IOActive
Hardware | Software | Wetware
SECURITY SERVICES

I Know Where Your Page Lives

De-randomizing the latest Windows 10 Kernel

Enrique Elias Nissim

whoami

- Senior Consultant at IOActive
- Information System Engineer
- Infosec Enthusiast (Exploit Writing, Reversing, Programming, Pentesting)
- Conference Speaker:
    - EKOParty 11
    - EKOParty 12
    - CansecWest 2016
- @kiqueNissim

ZERONIGHTS

IOActive
Hardware | Software | Wetware
SECURITY SERVICES

Introduction

- Back in March, Nicolas Economou and I presented several ways of taking control of the OS by leveraging write-what-where kernel primitives regardless of the presence of mitigations such as DEP, ASLR, SMEP, etc.

Introduction

- The techniques relied on the fact that all the paging structures used by Windows could be always located in a fixed region of virtual memory.
  - HAL's HEAP
  - Spraying Page Directories
  - Double NULL Write
  - Self-Ref of Death

- See "Getting Physical: Extreme abuse of Intel based Paging Systems": https://cansecwest.com/slides/2016/CSW2016_Economou-Nissim_GettingPhysical.pdf

CVE 2016-7255

- Disclosing vulnerabilities to protect users:
  - https://security.googleblog.com/2016/10/disclosing-vulnerabilities-to-protect.html

- The Windows vulnerability is a local privilege escalation in the Windows kernel that can be used as a security sandbox escape. It can be triggered via the win32k.sys system call NtSetWindowLongPtr() for the index GWLP_ID on a window handle with GWL_STYLE set to WS_CHILD.

CVE 2016-7255

```
typedef struct tagWND
{
    struct tagWND *parent;
    struct tagWND *child;
    struct tagWND *next;
    struct tagWND *owner;
[..]
    DWORD       dwStyle;      /* Window style (from CreateWindow) */
    DWORD       dwExStyle;    /* Extended style (from CreateWindowEx) */
    UINT        wIDmenu;      /* ID or hmenu (from CreateWindow) */
    HMENU       hSysMenu;     /* window's copy of System Menu */
[..]
} WND;
```

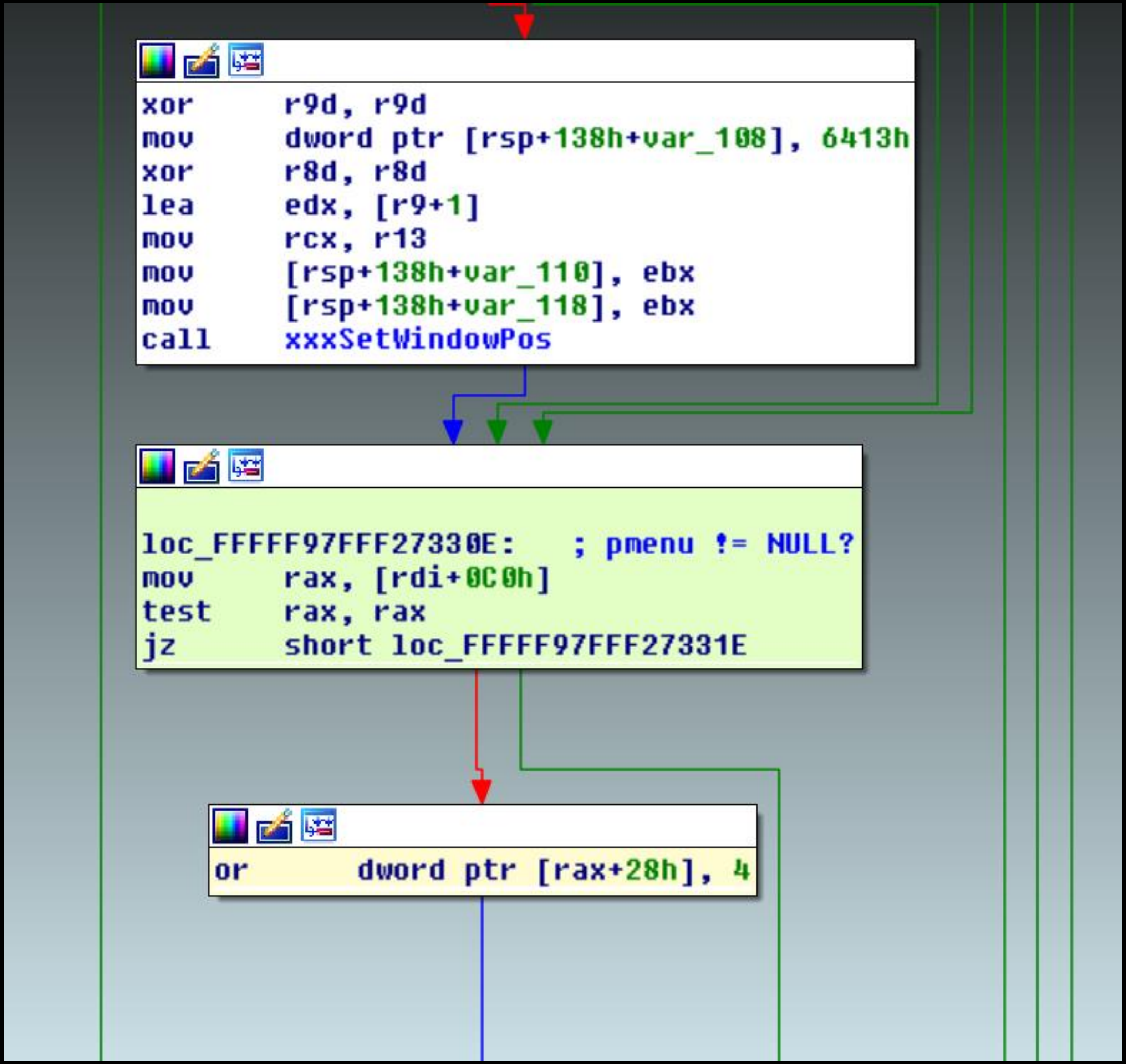
xxxSetWindowData

```
case GWL_ID:
    /*
     * Win95 does a TestWF(pwnd, WFCHILD) here, but we'll do the same
     * check we do everywhere else or it'll cause us trouble.
     */
    if (TestwndChild(pwnd)) {

        /*
         * pwnd->spmenu is an id in this case.
         */
        dwOld = (DWORD)pwnd->spmenu;
        pwnd->spmenu = (struct tagMENU *)dwData;
```

