Writing PoCs for processor software side-channels

Discovery of MDS vulnerabilities

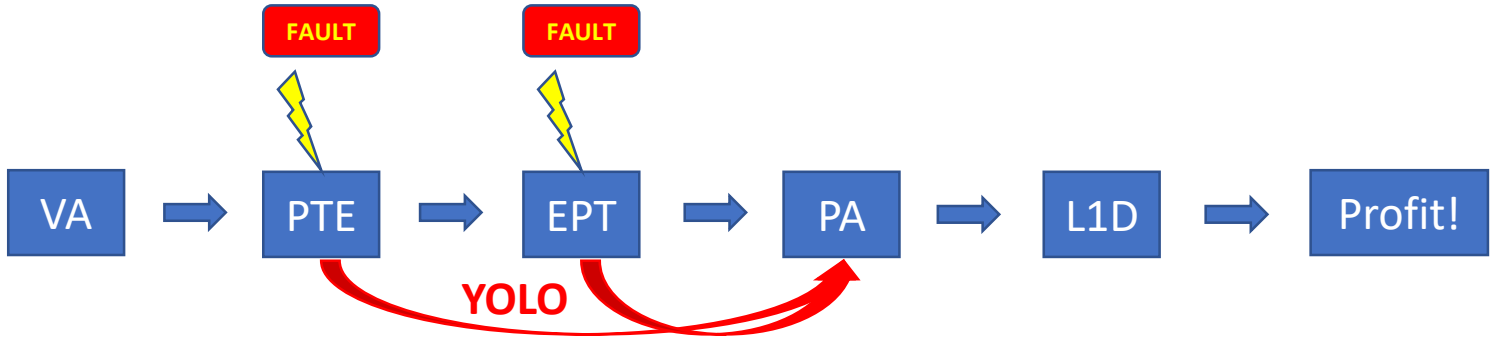
MFBDS (CVE-2018-12130) MDSUM (CVE-2019-11091)

About

Volodymyr Pikhur - security researcher from Portland, Oregon @vpikhur

How did it start?

L1TF (CVE-2018-3615, CVE-2018-3620, CVE-2018-3646)



Speculative execution engine (SEE) keeps executing instructions in parallel until actual delivery of the fault

SEE uses PTE.PFN or EPT.PFN directly to calculate Physical Address (PA)

From Meltdown vulnerability we know SEE doesn't check in which processor mode it is actually executing same applies here thus SGX, SMM, VMM are vulnerable to L1TF

https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault

Writing reproduction PoC

1. Code will be messy use source code version control
2. There will be bugs and frustration
3. Test against false positives/negatives to validate behavior consistency
4. Look for patterns like cache line size (64) or power of two
5. Typical bug - infinite amount of random results
6. Consult manual for other sizes of internal CPU structures
7. Use TSX if available
8. Use visualization of your results
9. Run multiple iterations of the same test
10. Use CPUID to flush pipeline

Reconnaissance

Intel: *"A paging structure entry is vulnerable to an L1TF exploit when it is not **Present (P=0)** or when it has a **reserved bit set**[2]. A fault delivered for either of these entries is a terminal fault because the condition causes the address translation process to be terminated immediately, without completing the translation. Although access is denied, speculative execution may still occur."*

Me: But what about other cases e.g. PTE.Write? Trying to write to RO memory should also trigger L1TF why Intel didn't mention it?🤔

https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault

Validation of results using CLFLUSH

After CLFLUSH has been issued we shouldn’t see any more leaks since it is not in L1D

```
00000772 PF[4] byte: 02: time: 54 (clocks)
0003657f PF[4] byte: 02: time: 63 (clocks)
000784a7 PF[4] byte: 02: time: 63 (clocks)
000ff789 PF[4] byte: 02: time: 48 (clocks)
001e5285 PF[4] byte: 02: time: 69 (clocks)
002149ab PF[4] byte: 02: time: 42 (clocks)
0021e726 PF[4] byte: 02: time: 57 (clocks)
```

In case of false-positives I wouldn’t see same byte ’02’ being leaked something wasn’t right I needed to validate it

Validation of results using PTE.UC

The only way to validate my data isn't being pulled into cache is to use uncacheable memory

Mitigations preventing data from being loaded into L1D

Software can prevent data from being reloaded into the L1D after a flush by removing cacheable mappings of that data.

For instance, to prevent secret data from being loaded into the L1D via a paging structure entry, software could clear the Present bit of the entry and flush the paging structure caches followed by flushing the L1D. An alternative to clearing the Present bit would be using uncacheable memory types specified via the *Page Attribute Table* (PAT) or *Memory Type Range Registers* (MTRRs), both of which also prevent data from being loaded in the L1D.

Further cache flushing operations are not required for mitigation after these entries have been put in place.

Loads cannot:

- Speculatively take any sort of fault or trap.
- Speculatively access uncacheable memory.

