OS HW4 0711529 陳冠儒

**• Simply describe how Meltdown exploits ooo execution/Speculative Execution/Flush+Reload to launch attacks**

1. 亂序執行(ooo execution, out-of-order execution)  
   在亂序執行中，CPU會依據可用性來決定執行順序，而不是原始程式資料的順序決定，因此可以避免原本因下一條指令所引起的等待，取而代之的可以處理下條可立即執行的指令。  
   e.g.  
   1. b = a + 3  
   2. v = \*b  
   3. c = a \* 5  
   1與3可以並行執行，而2之b需要等1，故可先計算1和3再執行2。
2. 推測執行(Speculative Execution)  
   在CPU資源過剩的情況下，會提前去執行一些以後可能用得上也可能用不上的指令，如果執行完發現用不上就會rollback將得到的結果拋棄。  
   e.g.  
   1. 傳送A需要花1分鐘  
   2. 如果A重送成功，就給出A的結果  
   3. 如果A不成功，就傳送B也需要花1分鐘，並給出B的結果。  
   如果沒有推測執行，那當A傳送後發現失敗，就要再花1分鐘去傳B，但如果傳送A過程中CPU資源是剩餘的情況下，就可以也一起傳送B，如果A失敗就不用多花時間去傳，如果成功再把B拋棄掉就可。

OOO Execution和Speculative Execution的差別只在於Speculative執行的並不一定會被執行到，而OOO Execution是一定會執行的，相同處就是他們會先將程式拆解成很多小部分，將沒有互相影響的，就先一起執行，加速處理的速度。

而因為這些先被執行的就會被存到cache裡面等待register去執行，但是CPU緩存的指令是不用經過安全、權限檢查的，只有加載到register後才會去檢查合法性，故這就是meltdown的漏洞所在，我們可以利用下面提到的Flush & Reload藉此去取得kernel資料。

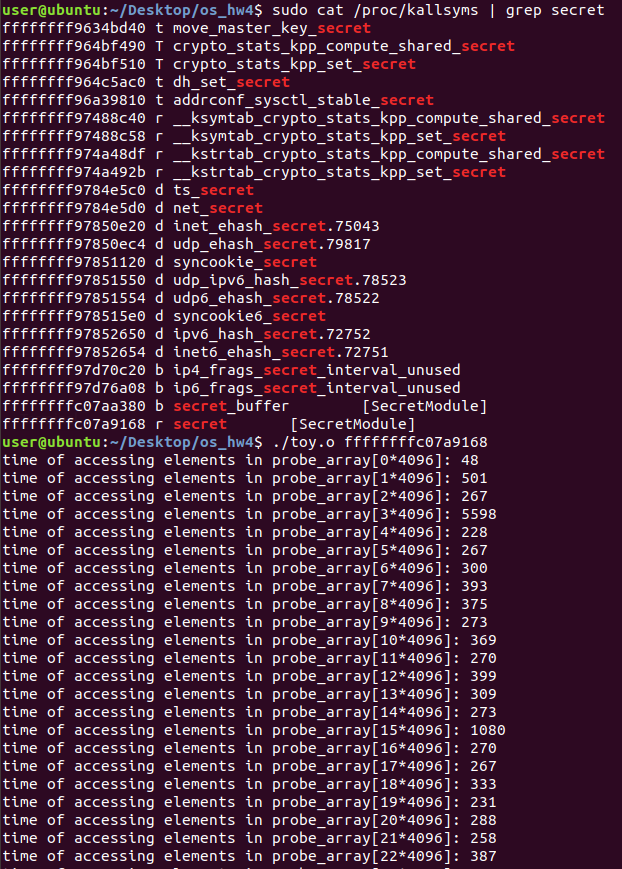
1. Flush & Reload  
   假設攻擊者和目標程序shared memory，攻擊者可以反複利用CPU指令將監控的內存塊（某些地址）從cache中驅逐出去，然後再等待目標程序訪問shared memory（Flush階段）。然後攻擊者重新加載監控的內存塊並測量讀取時間(Reload階段)，如果該內存塊被目標程序訪問過，其對應的內存會被導入到CPU cache中，則攻擊者對該cache的訪問時間將會較短。通過測量加載時間的長短，攻擊者可以清楚地知道該內存塊是否被目標程序讀取過。

Meltdown與Spectre利用這種Cache Side-Channel Attack可以進行越權內存訪問，甚至讀取整個kernel的內存數據。

Reference:

* <https://blog.gslin.org/archives/2018/01/05/8028/meltdown-%E8%88%87-spectre-%E9%83%BD%E6%9C%89%E7%94%A8%E5%88%B0%E7%9A%84-flushreload>
* <https://zhuanlan.zhihu.com/p/32848483>
* <https://www.freebuf.com/articles/system/159811.html>

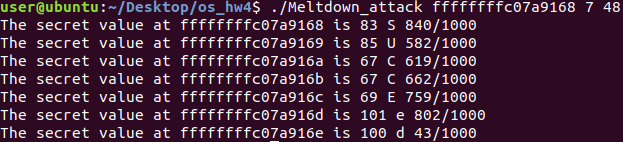
**• Task 1 (what you observed and explain it)**



