Coding part

1. 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.

1.0.1. 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.

1.0.1. 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).

1.0.1. 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.

1.0.1. 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.

1.0.1. 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!

2. 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)

2.0.1. 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.

2.0.1. 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.

2.0.1. 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.

2.0.1. 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?

---

3. 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 ->

3.1. 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.

4. 2 novembre 2023

4.1. 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.

5. 6 novembre 2023

5.0.1. 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`).

5.0.2. 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!

5.0.3. 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.

5.0.4. 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.