Coding part

1. Meltdown

da root scrivo password su indirizzo kernel space, ritorna [password, indirizzoKernelUsato]. Da altro terminale, senza root, con Meltdown, riesco a leggere da questo indirizzo, però lo devo conoscere. Comunque sia, poiché ciascun byte che viene letto dal kernel space può assumere 256 valori possibili, l'array A dovrà essere composto da 256 entry differenti. Si tratta di un attacco statistico: in realtà, i tempi di accesso alla cache e alla memoria RAM non sono deterministici; uno stesso accesso talvolta può risultare più lento, talvolta può essere più veloce, dipendentemente dallo scheduling delle istruzioni a livello hardware. Il successo dell'attacco è funzione anche di quanto differisce il tempo di accesso al valore memorizzato in cache da tutti i tempi di accesso agli altri valori che invece devono essere recuperati direttamente dalla RAM. Come detto più volte, in ISA non avremo nulla, ma vedremo dei residui micro-architetturali. La morale è che, sul processore, anche se divento offending io continuo lo stesso ad eseguire. Solo alla fine, vedendo il bit di trap cancello tutto. Il vantaggio di tale approccio risiede nel fatto di limitarmi all'utilizzo di un solo bit per la gestione di questi eventi. Cioè, dovrei avere meccanismi più complessi per arginare meglio le speculazioni. La patch attualmente più diffusa prevede il ritorno di un valore diverso dal valore vero. Quindi nell'esempio fatto da me con la scrittura di una pw su un indirizzo kernel che ritorna tale indirizzo, ne viene ritornato uno fittizio. In ISA NON posso leggere da cache, in quanto tale contenuto non è esposto in ISA.

2. Spectre

Analisi codice: `user-space-indirect-memory-reading-with-branches`.
In un'area di memoria metto un segreto, e non vi accedo più. Posso istruire però il predittore. Nel main uso `mmap` e prendo due pagine, oltre all'indirizzo secret-area in cui appunto scrivo. Ho un ciclo `while(1)` in cui cerco di capire la differenza di velocità tra ram e cache. L'istruzione `rdtscp` istanzia un timestamp contatore hw. Accedo svariate volte a `v[0]`, per averlo in cache. Successivamente eseguo nel secondo ciclo `clflush`, come per togliere il supporto alla cache, e avere cache miss. Prendo altra area di memoria usata come probing di 256 pagine (ovvero 256 possibilità). Scrivo un byte per ogni pagina, cosi posso materializzare l'oggetto.
Ora l'attacco: inizio ad addestrare, a volte con valori corretti, a volte con valori 'maliziosi' (circa uno ogni sei). In `side_effect_maker(x)` prendo in input x, se $x < 1$, ovvero $x = 0$ prendo l'info, vado nello zeresimo elemento per accedere al mio array. Se $x > 1$ eseguirò speculativamente, entrerà nello zeresimo byte e poi nella linea di cache che leggerò. Se ho casce hit va bene, sennò riprovo.
Nel `memory_reader`, per ogni entry delle 256, mi spiazzio di 12, leggo il *byte 0*, se *tempo* < *threshold*, ho cache hit e stampo tutte le info.

2.1. False cache sharing

Cache impattata quando creo struct ad esempio. Devo applicare politica *Loosely Related Fields*, cioè se un'operazione tocca due zone disgiunte, queste devono essere in due linee diverse di cache. Se fossero su stessa linea di cache, creerei un *False Cache Sharing*. Se le zone solo correlate, allora sono *Strong Related* e uso un'unica linea di cache. **Esempio**: Spawno due thread, uno legge e l'altro scrive su un vettore di interi (4 byte/entry), operazioni conflittuali. Writer aggiorna *v[0]*, Reader legge in una zona

v[TARGET]. Se TARGET >= 16, vuol dire che mi sto spaziando, dall'entry iniziale, di 16*4 = 64 byte, che è *una linea di cache*. Stiamo quindi dicendo che i due thread operano su due linee di cache diverse. Questa run è molto più veloce di un'altra con
TARGET <16, perchè i thread vanno in conflitto sulla singola linea di cache.
NB: Tutte le operazioni stanno nello **Store buffer**, non in cache, solo con **nfence** vanno in cache. (Nel codice c'è infatti **nfence**).
E' possibile avere una run veloce anche con TARGET < 16?
Si. Se la macchina è 2core/4thread (vedi slide *Baseline OOO, con due motori sotto)*, posso, tramite *0x5 ./a.out*, usare due hyperthread su stesso core (5 = 0101, uso lo 0-esimo e il secondo). Perchè lavora meglio? Perchè stanno **lavorando su stessa replica**, e non devono cambiare stato .
(MESI è tra cache "diverse", non nella singola). Essi accedono alla stessa memoria cache L1 e L2 del core e condividono la stessa copia dei dati. Senza *affinità*, i thread vengono mossi tra core diversi.

2.2. API kernel Level

- Il writer usa *synchronize_rcu()* per aspettare la fine del grace moment e "sganciare" l'area di memoria.
- Con *call_rcu()* non è il writer che aspetta, ma un demone di sistema.
- Con *rcu_read_lock()* e *rcu_read_**unlock()* segnalo che il lettore sta iniziando la sua attività, e poi la sua conclusione. Sono inutili in un contesto *non preemptive*.
Oltretutto, devo tener traccia di ciò che faccio, e quindi uno un *puntatore a struct srcu_struct* inizializzata tramite *INIT*, che poi dealloco con *CLEANUP*. Kernel Linux basato su queste API*: segnalo che non voglio abbandonare la cpu durante la lettura.* La lettura è non-preemtable, cioè in questo caso **non** lascio la cpu. Se nel mentre, ci fosse un servizio bloccante? Dovrei lasciarla io, ma questo potrebbe far pensare che ho finito di leggere. Quindi va bene solo con *servizi non bloccanti*. Esistono comunque API per gestire il caso di servizi bloccanti. ***RCU user level con thread preemptable a user-level!*** Uso istruzioni atomiche con Presence counter. Reader entra con +1, legge, e fa -1. Non basta un counter solo, perchè posso avere concorrenza rispetto al suo update, quindi chi viene dopo punta ad altro, chi viene prima no. Sostanzialmente dovrei mantenere due counter, attivo il nuovo e aspetto sul precedente, e rendo disponibile un nuovo *update* solo quando il precedente counter è libero, altrimenti potrei crearne troppi.

