COSE322-00
시스템프로그래밍
유혁 교수님

2차 과제

Netfilter 를 이용한 Packet Forwarding/Monitoring

제출일: 2019.12.19 (프리데이 사용일수: 0)
개발환경:
가상머신: Oracle VM VirtualBox
운영체제: Ubuntu 16.04 LTS

조장: 2015131102 강수진
조원: 2015131116 배은초

netfilter를 이용한 packet forwarding / monitoring

2015131102 강수진, 2015131116 배은초

목차

Netfilter 와 Hooking

Netfilter Framework는 패킷 맹글링에 대한 프레임워크로, 네트워크 프로토콜 스택과 독립적이다. 패킷 필터링이나 방화벽 설치, 네트워크 주소 해석, 패킷 맹글링 등을 하기 쉽게 한다.

Netfilter는 hook를 정의하고 있다. hook 이란 운영체제나 어플리케이션 등 컴퓨터 프로그램에서 소프트웨어 구성 요소 간의 함수 호출, 메시지, 이벤트 등을 중간에서 가로채거나 변경시키는 행위를 하는 코드를 의미한다. 리눅스 커널의 네트워크 스택엔 packet filtering hook point들이 존재하며, 프로토콜 코드는 hook point를 만나게 되면 Netfilter framework로 넘어간다. hook에는 kernel function을 등록할 수 있으며, 이 hook이 trigger되면 이 함수가 호출된다. 이 함수를 통해 패킷이 어떻게 처리될지 정해줄 수 있다. 함수가 실행완료 되면 패킷은 경로에 따라 계속 흘러간다.

넷필터에 정의된 hook 은 다음 다섯 개이다.

1. **NF_IP_PRE_ROUTING**: 패킷이 네트워크 스택에 들어온 직후 incoming traffic에 의해 트리거 된다. 이 hook은 라우팅 결정 전에 처리된다.
2. **NF_IP_LOCAL_IN**: 만약 패킷의 목적지가 local system인 경우, 패킷이 해당 목적지로 라우팅 되고 난 후에 trigger된다.
3. **NF_IP_FORWARD**: 패킷이 다른 호스트로 포워딩 되는 경우, 그 패킷이 해당 호스트로 포워딩 된 후에 trigger된다.
4. **NF_IP_LOCAL_OUT**: 네트워크 스택을 만나자마자 local에서 생성된 output traffic에 의해 trigger된다.
5. **NF_IP_POST_ROUTING**: 라우팅 발생 이후 실제 네트워크로 가기 직전에 그 outgoing/forwarding traffic에 의해 trigger된다.

넷필터 hook point에 등록하는 함수는 반드시 다음 다섯가지 value 중 하나를 반환해야한다.

- NF_DROP: 패킷 드롭
- NF_ACCEPT: 패킷 통과, 다음 routine으로 간다.
- NF_STOLEN: 패킷에 대한 것을 잊어버리도록 netfilter에게 지시
- NF_QUEUE: 사용자 공간에 패킷을 대기시킴
- NF_REPEAT: 다시 이 hook 호출

hook point에 등록하는 커널 함수에겐 우선순위를 지정해주어야 한다. 우선순위는 함수가 hook trigger시 호출되는 순서를 결정한다. 우선순위는 여러 함수를 호출하기 위한 수단으로 사용된다. 우선순위에 따라 함수는 순서대로 호출한다. 참고로 과제에서는 각 hook point에 하나의 함수만을 연결해주기 때문에 우선순위는 그냥 0으로 설정해주었다.

hook point에 함수를 등록하거나(`nf_register_hook()`), 해제하는 함수(`nf_unregister_hook()`)는 include/linux/netfilter.h 파일에 정의되어 있다. 이 두 함수는 parameter로 nf_hook_ops 구조체를 가지는데, nf_hook_ops 구조체는 우선순위(priority), 커널 함수 포인터(*hook), 네트워크 디바이스 포인터(*dev), 프로토콜 패밀리(pf), hook point(hooknum) 등을 가지고 있다.

IP protocol 네트워킹

IP Protocol 과 Network layer

IP 프로토콜은 OSI 관계 모델의 network layer에서 동작하며, TCP/IP 라는 프로토콜 suite의 일부이다. 그러므로 우선 네트워크 레이어에 대해서 알아본 후, 리눅스에서 ip layer가 어떻게 구현되는지 살펴본다.

Network Layer

Network Layer는 OSI 컴퓨터 네트워킹 모델의 3번째 레이어로, 라우팅을 포함한 패킷 포워딩을 담당한다. 네트워크 레이어는 변동적 길이의 네트워크 패킷을 하나, 또는 그 이상의 네트워크를 통해 source로부터 destination host로 보낸다. Network Layer는 Transport Layer에서 온 요청에 응답하며, Datalink Layer로 요청을 보낸다.

Network Layer의 기능

- Connectionless 통신

Ip 는 connectionless하다, 데이터 패킷은 recipient의 의사와 상관없이 sender로부터 recipient에게 전달된다. 커넥션에 기반한 프로토콜은 상위 레이어에 존재한다.

- Host Addressing

네트워크의 모든 호스트는 그 호스트가 어디에 위치하는지 알려주는 고유의 주소를 갖는다. 이 주소는 일반적으로 계층적 시스템을 통해 할당된다. 인터넷에서 이 주소는 ip 주소라고 불란다.

- 메세지 포워딩

많은 네트워크가 서브 네트워크로 나뉘어 있기 때문에 다른 네트워크로 연결하는 광역 통신을 위해 네트워크는 특별한 호스트를 사용하는데, 이 특별한 호스트를 게이트웨이 또는 라우터라고 보낸다. 라우터는 네트워크 간의 패킷 포워딩을 담당한다.

IP Layer in Linux

IP의 주요 기능은 다음과 같다.

- 네트워크 레이어 메세지 encapsulate, decapsulate
- 데이터그램을 목적지로 라우팅. 라우팅 테이블 업데이트
- fragmentation 및 reassemble 등

단, IP layer는 reliability를 보장하지 않는다.

IP는 논리 주소를 가지고 작동한다. 이 논리주소를 흔히 ip주소라고 부른다. 실제로 물리적 주소를 가지고 프레임을 전달하는 것은 레이어2에서 일어난다. IP layer에는 ARP 라는 프로토콜이 존재한다. ARP는 ip 주소와 mac 주소 간의 해석을 담당한다. 다시 말해, IP레이어가 IP 주소를 가지고 멀티캐스트 방식으로 ARP 리퀘스트를 보내면, 그 IP주소를 가지고 있는 네트워크 인터페이스가 그 리퀘스트를 받고 자신의 물리주소 정보를 보내준다.

IP layer 구현

- IP 레이어에서는 각 패킷의 IP 헤더를 표현하기 위해 iphdr라는 구조체를 갖고 있다.
- input, output, forwarding 경로에 대한 기능들은 net directory의 ipv4_input.c, ipv4_output.c, ipv4_forward.c, ipv4_fragment.c, ipv4_route.c 등에 정의되어있다.
- 커널 안에서 패킷은 sk_buff 구조체로 표현된다. 데이터그램 그 자체와 별도로, 이 구조체는 패킷을 성공적으로 라우팅하기 위해 필요한 모든 정보를 담고 있다. sk_buff는 IP stack에 있는 모든 레어이에 공통적인 구조체이다. 하지만 특정 시간동안, 한 레이어만이 이 구조체의 값을 조정할 수 있다. 몇몇의 경우, sk_buff의 복사본을 여러개 만들어서 동시에 프로세싱할 수도 있다.
- Traversal 논리엔 다양한 포인트에서 패킷을 필터링하기 위한 몇개의 netfilter hook이 포함되어 있다.

