

****

|  |  |
| --- | --- |
| 课程名称： | 信息系统安全 |
| 实验名称： | Meltdown Attack |
| 姓 名： | 王晨皓、谢华洋、宗威旭 |
| 学 号： | 3180103804、3180103576、3180102776 |

2021年 6月 7日

Lab 5：Meltdown Attack

# Purpose and Content实验目的与内容

Purpose

* understand the principle of Meltdown Attack
* practice the Meltdown attack and learn method to improve Meltdown attack success rate

Content

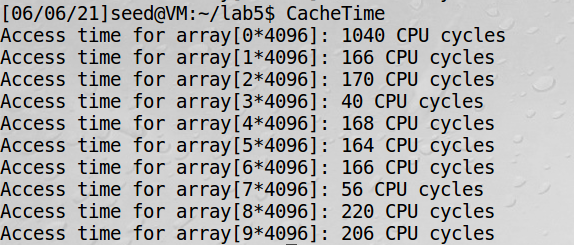
Meltdown Attack allow a user-level program to read data stored inside the kernel memory. It works for Many modern CPUs in 2018 include Intel, and it’s still worth learning during we learn security system. In this lab we understand its principle and launch Meltdown attack step by step, include understand CPU caching, side channel attack, Out-of-order execution inside CPU microarchitecture, and so on.

# Detailed Steps 实验过程

Task 1: Reading from Cache versus from Memory

This step helps us understand the speed difference between Main Memory and Cache when access memory. More importantly, we need to find a threshold to distinguish these two type of memory access. We use this threshold in the following step to locate the target cache block.

Then I run program the CacheTime about twenty times. It turns out that accessing cache memory is very fast compared to accessing main memory. The access time for cache memory ranges from 40 to 60 CPU cycles. So in the following step I still set threshold as 80 CPU cycles.



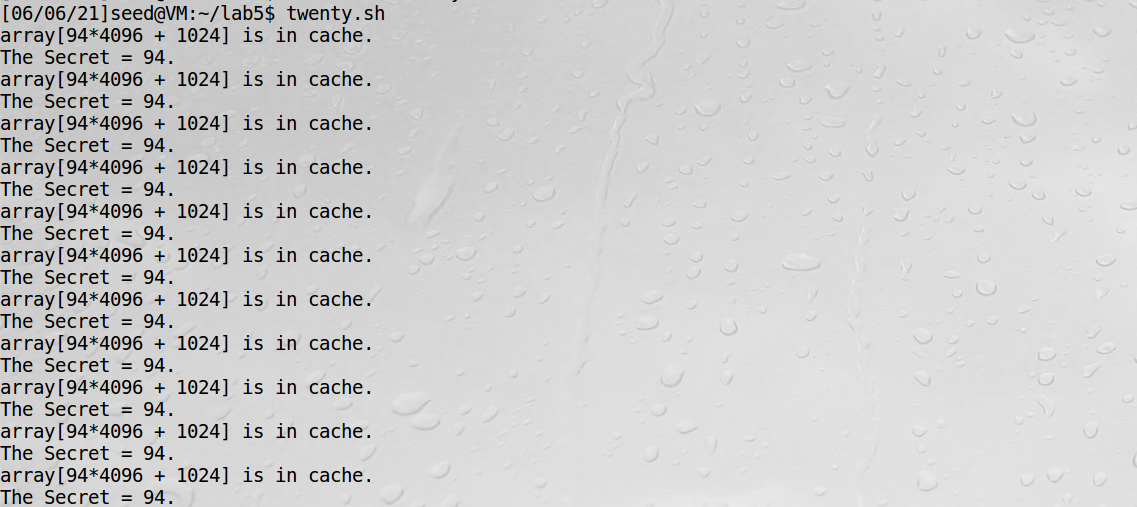
Task 2: Launching the Race Condition Attack

In this step we try to find a secret value used by a victim function which can’t be accessed from the outside. Based on the knowledge in step 1, we can find the secret value by distinguish different memory access time. So all we need to do is map each value to an array element which will be in different cache block. Here we use **side channel** to locate the target cache block:

* FLUSH: clear out the cache to ensure that the array is not index.
* execute victim function which cached a corresponding array element.
* RELOAD: reload the entire array and specific the array which has access time shorter than threshold

As there are 256 possible values for one-byte secret, we need to define 256 blocks’ element and in order to avoid different array mapping into the same cache block, we set elements’ size as 4096 bytes and DELTA as constant 1024.

Then run FlushReload for at least 20 times. Here I set threshold as 80. As the speed difference between main momory access and cache access is large enough, we can see that the program success to find out the correct secret value 20 out of 20 times.

