Lab4 meltdown漏洞实验报告

PB16111096 谢灵江

实验要求

从Meltdown（难度较低）和Spectre（难度较高）的变种中任选一种，仔细阅读对应的论文，并查阅相关资料，了解漏洞的背景、原理，以及漏洞可能的利用方式，并利用漏洞编写一个完整的攻击程序，实现越权内存访问。

本次实验选择了Meltdown漏洞，报告参考的文献主体为meltdown原论文 [1]以及Wenliang Du, Syracuse University的Meltdown Attack Lab文章 [2]。

实验原理

乱序执行技术(out-of-order execution)

CPU在工作时原则上应当按照程序规定的顺序执行其指令。但考虑到效率问题，完全按照原有顺序执行会导致大量的部件空闲，浪费资源。因此，现在的CPU使用了乱序执行技术，即根据各单元电路的空闲状态和各指令能否提前执行的具体情况分析后，CPU将可以提前处理的指令立即发送给相应电路执行的功能 [3]。此举能大大提升工作性能，但同时也引入了问题，即需要避免提前执行指令与顺序执行指令之间的差异，例如当理论上执行序列靠前的指令发生异常或中断时，靠后的指令可能已经进行完毕。此时需要消除其影响，即回复到未执行“不应当执行的指令”的情况下。

高速缓存(cache)

为了在主存的大的存储空间与寄存器的快速读写之间寻找一个平衡点，高速缓存cache产生了。其利用局部性原理，把一部分将会频繁访问内存块放入其中，加速处理器对它们的读写效率。当有未被放入cache的块被调用时，会按照一定的原则放入cache。cache本身的容量很小，但是因为其高速以及可以换入换出块的功能特性，在高命中率下能显著提升性能。

侧信道攻击(side channel attack)与FLUSH+RELOAD技术

对计算机的攻击方式是非常多样的。并非一定是要直接对数据做什么，通过迂回进攻也能达到效果。计算机在整个运行过程中的时间消耗、功率消耗、电磁辐射等信息实际上蕴含着很多含义。利用这些信息可以反推出计算机的处理工作的部分内容，从而达到攻击的目的，此种方式即被称为侧信道攻击。

侧信道攻击类型中，有一个与此次实验密切相关的攻击技术，被称为FLUSH+RELOAD，其主要通过监控cache的运行来达到确定内存调用等信息的情况。

FLUSH+RELOAD技术分为三个阶段：

1.第一阶段，从cache中剔除指定的内存（flush）；

2.第二阶段，允许目标进程访问内存。如果目标访问了指定内存，这个内存空间会被记录在cache中;

3.第三阶段，重新载入内存行，测量重新载入的时间。在cache中有记录的内存的访问速率会远远快于未被访问的内存，由此可以判断出目标进程对内存的使用情况。

Meltdown漏洞

上文已提到，处理器在进行乱序执行时，若遇到意外情况会将提前执行的指令回复至原样。但是许多处理器疏忽了一点，那就是仅回复了寄存器与内存等部件，cache却被保留了下来，而这可以得到极为宽泛的利用。当我们调用一个权限不足以访问的内存空间时（如操作系统内核），从宏观的角度来看，这个操作会触发异常，阻止这一次访问。然而，在实际情况下，可能有几段位于这次异常之后的代码其实已经得到了执行，只是在执行之后被还原了，但仍有信息滞留在了cache中，即能够通过监控访问速度来确定块是否曾被访问过。（这类指令被称为瞬态指令）那么利用这一点，我们只需想办法使数据能用“块是否被访问”这一信息来承载即可。于是，思路就很简单了：越权调用一个数据(此为异常指令)，并使用该数据去调用对应该数据的值的一个块（此为瞬态指令），在异常结束之后确定保留在cache中的为哪一块，便能逆推出数据值。

若该攻击方式生效，则可以越权调出整个内存的所有数据。

实验分析

处理器

在开始实验之前值得一提的是，meltdown的原论文 [1]之中，已经说明了该漏洞在intel的处理器上十分普遍，但是在amd的处理器上进行时虽然能捕捉到乱序处理的情况，但是攻击并不能成功。随后公布的调查和研究表明，amd处理器确实不受到meltdown类的影响，linus也让漏洞修复补丁作出了对amd的放宽：

```
(if (c->x86_vendor != X86_VENDOR_AMD) setup_force_cpu_bug(X86_BUG_CPU_INSECURE);)
```

我的笔记本电脑处理器为：AMD A8-5545M APU with Radeon(tm) HD Graphics × 4 ，因此无法进行meltdown攻击实验，于是借用了同学已配置好的电脑作为环境。

数据相关分析

由原理部分对meltdown的叙述知，一个实现meltdown的思路如下：

越权调用一个数据k，将数据k作为二进制的数字，再次用于调取一个数组array[k]，最后通过比较array不同位的访问速度来确定k的值。

这是最简单最基础的思考，实际上我们还需要进行一些改进。首先cache中的内存是以块为单位存储的，仅简单的拿一个数组地址会导致没有区分度，即array[k]、array[k+1]很可能在同一块内，统一被调入cache，导致访问速率相同。因此可以使用一个一定大于单个块的大小的数组值来确定区分，例如我们将数据k乘以4096,此时array[k * 4096] 、array[(k+1) * 4096]便没有空间局部性问题，调用可以得到显著区分。