IP layer의 함수 및 동작 방법

리눅스 IP layer 는 input, output, forwading 의 세 경로로 나누어질 수 있다. IP layer는 제대로 기능하기 위해 지원되는 프로토콜(예를 들어 ICMP)과 서브 시스템(예를 들어 라우팅 서브 시스템)들과도 interact한다. IP packet 은 IP layer에 들어와서 위에 언급된 세 경로 중 하나를 거친다. 아래는 그 세 경로, Input, Output, Forwarding에 대한 설명이다. 단, 아래 커널 레벨 코드 분석에 포함된 forwarding 관련 내용들은 생략하고, input의 경우 중복되는 부분은 간략히 언급하거나 생략했다.

Output

TCP 레이어에서 최종적으로 `ip_queue_xmit()`을 호출함으로서 IP 레이어의 output과정이 시작된다.

1) `ip_queue_xmit`: 목적지를 결정한다. 여기서 라우팅 캐시가 이용된다. 라우팅에 관련된 내용은 아래에 구체적으로 설명해두었다. 목적지를 결정한 후에는 ip 헤더를 초기화한다. 구체적으로 말하자면 sk_buff 구조체에 헤더를 위한 공간을 만들고 version , 헤더 길이, TOS, TTL, 주소, 프로토콜 필드 등을 채워넣어준다. 그리고 마지막으로 `ip_local_out()` 함수를 호출한다.

2) `ip_local_out()`: 체크섬을 계산한다. 이 함수엔 netfilter hook 인 NF_INET_LOCAL_OUTPUT 이 있어서, 이 훅을 처리한다. 이후 loopback인지를 확인한다. 만약 loopback이면 ip_local_deliver()을 호출한다. 아니라면 `ip_output()`을 호출한다.

3) `ip_output()`: sk_buff에 필드들을 업데이트 해준다. (Dev, protocl 필드 등) 이 함수 내에도 netfilter hook 인 NF_INET_POST_ROUTING이 있다. 마지막으로 `ip-finish_output()` 이 호출된다.

4) `ip_finish_output()`: fragmentation을 한다. 이를 위해 mtu값을 가져온다. 그리고 메시지 길이가 목적지 mtu보다 긴지 확인한다. 길다면 ip_fragment()를 호출하고, 그렇지 않으면 `ip_finish_output2()`를 호출한다.

5) `ip_finish_output2()`: L2헤더를 위한 공간을 만들어준다. 그리고 nexthop을 결정한다. neigh_table구조체에서 맥 주소를 가져온다. 만약 neigh_table에서 매치되는 주소를 찾을 수 없다면 ARP 메시지를 보낸다. 마지막으로 `dev_queue_xmit()`을 호출하여 디바이스로 패킷을 내려보낸다.

Input

1) `ip_rcv()`: sanity 체크 및 넷필터 훅 처리

2) `ip_rcv_finish()`: 라우팅 캐시에서 destination을 찾는다. 목적지를 찾는 이유는 자기한테 온 패킷인지 포워딩해야 할 패킷인지 알아보기 위해서이다. 여기서 local deliver와 forwarding을 판단하게 되는데, 포워딩의 경우 커널 레벨 코드 분석 부분에서 상세히 설명하므로 여기선 local_deliver() 과정을 설명한다.

3) `ip_local_deliver()`: reassemble을 한다. reassemble이 필요한지는 패킷에 있는 framented flag를 확인함으로써 알 수 있다. 이 함수 내에 netfilter hook인 NF_INET_LOCAL_IN이 있어 훅을 처리한다. 마지막으로 `ip_local_deliver_finish()`가 불린다.

4) `ip_local_deliver_finish()`: IP 헤더를 skb에서 제거한다. 상위 레이어의 핸들러를 invoke하는데, 이때 protocol field에 따라 호출되는 함수가 조금씩 다르다. TCP의 경우엔 `tcp_v4_rcv()`가 호출된다.

라우팅

패킷의 origin과 목적지에 따라 패킷은 리눅스 커널안에서 데이터그램의 경로를 계산하기 위해 라우팅된다. Send 과정에서의 라우팅과 Receive 하는 경우의 라우팅을 비교해 설명하자면, send를 할 경우, 패킷의 목적지 주소를 가지고 라우팅 테이블 lookup을 한다. 만약 loopback 패킷이라면 ip_local_deliver() 함수를 호출하여, 로컬로 다시 receive한다. Receive를 하는 경우, 패킷을 데이터 링크 레이어에서 받은 후에 local deliver인지, unicast forward인지, multicast forward인지 확인하여 각 항목에 맞게 함수를 호출한다.

라우팅을 위해서 3가지의 데이터 구조를 가진다:

1) *neigh_table*: 해당 노드와 물리적으로 연결된 호스트의 정보를 담고 있다. Arp 를 통해서 이 테이블이 유지, 업데이트 된다. 일정 시간동안 그 항목을 사용하지 않으면 그 항목은 사라진다. 이를 막기 위해 관리자는 그 항목을 permanent로 지정해 놓을 수 있다.

2) *rt_hash_table*: 라우팅 캐시, 최근에 사용된 루트 순으로 저장한다. 각 항목은 타이버와 카운터가 있어서 더이상 사용되지 않는 경우 삭제된다.

3) *fib_tables*: 라우팅 캐시에서 원하는 목적지에 대한 정보를 찾을 수 없는 경우, 즉 캐시 미스가 난 경우, fib_tables에 가서 정보를 찾는다. 가장 구체적인 넷마스크부터 매칭을 해보며, 매칭되는 것을 찾으면 그 주소를 라우팅 개시에 복사해두며, 라우팅이 필요한 (이 loopup을 하게 만든) 패킷을 찾은 정보로 포워딩할 수 있다.

커널 레벨 코드 분석

ip_rcv

해당 함수에서는 아래와 같은 기능을 수행한다

- host로 addressed되지 않은 패킷은 drop 된다.
- 다음과 같은 sanity cheking을 한다
  - packet이 ip header의 최소 사이즈를 갖고 있는지
  - ip version 4인지
  - checksum이 맞는지
  - packet이 잘못된 길이를 가지고 있진 않은지
- 만약 실제 패킷사이즈가 skb→len이면 skb_trim(skb,iph → total_len)을 invoke 한다.
- NF_INET_PRE_ROUTING 에 해당하는 netfilter hook 을 invoke한다
  - 이 과정에서 ip_rcv_finish이 불린다.

```
return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING,
           net, NULL, skb, dev, NULL,
           ip_rcv_finish);
```

NF_INET_PRE_ROUTING이라는 Netfilter Hook으로 넘겨 패킷을 필터링할지를 검사한 다음에 NF_HOOK() 매크로의 마지막 인자로 주어진 ip_rcv_finish 함수를 호출한다는 의미

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
{
  const struct iphdr *iph;
  struct net *net;
  u32 len;

  /* When the interface is in promisc. mode, drop all the crap
   * that it receives, do not try to analyse it.
   */

