Recitation 22: Meltdown

MIT - 6.033

Spring 2022

Henry Corrigan-Gibbs

Plan

- Recitation Qs

Meltdown

→ Load kernel data into cache

→ Read kernel data out of cache

Logistics

* DB Hands-on due TODAY!
* Design project due May 2
  ↳ 24 hr extension avail if you ask ahead
* If you like this, consider 6.5060 in Fall '22

Poll:

Do you know what a cache is?

Recitation Questions — in groups

1. What is the Meltdown attack?

   - Technique to read kernel memory from user space

   - Doesn't work on modern processors* or fully patched OSes (Linux, MacOS, etc.)

2. How does it work?

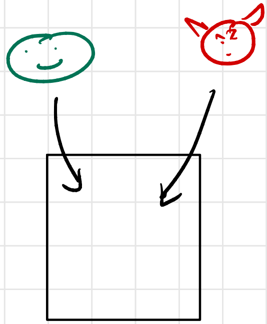
   a) Trick CPU into loading item into cache whose address depends on kernel data
   b) Use cache-timing to extract this info from cache
   c) Repeat

3. Why is this attack possible?

   - CPU designers prioritize speed
     ↳ Didn't really expect this "side-channel" leakage to be so problematic.

   - CPU "speculates past" permissions checks

Meltdown



Same machine
(login.csail.mit.edu)

Goal: Read data of another user on the same machine.

- Email
- Cryptographic keys
- Passwords
- ....

Assumes: Attacker running as unprivileged user

↳ e.g. two MIT users on the same cluster machine

e.g. two users on Amazon EC2

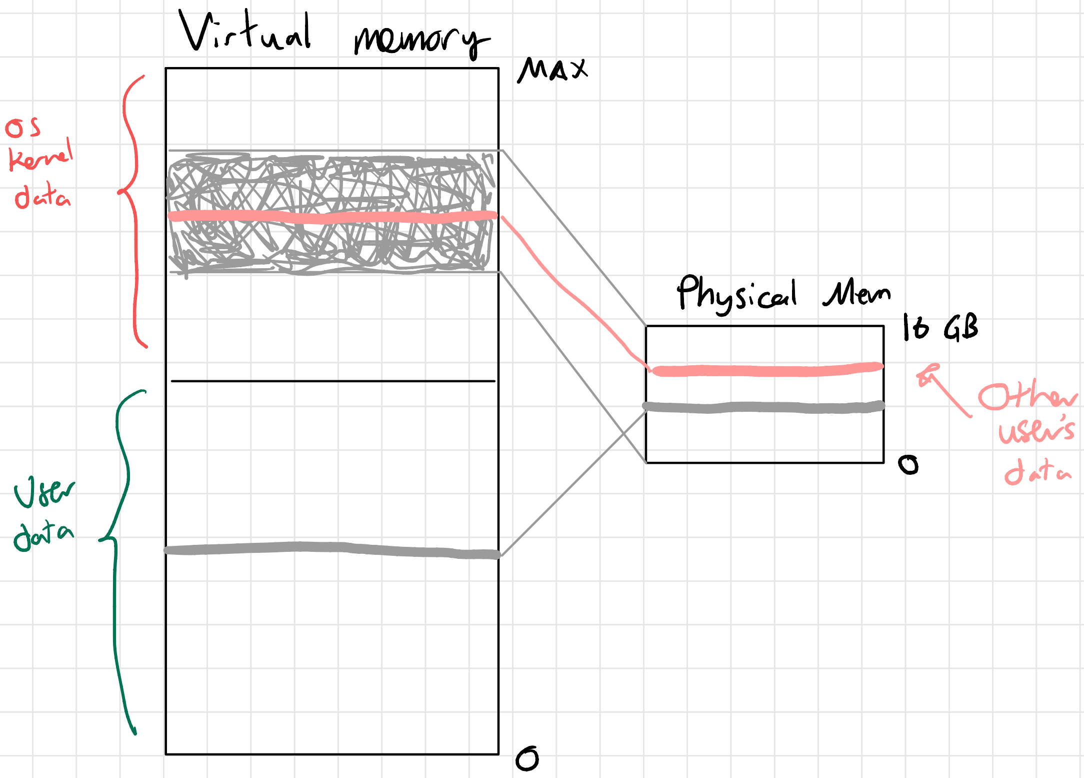
→ This particular attack will no longer work on a modern CPU/OS.