3. Locked List.c

Abbiamo due tipi di utilizzo: lock: `gcc try.c locked-list.c -DLOCK -lpthread -o test`

rcu: `gcc try.c rcu-list.c -lpthread -o test`

La versione "lock" lavora con i lock, la versione "rcu" senza lock. Si lavora a livello user, quindi un thread può cedere la cpu (è sleepable). La RCU-list con sleepable thread lavora con metadati. Ricordiamo che l'aggiunta di elementi alla lista non crea problemi a differenza di una rimozione, in quanto la rimozione comporta per thread che ci operano sopra la possibilità di "staccarsi" dalla lista, e perderne il puntamento.
La soluzione è quella già vista: quando si rimuove un elemento, inizia una nuova epoca, e quindi un nuovo contatore. Non posso usarne solo 1 di contatore, altrimenti potrebbero arrivare altri lettori, e non si svuoterebbe mai. L'idea è che ci sia una fase in cui aggiorno il contatore per i futuri lettori, mentre "aspetto" i vecchi lettori sul vecchio contatore. Ciò dipende dalla concorrenza con gli scrittori, cioè quando cambio contatatore, a che punto sono i lettori precedenti, se devo aspettarli o se hanno già visto l'update.

3.0.1. Uso dei counter

Il counter è aumentato in maniera atomica, mediante `__sync_fetch_and_add` : scrivo su un counter " c " che io lettore sto leggendo. scrivo su un counter " c' " che io lettore ho finito di leggere. Non potevo operare direttamente su " c ", e decrementarlo? No, perchè identificare il pointer e incrementare il contatore non è una operazione atomica. Con swap atomica posso porre c = 0 (azzero), ma chi arriva dove scriverà? Atomicamente associo " c " " per evitare confusione. Ho sempre counter per lettori concorrenti e lettori futuri! Alterno " c' " e " c" ", ma uso sempre " c " per l'accesso.

3.0.2. Graficamente

----------|-------+---------|---------->
past (vecchi let.) | rmv elements | future (no prob.)

Quando faccio la remove (all'istante "+", sgancio l'elemento, sto eseguendo una swap, ovvero resetto il counter e so quanti sono entrati prima, alternando il bit più significativo (0 o 1).

3.0.3. Thread house-keeper

Esegue una write_lock (esclude gli scrittori), aggiorna l'epoca, la incrementa, poi aspetta il 'grace period'. Evita l'overflow dell'epoca.

3.0.4. list.h

Implementa le logiche:

- rculist (spinlock solo writer)
- lock-list (spinlock writer/reader)

alcuni metadati sono: pointer al primo elemento (head), spinlock per lavorare, due contatori.

```
typedef struct _rcu_list{
 unsigned long standing[EPOCHS]; // lettori entranti
 unsigned long epoch; // indice epoca
 int next_epoch_index; // indice epoca successiva
 pthread_spinlock_t write_lock; // spinlock per scrivere
element * head; // pointer alla testa
} __attribute__((packed)) rcu_list; // packed: non ottimizzare
```

successivamente eseguo "aligned" perchè devo fare op. atomiche e mi allineo con linee cache.

3.0.5. Operazioni

- INIT: setta epoca = 0, ed epoca futura = 1, setta tutti i lettori 'standing' a 0.
- INSERT: alloca lo spazio per un nuovo elemento, setta valore e ptr nullo al successore. Abbiamo operazioni "mfence" per salvare in memoria. Lo store buffer è FIFO, se così non fosse avrei un

ordine inverso nelle operazioni di store.

- SEARCH: ho ptr alla testa e chiave da cercare come parametri. Scorro la lista, prima faccio `my_epoch = __sync_fetch_and_add(epoch,1);` per dire che sono "reader nell'epoca corrente" L'epoca può essere solo "0" o "1", mi basta per sapere dove rilasciare info che sansisce la mia terminazione in lettura.

- REMOVE: Quando concludo, sono al tempo 't', quindi i nuovi 'reader' non possono agganciarsi al vecchio elemento, tuttavia quelli vecchi potrebbero ancora essere collegati!

4. Lezione 26/10/23

Un modulo di linux è un kernel object, "ko", è un oggetto, non è un eseguibile, è reso eseguibile dal kernel (in particolare thread lato kernel). Noi analizziamo `usctm.c` e `the_usctm.ko` (usato per montare)

4.0.6. inserimento modulo

`sudo insmod the_usctm.ko` C'è funzione di startup che produce messaggi, posso vederli con `dmesg`, tutti i msg scritti in un array circolare. (come un log)

Mi stampa dove è sita la *syscall table*, *sys_ni_syscall* (indirizzi logici). Mi vengono dette anche le entry di *sys_ni_syscall*, anche dove è stata installata una syscall a due parametri (in sys table, posizione con entry nil).
Nb: syscall table compatibile con uno standard, quindi l'entry di spiazzamento 134 è perchè dallo standard risulta che 134 è ni_sys_call. Quindi già so quali aspettarmi (134,174,182,...)

Per ogni modulo che installo c'è anche la parte user, che è di test sostanzialmente, ovvero chiama il servizio che stiamo aggiungendo.

