

Iptables match su Segment Routing

Alex Ponzo Pier Francesco Contino

Settembre 2019

Il segment routing

Il **segment routing** è una tecnica di instradamento dove il percorso che il pacchetto deve seguire viene specificato dalla sorgente all'interno dell'header di quest'ultimo, così che il resto della rete ne segua le istruzioni.

Il nostro "progetto" si occupa di riuscire a riconoscere se una lista di indirizzi è contenuta nel path del segment routing, ovvero se un elenco ordinato di nodi appartiene al percorso che il pacchetto deve seguire, così da permettere la definizione di regole del firewall per la gestione di questo tipo di routing.

Iptables e Netfilter

Iptables

Iptables è una utility user-space che permette la configurazione delle tabelle del firewall del kernel linux e delle catene e regole di quest'ultimo.

Netfilter

Netfilter è un framework che permette di creare gestori personalizzati che si occupano di operazioni legate al networking. Per farlo offre degli hooks che permettono a specifici moduli kernel di registrare funzioni di callback, tali funzioni vengono richiamate ogni volta che un pacchetto attraversa il relativo hook.

Il nostro progetto si basa su un'estensione preesistente di iptables che gestisce varie regole di match dedicate al segment routing header.

Nel modificare i relativi file abbiamo deciso di aggiungere una nuova versione in modo da preservare quella antecedente.

Abbiamo sviluppato quindi **la versione 2** del modulo aggiungendo la funzionalità del match sulla lista di indirizzi.

La nostra patch

La nostra patch ha richiesto la modifica di 3 file:

- L'header file contenente le strutture dati relative alla regola:
ip6t_srh.h
- Il file relativo a Iptables: *libip6t_srh.c*
- Il modulo kernel che si occupa del riscontro del match:
ip6t_srh.c

Codice struttura dati: *ip6t_srh.h*

```
struct ip6t_srh2 {
    __u8          next_hdr;
    __u8          hdr_len;
    __u8          segs_left;
    __u8          last_entry;
    __u16         tag;
    struct in6_addr psid_addr;
    struct in6_addr nsid_addr;
    struct in6_addr lsid_addr;
    struct in6_addr psid_msk;
    struct in6_addr nsid_msk;
    struct in6_addr lsid_msk;
    struct in6_addr sid_list_addr[IP6T_SRH_LIST_MAX_LEN]; //NEW FIELD
    struct in6_addr sid_list_msk[IP6T_SRH_LIST_MAX_LEN];  //NEW FIELD
    __u32         sid_list_size; //NEW FIELD
    __u16         mt_flags;
    __u16         mt_invflags;
};
```

Evidenziati dai commenti ci sono le nuove parti: le liste contenenti indirizzi e maschere e la dimensione attuale di esse.

Attualmente la dimensione massima delle liste è di 16.

Codice Iptables: *libip6t_srh.c*

```
case O_SRH_SID_LIST: //NEW
    srhinfo->mt_flags |= IP6T_SRH_SID_LIST;
    struct in6_addr *sid_list_addr_ = NULL;
    struct in6_addr *sid_list_msk_ = NULL;
    xtables_ip6parse_multiple(cb->arg, &sid_list_addr_,
                             &sid_list_msk_, &srhinfo->sid_list_size);
    if (srhinfo->sid_list_size > IP6T_SRH_LIST_MAX_LEN) {
        fprintf(stderr, "size of address list must be <= %u\n",
IP6T_SRH_LIST_MAX_LEN);
        exit(EXIT_FAILURE);
    }
    for (unsigned int i = 0; i < srhinfo->sid_list_size; i++) {
        memcpy(srhinfo->sid_list_addr + i, sid_list_addr_ + i, sizeof(
struct in6_addr));
        memcpy(srhinfo->sid_list_msk + i, sid_list_msk_ + i, sizeof(
struct in6_addr));
    }
    free(sid_list_addr_);
    free(sid_list_msk_);
    if (cb->invert)
        srhinfo->mt_invflags |= IP6T_SRH_INV_SID_LIST;
    break;
```

Questa parte del codice effettua il parsing dei parametri della regola. La funzione fornita dal framework *xtables_ip6parse_multiple* si occupa del parsing di una lista di indirizzi ipv6 con maschere, aggiungendo quella di default (/128) nel caso venga omessa.

