시스템 프로그래밍 3 차 과제 보고서

제출자 2019320069 정채운

2019320122 이권은

제출일자 2021 년 12 월 5 일

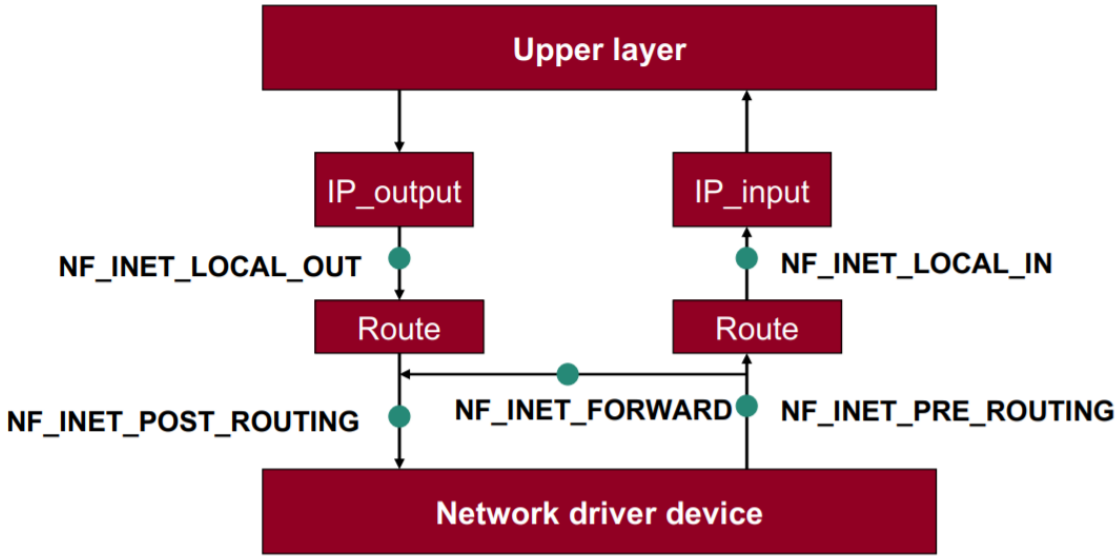
Freeday 사용일수 0 일

1. Netfilter 및 Hooking에 대한 설명

Netfilter는 packet mangling을 할 수 있도록 해주는 리눅스 커널 내부 프레임워크이다. 리눅스 2.4 이후의 커널에 내장되어 있으며, 네트워크 프로토콜 스택과는 독립적으로 작동한다. Netfilter를 통해 firewall 설정, 네트워크 주소 변환, 패킷 필터링, 패킷 mangling 등의 작업을 수월하게 수행할 수 있다.

Netfilter는 훅(hook)을 제공하여 네트워크와 관련한 다양한 연산을 핸들러 형태로 구현할 수 있게 해준다. 훅은 패킷이 네트워크 스택을 통해 지나갈 때, 그 경로 상에 있는 미리 정의된 지점들을 의미한다. 이러한 훅 포인트에 netfilter가 등록한 함수가 있으면, 프로토콜 코드는 훅 포인트에 도달했을 때 netfilter 프레임워크로 점프하여 등록된 함수를 수행한 뒤, 다시 프로토콜로 돌아간다.

각 프로토콜 family마다 서로 다른 훅 포인트들을 가질 수 있다. IPv4의 경우 5개의 훅 포인트를 아래 그림과 같이 정의한다.



NF_INET_PRE_ROUTING은 패킷이 들어왔을 때 sanity check을 진행한 후, routing decision을 내리기 이전의 단계에 있는 지점이고, NF_INET_LOCAL_IN은 routing decision을 내린 이후, 패킷의 목적지가 해당 호스트인 경우 지나가게 되는 지점이다. NF_INET_FORWARD는 패킷의 목적지가 해당 호스트가 아니고, 다른 인터페이스로 forwarding되어야 하는 경우에 지나가게 되는 지점이다.

NF_INET_LOCAL_OUT은 로컬 호스트에서 생성된 모든 패킷들이 전송되기 전에 지나가는 지점이며, NF_INET_POST_ROUTING은 routing이 끝나고 L2 layer로 가기 직전에 지나게 되는 지점이다.

