UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

Exploiting Linux kernel using dirty pagetables

Progetto di Network Security

Anno Accademico 2023/2024

Giuseppe Capasso
matr. M63001498

Indice

Introduzione

Oggigiorno, 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 offerte sia dai compilatori sia dai sistemi operativi. 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 quando un potenziale double possa essere sfruttato perchè la deallocazione può avvenire in contemporanea in altre applicazioni e può essere soggetta a operazioni di **quarantena**.

Tuttavia, le moderne protezioni per questo tipo di attacco sono presenti solamente come strumento di testing, ma sono impraticabili dal punto di vista pratico in quanto si basano sull'instrumentazione del codice che può portare ad una degradazione delle performance (sia dal punto di vista di memoria utilizzata che dal throughput del sistema) di diversi ordini di grandezza. Linux, su cui si basa questo progetto, mette a disposizione strumenti avanzati quali ***KMSAN, KASAN e KCMSAN***.

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 processi attualmente in esecuzione.

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

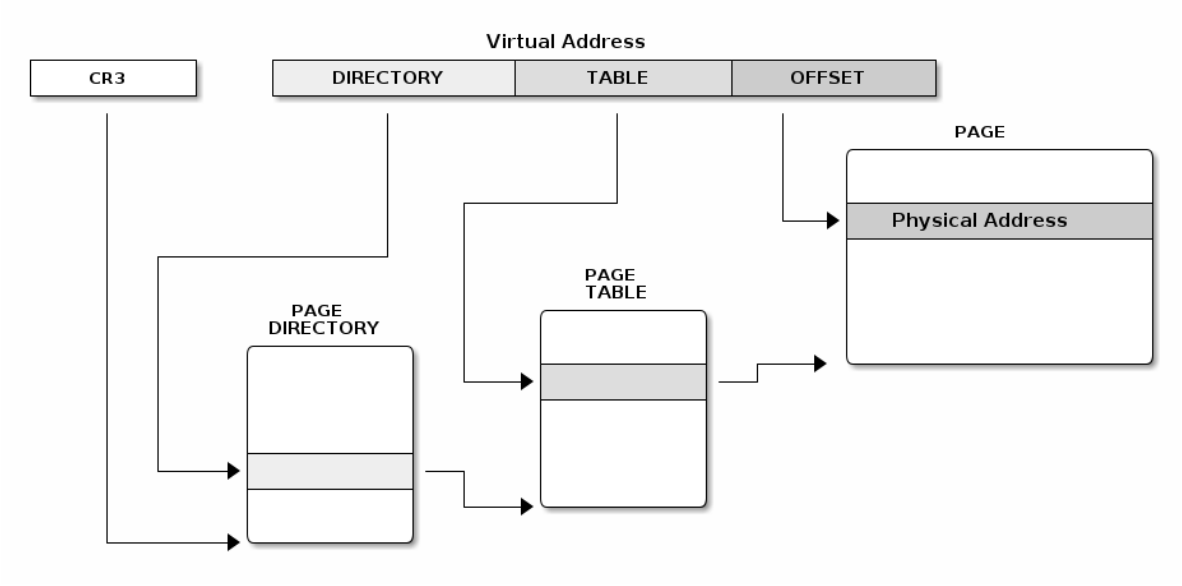

Figure 1: Formato indirizzo virtuale in Linux

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.

Infatti, la Figura 3[**?**] 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 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