Codice Riscontro del match: *ip6t_srh.c*

```
static bool srh_list_match(const struct sk_buff *skb, const struct ipv6_sr_hdr *
    srh, int srhoffs, const struct ip6t_srh2 *srhinfo)
{
    struct in6_addr *list_addr;
    struct in6_addr _list_addr;
    int listoffs, i, j;
    bool match;

    if (srh->first_segment + 1 < srhinfo->sid_list_size)
        return false;
    listoffs = srhoffs + sizeof(struct ipv6_sr_hdr);
    for (i = 0; i <= (srh->first_segment + 1 - srhinfo->sid_list_size); i++) {
        match = true;
        for (j = 0; j < srhinfo->sid_list_size && match; j++) {
            list_addr = skb_header_pointer(skb, listoffs + ((i + j) * sizeof(
                _list_addr)), sizeof(_list_addr), &_list_addr);
            if (ipv6_masked_addr_cmp(list_addr, &srhinfo->sid_list_msk[srhinfo->
                sid_list_size - 1 - j],
                                     &srhinfo->sid_list_addr[srhinfo->
                sid_list_size - 1 - j]))
                match = false;
        }
        if (match)
            return true;
    }
    return false;
}
```

Questo snippet di codice effettua il riscontro della lista nell'header

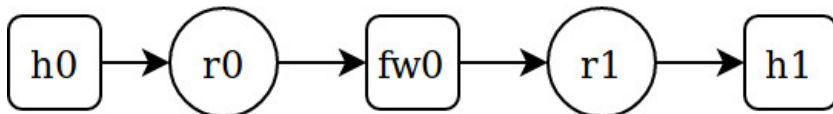
Codice Riscontro del match: *funzionamento*

La struttura dati contenente le informazioni nell'header del pacchetto: *ipv6_sr_hdr* fornisce un campo *first_segment* indicante il primo nodo, ma dato che i segmenti sono salvati in una pila esso è posizionato alla fine della struttura.

Noi abbiamo deciso quindi di leggere al contrario il vettore della regola per il confronto.

Il ciclo annidato è necessario, in caso la lista della regola sia più piccola di quella nell'header, per controllare che la prima sia contenuta nella seconda.

La topologia della rete nel testing



Data questa topologia e le regole contenute nello script un pacchetto ipv6 inviato da h0 (cafe::1) all'indirizzo cafe::2 (h1) effettuerà il percorso:

h0 - r0 - fw0 - r1 - fw0 - r1 - fw0 - r1 - h1

dove r0 ed r1 si occuperanno dell'incapsulamento (dopo h0) e decapsulamento (prima di h1) del pacchetto, oltre che dell'avanzamento della lista di routing tramite i vari passaggi attraverso fw0.

Lo script di testing del codice: srv6_iptables2.sh

Route ipv6 r0

```
$IPP netns exec r0 $IPP -6 route add fc00::1/128 \  
    encap seg6local action End.DX6 nh6 cafe::1 dev veth1
```

```
$IPP netns exec r0 $IPP -6 route add fc00::2/128 via fdff:0:1::2 dev veth2  
$IPP netns exec r0 $IPP -6 route add cafe::2/128 \  
    encap seg6 mode encap segs fc00::2,fc00::3,fc00::4 dev veth2
```

Route ipv6 fw0

```
$IPP netns exec fw0 $IPP -6 route add fc00::2/128 via fdff:1:2::2 dev veth4
```

```
$IPP netns exec fw0 $IPP -6 route add fc00::1/128 via fdff:0:1::1 dev veth3
```

```
$IPP netns exec fw0 $IPP -6 route add fc00::3/128 via fdff:1:2::2 dev veth4  
$IPP netns exec fw0 $IPP -6 route add fc00::4/128 via fdff:1:2::2 dev veth4
```

Route ipv6 r1

```
$IPP netns exec r1 $IPP -6 route add fc00::2/128 \  
    encap seg6local action End.X nh6 fdff:1:2::1 dev veth5  
$IPP netns exec r1 $IPP -6 route add fc00::3/128 \  
    encap seg6local action End.X nh6 fdff:1:2::1 dev veth5  
$IPP netns exec r1 $IPP -6 route add fc00::4/128 \  
    encap seg6local action End.DX6 nh6 cafe::2 dev veth6
```

```
$IPP netns exec r1 $IPP -6 route add fc00::1/128 via fdff:1:2::1 dev veth5  
$IPP netns exec r1 $IPP -6 route add cafe::1/128 \  
    encap seg6 mode encap segs fc00::1 dev veth5
```

Lo script di testing del codice: Le regole di iptables

Le regole di iptables

```
$IPP netns exec fw0 ip6tables -N test
$IPP netns exec fw0 ip6tables -A FORWARD -j test
$IPP netns exec fw0 ip6tables -A test -m srh --srh-sid-list fc00:6::1
$IPP netns exec fw0 ip6tables -A test -m srh --srh-sid-list fc00:6::2
$IPP netns exec fw0 ip6tables -A test -m srh --srh-sid-list fc00:6::2 , fc00:6::3 ,
fc00:6::4
$IPP netns exec fw0 ip6tables -A test -m srh --srh-sid-list fc00:6::2 , fc00:6::3 ,
fc00:6::5
$IPP netns exec fw0 ip6tables -A test -m srh --srh-sid-list fc00:6::2 , fc00:6::4
$IPP netns exec fw0 ip6tables -A test -m srh ! --srh-sid-list fc00:6::2 , fc00
:6::3 , fc00:6::4
```