하나의 훅 포인트에 여러 개의 함수가 등록될 수 있으므로, 각 함수의 priority를 설정해 주어야 한다. 패킷이 훅 포인트를 지나갈 때, 이러한 priority 순서에 따라 훅 포인트에 등록된 각 함수가 호출된다.

후킹 포인트에 함수를 등록하거나 해제할 때는 nf_hook_ops 구조체를 사용한다. 함수를 등록할 때는 nf_register_hook()에 nf_hook_ops 구조체를 넘겨주고, 함수를 해제할 때는 nf_unregister_hook()에 nf_hook_ops 구조체를 넘겨준다. nf_hook_ops 구조체의 field들중, 직접 채워 넣어야 하는 field는 *hook, pf, hooknum, priority이다. *hook에는 등록하려는 함수를 넣고, pf에는 PF_INET 등의 네트워크 family를 넣고, hooknum에는 해당 함수를 어떤 후킹 포인트에 등록할 것인지 넣어주며, priority에는 이 후킹의 우선순위를 넣어 준다.

후킹 함수의 입력으로는 sk_buff 구조체의 포인터인 skb 등이 주어지며, return 값으로는 현재 패킷을 drop하는 NF_DROP, 현재 패킷을 다음 루틴으로 넘기는 NF_ACCEPT, 현재 패킷을 커널이 잊어버리는 NF_STOLEN, 스택을 타지 않고 바로 사용자 공간에 올리는 NF_QUEUE, 훅을 다시 호출하는 NF_REPEAT의 다섯 가지 값들 중 하나를 return해야 한다.

2. 커널레벨 네트워킹 코드 분석

패킷이 data link layer에서 ip layer로 올라오면 ip_rcv() 함수가 받는다. 해당 함수는 net/ipv4/ip_input.c에 define되어 있다. 우선 sanity checking을 진행한다. sanity checking은 packet에 별 문제가 없고, 계속 진행해도 되는지를 확인하는 과정이다. Header size, IP version., checksum, length 등을 확인한다. 예를 들어 checksum이 맞지 않는다면 패킷을 버리게 된다. 이더넷 단에서는 checksum 확인이 필수였지만, IP layer에서는 확인하지 않기도 한다. 그래도 대부분의 경우 기본적으로 진행한다.

```
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));

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

iph = ip_hdr(skb);

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

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;
```

netfilter hook 중에서 NF_INET_PRE_ROUTING을 invoke하고 ip_rcv_finish()함수를 호출한다.

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

ip_rcv_finish()는 ip_rcv()와 마찬가지로 /net/ipv4/ip_input.c에 define되어 있다.

```
static int ip_rcv_finish(struct net *net, struct sock *sk, struct sk_buff *skb)
{
```

routing cache에서 패킷의 destination을 확인한다. 이 때 자신한테 온 패킷일 경우, 즉 locally delivered되어야 할 패킷이면 ip_local_deliver()을 호출하고, 아니라면 forwarding을 위해

ip_forward() 함수(unicast)나 ip_mr_input() 함수(multicast)를 호출한다. ip_local_deliver()가 호출된 경우 필요 시 fragment들을 reassemble하고, NF_INET_LOCAL_IN을 invoke한 뒤 ip_local_deliver_finish()함수를 호출한다. ip_local_deliver_finish()는 protocol의 종류에 따라 상위 layer의 handler를 호출한다. handler에는 tcp_v4_rcv() (TCP), udp_rcv() (UDP), icmp_rcv() (ICMP, 매칭되는 프로토콜이 없을 때 호출) 등이 있다.

포워딩을 위해 ip_forward()가 호출되는 구체적인 과정은 다음과 같다.