Usando anche qui `dmesg`, vediamo il thread che l'ha richiesta. Vediamo `usctm.c`, di cui ancora non possiamo capire tutto. vediamo init module, chiama `syscall_table_finder();`

Se le trova le scrive in variabile globale, se fallisce vuol dire che non l'ha trovata. Ho quindi le info, scorro le entry per vedere se contengono le entry della `ni_sys_call`. Tutte queste entry buone per scriverci sopra le metto in un altro vettore.

E' presente `ifdef Sys_call_Install`. Tuttavia abbiamo entry che sono read only. In CR0, il 16esimo bit ci dice se la protezione di page table, tlb è attiva o meno. Questo è aggiornabile, anche se a ring0, perchè quando inizializzo sono a ring0. Possiamo fare `read_cr0()`, non `write_cr0()`, dobbiamo implementarla noi! (avremmo problemi con la maschera di bit). Facciamo `unprotect_memory()` (cambio il bit). Nel vettore di appoggio per le ni_sys_call, ci metto `sys_trial`. Poi con `protect_memory()` rimetto lo stato protetto di cr0.

4.0.7. Come è fatta unprotect_memory?

Fa `write_cr0_forced` (passo cr0 e maschera bit per resettare il bit che mi serve). Essa come è fatta? prende unsigned long val, mette in un registro e fa mov su cr0. Fare una write su cr0 non vuol dire che leggo subito, perchè scrivo in un'altra zona della memoria. Devo quindi serializzarla. Lo faccio con

" `__force_order__` ". Se non lo facessi, andrei a scrivere sulla page table qualcosa che non è stato ancora scritto.

4.0.8. sys_call_install

Stiamo andando a puntare allo specifico oggetto syscall. Usiamo facility per vedere versione kernel, è > 4.17.0? E' da qui che esistono i wrapper, prima no. Se falso, opero con `asmlinkage`, dispatchato dal dispatcher, perchè deve sapere che deve prendere parametri dalla stack area. Altrimenti, ho syscall a due parametri con due parametri, A e B. sys_trail non esiste in versione >4.17.0, lo creo io in questi casi.

NB: Nel pc del prof non randomizzazione, quindi ciò che trovo non cambia di posizione. `grep sys_call_table /boot/System.map[versione]` mi dice syscall table a 32 e 64 bit. La "principale" che cerchiamo è 64 bit.

Su kernel > 5 ho randomizzazione, l'indirizzo non corrisponde a ciò che mi viene fornito a compile time dalla system map.

`cat /sys/module/` ogni cartella è un modulo montato, troviamo anche il modulo appena montato (the_usctm), e varie sottocartelle. Troviamo `sys_call_table_address` con sudo, ci viene dato un indirizzo della syscall table in decimale (prima era esadecimale). Qui trovo l'indirizzo della tabella usabile per fare installazioni.

`sudo cat /sys/module/the_usctm/parameters/free_entries` mi da' entry libere (134,174,177,180,...)

Queste operazioni avvengono anche quando eseguo update kernel a livello di release.

4.0.9. Se smonto syscall rimane qualcosa?

Nel modulo installato c'è una funzione per cleanup, in cui metto il valore che c'era prima. Senza non lo smonto. Ma basta cosi? Se smonto modulo ed elimino syscall, lo faccio in safe solo se tale syscall non è usata da thread. Se smonto, dove tornerebbe il thread?

---

5. TLS - soluzione

main crea thread con `pthread`, attività su memoria globale, locale, memoria tls (classica) e un tls alternativo.

- thread deve eseguire funzione target (`the work`).
  Per tale thread esiste già tls (by libreria). Come fa a farlo la libreria?
  Lungo il thread abbiamo starters , funzione e completamento. Quindi devo fare gestione tls negli "starters", potremmo far partire il "nostro starter" al posto di quello default.

`gcc main.c ./lib/tls.c -Xlinker --wrap=pthread_create -lpthread -DTLS -o test -1./include` con quel wrap uso wrapping, (dico di far partire un'altra cosa invece di quella di default) cioè due varianti di pthread_create. Il wrapper di `pthread_create` chiamerà prima o poi la libreria originale, ma diciamo che non deve partire (`*start_routine`), bensì dobbiamo far girare un'altra cosa. Chiamiamo la pthread vera ("real").

Adesso manca implementazione TLS. Come accedo? come metto variabili? Facciamo lavorare qualcuno a low-level. Per mettere oggetti nell'area, ognuno per thread, questi saranno accessibili tramite offset, posso farlo calcolare con "->". Esempio: per una struct uso operatore freccia per offsettarmi sulle componenti.

Con *architecture process control* mi faccio dare la base, applico freccia e mi faccio dare valore che mi interessa.
Ma questo vuol dire usare molte syscall. Alternativa:

Faccio accesso di offset rispetto GS, con offset = 0, abbiamo la base di GS.
Per l'operatore "->" mi serve indirizzo, quindi all'inizio di ogni area TLS scrivo indirizzo area TLS, cosi le altre entry sono solo offset rispetto ad indirizzo che ho salvato. Identifichiamo quindi `PER_THREAD_MEMORY_START` , `PER_THREAD_MEMORY_END` , e `TLS_SIZE` .
Se non la chiudessi, avrei errore dopo l'apertura. TLS position: fa mov di 0 in rax, (spiazzamento 0 a gs), poi faccio mov per ritornare ad indirizzo di questa area.

Per leggere, mi piazzo allo start, applico offset, e leggo con ->

5.1. TLS startup

`mmap` usa `TLS_SIZE` (dobbiamo usare il meno possibile le syscall).
Abbiamo ptr a tabella, registriamo due variabili e richiamo la funzione (stiamo tra starters e completamento). Il *main thread* è già tirato su con la libreria, quindi questi discorsi non valgono. Qui lancio thread, lancio wrapper. Posso wrappare il main, quando compilo e gli giro il mio di main.

