
mov eax, cr3 // A copy to modify for our new recursive PDE
and eax, 0xFFC00000 // Only the most significant bits stay for 4M pages
or eax, 0x93 // P | RW | PS | PCD
// EAX now holds the same PDE to put into the 'real' PDT
/********************** BEGIN CRITICAL SECTION
mov cr3, edx // Inject our new CR3

mov [ebx], eax // Add our mirthful PDE entry which should map in the PD
invlpg [ebx] // Invalidates the virtual address we used just in
// case it could cause later problems

mov cr3, ecx // Restore everything nicely
/********************** END CRITICAL SECTION

// Determine the virtual address of the base of the PDT
// (remembering the differences in alignment)
mov eax, cr3 // A copy to modify for our new recursive PDE
and eax, 0x003FFFFFF // Only the most significant bits stay for 4M pages
mov ebx, PDEoffset
shl ebx, 22 // Offset into the PDT
or eax, ebx
mov PDEoffset, eax
The above, on a 32-bit non-PAE system, will return the virtual address that maps in the page directory and allows you to map in arbitrary physical memory as a known location. It should be noted that kernel privileges are needed (to access CR3) and to operate on a kernel page marked as Global so as to persist through the CR3 changes. The author hopes you enjoyed this weird machine and remember to treat your input data as formally as code, for only you can prevent vulnerabilities!

Sincerely,
@JacobTorrey