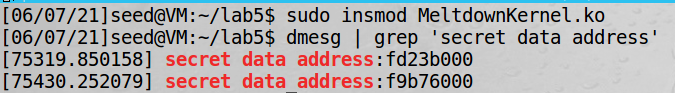


Task 3: Place Secret Data in Kernel Space

In this step we prepared a kernel module to write a secret data into kernel memory. Usually we cannot access data in kernel space from user space as to Linux kernel’s protection. So we need to perform Meltdown Attack to steal this secret data. To make our attack success two condition must be satisfied:

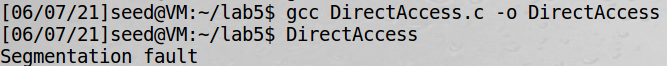
* The secret data’s memory address must be known (usually guess).
* The secret data need to be cached or the attack’s success rate will be low.

Then compile and install the kernel module MeltdownKernel given by seed project. Then we find the target data address in the kernel module log. In the next figure I carelessly installed this module twice so the last address is actually we need.



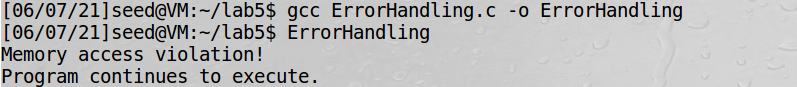
Task 4: Access Kernel Memory from User Space

In this step we try to access the target address (located in kernel memory) from user space. In the following figure we can figure out that it’s an illegal memory access. System throws out an error “Segmentation fault” because each user’s memory space are located by base address and offset. If the memory address it want to access doesn’t fall in the legal range in base address and base address plus offset, it throws out segmentation fault.



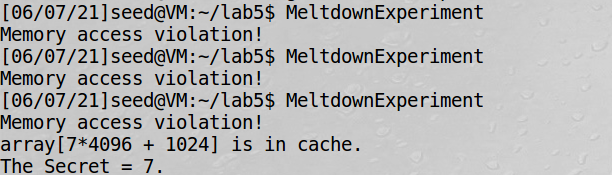
Task 5: Handle Error/Exceptions in C

In this step we try to handle exceptions caused by illegal memory access. When exception raised program sends a signal SIGSEGV and os will handle it and terminate our program. So we need to register a signal handler by ourselves and then continue to run our program. We use **signal** to register signal handler. Then we use **sigsetjmp** and **siglongjmp** to save/return the program status (stack context/environment). In the following figure we can see that the exception raised and it was successfully caught and handled by our handler and the program continues to run.



Task 6: Out-of-Order Execution by CPU

In this step we understand the key principle of Meltdown Attack. CPU uses out-of-order execution which means that CPU may run ahead once the required resources are available regardless of instruction order. It means that when the memory space is accessed, it is loaded into cache and then execute check. Though when the access is checked to be illegal, Some CPU designer forgets to clear out the effect on cache. So we may still find the secret data cache block by using the method in task 2. We can see that the third attempt below threshold and succeed to get secret data block.



Task 7: The Basic Meltdown Attack

In this step we approach the principle above to perform Meltdown Attack.

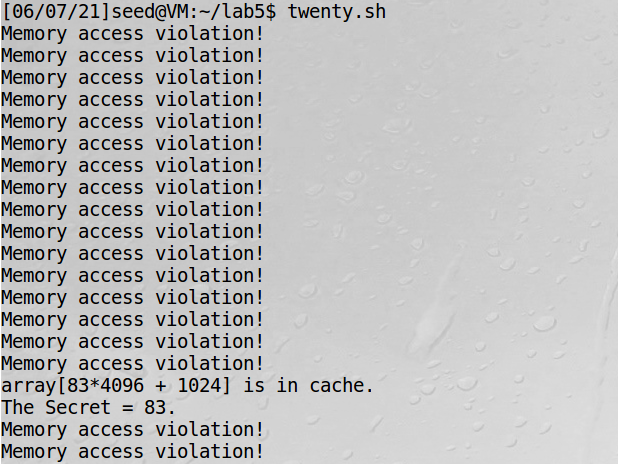
**Task 7.1: A Naive Approach**

A simple idea is that we locate the target value by access array[kernel\*4096+DELTA], and find which block’s access time is below threshold for cache access time. We can see that it’s indeed a naïve approach as its success rate is low. I run this attack program for 3000 times and unfortunately none of them succeed to find out the correct secret value.