Validation of results using PTE.UC

- Even after using uncacheable memory I still could observe leaks:

```
0001fbde PF[4] byte: EE: time: 45 (clocks)
0003846b PF[4] byte: 02: time: 36 (clocks)
0004d1e2 PF[4] byte: 02: time: 36 (clocks)
0007140c PF[4] byte: 02: time: 36 (clocks)
0007b457 PF[4] byte: 02: time: 36 (clocks)
0007ba39 PF[4] byte: EE: time: 54 (clocks)
0007d0f6 PF[4] byte: EE: time: 42 (clocks)
000bf472 PF[4] byte: 02: time: 42 (clocks)
000cc602 PF[4] byte: 02: time: 42 (clocks)
```

Digging into manuals

Table 2-18. L1 Data Cache Components

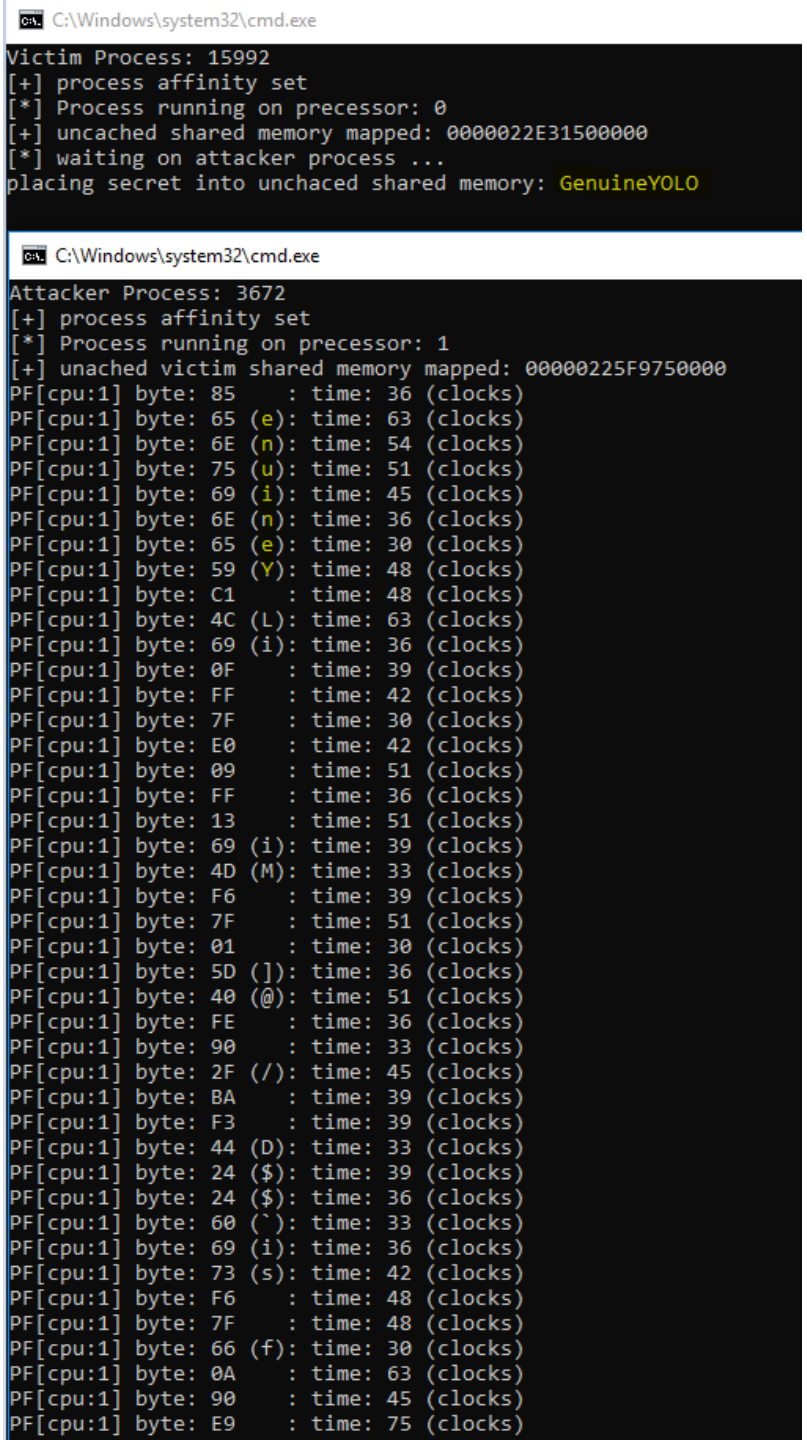
Component	Intel microarchitecture code name Sandy Bridge	Intel microarchitecture code name Nehalem
Data Cache Unit (DCU)	32KB, 8 ways	32KB, 8 ways
Load buffers	64 entries	48 entries
Store buffers	36 entries	32 entries
Line fill buffers (LFB)	10 entries	10 entries

Infrequent results due to small number of entries and they are shared between two physical threads.

https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/638034

MDS PoC

- Since number of buffers is really limited the only way to validate it is to make one physical thread constantly writing to UC memory and second triggering ~~L1TF~~ MDS
- Entry size is same as cache-line 64 bytes
- Researchers used same technique to constantly run Linux passwd and leak **/etc/shadow**
- Doing so would be immediately detected by EDR



MDS PoC DEMO

https://github.com/hwroot

Conclusions

- Researchers make mistakes
  - My original report claimed leak is coming from Load/Store in fact it was Line Fill Buffer
  - MDSUM was reported by TU Gratz as Meltdown UC on Mar 28 2018 they even correctly predicted data is coming from LFB but didn't provide any details and left it for future research.
- Validate your results!
  - CLFLUSH is out-of-order it is only in order in respect to other CLFLUSH instructions
  - Pay attention to SDM
  - Don't jump to conclusions

The L1 DCache can maintain up to 64 load micro-ops from allocation until retirement. It can maintain up to 36 store operations from allocation until the store value is committed to the cache, or written to the line fill buffers (LFB) in the case of non-temporal stores.

- No validation tools to inspect internal data structures

https://twitter.com/raphael_scarv/status/952088189379989505 2018-Jan-13
https://meltdownattack.com/meltdown.pdf

Questions?