**Progetto SOA**

**0** – **Introduzione:**

LKM 🡪 loadable linux kernel != base kernel

(hanno 3 maxi categorie)

- device driver

- filesystem driver

- system calls

**1** – **Specifiche progetto:**

• Scambio dati basato su TAG

• Montare un modulo

* + - Ricerca della tabella delle system call
    - Scrivere 4 chiamate di sistema
      * Get
      * Send
      * Receive
      * Control
    - Il software deve gestire 256 tag
    - Messaggi almeno 4 KB
    - 32 livelli
    - Device driver

**2 – Moduli Linux:**

• Comando per vedere tutti i moduli caricati

lsmod

• Comando per caricare i moduli

insmod + modulo

• Comando per vedere tutti i moduli caricati

rmmod + modulo

* **Hello world moduli**

<https://www.geeksforgeeks.org/linux-kernel-module-programming-hello-world-program/>

Modulo = pezzo di codice che può essere caricato o scaricato a domanda.

Un modulo può essere aggiunto in 2 modi:

1) Aggiungere un modulo alla “kernel source tree” e ricompilare il kernel.

2) Aggiungere un modulo in corsa.

* **Codice di Quaglia sui moduli**

**Kprobe-interceptor** 🡪 prende il contenuto dello standard input e lo passa al modulo.

**Hook\_init** 🡪 atomic counter che non ci da problemi di concorrenza.

large target pid 🡪 il processo per cui vogliamo osservare lo standard input.

current 🡪 all’interno della the\_hook prende l’indirizzo di memoria corrente del thread.

copy\_from\_user e inserisco il terminatore di stringa.

**User**: oggetto che scrive su standard input e legge da standard output.

**Kprobe usage** 🡪 aggiungiamo altri moduli alla nostra hook.

**Finish task with** 🡪 determina che si è concluso con un cambio di thread.

**Tail\_hook** 🡪 parla di hook\_function che è tipicamente pari a 0 ma se lo relaziono a qualcos’altro lui lo va a chiamare. Lo colleghiamo a kernel function pointer exposition che espone un function pointer (indirizzo di memoria associato a una funzione).

**3 – Makefile:**

La seconda riga del makefile ci dice che usiamo due moduli per creare un unico kernel object.

**4 – Multithread:**

Argomento utile per il progetto.

**5 – Riconciliazione temporale:**

Argomento utile per il progetto, top/bottom half.

**6 – System call:**

Trap e interrupt 🡪 migliorano il livello di privilegio.

Entry della tabella degli interrupts sono GATE.

Gestire le trap 🡪 meccanismo dei gate 🡪 inseriti in una tabella di livello kernel, interrupts table, come detto ogni entry identifica un gate che è a sua volta identificato da un id segmento ea quale offset punta.

Il gate ci indica anche qual è il valore massimo che può passare attraverso quel gate (passaggio cross segmento) e va confrontato con il CPL di ogni thread (di quello che fa l’accesso). Quando eseguiamo in user mode abbiamo un certo CPL e vogliamo iniziare a operare in modalità kernel dobbiamo usare un gate 🡪 quando entriamo il CPL viene portato a 0.

Passare da privilegio 0 a 3 è comunque un passaggio di cross segmento ma non ha bisogno dei gate.

Se viene fatto un passaggio cross segmento illegale viene lanciata un’eccezione e quindi un gestore di eccezione.

Passare a un segmento con un livelli di privilegio migliore si può fare attraverso l’istruzione INT 🡪 che prende come unico parametro il displacement all’interno della tabella delle trap che descrive il gate che vogliamo utilizzare (poi il gate indica anche il segmento dove vogliamo spiazzarci).

**7 – Gate**

ITD 🡪 interrupt descriptor table 🡪 ciascuna entry è un gate.

IDTR ci permette di accedere a IDT 🡪 il parametro che passiamo a INT deve essere compatibile con la size della IDT che è indicato in IDTR (riprogrammabile).

CPL contenuto nel registro CS.

Nel gate ci sono altri metadati che ci indicano come deve essere settato lo stato del processore all’ingresso del gate.

Il gate che ci fa accedere al kernel on demand è unico, in particolare negli x86 a 32 bit è all’interno della entry a spiazzamento 0x80 (esadecimale) (INT).