① ip_rcv_finish()에서 ip_route_input_noref() 호출

```
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;
    }
}
```

② ip_route_input_noref()에서 ip_route_input_slow() 호출

```
int ip_route_input_noref(struct sk_buff *skb, __be32 daddr, __be32 saddr,
                         u8 tos, struct net_device *dev)
{
    int res;
    ...
    res = ip_route_input_slow(skb, daddr, saddr, tos, dev);
    rcu_read_unlock();
    return res;
}
```

③ ip_route_input_slow()에서 ip_mkroute_input() 호출

```
static int ip_route_input_slow(struct sk_buff *skb, __be32 daddr, __be32 saddr,
                               u8 tos, struct net_device *dev)
{
    ...
    int     err = -EINVAL;
    ...

    err = ip_mkroute_input(skb, &res, &fl4, in_dev, daddr, saddr, tos);
    ...
}
```

④ ip_mkroute_input()에서 __mkroute_input() 호출

```
static int ip_mkroute_input(struct sk_buff *skb,
                            struct fib_result *res,
                            const struct flowi4 *fl4,
                            struct in_device *in_dev,
                            __be32 daddr, __be32 saddr, u32 tos)
{
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (res->fi && res->fi->fib_nhs > 1) {
        int h;

        if (unlikely(ip_hdr(skb)->protocol == IPPROTO_ICMP))
            h = ip_multipath_icmp_hash(skb);
        else
            h = fib_multipath_hash(saddr, daddr);
        fib_select_multipath(res, h);
    }
#endif

    /* create a routing cache entry */
    return __mkroute_input(skb, res, in_dev, daddr, saddr, tos);
}
```

⑤ __mkroute_input()에서 ip_forward()호출

```
static int __mkroute_input(struct sk_buff *skb,
                           const struct fib_result *res,
                           struct in_device *in_dev,
                           __be32 daddr, __be32 saddr, u32 tos)
{
    ...
    rth->dst.input = ip_forward;
    ...
}
```

다음은 forwarding에 대한 설명이다. ip_forward()는 net/ipv4/ip_forward.c에 define되어 있다.

```
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;
```

TTL을 체크하여 값이 1 이하면 drop하고 ICMP를 보낸다(ICMP_TIME_EXCEEDED).

```
if (ip_hdr(skb)->ttl <= 1)
    goto too_many_hops;
```

```
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);
```

길이가 MTU보다 큰 경우 fragmentation이 필요하기 때문에 마찬가지로 drop하고 ICMP를 보낸다(ICMP_FRAG_NEEDED).

```
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;
}
```

skb를 확인하여 L2헤더의 공간을 할당하고, TTL을 1만큼 감소시킨다.

```
/* 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);

/* Decrease ttl after skb cow done */
ip_decrease_ttl(iph);
```

마지막에는 NF_INET_FORWARDING을 invoke하고, ip_forward_finish()를 호출한다.

```
return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD,net, NULL, skb,
                   skb->dev, rt->dst.dev, ip_forward_finish);
```

ip_forward_finish()에서는 IP옵션을 처리하고 ip_output()을 호출한다.

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

    IP_INC_STATS_BH(net, IPSTATS_MIB_OUTFORWDATAGRAMS);
    IP_ADD_STATS_BH(net, IPSTATS_MIB_OUTOCTETS, skb->len);

    if (unlikely(opt->optlen))
        ip_forward_options(skb);