- Always checks if it is a child window with TestwndChild()

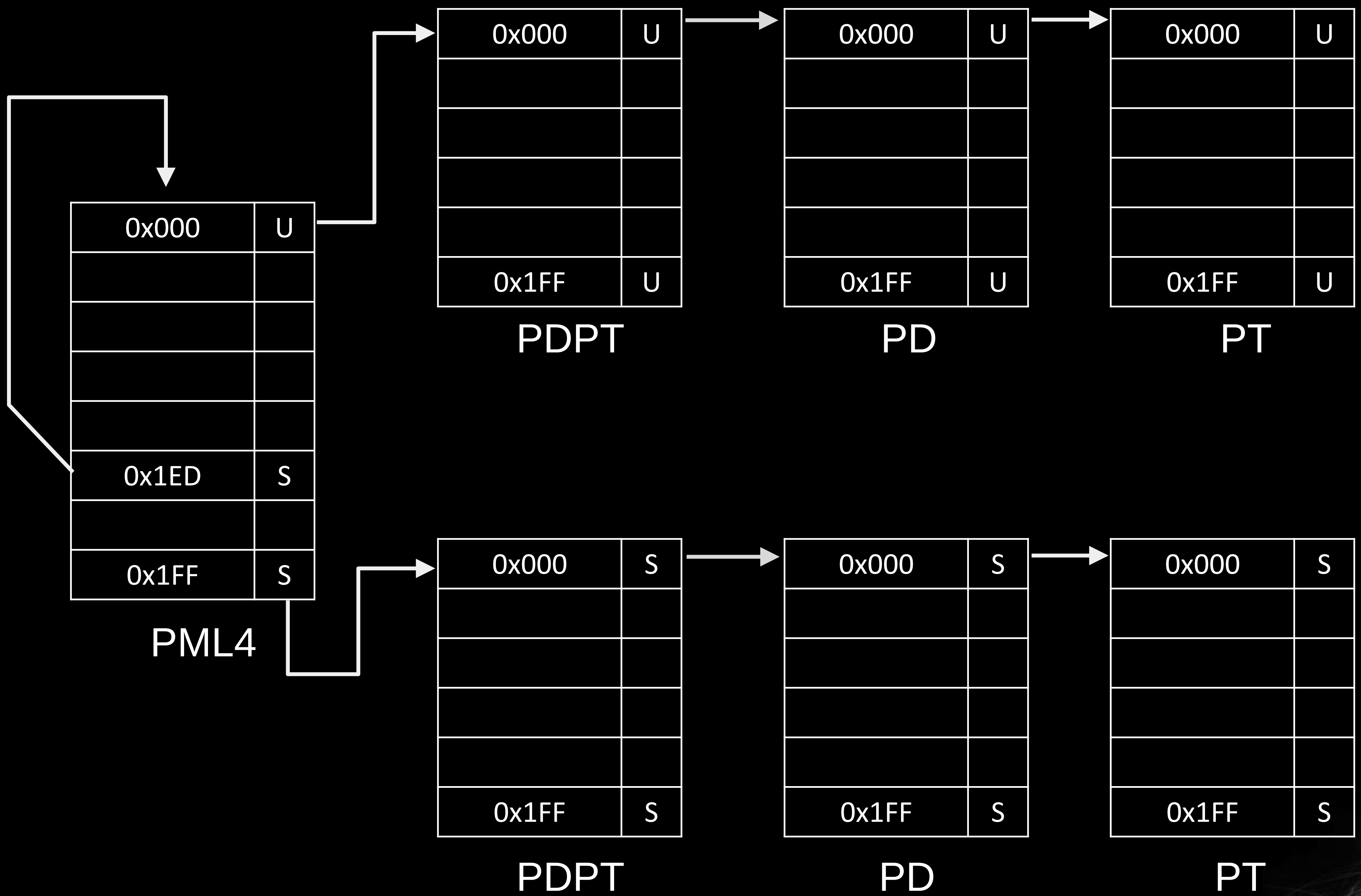
Except in Win32k!xxxNextWindow…



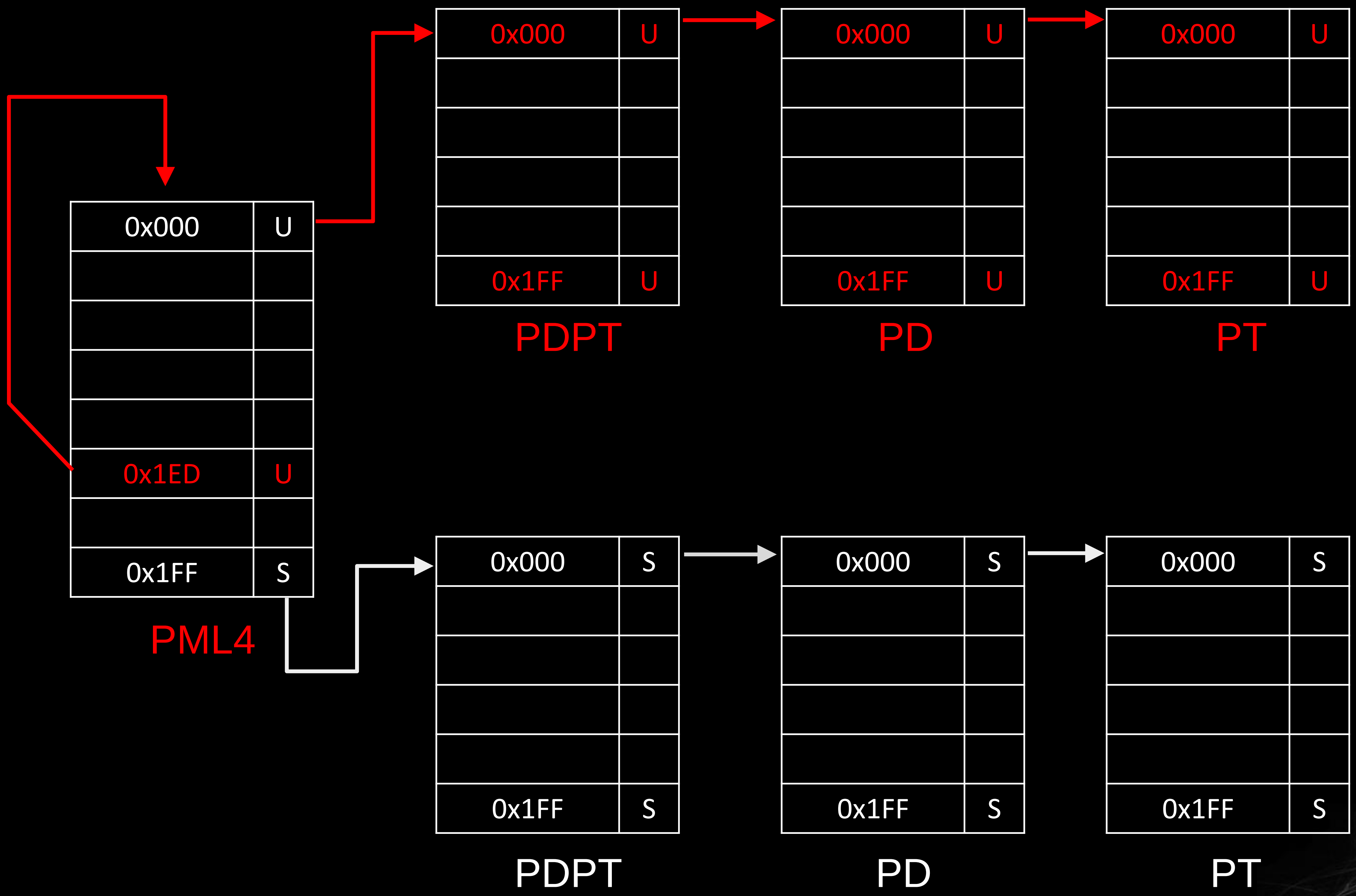
CVE 2016-7255

- PoC by @TinySecEx:
  - https://github.com/tinysec/public/tree/master/CVE-2016-7255

PML4 Self-Ref: 0xFFFFF6FB7DBEDF68



Self-Ref of Death: 0xFFFFF6FB7DBEDF68



Exploitation Steps

1. Leverage the vulnerability and flip the U/S bit in Self-Ref
2. Look for a free PML4E to use it as a spurious entry
3. Use the spurious entry to read the PTE of the HAL's Heap