// skb->pkt_type은 프레임의 L2 목적지 주소가 받는 인터페이스의 주소와 다를 때
// PACKET_OTHERHOST로 설정되어있다. 일반적으로 그런 패킷은 NIC에서 알아서 버려지지만
// 만약 인터페이스가 promiscuous mode로 되어 있으면 L2의 목적지 주소가 어디이든 모든 패킷을
// 상위 레이어로 올린다. 커널은 모든 패킷에 접근 요청한 sniffer를 invoke한다. 하지만
// ip_rcv는 다른 주소로 온 패킷을 다루지 않으며 간단히 드롭해버린다.
/* 다른 L2 주소로 온 패킷을 받는 건 다른 시스템으로 라우트되어야 하는 패킷을 받는 것과 다르다.
이후 케이스에서 인터페이스의 L2주소를 가지고 현재 수신자와 다른 L3 주소를 가지고 있다.
라우터는 이러한 패킷을 받아 라우트 하도록 설정되어있다.
*/

  if (skb->pkt_type == PACKET_OTHERHOST)
    goto drop;


  net = dev_net(dev);
  IP_UPD_PO_STATS_BH(net, IPSTATS_MIB_IN, skb->len);


/* 패킷의 참조 count가 1 이상인지 확인한다. 이는 커널의 다른 부분들이 그 버퍼를 참조하고 있다는 뜻이다.
sniffer들과 다른 유저들이 이 패킷에 관심이 있을 수 있기 때문에 각 패킷은 reference count를
가지고 있다. ip_rcv 함수를 부르는 netif_receive_skb 함수는
프로토콜 핸들러를 호출하기 전에 reference count를 늘린다. 만약 핸들러가 reference counter가
1이상인 것을 보면 자기가 사용할 버퍼 복사본을 만들어서 자신이 패킷을 바꿀 수 있게 한다.
후에 접근하는 핸들러들은 원래의, 바뀌지 않은 버퍼를 받게 된다. 만약 복사가 필요하지만
메모리 할당에 실패하면 패킷은 드롭된다. */

  skb = skb_share_check(skb, GFP_ATOMIC);
  if (!skb) {
    IP_INC_STATS_BH(net, IPSTATS_MIB_INDISCARDS);
    goto out;
  }

/* pskb_may_pull은 skb_data가 가리키는 공간이 적어도 ip헤더만큼 큰 데이터 블럭을 가질 수 있도록
 보장한다. 다시말해, IP헤더가 kmalloc'd 공간에 존재함을 보장한다. 왜냐면
각 IP패킷(fragmentation을 포함한)은 완전한 IP 헤더를 포함하고 있어야 하기 때문이다.
```

```
조건에 만족하면 할 게 없다. 그렇지 않은 경우 잃어버린 부분은 데이터 조각에서 복사되고
skb_shinfo(skb)→frags[]. 에 저장된다. 만약 이 과정에 실패하면 함수는 에러를 내며 종료된다.
만약 성공하면 함수는 iph를 다시 초기화한다. 왜냐면 pskb_may_pull이 버퍼 구조를 바꿀수도 있기
때문이다.  */

  if (!pskb_may_pull(skb, sizeof(struct iphdr)))
    goto inhdr_error;

  iph = ip_hdr(skb);

  /*
   *  RFC1122: 3.2.1.2 MUST silently discard any IP frame that fails the checksum.
   *
   *  Is the datagram acceptable?
   *
   *  1.  Length at least the size of an ip header
   *  2.  Version of 4
   *  3.  Checksums correctly. [Speed optimisation for later, skip loopback checksums]
   *  4.  Doesn't have a bogus length
   */

/* 그 후엔 ip 헤더에 sanity check을 한다. 기본적인 ip헤더의 사이즈는 20바이트이다.
그리고 헤더에 저장된 사이즈는 32비트의 곱(4바이트의 곱)으로 표현되기 때문에 그 값이
 5보다 작다는 것은 에러가 발생했다는 뜻이다. 두 번째 체크는 ipv4를 체크하기 위한 것이다.
현재 ip 프로토콜엔 두가지 버전 (4,6)이 존재한다. if 문은 패킷이 ipv4인 것을 보장한다.
하지만 두 프로토콜이 두 개의 다른 함수에 의해 핸들되기 때문에 ip_rcv 함수는 ipv6에서 애초에
 절대 불렸어선 안된다. */

  if (iph->ihl < 5 || iph->version != 4)
    goto inhdr_error;

  BUILD_BUG_ON(IPSTATS_MIB_ECT1PKTS != IPSTATS_MIB_NOECTPKTS + INET_ECN_ECT_1);
  BUILD_BUG_ON(IPSTATS_MIB_ECT0PKTS != IPSTATS_MIB_NOECTPKTS + INET_ECN_ECT_0);
  BUILD_BUG_ON(IPSTATS_MIB_CEPKTS != IPSTATS_MIB_NOECTPKTS + INET_ECN_CE);
  IP_ADD_STATS_BH(net,
      IPSTATS_MIB_NOECTPKTS + (iph->tos & INET_ECN_MASK),
      max_t(unsigned short, 1, skb_shinfo(skb)->gso_segs));

/* 전과 같은 체크를 반복한다. 하지만 이번엔 완전한 ip 헤더 사이즈를 사용한다.
(옵션을 포함한) 만약 ip헤더가 iph→ihl의 사이즈를 요구한다면 패킷은 적어도 iph→ihl만큼 길어야 한다.
 이 체크가 지금까지 남아있는 이유는 함수가 우선적으로 기본 헤더(옵션 없는 헤더)가 잘리지 않았음이
 보장되어야 하고, 기본적인 sanity check이 그것으로부터 무언가(이 경우엔 ihl)를 읽기 전에 일어나야
 하기 때문이다. */

  if (!pskb_may_pull(skb, iph->ihl*4))
    goto inhdr_error;

/* 이 두 프로토콜의 consistency 체크가 실행된 후 함수는 이게 헤더에 있는 체크섬과 맞는지
확인하기 위해 체크섬을 계산한다. 체크섬을 확인하기 전, iph에 다시 헤더 포인터를 load한다.
 pskb_may_pull()에서 skb를 다시 만들었을 수 있기 때문이다.
체크섬이 맞지 않으면 패킷은 드롭된다.  */

  iph = ip_hdr(skb);

  if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
    goto csum_error;

/* 체크섬 후에 두 가지의 sanity check이 이어진다.
  - 버퍼의 길이(받은 패킷)가 ip헤더에 기록된 길이와 같거나 큰지 확인한다.
  - 패킷 사이즈가 적어도 ip 헤더 사이즈만큼 큰지 확인한다.
첫 번째 체크는 L2 프로토콜(예를 들어 이더넷)이 payload를 pad out할 수도 있어서
ip payload 이후에 몇 바이트가 더 있을 수도 있기 때문에 필요하다.
(이런 상황은 L2 프레임 사이즈가 프로토콜에서 요구하는 최소 프레임 사이즈보다
 작은 경우 발생하곤 한다. 이더넷 프레임은 최소 64바이트 프레임 길이를 가지고 있다)
이런 경우 패킷은 ip 헤더에 기록된 길이보다 길어보일 것이다.*/

  len = ntohs(iph->tot_len);
  if (skb->len < len) {
    IP_INC_STATS_BH(net, IPSTATS_MIB_INTRUNCATEDPKTS);
    goto drop;
  } else if (len < (iph->ihl*4))
    goto inhdr_error;