Other related attacks ("Spectre") still do...

Meltdown (Restated)

Goal: Read arbitrary address in memory, bypassing HW permissions checks.

→ All of physical mem is mapped in vaddr space
→ Most of this is not available to user proc
  ↳ HW perm bits
→ Reading arbitrary vaddr is enough to read any location in physical memory?



A useful analogy:

- Go to Dr. LaCurts' favorite cafe, ask "I'll have the same thing Dr. LaCurts usually gets."

- Barista calls Dr LaCurts to ask if he can divulge her usual order.

- While the phone is ringing, he pulls 4 shots of espresso and froths 8 oz of almond milk

- Dr. Lacurts finally answers the phone. She tells the barista to not reveal her secret coffee order.

- Barista won't give you a cofee.

→ The barista leaked the secret info before performing the permissions check.

Step 1: Load kernel data into register.

```
int main() {
    char k = *kernel_addr;
    //print data
}
```

CPU will...

- Load data from memory
- Check permissions bits
- Crash program (exception) if perm check fails

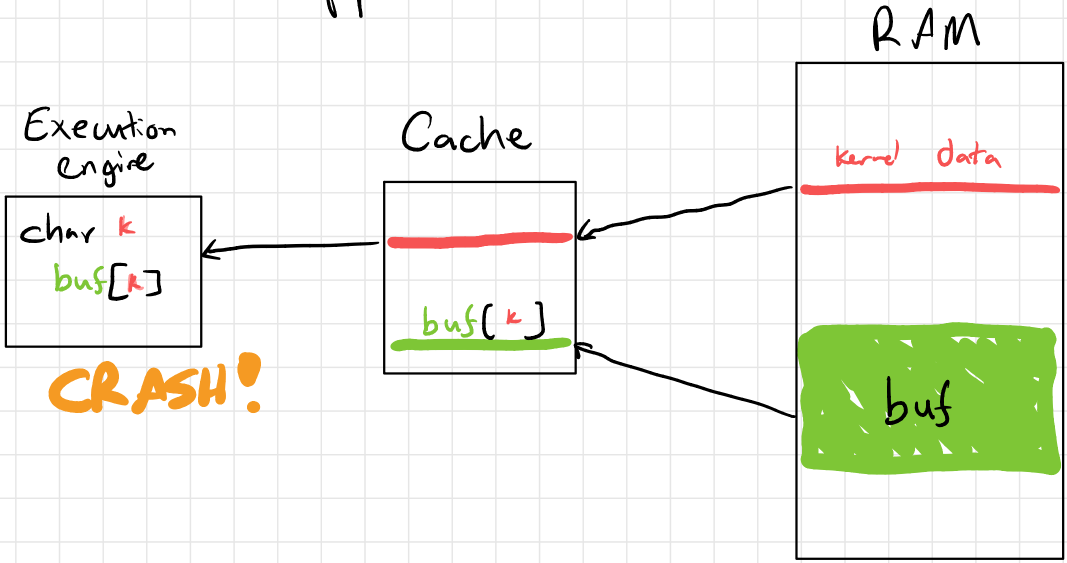
Step 2: Access data in cache based on register contents. [victim data]

```
int main() {
    char buf[4096];
    char k = *kernel_addr;
    char stuff = buf[k];
}
```

CPU will...

- Load data from memory
- Check permissions bits
- Execute next instruction (speculatively)
- Crash program (exception) if perm check fails

What happened here?