4. Use the spurious entry to write the Shellcode into a free space in HAL's Heap
5. Turn off NX through the Spurious entry in the corresponding PTE.
6. Overwrite the HalpApicInterruptController pointer to our shellcode (it could be the original VA or a new one as long as we clear NX)
7. Profit

Our conclusions back then

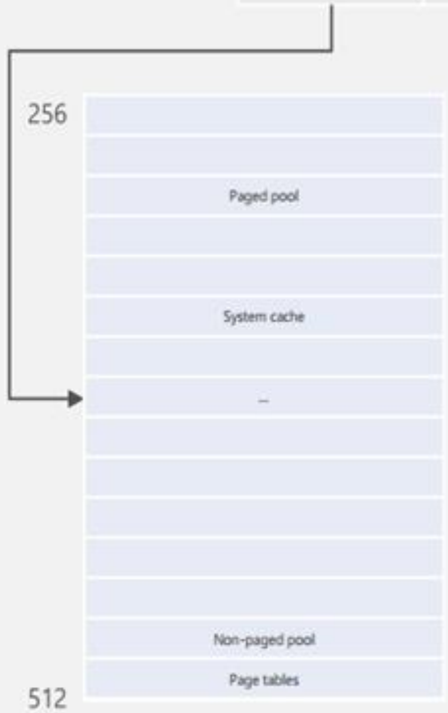
- Paging structures shouldn't be in fixed virtual region. It can be abused by local and remote kernel exploits
  - The PML4 entry (0x1ed) should be randomized