/* 두 번째 체크는 ip 헤더가 fragment되면 안된다는 점. 그리고 매 ip fragment는 적어도 하나의
ip 헤더를 항상 가지고 있어야 한다는 점에서 기원한다. 이 체크는 거의 실패하지 않는다.
 이 체크의 실패는 체크섬이 망가진 패킷에서 계산되었지만 우연히 원래 패킷과 같은 체크섬을
계산했다는 의미이다. (i.e. 체크섬이 에러를 발견하지 못했다)*/

//참고(두 문단)
/* 라우트에 관련된 최소 MTU는 사실 68이다. (RFC 791에서 정해진) IP 헤더는 최대 60바이트까지
길어질 수 있고, 최소의 fragment 길이(마지막 패킷 예외)는 8바이트이다. 즉, 모든 ip 라우터가
 68바이트의 패킷은 더이상의 fragmentation없이 포워딩할 수 있어야 한다는 뜻이다.   */

/* 이 함수 내에서 일어나는 sanity 체크는 시스템의 안정성을 위해 매우 중요하다.
만약에 우연히 sk_buff 구조체가 잘못 초기화되었거나, ip 헤더 자체가 망가졌을 경우,
커널이 패킷은 잘못된 방법으로 처리하거나, 유효하지 않은 메모리 공간에 접근하여 crash가
```

```
발생할 수 있다. */


/* L2 프로토콜은 패킷을 특정의 최소 길이만큼 pad out했을 수 있다.
pskb_trim_rcsum은 이런 경우가 발생했는지와 만약 그랬다면 __pskb_trim으로 패킷을 올바른
사이즈로 다듬고, 받은 NIC가 계산했을 경우를 고려해 L4 체크섬을 invalidate한다.
L4 체크섬이 네트워크 카드에서 하드웨어적으로 계산될 때 네트워크 카드의 능력에 따라
 L2 패딩을 빼지 못하고 포함할 수도 있다.여기에서 그런 상황인지, 아니면 안전한 상황인지 알 수
없기 때문에 pskb_trim_rcsum은 간단하게 체크섬을 invalidate하고 L4 프로토콜이 다시 계산하도록
강제한다.  */

  /* Our transport medium may have padded the buffer out. Now we know it
   * is IP we can trim to the true length of the frame.
   * Note this now means skb->len holds ntohs(iph->tot_len).
   */
  if (pskb_trim_rcsum(skb, len)) {
    IP_INC_STATS_BH(net, IPSTATS_MIB_INDISCARDS);
    goto drop;
  }

  skb->transport_header = skb->network_header + iph->ihl*4;

  /* Remove any debris in the socket control block */
  memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));

  /* Must drop socket now because of tproxy. */
  skb_orphan(skb);

/* 라우팅 결정이나 옵션 핸들링은 지금까지 되지 않았다. 이 일은 ip_rcv_finish의 몫이다.
이 함수는 netfilter subsystem을 부르며 끝이 나는데 이 부분은 이런 뜻을 가진다:

skb는 dev 디바이스에서 받은 패킷을 가지고 그 패킷이 그 이동을 계속해도 되는지 확인하고,
변화가 필요한지 확인하라. 네트워크 스택의 NF_IP_PRE_ROUTING 포인트에서 이를
요구하고 있다. 즉, 패킷이 수신되었지만 아직 라우팅 결정이 내려지지 않았다.
만약 패킷을 드롭하지 않기로 결정했다면 ip_rcv_finish를 실행하라. */

  return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING,
           net, NULL, skb, dev, NULL,
           ip_rcv_finish);

csum_error:
  IP_INC_STATS_BH(net, IPSTATS_MIB_CSUMERRORS);
inhdr_error:
  IP_INC_STATS_BH(net, IPSTATS_MIB_INHDRERRORS);
drop:
  kfree_skb(skb);
out:
  return NET_RX_DROP;
}
```

ip_rcv_finish

`ip_rcv`에 의해 `NF_INET_PRE_ROUTING` 후에 불리는 함수이다.

해당 함수에서는 아래와 같은 기능을 수행한다

- 패킷이 local하게 전달될 것인지, forward 될것인지를 결정해야한다. 후자의 경우는 나가는 디바이스(egress device)와 다음 홉을 찾아야한다. 이는 routing result에 따라 결정되고 아래와 같은 세가지 경우의 수가 있다.
  - ip_local_deliver()

    local host로 들어오는 unicast packet에 대해서는 dst 의 input callback 은 ip_local_deliver
  - ip_forward()

    들어오는 unicast packet이 forwarded되어야 하면 input callback 이 ip_forward로 셋된다.

    이번 과제에서는 들어오는 패킷에 대해 ip 주소를 바꿔주었기 때문에 해당 패킷은 forward 되어야하고 아래와 같이 세팅된 상태이다.

    ```
    rth->dst.input = ip_forward;
    ```
  - ip_mr_input() → forwarding(multicast)

    멀티캐스트 패킷에대해서는 input callback이 ip_mr_input으로 셋될수도 있다.
- 몇몇 IP 옵션에 대한 parsing과 처리가 필요하다.

- 만약 아이피 헤더가 options를 포함하고 있다면 ip_option 구조체가 만들어진다.

```
static int ip_rcv_finish(struct net *net, struct sock *sk, struct sk_buff *skb)
{
  const struct iphdr *iph = ip_hdr(skb);
  struct rtable *rt;

  if (sysctl_ip_early_demux && !skb_dst(skb) && !skb->sk) {
    const struct net_protocol *ipprot;
    int protocol = iph->protocol;

    ipprot = rcu_dereference(inet_protos[protocol]);
    if (ipprot && ipprot->early_demux) {
      ipprot->early_demux(skb);
      /* must reload iph, skb->head might have changed */
      iph = ip_hdr(skb);
    }
  }

  /*
   *  Initialise the virtual path cache for the packet. It describes
   *  how the packet travels inside Linux networking.
   */
  if (!skb_valid_dst(skb)) {
    int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
               iph->tos, skb->dev);
    if (unlikely(err)) {
      if (err == -EXDEV)
        NET_INC_STATS_BH(net, LINUX_MIB_IPRPFILTER);
      goto drop;
    }
  }

//Traffic Control에서 쓰일 몇몇 statistics을 업데이트 한다.
#ifdef CONFIG_IP_ROUTE_CLASSID
  if (unlikely(skb_dst(skb)->tclassid)) {
    struct ip_rt_acct *st = this_cpu_ptr(ip_rt_acct);
    u32 idx = skb_dst(skb)->tclassid;
    st[idx&0xFF].o_packets++;
    st[idx&0xFF].o_bytes += skb->len;
    st[(idx>>16)&0xFF].i_packets++;
    st[(idx>>16)&0xFF].i_bytes += skb->len;
  }
#endif