La funzione wrapper viene indicata, nel make si ha `wrap=pthread_create` cioè se trovo questa funzione, passo il rifermento ad una funzione del tipo `wrap_pthread_create` o simile. Poi se la richiamo con `real` , invece chiamo la versione originale.

6. 2 novembre 2023

6.1. run time detection of current page size for i386

`unsigned long addr = 3 << 30` mi posiziono in indirizzo superiore ai 3 GB, sarà il mio addr.
`asmlinkage int sys_page_size(){...}` asmlinkage perchè è pre-kernel 4, ritorna un intero, cioè la taglia della pagina a cui stiamo lavorando. Qui assumiamo che l'indirizzo `addr` abbia una rappresentazione di metadati nella tabella *identity page table*, cioè la entry sia valida, ma non è scontato!

La soluzione è prendere una referenza differente, cioè all'interno di `sys_page_size()`, Prendo un nuovo *addr* dentro tale funzione, ovvero l'address di `sys_page_size`, perchè tale blocco di codice parte da tale indirizzo e corrisponde ad una pagina, o una zona, di 4MB.

7. 6 novembre 2023

7.0.1. cartella *huge pages*

Nel makefile abbiamo vari commands, possiamo vedere:

- la configurazione attuale (*show config*) Se eseguo `make show-config`, abbiamo *always*, *never* o *madvise*, con questa ultima la gestisco io, in modo *dettagliato*.
- `make check-counter` ci dice il numero massimo di huge pages configurate e gestibili.
- `make get-huge-pages` ci permette di configurarle. Si esegue con root, e fa un echo di un numero su pseudofile (non file), per riconfiguare il sistema operativo. Se richiamo `check number`, il valore si aggiorna.
- `make huge-pages-current data` mi dice info su Huge pages, quelle gestibili e quelle effettivamente free.

Gli allocatori di memoria che permettono al kernel di prendere memoria per usarla, lavorano quasi sempre senza riscrivere le table, usando pagine directly mapped. Le pagine sono note, cambio solo metadati per dire quali ho. La huge page è risorsa importante che do in uso ad address space, quindi serve una gestione precisa.

In `prog.c` abbiamo nel main un *base ptr*, faccio una *mmap*, materializzata quando scrivo un carattere su questa area. Allora la huge page viene consegnata in uso a questa applicazione. Se vedo quante huge pages sono adesse disponibili, ce ne saranno di meno.

Possiamo fare anche la `releases-huge-pages`, azzero la possibilità di uso, una rimane in *automatic release*, se killo il programma torna tutto a 0.

Se kernel da in uso *huge page* ad address space, fornisce memoria ampia su cui il kernel ha potere limitato, non può farci swap-out, è meno controllabile. Se fornisce quell'unico oggetto, il kernel non può più discriminare cosa farci!

Il kernel può riservare *huge pages* per sè stesso ovviamente (`reserved`).

7.0.2. Attacco L1TF

Abbiamo una VM, all'attaccante do un indirizzo fisico per riconfigurare la sua page table per fare letture speculative. L'attacker sfrutta:

- Modulo linux per riscrivere la page table da livello user, non chiamao syscall per cambiare page table. Tale modulo abilita la possibilità di lavorare su *devmem*, pseudofile che ci fa osservare tutto il contenuto della memoria fisica vista dalla VM. (che in realtà è logica perchè VM). Con la nostra

page table posso identificare PTE, elemento basso livello, e cambiarne il contenuto, fornendo indirizzo fisico dove voglio attaccare, con un bit di *non validità*, e poi eseguirà accesso speculativo.

- Nella parte speculativa si fa search memoria di PTE, la aggiorno, accedo la memoria secondo quello che abbiamo appena detto.

Lato host attaccato, abbiamo:

- un *segreto*, cioè montiamo modulo kernel che prende in page cache una pagina, cioè buffer in memoria, che sta da qualche parte in memoria fisica, ci metto pseudofile che rappresentano il contenuto del file "segreto". Facciamo `./start-l1tf-demo`, tale oggetto lanciato ha un thread che legge su cache che condividiamo con altri thread (se girasse su altro core la L1 è condivisa e non posso attaccare).

La vm è a due thread, `ps eL |grep EMI` mi dice i due thread. Li mettiamo su *0x1* e *0x4*, per averli in cache L1 condivisa.

Con `taskset -p 0x4 4597` applico la modifica per il primo thread (dipende dal thread ovviamente).

Ritorno all'attaccante, innanzitutto montiamo il modulo con
`sudo insmod devmem allow.ko`; poi prendo indirizzo fisico ritornato dall'host vittima, e chiamo
`./attacker_process [address in cui avviene l'attacco] [0x20 cioè numero byte letti]` e otteniamo il segreto. L'address è allineato con pagina. Dall'attaccato, potrei cambiare contenuto del segreto con
`"ciao" < /proc/the_secret` e abbiamo il segreto aggiornato lato attaccante.

L'indirizzo lo abbiamo preso grazie a *devmem*, che mi permette di vedere tutto, sia leggere sia scrivere le page table. L'attaccante fa search, cerca su PTE di livello basso e la modifica, prende una entry e ci scrive l'indirizzo fisico che ho passato, che corrisponde indirizzo fisico pseudofile dell'host, e poi marca i bit di controllo non valido. L'indirizzo è passato da terminale. Sennò dovrei fare try differenti. Se fossi ospitato da sistema su cloud providing, potrei avere seri problemi! (o crearli). Oggi, gli intel dalla nona generazione ,non sono più affetti. Però andando avanti escono altri bug. A livello hardware la patch è simile a meltdown, c'è un default, se entry non è valida non posso passarla come voglio, bensì faccio attività di default. Però ad esempio, grazie a Meltdown abbiamo sviluppato side effect di delay di tempo, e anche qui! Tipicamente questi attacchi sono sempre di *natura speculativa*, sennò sarebbero bug gravissimi!

