Preventing CPU side-channel attacks with kernel tracking

Marian Marinov
mm@siteground.com

Chief System Architect
Head of the DevOps department
❖ Who am I?

- Chief System Architect of SiteGround.com
- Sysadmin since 1996
- Organizer of OpenFest, BG Perl Workshops, LUG-BG and others
- Teaching Network Security and Linux System Administration courses in Sofia University and SoftUni
What I'm proposing is NOT a general purpose solution!
Disclaimer

We are a shared hosting provider... we consider all code, hostile
Disclaimer

We haven't seen Meltdown attempts on our infrastructure
Intel's microcode updates and KPTI were supposed to result in 10-15% performance DEGRADATION.
a little bit of intro

Needless to say...
I was not a big fan of implementing both, across all of our servers
a little bit of intro

10-15% on a single machine are not a problem on 1000s of machines... that is a bit different
There are 4* different caches inside the CPU
There are 4* different caches inside the CPU

<table>
<thead>
<tr>
<th>Single Core</th>
<th>Single Core</th>
<th>Single Core</th>
<th>Single Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 instructions cache</td>
<td>L1 I cache</td>
<td>L1 I cache</td>
<td>L1 I cache</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>L1 D cache</td>
<td>L1 D cache</td>
<td>L1 D cache</td>
</tr>
<tr>
<td>L2 cache</td>
<td>L2 cache</td>
<td>L2 cache</td>
<td>L2 cache</td>
</tr>
</tbody>
</table>

L3 cache

* In some architectures, there is even L4 cache
Sharing the cache

L1 and L2 caches are shared between hyper-threads in a single core.

L2 cache is shared between different execution engines inside the core (ALU, FMA, ADD, etc.).

L3 cache is shared between all cores.
Some CPU architecture intro :)
Cache Side-Channel Attacks

- 2013 Flush + Reload
- 2016 Flush + Flush
Flush + Reload

1. Find a shared library location in memory
2. Clear the cache
3. Check if the victim has accessed it or not by comparing the time it takes to execute the code
Flush + Flush

1. Find a shared library location in memory
2. Clear the cache
3. Clear the cache again and observe the timing
   if the victim has accessed the code, clflush will
   take longer to finish
Thread 1: floating point
More architecture...

Thread 1: integer
Thread 2: floating point
So we will look at protections from Meltdown
Monitoring and analysis

➢ KPTI was already in the making
➢ Capsule 8 – wrote on Jan 5
➢ EndGame – shared their research on Jan 08
➢ I found out about EndGame and Capsule 8 on Jan 11, after we had already started on our work
Capsule 8 approach

Kernel tracepoints and monitor for:

exceptions/page_fault_user

Kernel perf counters

- PERF_COUNT_HW_CACHE_OP_READ
- PERF_COUNT_HW_CACHE_RESULT_ACCESS
- PERF_COUNT_HW_CACHE_RESULT_MISS

https://github.com/capsule8/capsule8/tree/master/examples
Capsule 8 approach

This is nice, but not enough...

since

Flush+Reload can be replaced by
Flush+Flush to achieve the same result without actual page miss

https://github.com/capsule8/capsule8/tree/master/examples
EndGame

They did not provide code examples...

However they explained a lot around the statistics and using the CPU performance counters.

Both examples by Capsule 8 and EndGame provide detection, but little to no countermeasures.
Fight the requirements not the attacks

➢ Successful meltdown exploitation prefers that both the SIGSEGV children and the victim are on the same CPU
➢ so we simply LIE to sched_setaffinity
➢ effectively **we do nothing**
➢ we save the requested affinity in the task_struct as `cpumask_t cpus_allowed`;
➢ we have patched `sched_getaffinity` to report only the cpu mask already stored for the current process
Fight the requirements not the attacks

➢ Successful meltdown exploitation requires that a process should have one of the following:
  ➢ SIGSEGV children or grandchildren
  ➢ SIGSEGV threads
  ➢ TSX instructions that do not finish successfully
Fight the requirements not the attacks

➢ On our infrastructure, there is no customer's software that has a valid case to have
  ➢ SIGSEGV children or threads
  ➢ our CPUs do not support TSX instructions :)

© SiteGround
So...

we decided to forbid SIGSEGV processes
What we did?