//ip header의 길이가 20 바이트보다 크면, 처리할 옵션이 없다는 것을 의미한다.
//ip_rcv_options는 그 안에서 drop이 불리면 true를, 나머지 경우에는 false를 반환한다.
//위의 함수 안에서는 skb_cow가 불린다. 그 이유는 후에 다른 프로세스와 shared되면 buffer의 복사본을 만들기 위함이다.
//옵션을 처리할 것이기 때문에, 또 ip 헤더를 바꿔야할 수 있기 때문에 해당 버퍼의 독점적인 소유권이 보장되어야 한다.
  if (iph->ihl > 5 && ip_rcv_options(skb))
    goto drop;

  rt = skb_rtable(skb);
  if (rt->rt_type == RTN_MULTICAST) {
    IP_UPD_PO_STATS_BH(net, IPSTATS_MIB_INMCAST, skb->len);
  } else if (rt->rt_type == RTN_BROADCAST)
    IP_UPD_PO_STATS_BH(net, IPSTATS_MIB_INBCAST, skb->len);

  return dst_input(skb);

drop:
  kfree_skb(skb);
  return NET_RX_DROP;
}
```

ip_forward

ip_forward()는 몇 가지 validation check를 한다. 예를 들어 pkt type이 PACKET_HOST인지 확인지 확인하고 아니면 패킷을 드롭한다. ip_forward()는 ttl도 확인한다. 만약 ttl이 1보다 작거나 같으면 ICMP 메세지로 패킷의 수명이 만료되었다고 알려야 한다. 패킷 길이(MAC 헤더 포함)가 너무 큰데 (skb→len > mtu) 프레그멘테이션을 허용하지 않는다면, ICMP 메세지를 보내기도 한다. (ICMP_FRAG_NEEDED)

skb_cow(skb, headroom)가 불려서 output device를 위한 충분한 MAC 헤더 공간이 남아있는지 확인한다. 만약 공간이 부족하다면 pskb_expand_head함수를 호출하여 충분한 공간을 만든다.

그리고 IP packet의 TTL 을 1 줄인다. 여기서 사용되는 ip_decrease_ttl()은 헤더 체크섬도 변경해준다. 마지막으로 netlfilter hook NF_INET_FORWARDING이 불린다.

```
int ip_forward(struct sk_buff *skb)
{

  u32 mtu;
  struct iphdr *iph;  /* Our header */
  struct rtable *rt;  /* Route we use */
  struct ip_options *opt  = &(IPCB(skb)->opt);
  struct net *net;

/*해당 체크가 이뤄지는 이유는 패킷이 host의 L2에 주소로 되어있었다는걸 확실히 하기 위해서이다.
skb->pkt_type은 L2 layer에서 초기화 되고 frame의 타입을 정의한다.
해당 인터페이스의 L2 주소로 frame이 addressed 되었을때 PACKET_HOST 값을 할당받는다.
만약 lower-level function이 제대로 동작했다면 해당 체크는 필요없지만,
혹여나 받으면 안됐던 패킷이 들어올때에 대한 에러 처리로 남겨 놓는다.*/
  if (skb->pkt_type != PACKET_HOST)
    goto drop;

  if (unlikely(skb->sk))
    goto drop;

  if (skb_warn_if_lro(skb))
    goto drop;

  if (!xfrm4_policy_check(NULL, XFRM_POLICY_FWD, skb))
    goto drop;

/*만약 헤더에 Router_Alert option이 없거나,
 존재하지만 interested processes들이 running하지 않은 상태라면
(ip_call_ra_chain가 FALSE를 반환하는 케이스),
ip_forward를 계속한다.*/
  if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
    return NET_RX_SUCCESS;
/*패킷을 포워드하기 때문에, 전체적으로 L3 layer에서 동작한다.
따라서 L4 checksum에 대해선 고려하지 않아도 된다.
CHECKSUM_NONE을 사용해 현재 체크썸이 ok 상태임을 나타낸다.
만약 transmission 전후에 ip 헤더나 tcp 헤더나 페이로드에 대한 처리가 바뀐다면
그때 커널에서 checksum을 재계산할것이다.*/
  skb_forward_csum(skb);
  net = dev_net(skb->dev);

  /*
   *  According to the RFC, we must first decrease the TTL field. If
   *  that reaches zero, we must reply an ICMP control message telling
   *  that the packet's lifetime expired.
   */
/*실제 포워딩 프로세스는 ttl field를 줄임으로써 시작한다.
ip protocol에 따르면 ttl 이 0이 되면 패킷은 드롭되고
그 사실을 알리기 위해 source로 ICMP 메시지를 보낸다.
(현재 <= 1로 체크하는 이유는 후에 ttl 감소시키고나면 0이 될것이기 때문)
ttl 필드가 이곳에서 감소하지 않고 몇라인 후에 실행되는 이유는
패킷이 이 시점에서 sniffer와 같은 다른 시스템에서 share되고 있을지도 모르기 때문이다.
이런 상황에서 헤더는 바뀌면 안된다.*/
  if (ip_hdr(skb)->ttl <= 1)
    goto too_many_hops;

  if (!xfrm4_route_forward(skb))
    goto drop;

/*rt 는 rtable 타입의 데이터 구조를 가리킨다.
rtable은 next hop (rt_gateway)을 포함해 forwarding 엔진에 필요한 모든 정보를 포함한다.
만약 ip header가 Strict Source Route 옵션을 포함하고,
다음 홉이 gateway면 Source Routing option은 실패하고 패킷은 드롭된다.
rt_uses_gateway는 다음 홉이 gateway면 1을 반환하고, direct route면 0을 반환한다.*/
  rt = skb_rtable(skb);

  if (opt->is_strictroute && rt->rt_uses_gateway)
    goto sr_failed;

  IPCB(skb)->flags |= IPSKB_FORWARDED;
  mtu = ip_dst_mtu_maybe_forward(&rt->dst, true);
  if (ip_exceeds_mtu(skb, mtu)) {
    IP_INC_STATS(net, IPSTATS_MIB_FRAGFAILS);
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
        htonl(mtu));
    goto drop;
  }
/*해당 상황에서는 또다른 ICMP message가 센더에게 전송된다.
대부분의 sanity check 가 충족된 후에
함수는 패킷 헤더에서 한 비트를 업데이트하고 ip_forward_finish에게 준다.
버퍼의 content를 수정할 예정이기 때문에 local copy를 만들어야 한다.
복사는 패킷이 shared 되거나 L2 헤더를 저장하기에 패킷의 헤드의 가용 공간이 부족할때만
skb_cow에 의해 이뤄진다.
만약 패킷이 shared 되지 않으면 안전하게 수정될 수 있다 */
  /* We are about to mangle packet. Copy it! */
  if (skb_cow(skb, LL_RESERVED_SPACE(rt->dst.dev)+rt->dst.header_len))
```

```
    goto drop;
  iph = ip_hdr(skb);

 //ttl 이 줄어든다.
 //IP checksum도 업데이트 한다.
  /* Decrease ttl after skb cow done */
  ip_decrease_ttl(iph);

/*만약 제안된것보다 더 좋은 다음 홉을 사용할 수 있다면
originating host는 ICMP REDIRECT에 의해 알림을 받는다.
하지만 이는 originating host가 source routing을 request 하지 않았을 때에 한정된다.
opt->srr 은 source routing이 requested 되었음을 의미한다.
즉 originating host가 더 나은 라우트를 찾는 것에 관심을 가지지 않는다는 얘기이다.
RTCF_DOREDIRECT flag는 cached route에 set되는데,
패킷의 소스가 ICMP REDIRECT message를 받아야한다는 것을 나타내기 위해 사용된다.*/
  /*
   *  We now generate an ICMP HOST REDIRECT giving the route
   *  we calculated.
   */
  if (IPCB(skb)->flags & IPSKB_DOREDIRECT && !opt->srr &&
      !skb_sec_path(skb))
    ip_rt_send_redirect(skb);