7.0.3. Page table

Installiamo modulo kernel che permette ad utente di chiamare indirizzo logico, e la page table restituisce l'indice del page frame contenente l'indirizzo logico, quindi non direttamente l'indirizzo fisico.

`virtual_to_physical_memory_mapper`: Abbiamo indirizzo system call table, indirizzo *0x0* di default, quando monto modulo devo poterlo cambiare. Abbiamo poi un insieme di macro, come:

- *page table address*, che ritorna indirizzo logico della propria page table, sia che sia isolata che non. Lo fa mediante `read_cr3`, lo legge, mette in un registro e lo ritorna. CR3 ha anche bit di controllo (parte iniziale) che devo scartare, applicando *address_mask* scartando i primi 12. Così ottengo indirizzi fisici tabella pagine. Ma serve indirizzo logico.

- *phys_to_virt*, mi da indirizzo logico, è directly mapped.
- Dell'indirizzo logico devo poter scendere nella page table, e prendere indirizzo frame che deve essere restituito. Varie macro in cui estraggo dei bit. Rappresentano possibilità di muoversi tra le page table e sapere quali page table sono coinvolte.

Ricaviamo, per il thread in esecuzione, *pml4 = PAGE_TABLE_ADDRESS*, con `down_read` prendo token di lettura per evitare problemi di concorrenza, sfrutta il thread control block corrente, prendo tabella di memory management, e poi da li prendo i costrutti di sincronizzazione, per indicare che sto leggendo. Poi vado su *plm4* ottenuto prima, prendo `plm4[target_address]`,
vado su *.pgd* e prendo la entry. Libero oggetto se non riesco ad andare oltre. Altrimenti, estraggo bit della tabella sottostante, cioè di estrarre indirizzo. Con questi ci faccio *virtual address* (perchè ho ottenuto indirizzo fisico). Con `pud` vedo se valido o meno. Se valido, ritorno nella PDP, prendo indirizzo, sto nella PDE, faccio check, ma qui la pagina associata ad indirizzo logico potrebbe essere *huge*, devo fare un check per forza. Eventualmente mando msg per vedere che sta succedendo. Poi prendo tabella PTE, faccio le stesse cose. Alla fine rilascio lock e faccio i calcoli, cioè dalla PTE prendo indirizzo fisico oggetto (frame), scarto i 12 bit e trovo indirizzo il frame number di interesse.

Queste cose le abbiamo già usate in *table-discovery*, cioè search dell'address space per cercare una tabella. Per fare search, pagina logica deve essere pagina presente in memoria fisica, se non c'è accade un disastro. Nel modulo c'è l'embedding del codice appena visto, di poco modificato. Quando il modulo fa nell'address space del kernel, scandisce tabella, verifica se quella pagina logica è presente in memoria fisica. Ovvero prendo tabella pagina corrente, è presente in memoria fisica? finchè non la trovo itero.

7.0.4. Modulo ko e usctm.c

Validate page passo l'indirizzo logico, e vedo se corrisponde a pagine veramente presente in memoria fisica tramite `vtpmo` (virtual time phisycal object), lo faccio anche per la pagina successiva. Cioè se cerco tabella a partire da un certo indirizzo, devo vedere quante pagine devo contenere, devo essere tutte valide. Il check lo faccio su due pagine, perchè la syscall può stare in una pagina o al più due pagine. Il check eseguito è:

- se pagina e successiva valida, allora cerco tabella in zona memoria, vedo se trovo coincidenza con almeno due entry dei servizi non offerti (gliene bastano due, ad esempio 134 e 137).
- se non trovo, vado 8 byte più avanti e cosi via.

Questo modulo non richiede *nulla*, cioè è facile da usare, l'unica cosa che faccio è lavorare con tale modulo ma implementazione mia, non facciamo syscall. Basta che il kernel permetta di montare un modulo.

Facciamo load dell'oggetto, otteniamo, tra gli altri, anche indirizzi syscall table e ni_syscall. Posso usarli per montare oggetti (`sudo make mount`) uso poi `virtual_to_physical_memory_mapper.ko the_syscall_table={$A}`, con A indirizzo della syscall table address.

In `user.c` abbiamo:

con `if (argc==2) buff[0]="q"` materializzo. Poi prendo frame number che deve essere ritornato, vedo i parametri passati al programma:

- se *read*, prendo contenuto su ciascuna delle pagine (mi spiazzo di 4096 e prendo un byte), passo indirizzo logico da cui ho letto, vedo dove sta memoria fisisca.
- se *write*, scrivo su 0-esimo byte un carattere, è uguale.
- se *vtpmo*, chiamo senza leggere nè scrivere, ma chiedo sempre indirizzo fisico.

compilando `gcc user.c` senza flag, errore. con `gcc user.c vtmpo` ho stesso errore, tranne per pagina 0-esima. (non materializzata)

con la `read` tutta la memoria è presente in memoria fisica (da array pagine logiche), le altre sono presenti nello stesso elemento della memoria fisica, il fenomeno è **empty zero memory**, pagine su stesso frame.

con `write` ho tutti frame fisici diversi, non è come prima. Materializzazione effettuata.

8. 8 novembre 2023

8.1. test NUMA

Ho un parametro `auto-affinity`, la quale prevede di usare un processore (cioè nodo NUMA) diverso da 0. Materializziamo sul nodo NUMA 0 invece. Nel `main` abbiamo `set_mem_policy`, cioè ci connettiamo a specifica maschera di nodi, scelta in funzione della configurazione fatta sopra. Abbiamo un ciclo, scrivo in un array, materializzo pagine array, abbiamo page fault, e per la mem policy le pagine vanno nel nodo 0.Abbiamo poi un for, dove accediamo alla memoria ma poi facciamo flush da cache, quindi al prossimo accesso prendiamo dalla ram (NUMA).