同时，由此我们还对k的数据长度n有个初步的限制。每次异常过后，在对cache的操作过程中，需要访问2^n个地址来确定k的值，若过大，则指数级增长的访问时间过于漫长。若n过小，则一次攻击能得到的数据太小，那么前期的异常、瞬态指令等预备工作的处理时间又会占比太大，以效率为出发点，取k为一字节的数据即n=8, k在[0~255]范围内。

另有一个问题，即当k=0时，array[0×4096]作为数组上界可能与其他的一些变量位置邻近，其他变量的调取可能会意外将array[0]调入cache中。为了避免这一点，可以再给数组的调用时设置一个偏移delta，即操作时调入array[k×4096+delta]，检查cache时也作出相应改动。[2]

时间相关分析

rdtsc指令用于read tsc寄存器，即time stamp counter寄存器的值。其延伸出的rdtscp指令利用cpuid使其保序执行，不受指令重排序影响。因此我们利用__rdtscp函数来记录一次调用某位数组操作的前后的两个时间戳，用其差值来表示一次访问消耗的时间。

因为瞬态指令其实是和处理器的异常处理模块竞速的，所以为了提高瞬态指令率先完成的可能性，我们需要一些额外的工作。这一部分在Wenliang Du的meltdown attack lab文章[2]中提到两个处理方式，一是使用预读将我们要进行meltdown的部分放入cache中，这可以提升瞬态指令的执行速率;另一种形式，即增加一段代码来使得处理器的计算模块忙碌，是可以采取的。事实上我们的内存调用并没有用到计算模块，cpu会采用乱序来处理指令，因此增加的计算会有效减缓异常被处理的速度。插入的指令如下，使eax连续进行400次计算。

```
asm volatile(
".rept 400;"
"add $0x141, %%eax;"
".endr;"
:
:
: "eax"
);
```

最后，由于即使是有漏洞的处理器，因其异常处理的时间以及处理时的其他条件不同，也不确定其一定会执行瞬态指令。一次meltdown攻击可能会失败，此时大小比较返回的结果便会出错。为了防止这种错误我们设置一个循环来对单个地址多次攻击，记录其每次攻击的情况，然后输出得到次数最多的返回值k。这样，整个meltdown的攻击流程便大致定下来了。

异常处理

当我们越权调用内存的程序引发出异常之后，一般而言程序会被终止。但是此次实验我们还需要程序继续进行对cache的判断，所以要处理异常。与C++及其他高级语言不同，C语言不提供对错误处理的直接支持，但我们可以模用sigsetjmp()和siglongjmp()来模仿try/catch语句[2]。

sigsetjmp()会保存目前堆栈环境，然后将目前的地址作一个记号，在程序其他地方调用siglongjmp()时便会直接跳到这个记号位置，然后还原堆栈，继续程序的执行。第一次使用sigsetjmp时返回值是0,而通过siglongjmp调用时不再为0,因此利用简单的if语句就可以作出正常情况工作和异常处理模块[4]。SIGSEGV是当进程执行了一个无效的内存引用时会有效的一个信号量，利用它和signal函数我们可以使异常发生时程序自动处理后续部分，具体为：进行越权调用前存储环境，越权调用触发异常后信号量有效，进行siglongjmp回到环境初，然后开始对cache进行观察。

实验过程

环境

内核：Linux 4.13.0-43-generic 操作系统 16.04.1-Ubuntu 体系架构 SMP x86_64 GNU/Linux

补丁情况：打开/etc/default/grub，在GRUB_CMDLINE_LINUX_DEFAULT加入nopti

代码

Wenliang Du, Syracuse University的Meltdown Attack Lab中代码的完成度已经非常高了，在不作出画蛇添足式的增删的情况下能修改的部分很少。不过还是有一些能够调整的地方和做法。

对于MeltdownKernel.c部分，仅作出了对写入部分的修改，让系统将一段secretdata："MeltDown provides"放在内核段，以便实验对其的攻击。