    skb_sender_cpu_clear(skb);
    return dst_output(net, sk, skb);
}
```

```
static inline int dst_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
    return skb_dst(skb)->output(net, sk, skb);
}
```

ip_output()은 sk_buff필드를 update하는데, 해당 패킷을 전송하기 위해 어떤 device를 사용할지, 해당 패킷이 IP와 있는지 specify한다. 그리고 NF_INET_POST_ROUTING을 invoke한 뒤, ip_finish_output()을 호출한다.

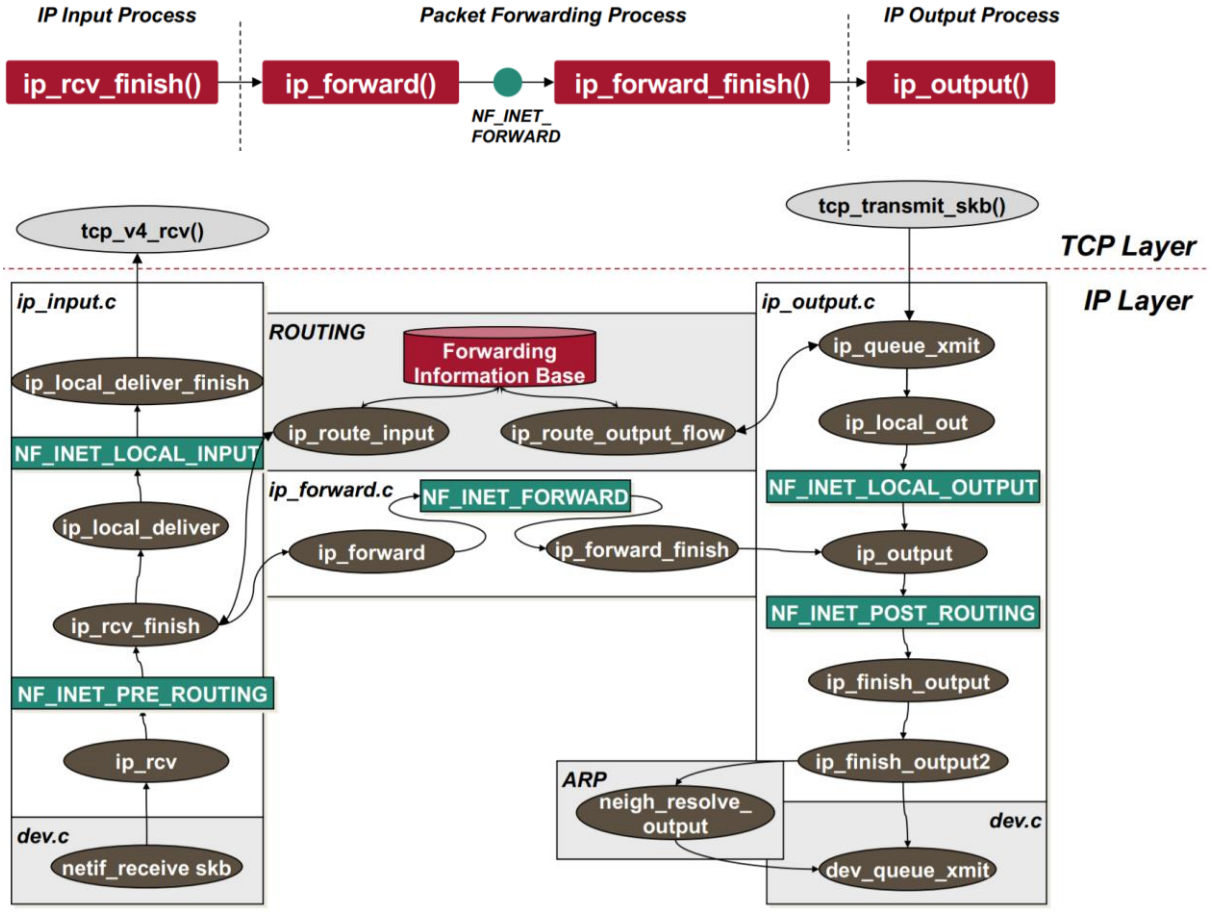
```
int ip_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
    struct net_device *dev = skb_dst(skb)->dev;

    IP_UPD_PO_STATS(net, IPSTATS_MIB_OUT, skb->len);

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

    return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING,
                net, sk, skb, NULL, dev,
                ip_finish_output,
                !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

전체 과정을 하나의 그림으로 나타내면 다음과 같다.



3. 작성한 소스코드에 대한 설명

```
#define PROC_DIRNAME "myproc"
#define PROC_FILENAME "myproc"
#define PROC_MAX 1024
#define NIPQUAD(addr) \
    ((unsigned char *)&addr)[0], \
    ((unsigned char *)&addr)[1], \
    ((unsigned char *)&addr)[2], \
    ((unsigned char *)&addr)[3]
```

proc file 을 저장할 디렉토리 명과 파일명, proc_buffer 의 최대 크기를 지정한다. NIPQUAD 는 ip header 의 source address(saddr), destination address(daddr)를 읽어오기 위해 만들어서 활용하였다.

```
static struct proc_dir_entry *proc_dir;
static struct proc_dir_entry *proc_file;

char proc_buffer[PROC_MAX];
//forward/drop할 포트번호 초기값 설정, proc write를 통해 사용자에게 입력받아 변경할 수 있음
unsigned short forward_port = 1111;
unsigned short drop_port = 2222;
```