Avvio con `gcc numa-test.c -lnuma -DAUTO_AFFINITY` e poi `time taskset 0x1 ./a.out`, lavoriamo su una cpu che accede ad un nodo numa diverso. Se ricompilo senza il flag, thread non cambia affinità, siamo locali, e siamo più veloci (circa metà tempo, considerando anche tempi flush e senza traffico sui nodi NUMA).

8.2. 13 novembre 2023

8.3. Table discovery

Carichiamo due moduli:

Il primo è associato `message-exchange-service-intermediate-buffering`. Vogliamo meccanismo per scambiare messaggi. Chi scrive chiama syscall, il messaggio viene portato in area user. In realtà questa area user, potrebbe essere valida ma non materializzata (uguale per la lettura), cioè può capitare page-fault. Se l'area, più o meno ampia, potremmo avere che una pagina la abbiamo e un'altra no, quindi abbiamo page-fault mentre leggiamo informazioni ad esempio. Se leggiamo, non possiamo scrivere (atomicità). La soluzione proposta è di usare una area intermedia nel kernel tipicamente directly mapped, non soggetta a page-fault. Il flag `SLAB_POISON` nella `kmem_cache_create` ci dice che la creazione viene fatta senza particolari vincoli.

Nel codice, in `sys_get_message`:

Facciamo check sulla size, per vedere se coerente, se lo è prendo tramite `get_zeroed` (dipende dal Buddy Allocator), in modalità Kernel. Probabilità bassa di andare in blocco. Faccio check per vedere se posso fare questo buffering intermedio. Se tutto ok, faccio copy from user, partire dall'area indicata. Il valore di ritorno ci dice il numero di byte *non* copiati. Poi blocchiamo con mutex, e facciamo mem_copy per copiare nel kernel_buffer dove le informazioni devono essere presente. Chiamiamo anche `print_k`, aggiorniamo VALID, pari a SIZE-RET, cioè alcuni byte parte user/kernel non ci sono finiti. Alla fine libero il buffer intermedio. Non potevo usare la stack area invece del buffer così fatto? No, perchè gli stack kernel hanno taglia precisa e limitata, devono essere contenuti, sennò rischiamo di andare fuori. Nel kernel, non c'è ricorsione (sennò userei in maniera massiva lo stack), è tutto iterativo. Se Kernel>4.17 usiamo simboli per rimappare syscall dentro la tabella.

Lo user non fa altro che prendere nome programma-nome syscall (134 o 137)- messaggio [se scrivo]. Poi chiamiamo la syscall passata e leggiamo le info prese dal kernel.

Il secondo è `queued-message-exchange-service`, ovvero abbiamo coda, non è che cancello ogni volta il messaggio precedente. Ci vogliono più buffer allora, li prendiamo con `kmemcache`, non la versione di default, ma la creiamo noi. Ha aree allocabili di una certa taglia, specifico anche il callback quando facciamo pre-reserving dal buddy-allocator, cioè per il setup dell'area di memoria. Indirizzo allineato all'area che devo consegnare. Quando si fa richiesta memoria da specifica cpu, è non c'è pre-reserving, esso viene fatto. Attualmente in `setup_area` non facicamo nulla. La lista in cui mettiamo i messaggi deve essere doppiamente collegata, perchè togliamo in testa e aggiungiamo in coda. Esse esistono entrambe, quindi lista mai vuota, così è più semplice da gestire, per ridurre gli if possibili. Viene fatto un insieme di check, e poi `kmemcache_alloc`, controllo che mi venga ritornata un'area. Se sì, faccio, fuori dalla sezione critica, `copy_from_user`, cosi i fault non vanno in sezione critica. Facciamo poi spinlock, e vedo se la lista è ben definita, poi aggiungo oggetto in testa, e ritorno SIZE-RET. Faccio similmente per `copy_to_user`. Abbiamo check anche sull'invocazione dei parametri del programma.

Per usarlo, montiamo il modulo, stesse entry lasciate dal modulo precedente. La parte user prevede un ciclo, perchè accodiamo messaggi. Compito con `./a.out 134 ciao` o `./a.out 174`

Noi abbiamo usato i moduli come strumento. Il thread kernel potrebbe andare in blocco con `kmem_cache_alloc(the_cache;GFP_USER)`, se smontiamo modulo, esso viene smontato. Ma il thread che è bloccato in attesa? Nel codice della funzione, c'è funzione di smontaggio `kmem_cache_destroy`, ma ciò porta ad un leak di memoria, inoltre rimuovo allocatore, ma non so se è stato correttamente rimosso.

Page table sfruttabili per arrivare direttamente in memoria. Se parametro è oltre addr_limit, sfrutto questo espediente.

16 novembre

9. usctm.c

Nel file abbiamo `sys_call_table_address` e un `module_param`, lo stesso per `ni_sys_call_address`, e sempre con i parametri li rendiamo visibili grazie al virtual file system (flag 0660). `free_entries[MAX_FREE]` è parametro del modulo, posso osservare tale oggetto mediante

`module_param_array(free_entries,NULL,0660)`. Rechiamoci in `cd /sys/module/the_utsmc/parameters` e vediamo gli elementi appena descritti.

Con `sudo cat free_entries` vediamo le entries non null marcate dal file.

10. l1tf - Kernel Level memory management

Nella cartella c'è codice `doit.c`, passo indirizzo fisico, a cui page table monta. Viene fatto tramite modulo. Il modulo è `devmem_allow.c`, che sfrutta `kallsyms_lookup_name`. I kernel devono lavorare in specifiche zone. Ok con kernel < 5.7.

Troviamo `devmem_is_allowed` e `text_poke`, che mi permette di applicare patch che voglio io! (diverso da kernel probe) Sostituisco a `devmem_is_allowed` istruzioni dummy per bypassarlo (slla fine sono delle push).