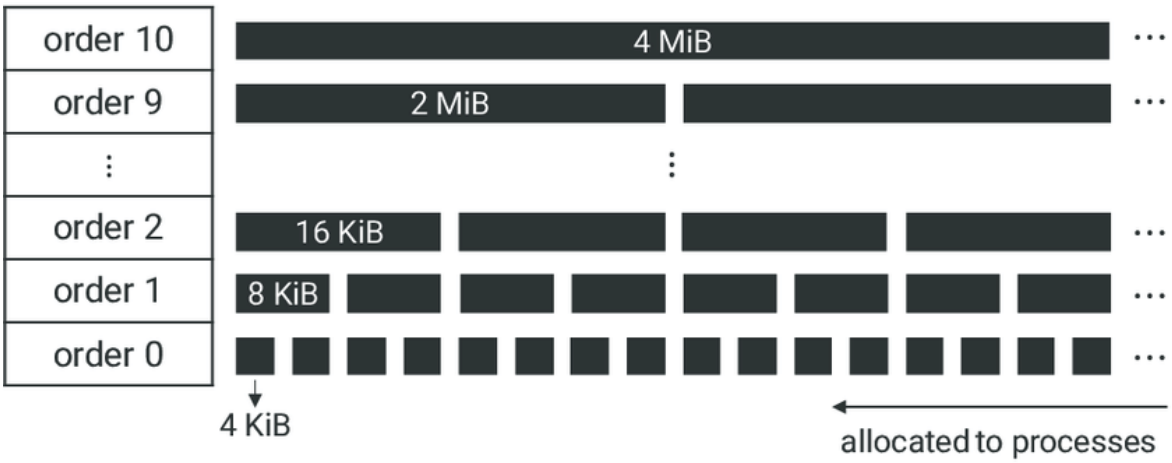

Figure 2: Allocazione delle pagine Linux

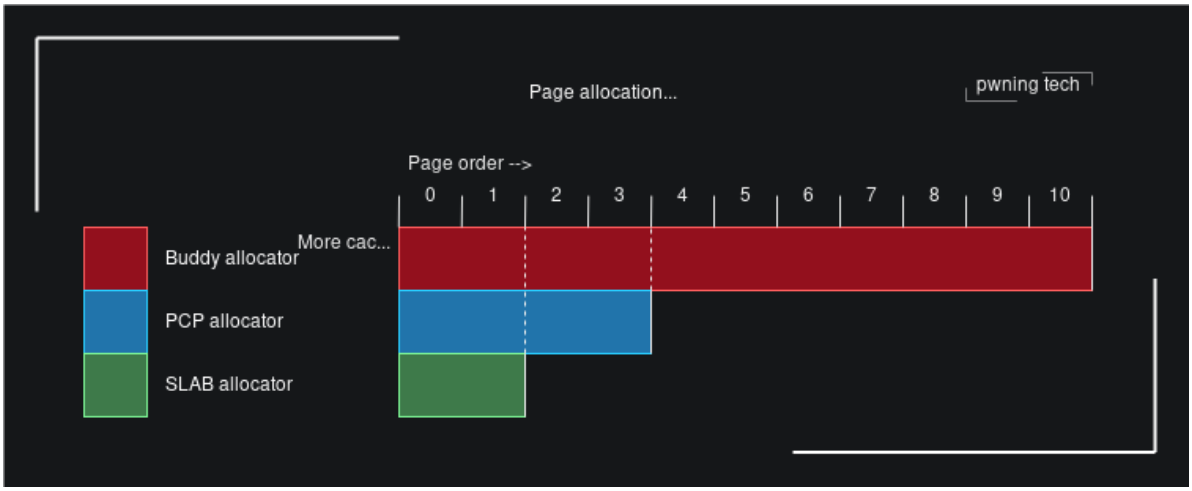

Figure 3: Allocatore utilizzato in base all'ordine delle pagine

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.

1.2.1 Tecnica *Dirty Pagetable*

1.2.2 Superare KASLR

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 (***call_modprobe()***).

```
1 static int call_modprobe(char *module_name, int wait)
2 {
3   struct subprocess_info *info;
4   static char *envp[] = {
5     "HOME=/",
6     "TERM=linux",
7     "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
8     NULL
9   };
10
11  char **argv = kmalloc(sizeof(char *[5]), GFP_KERNEL);
12  if (!argv)
13    goto out;
14
15  module_name = kstrdup(module_name, GFP_KERNEL);
16  if (!module_name)
17    goto free_argv;
18
19  argv[0] = modprobe_path;
20  argv[1] = "-q";
21  argv[2] = "--";
22  argv[3] = module_name;   /* check free_modprobe_argv() */
23  argv[4] = NULL;
24
25  info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
26           NULL, free_modprobe_argv, NULL);
27  if (!info)
28    goto free_module_name;
29
30  return call_usermodehelper_exec(info, wait | UMH_KILLABLE);
31
32 free_module_name:
33  kfree(module_name);
34 free_argv:
35  kfree(argv);
36 out:
37  return -ENOMEM;
38 }
```

Listing 1: "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 4, 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;