proc_dir_entry 변수들과, proc 파일 내부에서 사용할 버퍼 proc_buffer 를 선언해주었다. forwarding 할 패킷과 drop 할 패킷의 port 번호를 저장할 변수를 만들고, 각각 1111 과 2222 로 초기화하였다.

```
//forward/drop할 포트번호 전달할 때 proc 사용
static ssize_t my_write(struct file *file, const char __user * user_buffer, size_t count, loff_t *ppos) {
    //user_buffer에 있는 값으로 전역변수(포워딩/드랍할 포트번호) 설정 예) sudo echo 1111 2222 > myproc
    size_t size = count;
    //proc_buffer 크기만큼만 입력받음
    if (size > PROC_MAX) {
        size = PROC_MAX;
    }
    //user_buffer에 있는 값 size만큼 proc_buffer에 복사
    if (copy_from_user(proc_buffer, user_buffer, size)) {
        return -EFAULT;
    }
    //forward_port, drop_port 값 저장
    sscanf(proc_buffer, "%hu%hu", &forward_port, &drop_port);
    //저장한 값 출력
    printk(KERN_INFO "forward_port:%u, drop_port:%u\n", forward_port, drop_port);

    return size;
}
```

사용자로부터 입력 받은 forwarding 또는 drop 할 포트번호를 저장하는 과정이다. user_buffer 에 있는 값을 proc_buffer 로 복사한 후, 각각 forward_port 변수와 drop_port 변수에 저장한다.

```
static const struct file_operations myproc_fops = {
    .owner = THIS_MODULE,
    .write = my_write,
};
```

proc file 의 option 을 지정하였다.

```c
static int __init simple_init(void) {
    printk(KERN_INFO "module init");
    proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
    proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &myproc_fops);

    nf_register_hook(&forward_or_drop);
    nf_register_hook(&check_packet_drop);
    nf_register_hook(&check_packet_forward_f);
    nf_register_hook(&check_packet_forward_p);
    return 0;
}
```

모듈이 로드되었을 때 실행되는 부분이다. proc 파일과 저장될 디렉토리를 만들고, 후킹 포인트 등록 함수를 사용하여 nf_hook_ops 들을 등록하였다.

다음은 후킹함수와 nf_hook_ops 에 대한 설명이다.

```c
static unsigned int foward_or_drop_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state) {
    /* 후킹함수 작성 */
    struct iphdr *iph = ip_hdr(skb);
    struct tcphdr *tcph = tcp_hdr(skb);

    unsigned char protocol = iph->protocol;
    unsigned short sport = ntohs(tcph->source);
    unsigned short dport = ntohs(tcph->dest);
```

먼저 패킷 forward/drop 함수인 forward_or_drop_hook()에 대한 설명이다. socket buffer 로부터 ip header 와 tcp header 의 정보를 받아와 저장한다. ip header 의 protocol 과 tcp header 의 source port, destination port 도 각각 저장해 주었다. 이 때 TCP/IP 에서 사용되는 네트워크 바이트 오더 방식의 값을 호스트 바이트 오더 방식으로 바꾸기 위해 ntoh()를 사용하였다.

```c
if (sport == forward_port) {
    printk(KERN_INFO "forward:PRE_ROUTING packet(%u; %u; %u; %d.%d.%d.%d; %d.%d.%d.%d)\n", protocol,
            sport, dport, NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
    tcph->source = htons(7777);
    tcph->dest = htons(7777);
    iph->daddr = inet_addr("100.1.1.0");
    //page 11: NF_INET_PRE_ROUTING, NF_INET_FORWARD, NF_INET_POST_ROUTING 에서 hooking 되었는지 확인한다.
    return NF_ACCEPT;
}
```