10.1. Moduli

Abbiamo :

10.1.1. printk_example:

se andiamo a vedere il makefile, esso mi da alcuni commands, tra cui `mount`, in cui usiamo anche `the_syscall_table=${A}`, cioè passato da `sys_call_table_address`.

10.1.2. kprob_usage_example:

Abbiamo unsigned long = 0, possiamo osservare il valore di tale variabile. `hook_func=0` e `audit_counter=0`, il secondo conta le volte che succede "qualcosa" nel software. Se passiamo da threadA a threadB, prendiamo questo. Contiamo context switch. Il codice associato ad `hook` intercetta accesso a funzione target, e stiamo chiamando la finalizzazione. Il contatore interno è atomico, mentre "message_counter" non lo è. In `__init hook_init` installiamo kretprobe. Poi abbiamo anche unregistre del module.
La interception del context switch ci permette di dire che, se dei thread devono essere gestiti in modo diverso, possiamo innalzare delle barriere se sospetti.

10.2. Kprobe-read-interceptor

Supponiamo che thead chiama read. Se è su stdout (canale 0), la intercetto per uno specifico processo. A livello kernel intercetto anche password scritte su stdin. Quando entro in sys_read non ho snapshot cpu, quindi non so nè canale nè puntatore. Li so all'uscita, ed avrei altre cose nei registri. Devo installare handler h' di ingresso, mi devo registrare i parametri utili da qualche utili, e prenderli quando mi servono per fare le operazioni. Dove li metto? entro in sottosistema, potrei chiamare allocatore memoria in modo atomic, come relaziono tale memoria al thread? serve pointer. E dove lo prendo? Nell'handler ho solo variabili locali, dopo il pre_handler è perso.

Soluzione: Utilizziamo estensione logica del thread control block, quindi con più informazioni. Dove la prendiamo questa memoria per estendere? Semplicissimo: Il TCB ha pointer a stack area del thread. In

alcune versioni, la parte top dello stack ha un'estensione del TCB. Possiamo scriverci "sotto"? Basta non saturare la stack area. Quindi, l'handler registra in questa area l'info del dove verranno salvati. H' li riprenderà in quell'area di memoria. Tutto mediante pointer area user, settato quando entrato, e usiamo quando usciamo.

In `hook.c` vediamo che i metadati sono in struct `thread_info`, e mi spiazzo a seconda se il kernel includa quest'area iniziale allo stack. Prendo `current` , è pointer a TCB, come punto allo stack? `->stack` e poi mi spiazzo di offset. Quando lo riusco, uso `load_address`. Vengono usati dagli handler.

All'inizio `target_pid=-1`, non osservo alcun processo.

A seconda della versione kernel, manteniamo snapshot da una parte piuttosto che un'altra.

Ritornare "1" su pre-handler, vuol dire che l'handler di *uscita* NON viene eseguito. Se devo monitorare, prendo `regs->si`, e faccio store in quest'area.

Allora ritorna 0, e l'handler di uscita eseguirà. In questo secondo handler, alcuni check già li ho fatti prima, devo sapere quanti dati sono stati consegnati, lo vedo dal valore di ritorno di sys_read. Poi li loggo con `printk`.

Questo viene fatto da un thread associato ad uno specifico processo.

Il blocco `while(!copy_from_user((void...)` è bloccante, può generare dei fault. Non potrei eseguirla, allora installo altro modulo. Sostanzialmente, se la cpu va ad altri, viene installato il contesto dell'altro e viene eseguito. Quando ritorno io, mi viene restorato il mio contesto, preso dalla variabile `per-cpu`.

Nel modulo dichiaro variabile `per-cpu` per la discovery, per spostarmi nell'area in cui c'è oggetto il cui valore è pari a `kp`, cioè variabile di cui devo fare l'aggiustamento. C'è codice `brute force`, si può fare solo quando installiamo il modulo, una volta solo. Devo fare eseguire tale funzione su tutte le cpu.

22 novembre

Ci troviamo in **Kernel LEVEL TASK MANAGEMENT**.

In particolare, iniziamo con **Tasklets.c**

Abbiamo struttura dati `_packed_task` che include a sua volta un'altra struttura dati. Nell'*init module*, c'è del lavoro che deve essere processato deferred, svariati controlli. Alla fine abbiamo una `put_work`, in base alla presenza o assenza del wrapper. Esaminiamo la syscall `put_work`, facciamo un `try_module_get`, per la richiesta del deferred work. Poi allochiamo, con `zalloc`, in modo atomico, memoria. Se ho problemi rilasciamo. Altrimenti andiamo in `the_task`, campo buffer, e vi scriviamo dei task. Ha un pointer che punta a se stessa, per il retrieve automatico. Il campo request_code è il parametro passato alla syscall.

Inizializziamo la tasklet, in particolare la struttura presente nella struttura allocata. `&the_task->the_tasklet.audit`. Scheduliamo sulla cpu *corrente*, dove sto eseguendo il servizio. In `printk`, o `tasklet_init`, potrei essere migrato, quando scheduliamo su cpu corrente, no. In `code` c'è il codice della specifica chiamata. Viene usato su `data` il `container_of` per il retrieve. Poi facciamo `kfree` per liberare la memoria, e successivamente decremento il contatore d'uso, perchè c'è stato processamento.

Ora ritorniamo al demone, che lavora su software esterno. Posso smontarlo? dipende dal valore del counter.