Gli altri gate vengono usati per altri motivi, come ad esempio accedere agli interrupt.

Il dispatcher, che è in grado di dispatchare ogni servizio all’interno del kernel, è il blocco di codice a cui ci rimanda quest’unico gate.

Per far capire al (unico) dispatcher quale servizio vogliamo attivare utilizziamo un’ulteriore tabella puramente software 🡪 system call table 🡪 ciascuna entry è un puntatore a una funzione di livello kernel. Quindi il dispatcher deve ricevere l’indice numerico del servizio che deve richiamare.

Il dispatcher eesegue già a CPL 0, attraverso CALL<offstet> andiamo a chiamare la syscall. Il suo output andrà a finire nei registri di sistema. Poi il dispatcher ci deve riportare in modalità user (DTI).

**8 – System call** **2:**

1. Servizio non dispatchabile dall’attuale dispatcher, quindi dobbiamo crearne uno nuovo nostro
2. Oppure usando lo stesso seguendo le regole che segue per altre syscall

Dobbiamo sicuramente andare a modificare la system call table 🡪 trovare una entry libera e agganciare il pointer alla nostra funzione oppure espanderla (ma è molto difficile).

La syscall che implementiamo deve essere compliant a livello di formato e di regole rispetto al dispatcher.

Indicizzare le syscall 🡪 associare il codice numerico k a una funzione (o meglio ad un pointer di una funzione) 🡪 con unistd\_63 cambiano i codici numerici rispetto a unistd\_32 (che è ancora supportato).

A livello user le info riguardanti l’indicizzazione le troviamo all’interno di alcuni header come, ad esempio, unistd\_64.h

Un user per accedere al gate deve passare il codice numerico e anche i parametri.

Il meccanismo dei gate necessita di linguaggio assembly.

**9 – Stub di system call:**

Entra in modo kernel e chiama la reale attività, gli stub vengono costruiti attraverso macro livello user (che hanno anche istruzioni machine dependent) a cui passiamo i 7 parametri.

Syscall0(type, nome)

0 🡪 perchè prende 0 parametri

Type 🡪 del valore di ritorno

Nome 🡪 della funzione C deve essere generata

**10 – Utili:**

<https://www.generacodice.com/it/articolo/36965/>

asm line prende i parametri dai registri di processore e non dallo stack

**11 – Syscall:**

Immagine che contiene testo, ricevuta

Descrizione generata automaticamente

I registri MSR ci aiutano perché avito di accedere alla memoria ogni volta. Sta tutto nel processore.

Abbiamo 2 dispatcher diversi proprio grazie a MSR.

Per utilizzarli in passato si usava sysenter (32bit) ma ora usiamo syscall (64bit) 🡪 con questa usiamo i registri MSR e non serve lo stack pointer.

Non abbiamo quindi uno stack switch fino a che non prende il controllo il kernel; per tornare in modo user a 64bit dobbiamo usare sysret (trova in rcx il punto di ritorno).

Troviamo syscall() in stdlib.h 🡪 API che diventa una sorta di dispatcher livello user.

**12 – System call table:**

Difficilmente possiamo operare resizing della tabella, eseguiamo quindi l’hacking della system call table per cercare entry libere. Tutte le entry ora sono occupate ma alcune puntano allo stesso servizio x, ovvero un servizio che implementa tutti i servizi che non hanno una reale implementazione 🡪 ni\_syscall.

**13 – Sys ni\_syscall:**

La tabella possiamo trovarla in arch/i386/kernel/entrys.c

Arch/x86/entry/syscall64.c

sys\_qualcosa ci permette di accedere al frontend di ‘qualcosa’ all’interno del kernel.

Quaglia a lezione ci ha fatto vedere l’uso delle macro per utilizzarle come trampolino per i frontend delle syscall che servono ad offuscare la stack area.

**14 – Utili:**

Ricordare di fare alla fine il cleen up dei moduli rimettendo le ni\_syscall.

**15 – Codice:**

1. Lato user 🡪 scrivere il codice che lato user richiama la nostra system call tramite syscall()
   1. Il codice user Quaglia lo spiega a lezione 16 al minuto 97.
2. Lato kernel 🡪 all’interno del modulo da mantenere inserire la funzione che hackera la tabella delle system call e il frontend delle chiamate di sistema (ovvero la loro effettiva implementazione).