Microsoft Presentation at BlackHat 2016

Windows Kernel 64-bit ASLR Improvements

Predictable kernel address space layout has made it easier to exploit certain types of kernel vulnerabilities

64-bit kernel address space layout is now dynamic



System region PML4 entries are randomized

- ✓ Non-paged pool
- ✓ Paged pool
- ✓ System cache
- ✓ PFN database
- ✓ Page tables
- ✓ ... and so on

Various address space disclosures have been fixed

- ✓ Page table self-map and PFN database are randomized
  - Dynamic value relocation fixups are used to preserve constant address references
- ✓ SIDT/SGDT kernel address disclosure is prevented when Hyper-V is enabled
  - Hypervisor traps these instructions and hides the true descriptor base from CPL>0
- ✓ GDI shared handle table no longer discloses kernel addresses

Design Implications

- Total address space is partitioned in two halves:
  - User Address Space
  - Kernel Address Space
- Virtual Addresses are canonical: bit 48-63 has the same value as bit 47.
- The Self-Ref design requires an entire PML4 space for itself (512 GB).
- This means the randomization of the entry is restricted to: 2^47 / 2^39 = 256
- Keeping the same design, only 256 regions can be used to store the paging structures.

Potential new Self-Reference Entries

0xffff804020100800	0xffff904824120900	0xffffa05028140a00	0xffffb0582c160b00	0xffffc06030180c00	0xffffd068341a0d00	0xffffe070381c0e00	0xfffff0783c1e0f00
0xffff80c060301808	0xffff90c864321908	0xffffa0d068341a08	0xffffb0d86c361b08	0xffffc0e070381c08	0xffffd0e8743a1d08	0xffffe0f0783c1e08	0xfffff0f87c3e1f08
0xffff8140a0502810	0xffff9148a4522910	0xffffa150a8542a10	0xffffb158ac562b10	0xffffc160b0582c10	0xffffd168b45a2d10	0xffffe170b85c2e10	0xfffff178bc5e2f10
0xffff81c0e0703818	0xffff91c8e4723918	0xffffa1d0e8743a18	0xffffb1d8ec763b18	0xffffc1e0f0783c18	0xffffd1e8f47a3d18	0xffffe1f0f87c3e18	0xfffff1f8fc7e3f18
0xffff824120904820	0xffff924924924920	0xffffa25128944a20	0xffffb2592c964b20	0xffffc26130984c20	0xffffd269349a4d20	0xffffe271389c4e20	0xfffff2793c9e4f20
0xffff82c160b05828	0xffff92c964b25928	0xffffa2d168b45a28	0xffffb2d96cb65b28	0xffffc2e170b85c28	0xffffd2e974ba5d28	0xffffe2f178bc5e28	0xfffff2f97cbe5f28
0xffff8341a0d06830	0xffff9349a4d26930	0xffffa351a8d46a30	0xffffb359acd66b30	0xffffc361b0d86c30	0xffffd369b4da6d30	0xffffe371b8dc6e30	0xfffff379bcde6f30
0xffff83c1e0f07838	0xffff93c9e4f27938	0xffffa3d1e8f47a38	0xffffb3d9ecf67b38	0xffffc3e1f0f87c38	0xffffd3e9f4fa7d38	0xffffe3f1f8fc7e38	0xfffff3f9fcfe7f38
0xffff844221108840	0xffff944a25128940	0xffffa45229148a40	0xffffb45a2d168b40	0xffffc46231188c40	0xffffd46a351a8d40	0xffffe472391c8e40	0xfffff47a3d1e8f40
0xffff84c261309848	0xffff94ca65329948	0xffffa4d269349a48	0xffffb4da6d369b48	0xffffc4e271389c48	0xffffd4ea753a9d48	0xffffe4f2793c9e48	0xfffff4fa7d3e9f48
0xffff8542a150a850	0xffff954aa552a950	0xffffa552a954aa50	0xffffb55aad56ab50	0xffffc562b158ac50	0xffffd56ab55aad50	0xffffe572b95cae50	0xfffff57abd5eaf50
0xffff85c2e170b858	0xffff95cae572b958	0xffffa5d2e974ba58	0xffffb5daed76bb58	0xffffc5e2f178bc58	0xffffd5eaf57abd58	0xffffe5f2f97cbe58	0xfffff5fafd7ebf58
