张宇杰 2022113573 2203101

一、事件背景

2018年1月3日，Google Project Zero团队公开了熔断(Meltdown)和幽灵(Spectre)这两个芯片漏洞。该漏洞能够影响几乎所有的Intel CPU和部分的ARM CPU，受影响的操作系统平台有Windows、Linux、Android、IOS以及Mac OS等。

本次事件涉及到如下三个漏洞：

1、Spectre variant1 CVE-2017-5753：Bounds Check Bypass

2、Spectre variant2 CVE-2017-5715：Branch Target Injection

3、Meltdown CVE-2017-5754：Rogue Data Cache Load

二、Spectre漏洞v1 —— Bounds Check Bypass

Spectre攻击的v1变种可以用于绕过内存访问的边界检查。

由于CPU的预测执行，在存在条件分支的场景中，可能会在条件分支判断操作执行之前(或者同时)执行(或预取)分支后的数据(可能是未授权的内核隐私数据)，预取的数据会在CPU Cache中存在一段时间。此时，恶意的用户态程序就可能通过侧信道攻击推算出Cache中的数据，导致关键数据的泄露。

可以被Bounds Check Bypass攻击的代码通常具有以下特征：它们通常操作了可被攻击者控制或影响的数据，并且有着能被攻击者观察到的某种副作用，为侧信道攻击创造机会。代码需要包含两个成分：一是根据不信任的值对数组或指针进行解引用，二是将一中解引用得到的值用于加载或存储一个地址。为了攻击这样的代码，这个“不信任的值”需要在攻击者的掌控之下。

下面以[4]中的代码为例，分析Spectre v1漏洞的原理。

1. **if** (user\_value >= 0 && user\_value < LIMIT) {
2. x = table[user\_value];
3. node = entry[x];
4. } **else**
5. **return** ERROR;

上述代码中，table中处于LIMIT之内的数据是允许访问的，而处于LIMIT之外的数据是攻击者不可见的、需要获得的，entry是攻击者可以控制的数据。当上述代码执行时，在判断user\_value < LIMIT这个条件是否成立之前，由于分支预测的存在，CPU将预先执行if中的语句。由于在执行if中的语句时，CPU访问了数据table[user\_value]与entry[x]。在CPU访问entry[x]时，就有可能将其加载到L1 Cache中。一旦CPU执行分支判断后发现分支预测错误，将丢弃先前的计算结果，但缓存在L1 Cache中的数据却不会进行回滚。这样之后，攻击者通过遍历entry数组，并测量其中每个元素的访问时间，访问最快的就是先前被缓存在L1 Cache中的entry[x]，进而推断出对应的x值。如此反复，table中处于LIMIT之外的数据就可以通过攻击进行窥探。

三、Spectre漏洞v2 —— Branch Target Injection

Spectre攻击的v2变种可以通过分支预测注入(对CPU分支预测机制的干扰)来获取数据。

该种攻击通过欺骗CPU的间接分支预测器的预测结果，来指导CPU在接近间接分支跳转指令之后应该执行哪些操作。通过控制间接分支预测器的操作方式，攻击者可以诱导执行某些指令“gadget”的执行，之后通过侧信道攻击窃取预测执行时预取到CPU Cahce中的数据。

对于带有条件的直接跳转语句，分支预测器在进行预测时，只有两种选择：跳转的目标(即跳转条件为真时)或者跳转语句的下一条指令(跳转条件为假)，攻击者无法让控制转移到这些位置之外。然而，间接分支跳转可能导致控制能被转移到更大范围的位置上。Branch Target Injection的工作原理是，通过分支预测注入，导致间接分支跳转在分支预测时“总是”执行特定的指令“gadget”，而这些特定的指令为侧信道攻击提供了可能性。

CPU在执行以下指令时才会使用间接分支预测器：

|  |  |  |
| --- | --- | --- |
| 分支类型 | 指令 | 操作码(机器码) |
| 间接近调用(Near call indirect) | CALL r/m16, CALL r/m32, CALL r/m64 | FF /2 |
| 间接近跳转(Near jump indirect) | JMP r/m16, JMP r/m32, JMP r/m64 | FF /4 |
| 近返回(Near return) | RET, RET Imm16 | C3, C2 lw |