source port 번호가 forwarding 할 port 번호일 경우, 우선 protocol, source port 번호, destination port 번호, ip header 의 source address, destination address 를 출력한다. ip header 의 address 는 string 형식이 아니기 때문에 앞서 선언한 NIPQUAD 함수를 사용하여 출력하였다. 이어서 tcp header 의 source port 와 destination port 를 각각 7777 로 변경한다. 이번에는 호스트 바이트 오더 방식의 값을 네트워크 바이트 오더 방식으로 변환하기 위해 htons()를 사용하였다. 그리고 forwarding 을 위하여 ip header 의 destination address 값을 미리 라우팅 테이블에 저장한 주소로 변경한다. 앞서 NIPQUAD 를 이용하여 주소를 받아왔던 것처럼, 저장할 때도 ip header 의 주소 형식에 맞게 저장하도록 변환이 필요하다. 원래 inet_addr()함수가 이 역할을 수행하지만, 커널 영역에서는 inet_addr()를 사용할 수 없어 별도의 함수를 선언하여 사용하였다. forwarding 할 패킷이므로 현재 패킷을 다음 루틴으로 넘기기 위해 NF_ACCEPT 를 리턴한다.

```c
unsigned int inet_addr(char *str)
{
    int a, b, c, d;
    char arr[4];
    sscanf(str, "%d.%d.%d.%d", &a, &b, &c, &d);
    arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d;
    return *(unsigned int *)arr;

}
```

직접 선언한 inet_addr()이다. 문자열 형식을 network 형식에 맞게 변경해준다.

```c
else if (sport == drop_port) {
    printk(KERN_INFO "drop:PRE_ROUTING packet(%u; %u; %u; %d.%d.%d.%d; %d.%d.%d.%d)\n", protocol,
            sport, dport, NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
    tcph->source = htons(3333);
    tcph->dest = htons(3333);
    //page 10: 넷필터의 후킹 포인트 중 '패킷 drop 함수를 등록한 후킹 포인트'에 drop한 패킷을 확인하는 함수를 등록
    //page 12: NF_INET_PRE_ROUTING, NF_INET_LOCAL_IN 에서hooking 되었는지 확인한다.
    printk(KERN_INFO "drop:PRE_ROUTING packet(%u; %u; %u; %d.%d.%d.%d; %d.%d.%d.%d)\n", protocol,
            ntohs(tcph->source), ntohs(tcph->dest), NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
    return NF_DROP;
}
else {
    return NF_ACCEPT;
}
```

source port 번호와 drop 할 port 번호가 같은 경우 실행되는 부분이다. 앞서 설명한 forwarding 을 위한 전처리과정과는 달리 tcp header 의 source port 와 destination port 를 각각 3333 으로 변경한다. 이번에도 호스트 바이트 오더 방식의 값을 네트워크 바이트 오더 방식으로 변환하기 위해 htons()를 사용하였다. port 번호를 변경한 뒤 잘 변경되었는지 확인하기 위하여 처음에 출력했던 정보들을 다시 한 번 출력하였다. drop 할 패킷이므로 NF_DROP 을 리턴한다.

port 번호가 forwarding 또는 drop 할 번호가 아닌 경우 NF_ACCEPT 를 리턴한다.

```c
static struct nf_hook_ops forward_or_drop = {
    .hook = foward_or_drop_hook, //등록하려는 함수
    .pf = PF_INET, //네트워크 family
    .hooknum = NF_INET_PRE_ROUTING, //후킹 포인트
    .priority = NF_IP_PRI_FIRST, //우선순위
};
```

forward_or_drop()을 NF_INET_PRE_ROUTING 후킹 포인트에 등록하기 위해 위와 같이 nf_hook_ops 구조체 forward_or_drop 을 선언하였다. 해당 후킹 포인트를 사용한 이유는 이 포인트가 패킷을 locally deliver 할지 forwarding 할지 결정하는 부분이기 때문이다(ip_rcv()에서 invoke 하는 후킹 포인트).

```c
static unsigned int check_drop_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state) {
    /* 후킹함수 작성 */
    struct iphdr *iph = ip_hdr(skb);
    struct tcphdr *tcph = tcp_hdr(skb);

    unsigned char protocol = iph->protocol;
    unsigned short sport = ntohs(tcph->source);
    unsigned short dport = ntohs(tcph->dest);

    if(sport==3333){
        printk(KERN_INFO "drop:LOCAL_IN packet(%u; %u; %u; %d.%d.%d.%d; %d.%d.%d.%d)\n", protocol,
                sport, dport, NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
        return NF_DROP;
    }

    return NF_ACCEPT;
}
```