Il testing del codice

```
fw0
Chain FORWARD (policy ACCEPT 16 packets, 3072 bytes)
pkts bytes target prot opt in out source destination
 16 3072 test all any any anywhere anywhere

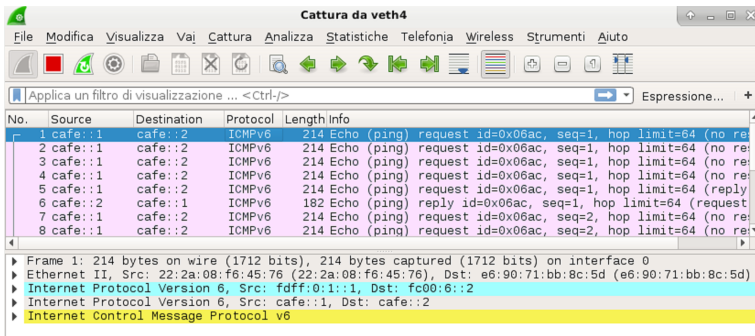
Chain OUTPUT (policy ACCEPT 3 packets, 208 bytes)
pkts bytes target prot opt in out source destination

Chain test (1 references)
pkts bytes target prot opt in out source destination
 4 672 srh sid-list fc00:6::1/128 all any anywhere anywhere
12 2400 srh sid-list fc00:6::2/128 all any anywhere anywhere
12 2400 srh sid-list fc00:6::2/128 all any anywhere anywhere
0 srh sid-list fc00:6::2/128 fc00:6::3/128 fc00:6::4/128
0 all any anywhere anywhere
0 srh sid-list fc00:6::2/128 fc00:6::3/128 fc00:6::5/128
0 all any anywhere anywhere
4 srh sid-list fc00:6::2/128 fc00:6::4/128
4 672 all any anywhere anywhere
srh sid-list ! fc00:6::2/128 fc00:6::3/128 fc00:6::4/128
root@debian:/home/lex/1819-sr-iptables-sl-match#

h0
root@debian:/home/lex/1819-sr-iptables-sl-match# ping6 cafe::2
PING cafe::2(cafe::2) 66 data bytes
64 bytes from cafe::2: icmp_seq=1 ttl=63 time=0.070 ms
64 bytes from cafe::2: icmp_seq=2 ttl=63 time=0.069 ms
64 bytes from cafe::2: icmp_seq=3 ttl=63 time=0.073 ms
64 bytes from cafe::2: icmp_seq=4 ttl=63 time=0.071 ms
^C
--- cafe::2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3067ms
rtt min/avg/max/ndev = 0.069/0.070/0.073/0.010 ms
root@debian:/home/lex/1819-sr-iptables-sl-match#
```

In questo esempio h0 invia 4 ping ad h1:
per ogni ping il nodo fw0 viene attraversato 3 volte per via del
segment routing header **fc00:6::2/128**, **fc00:6::3/128**,
fc00:6::4/128 che segue il percorso sopra indicato e da un
pacchetto di ritorno con segment routing header **fc00:6::1/128**.

L'analisi del percorso dei pacchetti



Cattura da veth4

File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto

Applica un filtro di visualizzazione ... <Ctrl-/> Espressione... +

No.	Source	Destination	Protocol	Length	Info
1	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=1, hop limit=64 (no re
2	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=1, hop limit=64 (no re
3	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=1, hop limit=64 (no re
4	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=1, hop limit=64 (no re
5	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=1, hop limit=64 (reply
6	cafe::2	cafe::1	ICMPv6	182	Echo (ping) reply id=0x06ac, seq=1, hop limit=64 (request
7	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=2, hop limit=64 (no re
8	cafe::1	cafe::2	ICMPv6	214	Echo (ping) request id=0x06ac, seq=2, hop limit=64 (no re

Frame 1: 214 bytes on wire (1712 bits), 214 bytes captured (1712 bits) on interface 0
Ethernet II, Src: 22:2a:08:f6:45:76 (22:2a:08:f6:45:76), Dst: e6:90:71:bb:8c:5d (e6:90:71:bb:8c:5d)
Internet Protocol Version 6, Src: fdff:0:1::1, Dst: fc00:6::2
Internet Protocol Version 6, Src: cafe::1, Dst: cafe::2
Internet Control Message Protocol v6

Posizionandoci sull'interfaccia veth4 (quella di fw0 verso r1) possiamo utilizzare wireshark per tracciare i vari passaggi dei ping ipv6 attraverso quest'ultima; come si può notare, per ogni ping l'interfaccia viene attraversata per 6 volte, 5 delle quali per la "request" con sorgente h0 e destinazione h1 (proprio come specificato nel percorso delle precedenti slide) ed 1 al ritorno per la "reply" con sorgente h1 e destinazione h0.