0xffff86432190c860	0xffff964b2592c960	0xffffa6532994ca60	0xffffb65b2d96cb60	0xffffc6633198cc60	0xffffd66b359acd60	0xffffe673399cce60	0xfffff67b3d9ecf60
0xffff86c361b0d868	0xffff96cb65b2d968	0xffffa6d369b4da68	0xffffb6db6db6db68	0xffffc6e371b8dc68	0xffffd6eb75badd68	0xffffe6f379bcde68	0xfffff6fb7dbedf68
0xffff8743a1d0e870	0xffff974ba5d2e970	0xffffa753a9d4ea70	0xffffb75badd6eb70	0xffffc763b1d8ec70	0xffffd76bb5daed70	0xffffe773b9dcee70	0xfffff77bbddeef70
0xffff87c3e1f0f878	0xffff97cbe5f2f978	0xffffa7d3e9f4fa78	0xffffb7dbedf6fb78	0xffffc7e3f1f8fc78	0xffffd7ebf5fafd78	0xffffe7f3f9fcfe78	0xfffff7fbfdfeff78
0xffff884422110880	0xffff984c26130980	0xffffa8542a150a80	0xffffb85c2e170b80	0xffffc86432190c80	0xffffd86c361b0d80	0xffffe8743a1d0e80	0xfffff87c3e1f0f80
0xffff88c462311888	0xffff98cc66331988	0xffffa8d46a351a88	0xffffb8dc6e371b88	0xffffc8e472391c88	0xffffd8ec763b1d88	0xffffe8f47a3d1e88	0xfffff8fc7e3f1f88
0xffff8944a2512890	0xffff994ca6532990	0xffffa954aa552a90	0xffffb95cae572b90	0xffffc964b2592c90	0xffffd96cb65b2d90	0xffffe974ba5d2e90	0xfffff97cbe5f2f90
0xffff89c4e2713898	0xffff99cce6733998	0xffffa9d4ea753a98	0xffffb9dcee773b98	0xffffc9e4f2793c98	0xffffd9ecf67b3d98	0xffffe9f4fa7d3e98	0xfffff9fcfe7f3f98
0xffff8a45229148a0	0xffff9a4d269349a0	0xffffaa552a954aa0	0xffffba5d2e974ba0	0xffffca6532994ca0	0xffffda6d369b4da0	0xffffea753a9d4ea0	0xfffffa7d3e9f4fa0
0xffff8ac562b158a8	0xffff9acd66b359a8	0xffffaad56ab55aa8	0xffffbadd6eb75ba8	0xffffcae572b95ca8	0xffffdaed76bb5da8	0xffffeaf57abd5ea8	0xfffffafd7ebf5fa8
0xffff8b45a2d168b0	0xffff9b4da6d369b0	0xffffab55aad56ab0	0xffffbb5daed76bb0	0xffffcb65b2d96cb0	0xffffdb6db6db6db0	0xffffeb75badd6eb0	0xfffffb7dbedf6fb0
0xffff8bc5e2f178b8	0xffff9bcde6f379b8	0xffffabd5eaf57ab8	0xffffbbddeef77bb8	0xffffcbe5f2f97cb8	0xffffdbedf6fb7db8	0xffffebf5fafd7eb8	0xfffffbfdfeff7fb8
0xffff8c46231188c0	0xffff9c4e271389c0	0xffffac562b158ac0	0xffffbc5e2f178bc0	0xffffcc6633198cc0	0xffffdc6e371b8dc0	0xffffec763b1d8ec0	0xfffffc7e3f1f8fc0
0xffff8cc6633198c8	0xffff9cce673399c8	0xffffacd66b359ac8	0xffffbcde6f379bc8	0xffffcce673399cc8	0xffffdcee773b9dc8	0xffffecf67b3d9ec8	0xfffffcfe7f3f9fc8
0xffff8d46a351a8d0	0xffff9d4ea753a9d0	0xffffad56ab55aad0	0xffffbd5eaf57abd0	0xffffcd66b359acd0	0xffffdd6eb75badd0	0xffffed76bb5daed0	0xfffffd7ebf5fafd0
0xffff8dc6e371b8d8	0xffff9dcee773b9d8	0xffffadd6eb75bad8	0xffffbddeef77bbd8	0xffffcde6f379bcd8	0xffffddeef77bbdd8	0xffffedf6fb7dbed8	0xfffffdfeff7fbfd8
0xffff8e472391c8e0	0xffff9e4f2793c9e0	0xffffae572b95cae0	0xffffbe5f2f97cbe0	0xffffce673399cce0	0xffffde6f379bcde0	0xffffee773b9dcee0	0xfffffe7f3f9fcfe0
0xffff8ec763b1d8e8	0xffff9ecf67b3d9e8	0xffffaed76bb5dae8	0xffffbedf6fb7dbe8	0xffffcee773b9dce8	0xffffdeef77bbdde8	0xffffeef77bbddee8	0xfffffeff7fbfdfe8
0xffff8f47a3d1e8f0	0xffff9f4fa7d3e9f0	0xffffaf57abd5eaf0	0xffffbf5fafd7ebf0	0xffffcf67b3d9ecf0	0xffffdf6fb7dbedf0	0xffffef77bbddeef0	0xffffff7fbfdfeff0
0xffff8fc7e3f1f8f8	0xffff9fcfe7f3f9f8	0xffffafd7ebf5faf8	0xffffbfdfeff7fbf8	0xffffcfe7f3f9fcf8	0xffffdfeff7fbfdf8	0xffffeff7fbfdfef8	0xfffffffffff8