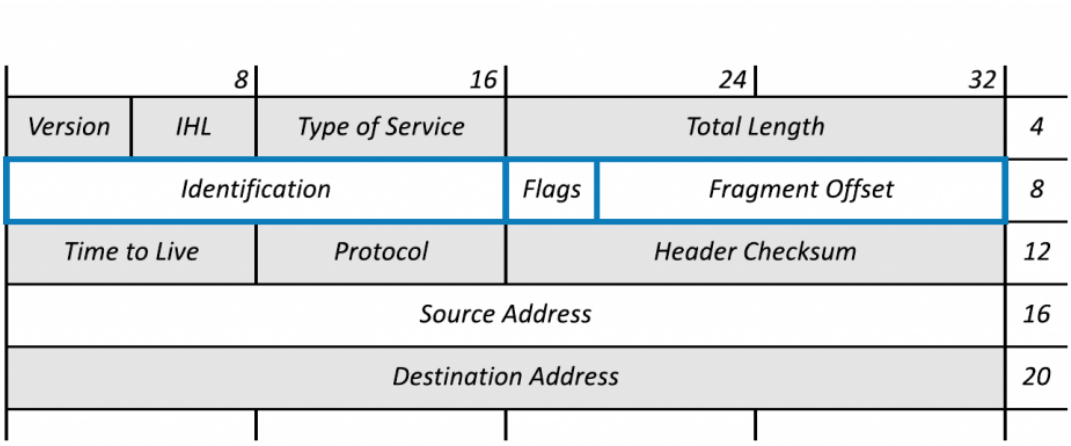

Figure 4: Header di un pacchetto IPv4

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 5), 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:

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

```
4 transmit(iph1, iph1_body, 8);
5
6 iph2->len = sizeof(struct ip_header)*2 + 64;
7 iph2->offset = ntohs(8 >> 3); // don't set IP_MF since this is the last packet
8 memset(iph2_body, 'A', 56);
9 transmit(iph2, iph2_body, 56);
```

Listing 2: "Creazione di due frammenti per un pacchetto IP"

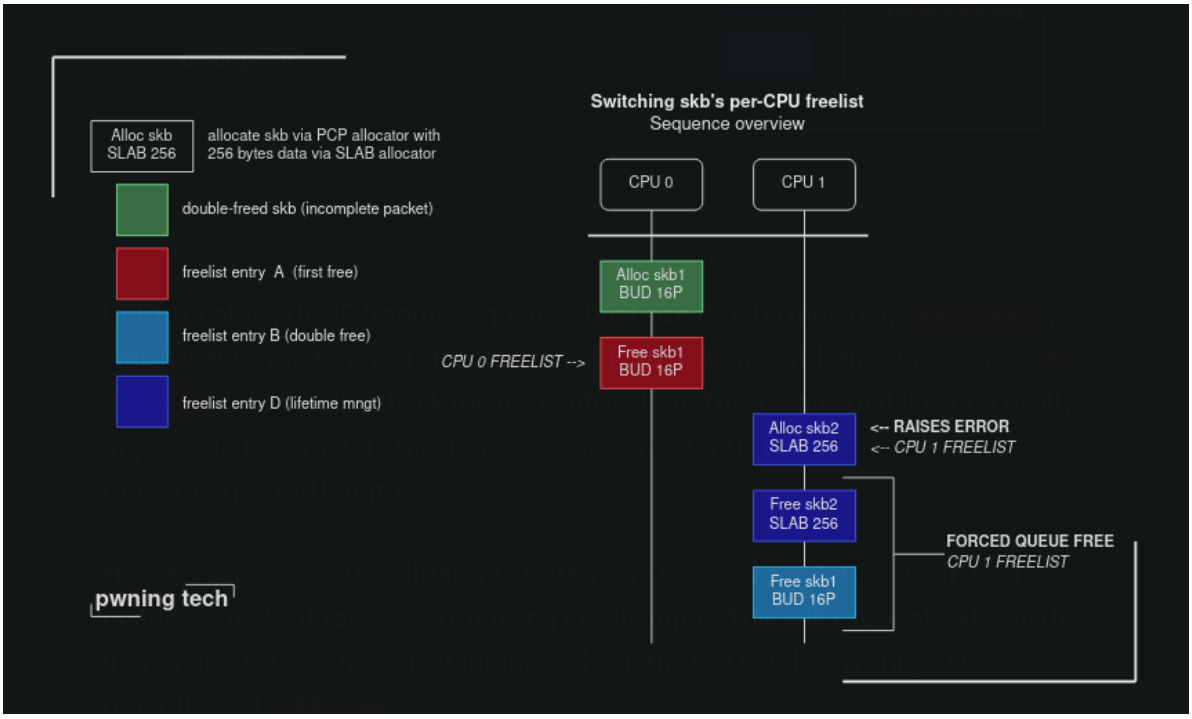

Figure 5: Utilizzo della frammentazione IP per spostare un skb da una CPU ad un'altra

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.

```
1 static void privesc_flh_bypass_no_time(int shell_stdin_fd, int
      shell_stdout_fd)
2 {
3     // ... (skb spray)
4
5     // allocate and free 1 skb from freelist
6   df_ip_header.ip_id = 0x1337;
7   df_ip_header.ip_len = sizeof(struct ip)*2 + 32768 + 24;
8   df_ip_header.ip_off = ntohs((0 >> 3) | 0x2000);  // wait for other
        fragments. 8 >> 3 to make it wait or so?
9   trigger_double_free_hdr(32768 + 8, &df_ip_header);
```

```
10
11      // ... (rest of the exploit)
12 }
```

Listing 3: "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. 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 6.