****

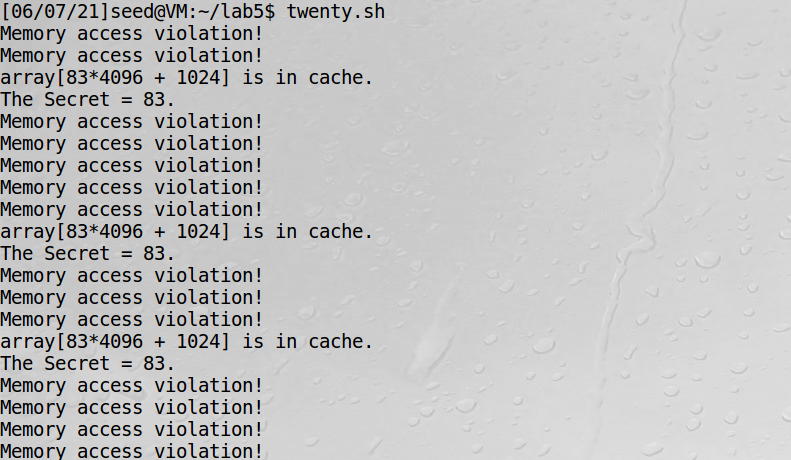
**Task 7.2: Improve the Attack by Getting the Secret Data Cached**

We can see that actually the Meltdown Attack is a race condition vulnerability. We need to load the secret data faster than security check then we succeed to find the correct block. So if we make data load process faster we can make our success rate high. We can make load kernel data faster by access this secret data in advance to move it to cache block before triggering the out-of-order execution. In following figure we can see a success attack one time out of 140.

****

**Task 7.3: Using Assembly Code to Trigger Meltdown**

Apart from make data load part faster, we can also perform a trick by using assembly code to give algorithmic units extra operations. Then our attack function (find the correct cache block) can have longer time to find target data and increase success rate.



**Summary:** In this part we perform basic Meltdown Attack. The procedure of our attack goes as follow:

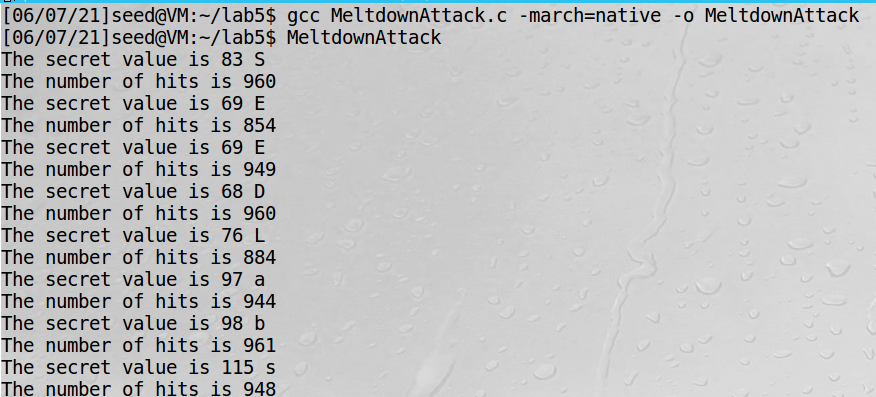
* load target data from Memory. At the same time system start security check.
* write the array element mapped by our target value to load it to cache.
* scan the entire array to find the block that has been loaded to cache.

A basic method is to use the target value directly. To increase attack success rate, we can use two tricks:

* Get the kernel secret data cached before launching the attack.
* Use assembly code to delay algorithmic unit.

Task 8: Make the Attack More Practical

In this step we practice the attack we learn above by steal the secret value from Meltdown Kernel. To improve the accuracy, we can run our attack program for multiple time and store the attack result. In the end we use the value k with the highest score as our final estimation of the secret. And Here we modify the MeltdownAttack.c to get all 8 bytes of the secret. Here we use an iteration to implement.

****

# Analysis and Conclusion 实验分析与结论

In this lab we finally perform a simple Meltdown Attack. And here we can make a summary to deepen the impression:

First, to steal the secret value from kernel space, we need to find **correct memory address**. Even if we find it hard to know the proper address, we can still guess a wide range to traversal.

Then, we can do pre-experiment to know the distinguish time between main memory access and cache access time. Then we use the threshold for the following step.

After that we should prepare a attack program. It will trigger meltdown attack by accessing this memory space. We should make use of side channel attack and find the correct cache block. Moreover, we should register a exception handler to make our attack program continue and store our attack result.

Before our attack perform, we can make some preparation to increase success rate. We can try to preload the target block to CPU cache to reduce load time. During our attack, we can perform some algorithmic operation to delay security check. In order to make our result reliable we can record attack result and find the maximum to confirm.

This lab is interesting to learn as it applies simple principle we learn from Operating System and circumvents kernel protection by using Out-of-order microarchitecture and CPU cache.