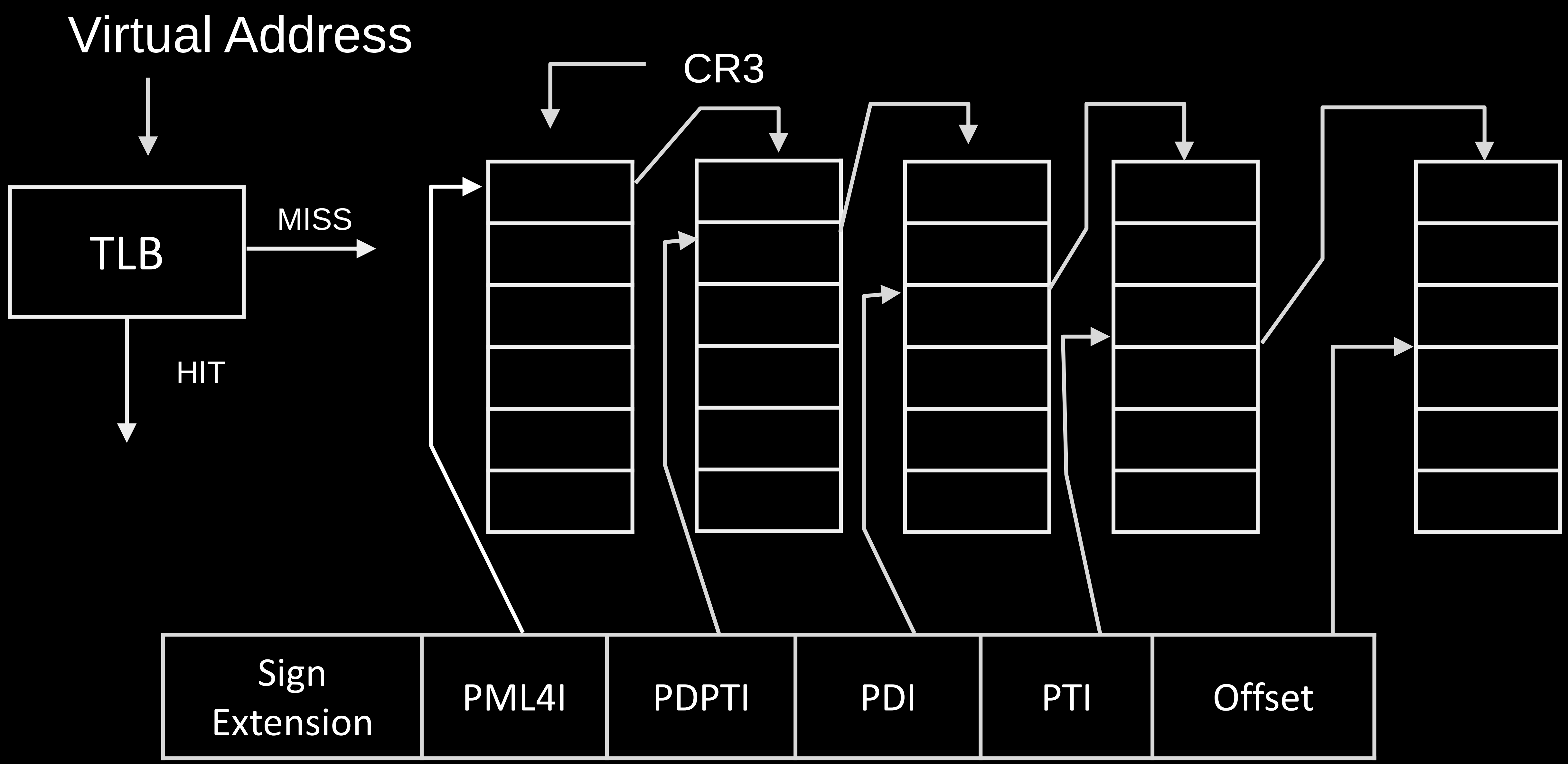
256 Posibilities?

- Not all of them are always mapped
  - A mistake leads to a BSOD
- There are regions which even now remain “fixed” (and mapped):
  - HAL’s HEAP - 0xFFFFFFFFFFD00000
  - (This means the last PML4 Entry cannot be the Auto-Mapping one).

KASLR Timing Attacks

- “Practical Timing Side Channel Attacks Against Kernel Space ASLR” - http://www.ieee-security.org/TC/SP2013/papers/4977a191.pdf
  - No memory disclosure bug required!
  - The “Double Page Fault” Technique could be used to detect whether a page is mapped or unmapped.

Theory behind it



Theory behind it

There is a measurable time difference between accessing a mapped page and accessing an unmapped page.

- Accessing an unmapped kernel address from Ring3:
  - Look in the TLB
  - Do the PageWalk
  - No Entry => PageFault

- Accessing a mapped kernel address from Ring3:
  - Look in the TLB
  - Do the PageWalk
  - Cache the entry in the TLB
  - Access Violation => PageFault

Double Page Fault

```
UINT64 side_channel_exception(PVOID lpAddress) {
    UINT64 begin = 0;
    UINT64 difference = 0;

    unsigned int tsc_aux1 = 0;
    unsigned int tsc_aux2 = 0;
    __try {
        begin = __rdtscp(&tsc_aux1);
        *(char *)lpAddress = 0x00;
        difference = __rdtscp(&tsc_aux2) - begin;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        difference = __rdtscp(&tsc_aux2) - begin;
    }
    return difference;
}
```

In Practice…

- Problem: a lot of noise!… Even a small CPU burst could affect the end result.
  - Unmapped Time: 3168
  - Mapped Time: 3048
  - Initial Difference: ~120 cycles
- During the actual execution:
  - Unmapped: 3320
  - Mapped: 3327
- Yes, there are times that the Mapped time is higher than the Unmapped one in the same run!