//priority 필드는 ip 헤더의 Type of Service field를 이용해 이곳에서 set된다.
//우선 순위는 추후 QoS layer의 Traffic Control에 이용된다.
  skb->priority = rt_tos2priority(iph->tos);

//만약 포워딩을 막는 필터링 규칙이 없다면 Netfilter에 ip_forward_finish를 실행시킬 것을 요구하며 종료된다.
  return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD,
           net, NULL, skb, skb->dev, rt->dst.dev,
           ip_forward_finish);

sr_failed:
  /*
   *  Strict routing permits no gatewaying
   */
   icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
   goto drop;

too_many_hops:
  /* Tell the sender its packet died... */
  IP_INC_STATS_BH(net, IPSTATS_MIB_INHDRERRORS);
  icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
drop:
  kfree_skb(skb);
  return NET_RX_DROP;
}
```

ip_forward_finish

이 함수에 도달했다는 것은 패킷이 모든 체크를 통과했고 다른 시스템으로 전송될 준비가 다 되었다는 것을 의미한다. IP헤더에서 가능한 두 가지 옵션(Router Alert and Strict Source Alert) 은 이미 핸들링이 되었다.

```
static int ip_forward_finish(struct net *net, struct sock *sk, struct sk_buff *skb)
{
 //
  struct ip_options *opt  = &(IPCB(skb)->opt);

// STATS 업데이트
  __IP_INC_STATS(net, IPSTATS_MIB_OUTFORWDATAGRAMS);
  __IP_ADD_STATS(net, IPSTATS_MIB_OUTOCTETS, skb->len);

#ifdef CONFIG_NET_SWITCHDEV
  if (skb->offload_l3_fwd_mark) {
    consume_skb(skb);
    return 0;
  }
#endif

// 옵션에 필요한 마지막 작업들을 위한 함수.
// 플래그, offset 등을 확인하여 어떤 작업이 남았는지 확인하고 수행한다.
// IP header에 업데이트가 있었을 수 있기 때문에 IP checksum을 다시 계산한다.
  if (unlikely(opt->optlen))
    ip_forward_options(skb);

  skb->tstamp = 0;
  return dst_output(net, sk, skb);
  // dst_output함수는 skb->dest->output(skb)함수이다. 이 함수는 결국 ip_output()함수이다.
}
```

`dst_output()` 는 local에서 만들어졌든, 다른 호스트에서 만들어진 패킷을 포워딩하든 거치게 되는 함수이다. `dst_output()` 이 불리는 시점에서 IP헤더는 완성되어있다. 이때의 패킷은 전송에 필요한 정보를 모두 가지고 있으며 로컬 시스템이 더해줘야 할 정보들도 모두 탑재되어 있는 상태다. `dst_output()` 함수는 가상 함수인 output함수를 `skb->dst->output(skb)` 로 invoke하는데, 만약 목적지 주소가 anycast라면 이 output 가상함수는 `ip_output()` 으로 초기화 되어있다. 덧붙이자면 `dst_output()` 에선 fragmentation이 일어난다.

ip_output

```
int ip_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
// sk_buff에 dev필드을 업데이트 해준다.
  struct net_device *dev = skb_dst(skb)->dev, *indev = skb->dev;

// STATS 업데이트
  IP_UPD_PO_STATS(net, IPSTATS_MIB_OUT, skb->len);

  skb->dev = dev;
  skb->protocol = htons(ETH_P_IP);

// HOOK 포인트로 가게된다. hook에서 accept하면 ip_finish_output이 호출된다.
// NF_HOOK_COND는 조건을 만족시키지 못하면 okfn(여기서 ip_finish_output)를 바로 실행시킨다.
// 조건을 만족시키지 못하면 nf_hook 함수를 실행하고 일반 NF_HOOK과 같이 동작한다.
  return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING,
          net, sk, skb, indev, dev,
          ip_finish_output,
          !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

작성한 소스 코드 설명

각 Hooking Point 선택 이유

PRE_ROUTING

패킷 포워드를 위한 작업을 `NF_INET_PRE_ROUTING` 에서 수행했다. 그 이유는 아래와 같다.

패킷이 net device driver에서 ip 레이어로 넘어오면 처음 불리는 함수가 `ip_rcv()` 인데, `ip_rcv()` 에서는 후킹 포인트로 훅을 할 수 있다. 해당 후킹 포인트가 바로 `NF_INET_PRE_ROUTING` 이다.

`NF_INET_PRE_ROUTING` hook은 라우팅 테이블을 거치기 전이기 때문에 이곳에서 ip 주소를 바꾸어 주어야 한다. 해당 정보를 바탕으로 이후에 불리는 `ip_rcv_finish` 에서 패킷이 local로 갈지 forward될지 판단하게 된다.

FORWARD

ip_forward() 함수에서 FORWARD hook point로 접근할 수 있다. 이 hook point에서 패킷 정보를 프린트함으로써 패킷이 제대로 변조되었고, Forward되었는지 확인할 수 있다.

POST_ROUTING

ip_output() 함수에서 POST_ROUTING hook point 로 접근할 수 있는데 이 hook point에서 패킷이 제대로 변조되었는지 확인하고, 제대로 routing 되고 있는지 확인하려고 이 후킹 포인트에서 패킷 정보를 프린트 해 본다.

바꾼 부분

`NF_INET_PRE_ROUTING` 해당 함수 안에서 우선 아래와 같은 작업을 통해 `iphdr` 구조체와 `tcphdr` 구조체를 얻었다.

```
struct iphdr *iph = ip_hdr(skb);
struct tcphdr *tcph = tcp_hdr(skb);
```

`iphdr` 구조체 안에는 다음과 같이 정의되어 있는데 여기서 `daddr` 는 destination ip address를 나타낸다.

```
struct iphdr {
  __u8  tos;
  __be16  tot_len;
  __be16  id;
  __be16  frag_off;
  __u8  ttl;
  __u8  protocol;
```

```
  __sum16 check;
  __be32  saddr;
  __be32  daddr;
};
```

따라서 들어온 패킷의 `daddr` 값을 다른 곳으로 바꿔주면 해당 패킷을 로컬로 보내지 않고 forward 시킬 수 있다. 이때 네크워크에서는 빅엔디안을 표준으로 삼고 있기 때문에 `htonl` 를 통해 바이트 순서를 맞춰줘야한다.

또한 tcp 헤더의 `dest` 에 접근함으로써 destination port 번호를 바꿀 수 있다. 마찬가지로 `htons` 를 통해 Host(사용자 컴퓨터에서 사용하는 엔디안)에서 Network(빅엔디안 방식)로 바이트 순서를 맞춰줘야하는데 다만 ip와 다른 점은 short를 사용한다는 점이다.

`tcphdr` 구조체는 다음과 같이 정의된다.

```
struct tcphdr {
  __be16  source;
  __be16  dest;
  __be32  seq;
  __be32  ack_seq;
  __be16  window;
  __sum16 check;
  __be16  urg_ptr;
};
```

자세한 코드는 아래와 같다.