PRE_ROUTING 후킹 포인트에서 드랍할 패킷이 잘 처리되었는지 확인하기 위해 만든 check_drop_hook()이다. NF_INET_LOCAL_IN 후킹 포인트에서 source port 번호가 3333 인 패킷, 즉 드랍할 패킷이 있는지 확인한다. 해당 포인트를 선택한 이유는 locally deliver 될 때 실행되는 ip_local_deliver()함수에서 invoke 하는 후킹 포인트이기 때문이다. 드랍할 패킷이 있는 경우 정보를 출력하고, NF_DROP 을 리턴한다. 그 외의 패킷들은 NF_ACCEPT 를 리턴한다.

```c
static struct nf_hook_ops check_packet_drop = {
    .hook = check_drop_hook, //등록하려는 함수
    .pf = PF_INET, //네트워크 family
    .hooknum = NF_INET_LOCAL_IN, //후킹 포인트
    .priority = NF_IP_PRI_FIRST, //우선순위
};
```

후킹 함수를 등록하기 위한 nf_hook_ops 구조체 변수 check_packet_drop 은 위와 같다.

```c
static unsigned int check_forward_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state) {
    /* 후킹함수 작성 */
    struct iphdr *iph = ip_hdr(skb);
    struct tcphdr *tcph = tcp_hdr(skb);

    unsigned char protocol = iph->protocol;
    unsigned short sport = ntohs(tcph->source);
    unsigned short dport = ntohs(tcph->dest);

    if(sport==7777) {
        if(state->hook == NF_INET_FORWARD)
            printk(KERN_INFO "forward:FORWARD packet(%u; %u; %u; %d.%d.%d.%d; %d.%d.%d.%d)\n", protocol,
                    sport, dport, NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
        else
            printk(KERN_INFO "forward:POST_ROUTING packet(%u; %u; %u; %d.%d.%d.%d; %d.%d.%d.%d)\n", protocol,
                    sport, dport, NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
    }
    return NF_ACCEPT;
}
```

forwarding 할 패킷이 잘 처리되었는지 확인하기 위한 check_forwrd_hook()이다. 이 함수는 NF_INET_FORWARD 와 NF_INET_POST_ROUTING 후킹 포인트에 등록한다. 해당 포인트들을 선택한 이유는 포워딩 중에 invoke 되는 후킹 포인트 이기 때문이다(NF_INET_FORWARD 는 ip_forward()에서, NF_INET_POST_ROUTING 은 ip_output()에서 invoke). 만약 source port 번호가

7777 인 패킷이 있을 경우 정보를 출력한다. 이 때 어느 포인트에서 출력되는 것인지 구분하기 위해 if 문을 사용하였다.

```
//후킹포인트 : FORWARD
static struct nf_hook_ops check_packet_forward_f = {
    .hook = check_forward_hook, //등록하려는 함수
    .pf = PF_INET, //네트워크 family
    .hooknum = NF_INET_FORWARD, //후킹 포인트
    .priority = NF_IP_PRI_FIRST, //우선순위
};
//후킹포인트: POST_ROUTING:
static struct nf_hook_ops check_packet_forward_p = {
    .hook = check_forward_hook, //등록하려는 함수
    .pf = PF_INET, //네트워크 family
    .hooknum = NF_INET_POST_ROUTING, //후킹 포인트
    .priority = NF_IP_PRI_FIRST, //우선순위
};
```

후킹 함수를 등록하기 위한 nf_hook_ops 구조체 변수들은 위와 같다. check_packet_forward_f 는 NF_INET_FORWARD 에, check_packet_forward_p 는 NF_INET_POST_ROUTING 에 함수를 등록하고 있는 것을 볼 수 있다.

```
static void __exit simple_exit(void) {
    printk(KERN_INFO "module exit");
    proc_remove(proc_file);
    proc_remove(proc_dir);

    nf_unregister_hook(&forward_or_drop);
    nf_unregister_hook(&check_packet_drop);
    nf_unregister_hook(&check_packet_forward_f);
    nf_unregister_hook(&check_packet_forward_p);
}
```

모듈이 내려갈 때 실행되는 부분이다. proc_file 과 proc_dir 를 없애고, 등록했던 nf_hook_ops 들을 해제한다.