Conclusion: it might work, but it would take a considerable amount of time to get consistency and reliable results

TSX to the Rescue!

https://www.blackhat.com/docs/us-16/materials/us-16-Jang-Breaking-Kernel-Address-Space-Layout-Randomization-KASLR-With-Intel-TSX-wp.pdf

Transactional Synchronization Extensions

- A feature included since Haswell (2013)
  - New instructions that allows to improve the performance of multithreaded applications:
    - Hardware Lock Elision (HLE)
      - XAQUIRE / XRELEASE
    - Restricted Transactional Memory (RTM)
      - XBEGIN / XEND / XABORT
- The processor determines dynamically whether threads need to serialize through lock-protected critical sections and performs serialization only when required.
- Rafal Wojtczuk: was the first one to talk about how can be used to attack ASLR
  - https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/

Transactional Synchronization Extensions

- Programmer specified code regions are executed transactionally. If the operation is successful it is called an atomic commit
- If something goes wrong, a transactional abort occurs, and the processor continues the execution at the fallback address provided.

8.3.8.2 Runtime Considerations

In addition to the instruction-based considerations, runtime events may cause transactional execution to abort. These may be due to data access patterns or micro-architectural implementation causes. Keep in mind that the following list is not a comprehensive discussion of all abort causes.

Any fault or trap in a transaction that must be exposed to software will be suppressed. Transactional execution will abort and execution will transition to a non-transactional execution, as if the fault or trap had never occurred. If any exception is not masked, that will result in a transactional abort and it will be as if the exception had never occurred.

Transactional Aborts

- The execution never leaves UserMode!
- This means we don't have all the overhead associated with the page fault handling mechanism as with __try __except blocks.
- LESS NOISE => MORE ACCURACY

Using TSX RTM as a Side Channel

```
UINT64 side_channel_tsx(PVOID lpAddress) {
    UINT64 begin = 0;
    UINT64 difference = 0;
    int status = 0;

    unsigned int tsc_aux1 = 0;
    unsigned int tsc_aux2 = 0;
    begin = __rdtscp(&tsc_aux1);
    if ((status = _xbegin()) == _XBEGIN_STARTED) {
        *(char *)lpAddress = 0x00;
        difference = __rdtscp(&tsc_aux2) - begin;
        _xend();
    }
    else {
        difference = __rdtscp(&tsc_aux2) - begin;
    }
    return difference;
}
```

Using TSX RTM as a Side Channel

- Testing on Core i7 6700K:
  - Unmapped Time: 221
  - Mapped Time: 191
  - Difference: ~40 cycles
- A lot faster than exceptions!

Getting References

We need to get measures that will serve as references for both Mapped and Unmapped pages.

- Getting the unmapped one is easy:
  - Target the virtual address 0x0000000000000000
- For the mapped one we could use the HAL's HEAP: 0xFFFFFFFFFFD00000 - We know this is fixed and mapped even after Anniversary Patch.
- But that might change in the near future. So it's better to just map our own, and simulate a privilege access failure through the R/W permission.
  - VirtualAlloc()
  - memset() => to actually allocate the page (otherwise it's just reserved)
  - VirtualProtect() => change the page to be Read-Only

How to determine the actual one?

Now we are able to know which ones of the former list are mapped:

- Potential: 0xffffd96cb65b2d90
- Potential: 0xffffec763b1d8ec0
- Potential: 0xffffecf67b3d9ec8
- Potential: 0xffffed76bb5daed0
- Potential: 0xffffedf6fb7dbed8
- Potential: 0xffffee773b9dcee0
- Potential: 0xffffeef77bbddee8

Better… but not enough

Back to the entries!