從上圖結果來說，我們可以觀察到幾個地方：

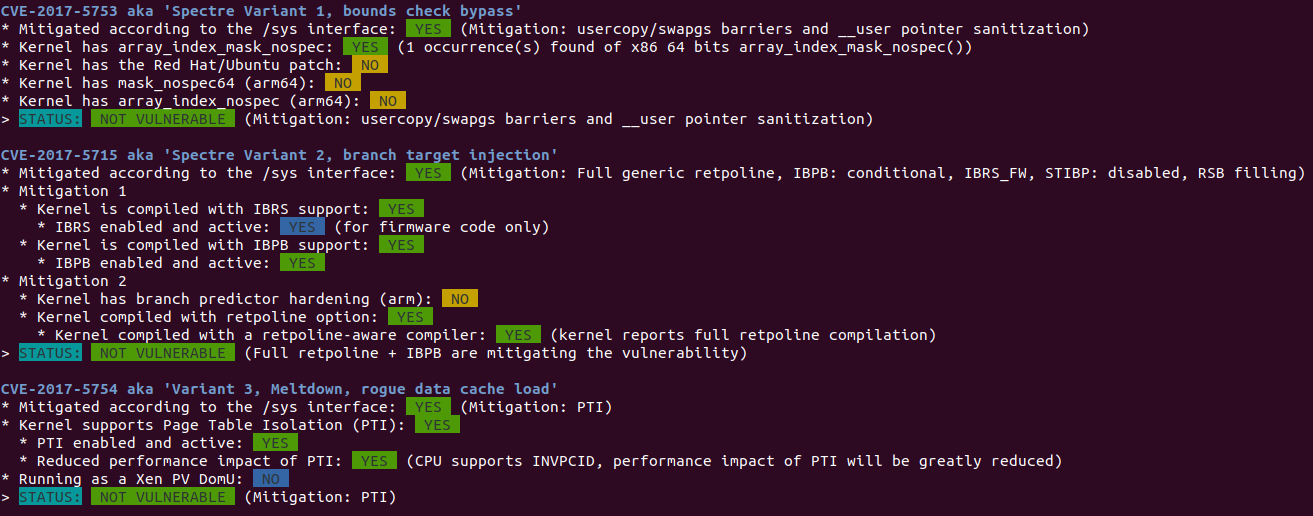
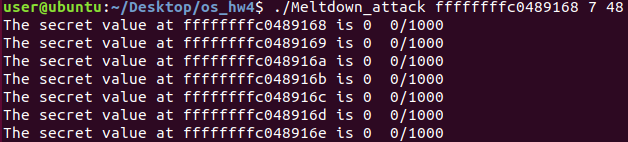
1. probe\_array[al\*4096]這裡的al變數會從0到255，因為一個字節的取值範圍為0-255，共256個數，而4096是x86架構中一個page的大小(4KB)，故這裡的probe\_array數組填充了256個page。
2. 第一個page(probe\_array[0\*4096])所需時間每次都會是最小的，我覺得是因為，他裡面有數據存進cache中了，所以每次都只要access from cache，而其他page因為還沒有數據存進，故要access from kernel時間會花得比較長。所以我們得知access from cache的時間後，就可以利用他來設置我們的threshold，threshold設的比他大一點點即可，如上48的話大概設置為5~60左右即可

**• Task 2 (the result you get)**



由結果可以看到，我們可以從user privilege取得在kernel中的資料。

**• Task 3 (compare the results before and after patch and explain pti mitigation for Ubuntu)**

由上面的結果顯示，用了patch後，就不會有meltdown的問題了。故所以此時就不會有access from cache的情況發生。

PTI又稱為KPTI(kernel page-table isolation, 核心頁表隔離)，他可以讓現在user space和kernel space被分成兩套，各自使用一個。

在當前的kernel中，每個CPU有一個單個的 PGD；在 KPTI系列補丁中所採取的第一步的其中一個措施是，去建立一個第二個 PGD。當kernel執行時，原來的仍然在使用；它對映所有的地址空間。當CPU執行在user space時，（在打完該系列補丁之後）第二個被啟用。它指向屬於該程序的頁面的相同目錄層次，但是，描述kernel space（位於虛擬地址空間的頂端）的部分通常都不在這裏。

但他的缺點就是會有執行的成本，估計是5%，不過如果是沒有問題的CPU的話，可以使用「nopti」指令去禁用他。

**• What do you learn from this homework?/Conclusion**

我覺得這是個很好的功課，藉由上課內容的延伸，讓我們更加了解到為了加速程式執行，會引起的一些memory和cache存取上的問題，雖然有些東西上課時沒有說過，但是在查資料的過程中，因為基本觀念都有說過了，所以花些時間就能看懂並吸收，進而更完整的了解課程的內容。

OOO Execution和Speculative Execution其實是很簡單的概念，學完了mutli-processing後不難理解他的做法，但是提早被執行的結果他是怎麼存放的、他會引發什麼問題，這些都是我之前沒有想過的，在看完meltdown後，我更理解了memory stall跟processor scheduling等觀念。

而最後他的patch原理是利用pti，這部分又用到了page table的觀念，page table的管理方式有Page Global Directory(PGD)、Page Upper Directory(PUD)、Page Middle Directory(PMD)等等又是個很複雜的結構，真的覺得上完OS後真的只學到了概論，其實還有很多很多更深入的東西沒有學到，這份的作業也算是開啟了我們自學OS的一扇大門。