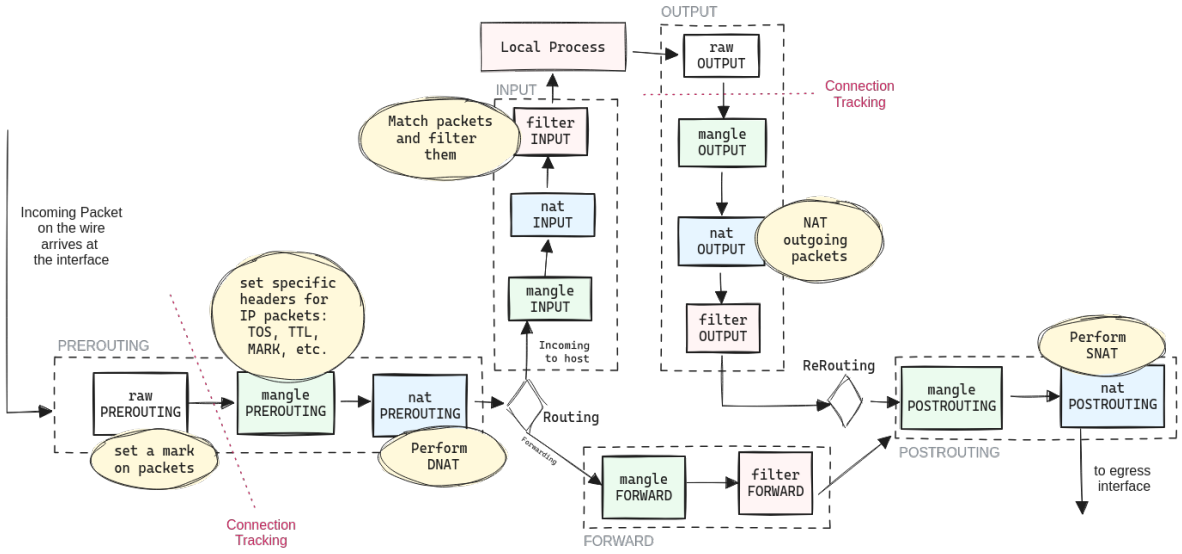

Figure 6: Flusso di un pacchetto all'interno di netfilter

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 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.

```
1 // looping over existing rules when skb triggers chain
2 int nf_hook_slow(struct sk_buff *skb, struct nf_hook_state *state,
3           const struct nf_hook_entries *e, unsigned int s)
4 {
5     unsigned int verdict;
6     int ret;
7
8     for (; s < e->num_hook_entries; s++) {
9         // malicious rule: verdict = 0xffff0000
10        verdict = nf_hook_entry_hookfn(&e->hooks[s], skb, state);
11
12        // 0xffff0000 & NF_VERDICT_MASK == 0x0 (NF_DROP)
13        switch (verdict & NF_VERDICT_MASK) {
14        case NF_ACCEPT:
15            break;
16        case NF_DROP:
17            // first free of double-free
18            kfree_skb_reason(skb,
19                     SKB_DROP_REASON_NETFILTER_DROP);
20
21            // NF_DROP_GETERR(0xffff0000) == 1 (NF_ACCEPT)
22            ret = NF_DROP_GETERR(verdict);
23            if (ret == 0)
24                ret = -EPERM;
25
26            // return NF_ACCEPT (continue packet handling)
27            return ret;
28
29        // [snip] alternative verdict cases
30        default:
31            WARN_ON_ONCE(1);
32            return 0;
33        }
34    }
35
36    return 1;
37 }
```

Listing 4: "Funzione 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).

```
1 static void do_unshare()
```

```
2 {
3     int retv;
4
5     printf("[*] creating user namespace (CLONE_NEWUSER)...\n");
6
7   // do unshare seperately to make debugging easier
8     retv = unshare(CLONE_NEWUSER);
9   if (retv == -1) {
10          perror("unshare(CLONE_NEWUSER)");
11          exit(EXIT_FAILURE);
12    }
13
14    printf("[*] creating network namespace (CLONE_NEWNET)...\n");
15
16    retv = unshare(CLONE_NEWNET);
17    if (retv == -1)
18  {
19    perror("unshare(CLONE_NEWNET)");
20    exit(EXIT_FAILURE);
21  }
22 }
```

2 Attacco

2.1 Preparazione del kernel

2.2 Configurazione kernel: socket Netlink

2.3 Exploit della vulnerabilità

3 Mitigazione della vulnerabilità

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);
--
```