表1

下面通过描述[6]中的一个实际的攻击过程，来更好的阐明Spectre v2漏洞的原理。

在这个例子中，victim.c接收外界传来的字符串，通过sscanf获取字符串中的整数%ld，再通过sprintf将该整数格式化为字符串%ld后输出至外界。

sscanf(buf, "%ld\n", &a);

sprintf(send, "%ld\n", a);

victim.c将作为victim进程独立运行。victim.c中存放着一个外界不可见的secret字符串，需要我们攻击获取。

攻击者使用attack.c通过套接字与victim进程进行简单通信，向其发送字符串并接受传回的字符串。

为了实现攻击，两个进程之间需要共享一片内存区域。这个区域可以建立在系统的共享库中，这里原作者为了清晰讲解攻击原理，直接在victim的可执行文件的.rodata段插入了一个64KB的ProbeTable数组，并通过mmap将正在运行的victim的ProbeTable映射到attack进程。通过之后的分支预测注入，victim将把secret中的字符映射到ProbeTable中，并加载到Cache中。此时attack进程即可通过侧信道攻击的方式探测出ProbeTable中被victim映射的secret字符，进而还原出整个secret字符串，完成跨进程攻击。具体实现见下。

攻击者分析了victim.c的反汇编语句，找到了语句mov -0x88(%rbp),%rdx，在这里%rdx作为sprintf的第三个参数。在先前-0x88(%rbp)的地址作为了sscanf的第三个参数，-0x88(%rbp)存放了攻击者传入的整数，这是攻击者可以控制的。从而，通过分析，攻击者发现可以对寄存器%rdx的值进行控制。以下为victim.c的部分反汇编语句：

1. 400c03: 48 8d 95 78 ff ff ff     lea    -0x88(%rbp),%rdx     #变量a的地址
2. 400c0a: 48 89 ce                 mov    %rcx,%rsi            #格式化字符串"%ld\n"的地址
3. 400c0d: 48 89 c7                 mov    %rax,%rdi            #buf的地址
4. 400c10: b8 00 00 00 00           mov    $0x0,%eax
5. 400c15: e8 f6 fb ff ff           callq  400810 <sscanf@plt>
6. 400c1a: 48 8b 95 78 ff ff ff     mov    -0x88(%rbp),%rdx     #攻击者传入的整数被装入rdx，并作为sprintf的第三参数
7. 400c21: b9 51 00 41 00           mov    $0x410051,%ecx
8. 400c26: 48 8d 45 80             lea    -0x80(%rbp),%rax
9. 400c2a: 48 89 ce                 mov    %rcx,%rsi            #send的地址
10. 400c2d: 48 89 c7                 mov    %rax,%rdi            #格式化字符串"%ld\n"的地址
11. 400c30: b8 00 00 00 00           mov    $0x0,%eax
12. 400c35: e8 c6 fb ff ff           callq  400800 <sprintf@plt>

此外，攻击者还发现了secret字符串的地址0x630480，但由于虚拟内存的独立性，攻击者并不能直接通过该虚拟地址获取secret的信息。

为了达到目的，攻击者在victim中编写了找到了一段gadget代码(简化操作，实际上应该在动态库中寻找gadget)，该段代码把攻击者控制的%rdx所指向的值作为下标，访问处于共享物理内存区域中的ProbeTable的数据。gadget的大致代码见下：

1. xorl %eax, %eax
2. movb (%rdx), %ah
3. movl ProbeTable(%eax), %eax
4. retq

在编写完gadget之后，为了完成分支预测注入，需要包含表1中提到的指令才能利用间接分支预测器进行攻击。而victim.c中调用了外部函数sprintf，链接器将使用延迟绑定的动态链接技术对sprintf进行动态链接。在实现动态链接的汇编代码中，有一行jmpq \*0x22fbda(%rip) # 6303e0 <sprintf@GLIBC\_2.2.5>，正好是表1中提到的指令。

1. 0000000000400800 <sprintf@plt>:
2. 400800: ff 25 da fb 22 00    jmpq   \*0x22fbda(%rip)     # 6303e0 <sprintf@GLIBC\_2.2.5>
3. 400806: 68 06 00 00 00       pushq  $0x6
4. 40080b: e9 80 ff ff ff       jmpq   400790 <.plt>

这行中提到的0x6303e0是sprintf的GOT表项地址，里面存放了sprintf的真实地址。攻击者通过将来提到的evictor进程，将0x6303e0对应的物理页面排出Cache缓存，导致victim在取得sprintf地址的过程中出现了时间窗口。在该时间窗口之中，由于将来提到的trainer进程的作用，间接分支预测器在jmpq指令给出的预测地址被毒化(“调教”)为gadget的地址，从而CPU预测执行gadget将secret的字符信息泄漏到共享区域中。