들어온 패킷의 프로토콜이 tcp이고 port가 유저 버퍼에서 받은 inputPort 번호와 같다면 해당 패킷의 destination ip와 destination port 번호를 변경한다.

```
static  unsigned int my_hook_fn_pre(void *priv, struct sk_buff *skb, const struct nf_hook_state *state) {
    struct iphdr *iph = ip_hdr(skb);
    struct tcphdr *tcph = tcp_hdr(skb);
    printk("PRE_ROUTING[(%u, %u, %u, %pI4, %pI4)]\n",iph->protocol, ntohs(tcph->source), ntohs(tcph->dest),&iph->saddr, &iph->d
    if (iph->protocol == IPPROTO_TCP)
    {
        if ((tcph->source) == htons(inputPort))
        {
            char *ip = "111.1.1.0";
            unsigned int IpAddr;
            unsigned int changedIpAddr;
            ipStringToNumber(ip, &IpAddr);
            changedIpAddr = htonl(IpAddr);
            tcph->dest = htons(7777);
            iph->daddr = changedIpAddr;
            return NF_ACCEPT; // Tells the system to accept the packet, and process the next one.
        }
    }
    return NF_ACCEPT;
}
```

Trouble Shooting

echo 문제 발생

sudo echo를 통해 ip_forward 파일을 1로 바꾸려 했으나 되지 않았다. 전에 echo가 되지 않았을 때 사용했던 sudo bash -c echo "~" 를 사용하려 했으나, 이 방법도 permission 문제가 발생했다. 검색을 통해 다른 명령어를 찾았고, 이 명령어는 제대로 실행되었다.

`echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward`

주소 "111.1.1.0" 변환

단순히 "111.1.1.0"은 string 형태이기 때문에 커널에서 이 string을 ip주소로 변환하여 인식하지 못한다.

기존 2차 웜업과제를 할 때 는 arpa.inet.h 를 include 한 후에 inet_addr 함수를 통해 string 형식의 ip 주소를 long형의 숫자 IP 주소로 바꾸어 줄 수 있었다.

```
inet_addr(SERVER_IP_ADDR);
```

하지만 같은 방법으로 커널을 컴파일을 시도했을때 arpa.inet.h 파일을 찾을 수 없어서 해당 함수를 사용하지 못한다는 에러 메시지를 확인할 수 있었다.

커널 소스 트리에 있는 파일만을 include 할 수 있음을 알고 다른 방법을 사용하기로 했다.

iph→daddr 의 자료형을 찾아보니 be32라고 되어있다. Big Endian 32 bit라는 것 같아서 처음에는 111.1.1.0을 32비트로 직접 계산해서 unsigned int 값을 만들었다. (1862336768) 하지만 비트를 직접 계산하는 건 아닌 것 같아 다른 방법을 찾아보았다. 그 결과 string을 int 자료형으로 바꾸는 아래와 같은 함수를 정의하여 사용했다.

```
//------------------chage IP formmat------------------//
int ipStringToNumber (const char* pDottedQuad, unsigned int * pIpAddr)
{
    unsigned int byte3;
    unsigned int byte2;
    unsigned int byte1;
    unsigned int byte0;
    char dummyString[2];
    if (sscanf(pDottedQuad, "%u.%u.%u.%u%1s", &byte3, &byte2, &byte1, &byte0, dummyString) == 4) {
        if ( (byte3 < 256)
            &&(byte2 < 256)
            &&(byte1 < 256)
            &&(byte0 < 256)
            )
        {
            *pIpAddr = (byte3 << 24) + (byte2 << 16) + (byte1 << 8) + byte0;
            return 1;
        }
    }
    return 0;
}
```

유저 프로세스 공간으로 부터 커널로 데이터 복사(copy_from_user)

`echo 3333 > /proc/myproc_dir/myproc_file` 명령어 수행시 실행되는 proc의 custom write 함수에서는 유저 프로세스 공간으로부터 커널로 데이터 복사를 위해 별도의 처리를 해야했고 이를 위해 copy_from_user 함수를 사용했다. `copy_from_user(to, from, n)` 함수는 `from` 에서 `n` 바이트만큼 커널 가상 주소 `to`로 데이터를 복사한다.

```
copy_from_user(my_buffer, user_buffer, BUFF_LEN)
```

char array를 int number로 변환(sscanf)

`copy_from_user` 함수를 통해 유저 프로세스 공간으로부터 커널로 데이터를 받아왔지만 이는 char array였기에 숫자의 변환이 필요했다. 이를 위해 문자열에서 형식화 된 데이터를 읽어오는 `sscanf` 함수를 사용했다. `sscanf(str, format, ...)` 함수는 `str` 에서 데이터를 `format`에서 지정하는 바에 따라 읽어와 해당 형식대로 데이터를 뒤에 부수적인 인자들이 가리키는 메모리 공간에 저장한다.

```
sscanf(my_buffer, "%u", &inputPort);
```

수행 방법

기본 세팅

1. route 명령어 통해서 라우트 정보 확인
2. sudo route add -net 111.1.1.0 netmask 255.255.255.0 dev enp0s3 (111.1.1.0은 우리가 포워딩으로 보낼 ip 주소)
3. echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward

모듈 세팅

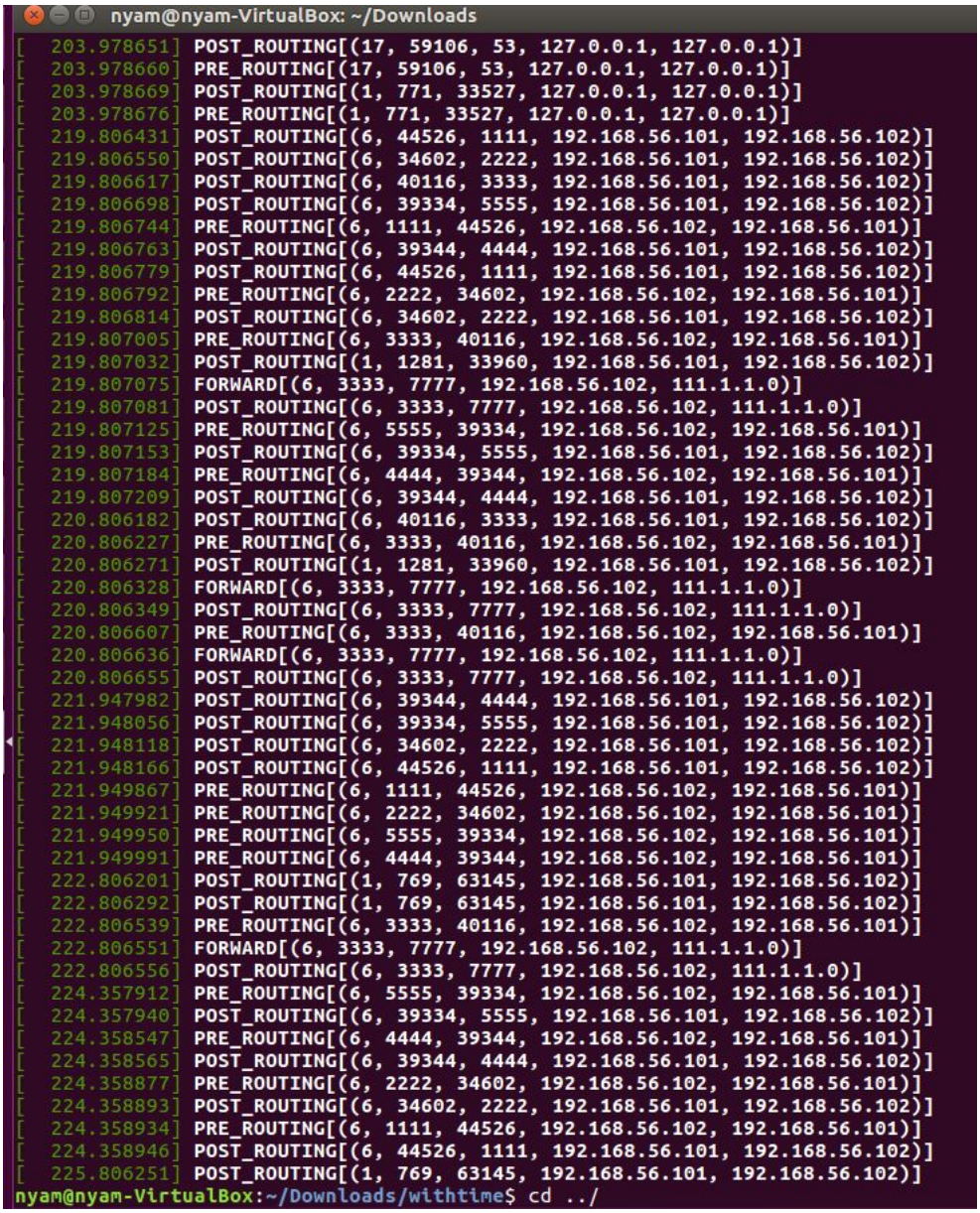
1. make 명령어 수행 (mybasic.c 컴파일)
2. sudo insmod mybasic.ko (반대는 sudo rmmod mybasic)
3. sudo bash -c 'echo 3333 > /proc/myproc_dir/myproc_file' 명령어 수행 (모듈 변수에 포트번호 전달)