Montiamo con `sudo make mount`, ci da indirizzo SYSCALL TABLE, e prende indirizzo 134, il primo libero. In `ser.c` ,la chiamata della funzione è `syscall-num`. L'avvio del codice è sul cpu-core1, quindi la print è software queue demon. Se mettessi `taskset 0x1` viene tutto dirottato sulla cpu0. Rimuoviamo con `sudo rrmod the_tasklets.ko`

11. Work queue

In `work_queues.c`, abbiamo sempre `audit`, la quale manda in print qualcosa dalla workqueue. Anche qui abbiamo `module_put`, perchè esso deve essere presente. Nella syscall, prendo e blocco il modulo, il parametro ulteriore è la cpu su cui eseguire il lavoro, ovviamente deve essere presente! Sempre `kzalloc` atomica, prendo il buffer, controllo se sia stato preso. Poi inizializziamo con `__INIT_WORK`, in cui eseguiamo la funzione `the_audit`, il parametro ulteriore è la tabella interna, non quella esterna, che possiamo trovare con `container_of`. In `schedule_work` passiamo proprio l'address della tabella interna.

11.0.1. Parte User

Vuole usage con numero di processing unit. Opera similmente a quello già visto.

11.1. Lancio

`./a.out 134 1`, eseguiamo sempre su cpu1, anche se uso `tasket 0x1`, sempre su 1 eseguo il deferred work. Poi rimuoviamo anche questo modulo.

12. Interceptor

In `hook.c`, viene fatta `copy_from_user`, è bloccante, perchè è possibile page fault. Ma stiamo in `kernel return prob`, non potrei scrivere codice bloccante. Soluzione: annullo contesto corrente, e poi lo rimetto a posto. Ma quindi devo abiliare *preemption*, perchè se annullo il concetto del contesto, annullo anche il concetto di preemtability. Se così non fosse, il codice avrebbe dei safe-places non usabili se non lo riattivo con `preempt_enable`, poi alla fine `preempt_disable`, quando passo il controllo al sottosistema di kernel probe.

La logica del codice è: Ho un thread, che chiama `sys_read()` (siamo scesi nel kernel) e poi `return probe`, la quale sa dove e quanti dati sono stati consegnati, vedendo snapshot cpu del ritorno del blocco di codice, mediante MACRO. Se ho stato di blocco nella sycall, resto fermo (es: aspetto che page fault sia risolto). Ad esempio, la `copy_from_user`, avvenendo in un ciclo, potrebbe accedere a pagine "più avanti" quando la richiamo. Nella return probe stiamo lavorando su **per-cpu variabile***, puntata da *temp. Inizialmente c'è null, non sto in contesto di kernel probe, poi alla fine rimetto a posto il contesto originale. E se venissi rischedulato su **altra cpu**?

La variabile per-cpu non è all'indirizzo temp, ma in un altro indirizzo. Sto corrompendo la variabile per-cpu di un'altra cpu. Ciò avviene se i servizi vanno in pree-emption oppure vanno in blocco, posso essere

rimesso su altra cpu. Soluzione? Devo aggiungere altro! Soluzione parziale: affinità, ma poi lo forzo a lavorare solo lì!

13. 23 novembre

14. Servizio usleep()

Possiamo dormire a grana fine, ovviamente esiste già, però sappiamo che se chiediamo di dormire x millisecondi, il kernel ci fa dormire $x + \Delta$ millisecondi. La sys call `sys_goto_sleep(unsigned long microsecs)` ci fa dormire per un tempo pari al parametro. Possiamo creare `wait_queue` locale, cioè nella stack area dell'oggetto. Ha senso se la usiamo solo noi qui dentro, relazionabili a questo thread. Così non chiamiamo nè allocatore nè altro. Quando il timer scorre, usciamo. Altrimenti, settiamo `control = &data` , variabile locale associata a `control_record` , una struct contenente `struct task_struct *task` in cui scriviamo il nostro TCB, nel pid il pid e un `awake` , per il risveglio. Abbiamo anche `struct htimer hr_timer` . Tutto questo è puntato da `control_record` , allora riempio le informazioni (esempio: `control ->stack = current`) . Poi inizializzo `hr_timer` per inizializzare timer e farci risvegliare. In origine, mettiamo NO nell'awake, e poi chiediamo di metterci nella coda di attesa con una `control->awake=YES` e un certo intervallo. Perchè viene messo a dormire più del dovuto se non è un thread real time? Perchè non è così importante non essendo real-time? NO. Se è a bassa priorità, viene messo nella coda e non verrà selezionato in favore di altri, quindi perchè svegliarlo poco più tardi? Quando ci risvegliamo, mettiamo TCB su runqueue, ma è ciò che avviene davvero? chi lo fa? Noi giriamo top_half dell'hr_timer, marca la software queue_deamon che mette sulla runqueu il demone ad alta priorità, quindi dobbiamo togliere altre cose dalla cpu. Allora, tanto vale farlo per attività sensibili. Thread di bassa priorità posso interferire su quelli ad alta priorità, quindi se il thread non è real-time dorme di più. Il demone non cambia priorità. Con la nostra versione, ciò non avviene, inizializziamo struttura di controllo e timer, che lavora col nostro intervallo, saltando le regole del kernel. Alla fine, la tabella inserita in hr_timer la cancello con `cancel` . Perchè posso uscire o se il demone mette condizione a yes, o se lo ha fatto qualcun altro. Se così non faccio, demone riferisce stack area in cui non ha più senso controllare informazioni. Bisogna ragionare in termine di ecosistema, non solo quello che vediamo noi.

Quando arriva interrupt, parte il demone che processa la funzione da noi definita, con `container_of` prendo tabella esterna, contenente info thread, tcb, condizioni etc... qui settiamo `awake=YES` etc. All'uscita, il software che lo ha chiamato non la reinserirà. Li dentro uso stack area di qualcuno, quel qualcuno deve esistere. Se montiamo ed avviamo con `./a.out [numero millisecondi]` , vediamo che con un numero ridotto di millisecondi c'è una differenza tra *usleep classica* e la *nostra implementazione*. Se disattivo macro, userò due volte le stesse cose, con stessi risultati.