Kernel module
What we did?

Kernel module

- detecting processes that had more than 1 child dying with SIGSEGV
What we did?

Kernel module

- detecting processes that had more than 1 child dying with SIGSEGV
- when such process is detected it is STOPPED, not KILLED
What we did?

Kernel module

- detecting processes that had more than 1 child dying with SIGSEGV
- when such process is detected it is STOPPED, not KILLED
- only the root on the host machine can send any type of signals to this process
What we had to change?

- Introduced a per process counter of its SIGSEGV children
  
  ```c
  unsigned int pids[MAX_PID];
  ```

- Implement a workqueue to check for abusers
  
  ```c
  create_singlethread_workqueue()
  ```

- Implement a `/proc` interface to monitor and change the max segfaults
Similar attack pattern?

➢ Forshadow-OS
➢ Forshadow-VMM
➢ Forshadow-SGX

➢ All of the above require the generation of page faults, which is essentially the same side effect that we see with Meltdown
Cache Line Flush

- Limiting clflush and clflushopt effectively stops Flush+Reload and Flush+Flush attacks
- Cache flush can be indirectly called when invalid instruction is issued
- This greatly limits the options for executing Meltdown, by leaving only TSX instructions
Cache Line Flush

- both clflush and clflushopt are unprivileged instructions. Trapping them is not directly possible.

- We discussed different approaches:
  
  - Inspecting the instructions of each binary, before it is executed and marking it clean.
  
  - Inspecting the binary in parallel while the program is executing.

  - Virtualizing the system and actually trapping the instructions after they have been evaluated by the guest kernel.
Cache Line Flush

- adding noclflush on the kernel cmdline does NOT disable clflush!!!

- Events you can monitor for clflush:
  - L2_LINES_OUT.DEMAND_CLEAN
  - MEM_LOAD_UOPS RETIRED.L3_MISS
  - PERF_COUNT_HW_CACHE_LL
  - PERF_COUNT_HW_CACHE_OP_READ
  - PERF_COUNT_HW_CACHE_RESULT_ACCESS
  - PERF_COUNT_HW_CACHE_RESULT_MISS
TSX

Transactional Synchronization eXtensions (TSX)

- Because of issues with the implementation, TSX instructions should be disabled on Haswell CPUs.
- However if the microcode is not applied, your Haswell CPUs support TSX :)
- TSX instructions are supported on Skylake...
TSX

- One thing that EndGame showed us is that TSX instructions can be counted

- RTM RETIRED.ABORTED
TSX

- I considered lying to userspace by reporting that TSX is not supported by the CPU...

  - but the cpuid instruction is unprivileged so trapping it is a non-trivial job 😞
So relying on CPU counters is currently the only sensible way of detecting if TSX is being used for Meltdown exploits.

The legitimate use of TSX instructions is very limited, and on shared hosting we will most likely not see such software.
**TSX events**

- **RTM_RETIRED.START** - Number of times we entered an RTM region. Does not count nested transactions

- **RTM_RETIRED.ABORTED** - Number of times an RTM execution aborted due to any reasons (multiple categories may count as one)

- **RTM_RETIRED.ABORTED_TIMER** - Number of times an RTM execution aborted due to uncommon conditions
you can find the list of Processor Monitor Unit (PMU) events by running:

```
# perf list
```

Perf can be build from the linux kernel source tree in tools/perf:

```
# make
# mv perf /usr/bin
```
Glossary

ALU - Arithmetic Logic Unit
AGU - Address Generation Unit
TLB - Translation Lookaside Buffer
BTP - Branch Target Predictor
BP - Branch Predictor
BTB - Branch Target Buffer
LLC - Last Level Cache
WB Cache - Writeback cache
W.C. Cache - Write combining cache
Trace Cache - execution trace cache
Links

Flush + Reload paper
Flush + Flush paper
Spectre & Meltdown attacks
TLBleed attack
Forshadow attacks
EndGame research
Capsule 8 meltdown detection

Meltdown PoC
Collection of Speculation bugs info