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。

(3) 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)**

```
user@ubuntu:~/Desktop/os_hw4$ sudo cat /proc/kallsyms | grep secret
ffffffff9634bd40 t move_master_key_secret
ffffffff964bf490 T crypto_stats_kpp_compute_shared_secret
ffffffff964bf510 T crypto_stats_kpp_set_secret
ffffffff964c5ac0 t dh_set_secret
ffffffff96a39810 t addrconf_sysctl_stable_secret
ffffffff97488c40 r __ksymtab_crypto_stats_kpp_compute_shared_secret
ffffffff97488c58 r __ksymtab_crypto_stats_kpp_set_secret
ffffffff974a48df r __kstrtab_crypto_stats_kpp_compute_shared_secret
ffffffff974a492b r __kstrtab_crypto_stats_kpp_set_secret
ffffffff9784e5c0 d ts_secret
ffffffff9784e5d0 d net_secret
ffffffff97850e20 d inet_ehash_secret.75043
ffffffff97850ec4 d udp_ehash_secret.79817
ffffffff97851120 d syncookie_secret
ffffffff97851550 d udp_ipv6_hash_secret.78523
ffffffff97851554 d udp6_ehash_secret.78522
ffffffff978515e0 d syncookie6_secret
ffffffff97852650 d ipv6_hash_secret.72752
ffffffff97852654 d inet6_ehash_secret.72751
ffffffff97d70c20 b ip4_frags_secret_interval_unused
ffffffff97d76a08 b ip6_frags_secret_interval_unused
ffffffffc07aa380 b secret_buffer        [SecretModule]
ffffffffc07a9168 r secret       [SecretModule]
user@ubuntu:~/Desktop/os_hw4$ ./toy.o ffffffffc07a9168
time of accessing elements in probe_array[0*4096]: 48
time of accessing elements in probe_array[1*4096]: 501
time of accessing elements in probe_array[2*4096]: 267
time of accessing elements in probe_array[3*4096]: 5598
time of accessing elements in probe_array[4*4096]: 228
time of accessing elements in probe_array[5*4096]: 267
time of accessing elements in probe_array[6*4096]: 300
time of accessing elements in probe_array[7*4096]: 393
time of accessing elements in probe_array[8*4096]: 375
time of accessing elements in probe_array[9*4096]: 273
time of accessing elements in probe_array[10*4096]: 369
time of accessing elements in probe_array[11*4096]: 270
time of accessing elements in probe_array[12*4096]: 399
time of accessing elements in probe_array[13*4096]: 309
time of accessing elements in probe_array[14*4096]: 273
time of accessing elements in probe_array[15*4096]: 1080
time of accessing elements in probe_array[16*4096]: 270
time of accessing elements in probe_array[17*4096]: 267
time of accessing elements in probe_array[18*4096]: 333
time of accessing elements in probe_array[19*4096]: 231
time of accessing elements in probe_array[20*4096]: 288
time of accessing elements in probe_array[21*4096]: 258
time of accessing elements in probe_array[22*4096]: 387
```

從上圖結果來說，我們可以觀察到幾個地方：

1. probe_array[a1*4096]這裡的 a1 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)

```
user@ubuntu:~/Desktop/os_hw4$ ./Meltdown_attack ffffffffc07a9168 7 48
The secret value at ffffffffc07a9168 is 83 S 840/1000
The secret value at ffffffffc07a9169 is 85 U 582/1000
The secret value at ffffffffc07a916a is 67 C 619/1000
The secret value at ffffffffc07a916b is 67 C 662/1000
The secret value at ffffffffc07a916c is 69 E 759/1000
The secret value at ffffffffc07a916d is 101 e 802/1000
The secret value at ffffffffc07a916e is 100 d 43/1000
```

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)

```
CVE-2017-5753 aka 'Spectre Variant 1, bounds check bypass'
* Mitigated according to the /sys interface: YES (Mitigation: usercopy/swapgs barriers and __user pointer sanitization)
* Kernel has array_index_mask_nospec: YES (1 occurrence(s) found of x86 64 bits array_index_mask_nospec())
* Kernel has the Red Hat/Ubuntu patch: NO
* Kernel has mask_nospec64 (arm64): NO
* Kernel has array_index_nospec (arm64): NO
> STATUS: NOT VULNERABLE (Mitigation: usercopy/swapgs barriers and __user pointer sanitization)

CVE-2017-5715 aka 'Spectre Variant 2, branch target injection'
* Mitigated according to the /sys interface: YES (Mitigation: Full generic retpoline, IBPB: conditional, IBRS_FW, STIBP: disabled, RSB filling)
* Mitigation 1
  * Kernel is compiled with IBRS support: YES
    * IBRS enabled and active: YES (for firmware code only)
  * Kernel is compiled with IBPB support: YES
    * IBPB enabled and active: YES
* Mitigation 2
  * Kernel has branch predictor hardening (arm): NO
  * Kernel compiled with retpoline option: YES
    * Kernel compiled with a retpoline-aware compiler: YES (kernel reports full retpoline compilation)
> STATUS: NOT VULNERABLE (Full retpoline + IBPB are mitigating the vulnerability)

CVE-2017-5754 aka 'Variant 3, Meltdown, rogue data cache load'
* Mitigated according to the /sys interface: YES (Mitigation: PTI)
* Kernel supports Page Table Isolation (PTI): YES
  * PTI enabled and active: YES
  * Reduced performance impact of PTI: YES (CPU supports INVPCID, performance impact of PTI will be greatly reduced)
* Running as a Xen PV DomU: NO
> STATUS: NOT VULNERABLE (Mitigation: PTI)
```

```
user@ubuntu:~/Desktop/os_hw4$ ./Meltdown_attack ffffffffc0489168 7 48
The secret value at ffffffffc0489168 is 0  0/1000
The secret value at ffffffffc0489169 is 0  0/1000
The secret value at ffffffffc048916a is 0  0/1000
The secret value at ffffffffc048916b is 0  0/1000
The secret value at ffffffffc048916c is 0  0/1000
The secret value at ffffffffc048916d is 0  0/1000
The secret value at ffffffffc048916e is 0  0/1000
```

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。