The data buf[k] gets loaded into CPU cache.

↳ Then program crashes. (Segfault)

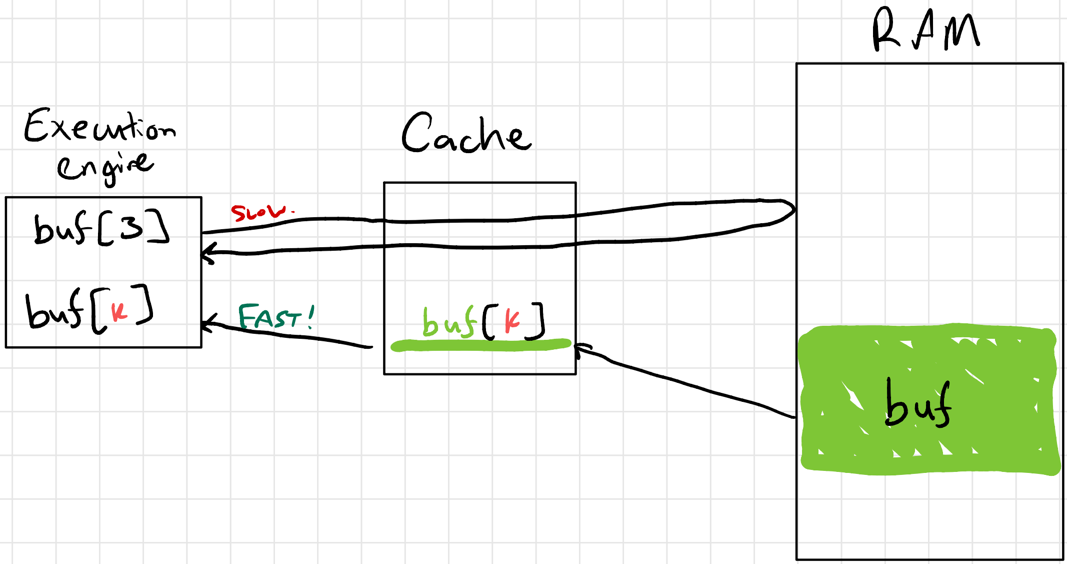
↳ Cache stays as is.

⇒ Learning which element of buf got cached reveals k! ← kernel data.

Key: possible for program to handle the exeption and continue running?

Step 3: Figure out which element of buf the CPU accessed.

* Access to buf[k] → Fast (CACHED!)
* Access to all other parts of buf
  ↳ Slow



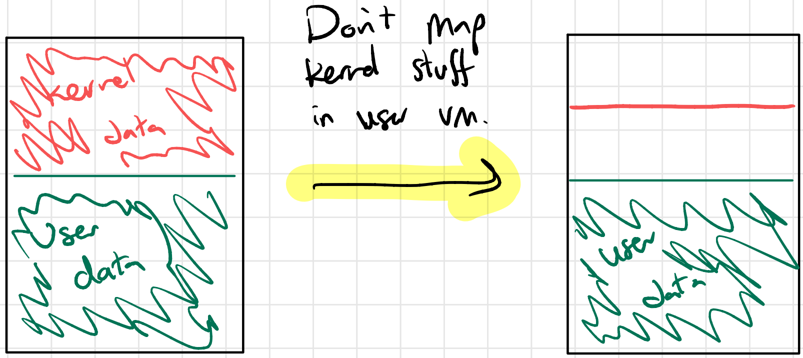
→ 256 possible values of k.
Try them all and time accesses!

Mitigations

→ Can't trust HW to enforce mem perm checks

Software/OS: KAISER/KPTI

Protected by HW permissions



→ HW was too greedy

CPU design: Do not speculate past permissions checks

CPU will...

- Load data from memory
- Check permissions bits
- Crash program (exception) if perm check fails
- Execute next instruction

← Enforced ordering