攻击者使用了train.c对间接分支预测器进行“训练”。train.c创造了两个进程，trainer进程与evictor进程。

trainer进程中使用execv函数加载victim，并利用ptrace代码注入的手段在sprintf@got (0x6303e0)的位置处填入gadget的地址。由于trainer模拟了正在运行的victim进程。当victim执行sprintf函数时，入口处的间接跳转指令被毒化，从而使CPU预测执行gadget。

evictor进程用于不停地刷新sprintf@got，使得jmpq指令执行时产生Cache不命中，进而引发间接分支预测器的工作。

最后，attack进程向victim进程循环发送secret的地址0x630480，触发victim执行gadget，然后对ProbeTable逐个探测命中情况，ProbeTable中命中的那个单元的下标即为地址0x630480对应的secret中的字符。

下图展现了各进程之间的关系：

图示

描述已自动生成

四、Meltdown漏洞 —— Rogue Data Cache Load

Meltdown攻击通过用户态应用程序直接访问内核中的私有数据。

正常情况下，用户态应用程序访问内核时会触发无效缺页错误(Invalid Page Fault)。由于权限检查的存在，工作在第3环的程序显然不能直接访问第0环中的页面。但是，CPU实际上是乱序执行的，意味着访问数据的操作可能在权限检查之前执行。由于权限检查是相对比较耗时的操作，而数据读取操作通常很快。在这种情况下，数据读取和权限检查二者实际执行的时间差可能会达到数百个CPU周期。这给攻击者通过侧信道攻击留下了比较充分的时间。

利用Meltdown漏洞，恶意的用户态程序可以突破用户态和内核态边界，窃取内核隐私数据。该漏洞只能窃取在页表中已经映射的内核数据，不能窃取未映射的数据。

下面通过[8]中出现的简单例子，阐明Meltdown攻击的原理。

考虑以下操作序列

1. clflush **for** user\_probe[]; // 把user\_probe\_addr对应的cache全部都flush掉
2. u8 index = \*(u8 \*)attacked\_mem\_addr; // attacked\_mem\_addr存放被攻击的地址
3. data = user\_probe\_addr[index \* 4096]; // user\_probe\_addr存放攻击者可以放访问的基地址

如果attached\_mem\_addr存放的地址位于内核, 我们就可以利用它解引用来读取内核空间的内容。

假如CPU顺序执行, 在第2行就会发现它访问了一个无权限的地址, 产生Invalid Page Fault, 进而被内核捕获, 第3行就没有机会运行。不幸的是, 由于CPU的乱序执行, 在某些条件满足的情况下, 它取得了attacked \_mem\_addr里的值, 并在CPU将该指令标记为异常之前将它传递给了下一条指令, 并且随后的指令利用它来触发了内存访问。在指令提交的阶段，CPU发现了异常，再将先前乱序执行指令的结果丢弃掉。这样虽然没有对指令的正确性造成影响, 但是乱序执行时被缓存在Cache中的user\_probe\_addr[]却不会被回滚。这样，之后通过对user\_probe\_addr[]进行遍历探测，比较访问时间，即可得知先前访问user\_probe\_addr[]的下标，进而得知index的值，从而得知了内核空间中的内容。

需要指出的是，attached\_mem\_addr指向的内容必须要已经被缓存到了L1 Cache中，这样CPU才有可能将指令标记为异常之前指数据传给后续的指令。

五、参考文献

1. 漏洞惊魂，服务器出现问题：熔断（Meltdown）和幽灵（Spectre）

<https://zhuanlan.zhihu.com/p/33488943>

1. CPU漏洞-Spectre&Meltdown原理分析及修复原理

<https://happyseeker.github.io/kernel/2018/02/08/what-are-Spectre-Meltdown.html>

1. Meltdown & Spectre 攻击及缓解措施（一）

<https://paper.seebug.org/501/>

1. Analyzing Potential Bounds Check Bypass Vulnerabilities

<https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analyzing-bounds-check-bypass-vulnerabilities.html>

1. Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088

<https://www.intel.cn/content/www/cn/zh/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html>

1. [原创]深入Spectre V2——跨进程泄露敏感信息

<https://bbs.kanxue.com/thread-254288.htm>

1. Rogue Data Cache Load / CVE-2017-5754 / INTEL-SA-00088

<https://www.intel.cn/content/www/cn/zh/developer/articles/technical/software-security-guidance/advisory-guidance/rogue-data-cache-load.html>

1. 首发：Meltdown漏洞分析与实践

<https://zhuanlan.zhihu.com/p/405558470>