In the old days, we had 0x1ed as the self-reference entry

0x1ED = 1 1110 1101

1111	**1111**	**1111**	**1111**	**1111**	**0101**	**1XXX**	**XXXX**
F	F	F	F	F	6	8-F	0-F

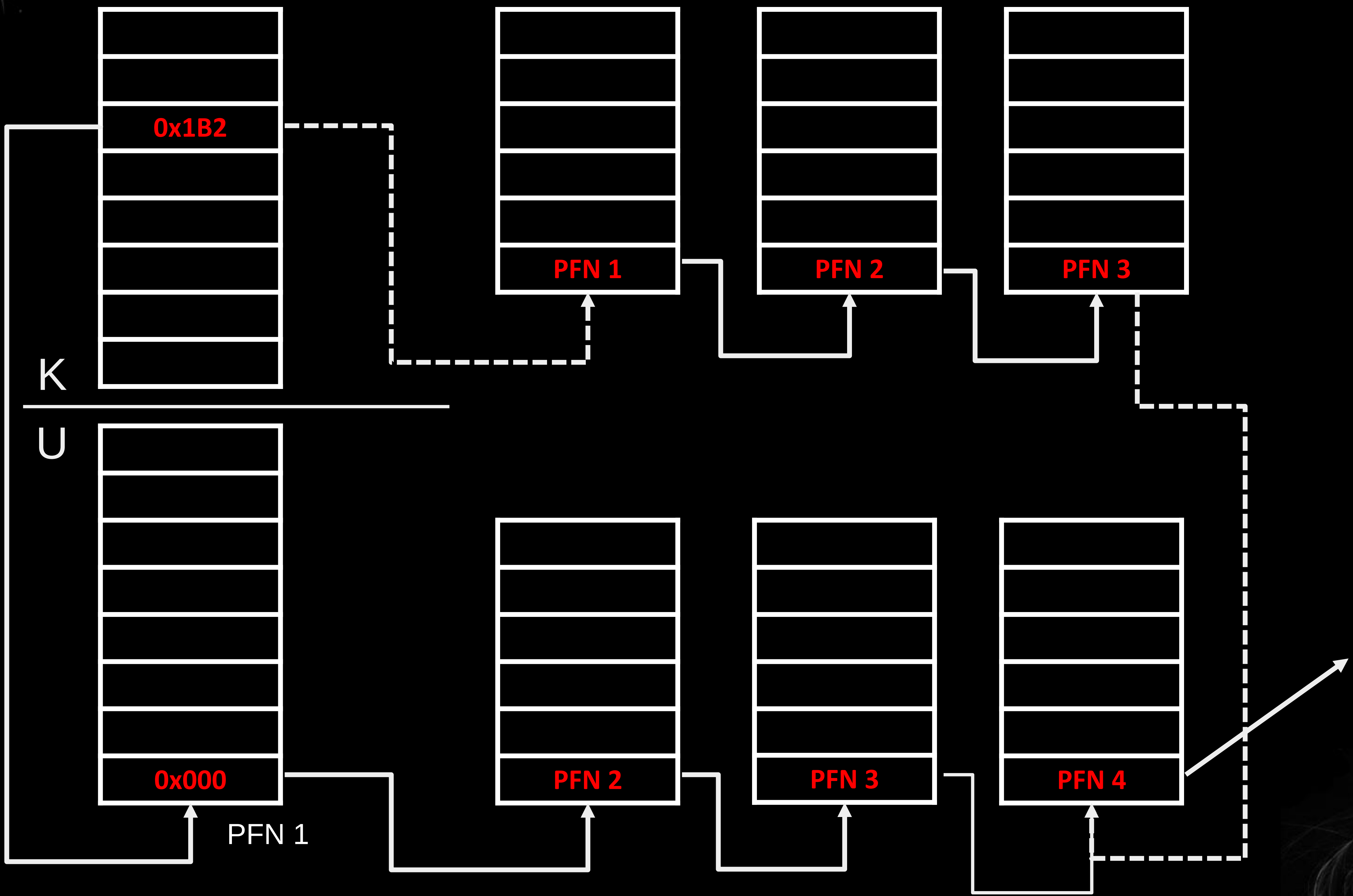
And we had the formula:

```
UINT64 get_pxe_address(UINT64 address) {
    UINT64 result = address>>9;
    result = result | 0xFFFFF68000000000;
    result = result & 0xFFFFF6FFFFFFFFF8;
    return result;
}
```

Insight: Allocate Probing Pages!

- The key here is that we know that the real PML4 will have all these allocated!
- Let’s assume 0xffffd96cb65b2d90 is the PML4 SelfRef.
  - That means the index is 0x1B2
- So let’s assume we can allocate a page at virtual address 0x0; then the PTE describing this page should be at 0xFFFFD90000000000

Insight: Allocate Probing Pages!



New !pte formula

```
UINT64 get_pxe_address(UINT64 address, UINT entry)  {
    UINT64 result = address>>9;
    UINT64 lower_boundary = ((UINT64)0xFFFF << 48) |
((UINT64)entry << 39);
    UINT64 upper_boundary =  \\
      (((UINT64)0xFFFF << 48) | ((UINT64)entry << 39) +
0x8000000000 - 1) & 0xFFFFFFFFFFFFFFF8;
    result = result | lower_boundary;
    result = result & upper_boundary;
    return result;
}
```

Dummy(s) PML4E

- I found that there was more than one PML4E that had all the entries mapped to a dummy page, so the previous check is going to pass but it isn’t actually the PML4 Self Ref.
- Solution: probe for one address we know is UNMAPPED => The dummy PML4E will succeed, while the real one will fail.

Consistency in the Measures

- Problem: each processor has its own set of TLBs
- Solution: We need to make sure we always run on the same core:

```
void set_processor_affinity(void) {
    GROUP_AFFINITY affinity = { 0 };
    affinity.Group = 0;
    affinity.Mask = 1;
    SetThreadGroupAffinity(
          GetCurrentThread(),
          &affinity,
          NULL);
}

void set_thread_priority(void) {
     SetThreadPriority(GetCurrentThread(), 31);
}
```

Consistency in the Measures

- Problem: The load on the CPU affects every measure we take.
- Solution: Probe an address 200.000 and get the average. Then repeat the process X times and get the median. Also use global counters to keep track of mistakes (measures that are too far from our references). If the Global mistakes are above a threshold, cancel the process and start over (keeping the work done).

Live Demo

What about AMD?

- Advanced Synchronization Facilities
- Instructions: SPECULATE / COMMIT / ABORT
- Wikipedia: “As of October 2013, it was still in the proposal stage. No released microprocessors implement the extension.”
- It seems it remains as a proposal. No opcodes for the instructions: http://developer.amd.com/wordpress/media/2008/10/24594_APM_v3.pdf

Countermeasures / Ideas

- TLB Cache modification: do not cache the PFN in the TLB if the privileges are not met.
  - Requires hardware modification
- Separate page tables for Kernel and User.
  - Performance degradation
- Change the memory type for the region to something different than Write-Back
- Switch to AMD? :)

Questions?

Thank you