실행

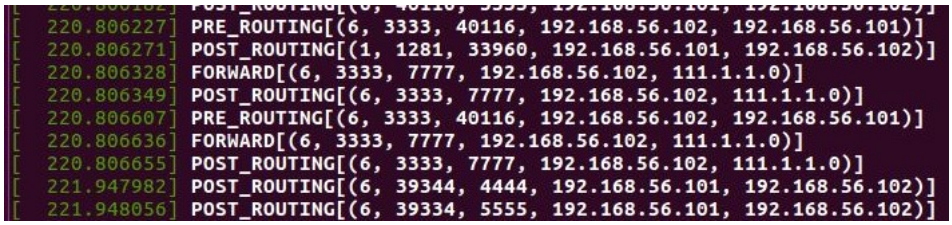
1. 서버 사이드 ./server 명령어 통해 서버 동작 (Virtural Box id : syspro / pwd : oslab)
2. 클라이언트 사이드 ./main 통해 클라이언트 동작
3. Ctrl+c를 통해 클라이언트를 종료시킨 후 dmesg 명령어를 통해 log를 확인한다.

로그파일 결과 분석

- LOG 화면 캡처



- 상세한 패킷 분석:



패킷 캡처의 각 줄의 형식은 다음과 같다.

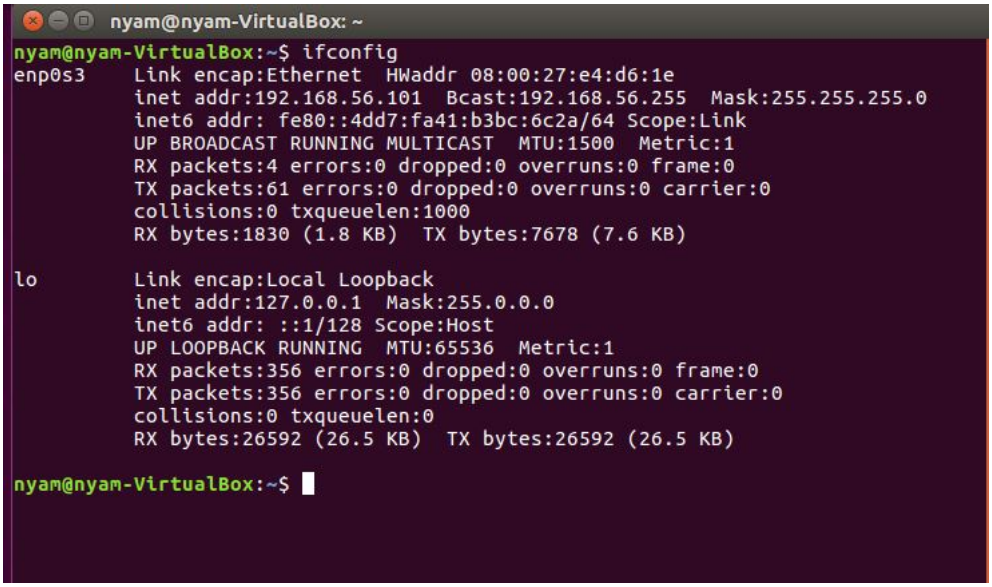
HOOKPOINT명[(protocol, source포트, dest포트, source IP주소, destination IP주소)]

1. 첫 줄의 패킷은 PRE_ROUTING hook point에서 프린트된 패킷이다. 프로토콜 6번은 TCP를 사용하고 있다는 것을 뜻한다. source포트는 3333이다. 서버에서 보낸 패킷 중 3333포트를 통해 보내온 패킷임을 뜻한다. 40116는 클라이언트의 포트번호이

다. 192.168.56.102는 서버의 IP주소이다. 192.168.56.101는 클라이언트의 IP주소이다. PRE_ROUTING에서 프린트 한 패킷은 hook function을 통해 port 번호, IP주소 등을 바꿔주기 전의 패킷이기 때문에, 서버에서 보낸 패킷 형태 그대로 프린트되었다.

2. 세 번째 줄의 패킷은 FORWARD hook point에서 프린트된 패킷으로 source port가 3333이고, destination port가 7777이다. PRE_ROUTING hook point에서 하드코딩으로 destination port를 바꾸어주었기 때문에 destination port 가 기존의 40116가 아닌 7777로 바뀐 것을 확인할 수 있다. source IP 주소는 원래대로 서버의 주소인 192.168.56.102이다. 마지막으로 destination IP가 111.1.1.0인 것을 확인할 수 있는데, 역시 PRE_ROUTING hook function에서 하드코딩으로 ip주소를 111.1.1.0 으로 바꾸어 주었기 때문에 `ip_rcv_finish()`에서 패킷을 포워딩하여 FORWARD hook point 지점으로 패킷이 전달될 수 있었다.
3. 네 번째 줄의 패킷은 POST_ROUTING hook point에서 프린트된 패킷으로 source port가 3333, destination port 가 7777이고, destination IP 주소가 111.1.1.0으로 바뀐 패킷이 목표했던대로 output으로 나가고 있는 것을 확인할 수 있다.

- 참고 - Client의 IP주소 확인



각자 맡은 부분, 주요 기여점

- 강수진: mybasic.c 작성 및 코드 설명, 커널 레벨 코드 분석(ip_rcv, ip_rcv_finish, ip_forward) 수행방법 작성, trouble shooting (주소 변환, copy_from_user, sscanf사용 등)
- 배은초: mybasic.c 작성, 커널 레벨 코드 분석(ip_rcv, ip_forward_finish, ip_output) 로그 파일 생성 및 설명, trouble shooting(echo 문제), ip layer 설명, netfilter & hooking 설명