MeltdownAttack.c部分：

CACHE_HIT_THRESHOLD是对访问所需时间进行界定的常量，使用CacheTime.c对系统进行多次测试：

```
xielingjiang@xielingjiang-Inspiron-5435:~/4-Hipugna$ gcc -march=native CacheTime.c -o CacheTime.o -O0
xielingjiang@xielingjiang-Inspiron-5435:~/4-Hipugna$ ./CacheTime.o
Access time for array[0*4096]: 225 CPU cycles
Access time for array[1*4096]: 300 CPU cycles
Access time for array[2*4096]: 296 CPU cycles
Access time for array[3*4096]: 139 CPU cycles
Access time for array[4*4096]: 283 CPU cycles
Access time for array[5*4096]: 302 CPU cycles
Access time for array[6*4096]: 289 CPU cycles
Access time for array[7*4096]: 135 CPU cycles
Access time for array[8*4096]: 306 CPU cycles
Access time for array[9*4096]: 293 CPU cycles
xielingjiang@xielingjiang-Inspiron-5435:~/4-Hipugna$ ./CacheTime.o
Access time for array[0*4096]: 247 CPU cycles
Access time for array[1*4096]: 294 CPU cycles
Access time for array[2*4096]: 303 CPU cycles
Access time for array[3*4096]: 136 CPU cycles
Access time for array[4*4096]: 303 CPU cycles
Access time for array[5*4096]: 301 CPU cycles
Access time for array[6*4096]: 300 CPU cycles
Access time for array[7*4096]: 136 CPU cycles
Access time for array[8*4096]: 314 CPU cycles
Access time for array[9*4096]: 298 CPU cycles
xielingjiang@xielingjiang-Inspiron-5435:~/4-Hipugna$ ./CacheTime.o
Access time for array[0*4096]: 380 CPU cycles
Access time for array[1*4096]: 370 CPU cycles
Access time for array[2*4096]: 283 CPU cycles
Access time for array[3*4096]: 136 CPU cycles
Access time for array[4*4096]: 332 CPU cycles
Access time for array[5*4096]: 335 CPU cycles
Access time for array[6*4096]: 300 CPU cycles
Access time for array[7*4096]: 137 CPU cycles
Access time for array[8*4096]: 270 CPU cycles
```

部分测试结果见上图。其中，所有的位都经历了一次flush，但3和7在flush之后再次调用过。观察发现，cache命中时一般两次rdtscp函数差值（即 访问时间）会在140左右，最高不超过200;而一般情况即flush清除掉且未再次调入时该值一般大于280,且从未低于250。

这是在本人的计算机中运行的结果，更换到同学的计算机环境后测试方法相同，所得结果比上述值更大，但情况类似。命中时从不超过300,而miss时绝不低于300.因此我们将CACHE_HIT_THRESHOLD设为300,若访问块时间低于该值则认为该块被调入了cache中。

偏移量delta理论上只要控制k为0和255时的调入数组位不与数组上下界紧邻即可，可取为2048.

在main函数模块加入循环，对地址及其后20位的每一位实施meltdown攻击，并输出攻击所得的ascii码。地址值不是固定的，是当meltdownkernel模块工作完成后得知secret_data即我们的目标内核部分的具体地址时进行实时更改的。

步骤

1.首先对meltdownkernel部分进行处理。编译内核模块：

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep ’secret data address’
```

情况如下：

```
dingfeng@ding-laptop:~/NewDisk/code/xlj/Meltdown_Attack$ sudo rmmod MeltdownKern
el
dingfeng@ding-laptop:~/NewDisk/code/xlj/Meltdown_Attack$ sudo insmod MeltdownKer
nel.ko
dingfeng@ding-laptop:~/NewDisk/code/xlj/Meltdown_Attack$ dmesg | grep 'secret da
ta address'
[  362.439554] secret data address:ffffffffc0cac000
[ 4363.144664] secret data address:ffffffffc0cbf000
dingfeng@ding-laptop:~/NewDisk/code/xlj/Meltdown_Attack$ gcc -march=native Meltd
```

其中第一个输出是我在研究原代码时进行的操作放入的结果，第二个输出为修改后自己的代码放入的数据。因此我们取第二个输出的地址，放入MeltdownAttack代码的对应段。

2.更改后编译运行MeltdownAttack.c：

```
$ gcc -march=native MeltdownAttack.c -o MeltdownAttack.o -O0
$ ./MeltdownAttack.o
```

运行情况：

```
dingfeng@ding-laptop:~/NewDisk/code/xlj/Meltdown_Attack$ ./MeltdownAttack.o
character: M
character: e
character: l
character: t
character: D
character: o
character: w
character: n
character:
character: p
character: r
character: o
character: v
character: i
character: d
character: e
character: s
character:
character:
character:

Work successfully!
```

与预期输出一致，且通过简陋代码对该地址的直接访问被拒绝，可以认为是成功复现了简易的meltdown攻击。

具体实现请参见代码。

参考文献

[1] Meltdown Moritz Lipp , Michael Schwarz, Daniel Gruss, Thomas Prescher , Werner Haas ,Stefan Mangard, Paul Kocher, Daniel Genkin , Yuval Yarom, Mike Hamburg,Graz University of Technology Cyberus Technology GmbH Independent University of Pennsylvania and University of Maryland University of Adelaide and Data Rambus, Cryptography Research Division

[2] Meltdown Attack Lab 2018 Wenliang Du, Syracuse University.

[3]乱序执行 百度百科, https://baike.baidu.com/item/%E4%B9%B1%E5%BA%8F%E6%89%A7%E8%A1%8C/4944129

[4]sigsetjmp用法 泞途 - 泊客网 ningto.com, https://blog.csdn.net/tujiaw/article/details/7230990