4. 실험 방법에 대한 설명 및 로그파일 결과 분석

(1) 실험 방법

① echo 1 > /proc/sys/net/ipv4/ip_forward 명령어를 이용하여 forwarding 기능을 enable한다.

② route add 명령어를 이용하여 라우팅 테이블에 포워딩을 위한 엔트리를 추가한다. 이때 destination IP 주소는 100.1.0으로, netmask는 255.255.255.0으로, 인터페이스는 enp0s3로 설정해 주었다.

```
filled@filled-VirtualBox:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
100.1.1.0       *               255.255.255.0   U     0      0        0 enp0s3
link-local      *               255.255.0.0     U     1000   0        0 enp0s3
192.168.56.0    *               255.255.255.0   U     100    0        0 enp0s3
```

③ lkm과 makefile을 작성하고 make 명령어로 컴파일한다.

```
filled@filled-VirtualBox:~/module$ make
make -C /usr/src/linux-4.4 SUBDIRS=/home/filled/module modules
make[1]: Entering directory '/usr/src/linux-4.4'
  CC [M]  /home/filled/module/lkm.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/filled/module/lkm.mod.o
  LD [M]  /home/filled/module/lkm.ko
make[1]: Leaving directory '/usr/src/linux-4.4'
```

④ insmod 명령어를 통해 lkm을 커널에 올린다.

```
filled@filled-VirtualBox:~/module$ sudo insmod lkm.ko
```

⑤ 생성한 proc 파일에 write하여 forwarding/drop할 포트 번호를 전달한다.

```
filled@filled-VirtualBox:/proc/myproc$ sudo sh -c "echo 1111 2222 > myproc"
```

⑥ server-side 프로그램을 실행한다. 이때 포트 번호는 1111, 2222, 4444, 5555, 6666으로 설정해 주었다.

```
syspro@oslab:~$ ./server
Type Server Ports(format:<Port1> <Port2> <Port3> <Port4> <Port5> :
1111 2222 4444 5555 6666
1111
2222
4444
5555
6666
```

⑦ client-side 프로그램을 실행한다.

```
filled@filled-VirtualBox:~/2$ ./client
```

⑧ 실험이 끝나면 rmmod 명령어로 모듈을 커널에서 내리고 dmesg로 로그를 확인한다.

(2) 로그파일 결과 분석

```
[19502.835633] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19502.835636] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19502.835639] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19502.835659] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19502.835661] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19503.832655] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19503.832695] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19503.832702] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19503.832709] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19503.832715] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19503.835587] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19503.835598] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19503.835603] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19503.836678] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19503.836687] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19505.832872] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19505.832901] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19505.832903] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19505.836744] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19505.836746] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19505.838872] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19505.838879] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19505.838881] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19505.840323] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19505.840325] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19509.836473] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19509.836498] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19509.836506] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19509.836534] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19509.836542] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19509.851915] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19509.851929] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19509.851938] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19509.851950] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19509.851958] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19517.852935] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19517.852954] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19517.852963] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19517.852976] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19517.852983] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19517.867656] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19517.867673] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19517.867679] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19517.867685] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19517.867691] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19533.864644] forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
[19533.864699] forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19533.864731] forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
[19533.864785] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
[19533.864793] drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
[19533.883862] drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
```

모든 로그에서 프로토콜은 6으로 출력되었는데, 이는 해당 패킷이 TCP를 사용한다는 것을 의미한다.

0x06	6	TCP	Transmission Control Protocol

Source port가 1111인 패킷들은 source port와 destination port가 모두 7777로 바뀐 후 forwarding된다.

```
forward: PRE_ROUTING (6; 1111; 52272; 192.168.56.101; 192.168.56.102)
```

PRE_ROUTING 후킹 포인트에서 패킷 변조 전, source port가 1111인 패킷 정보가 출력된다. Source IP 주소는 server-side program이 실행되는 VM의 IP 주소이고, destination IP 주소는 이 VM(client)의 IP 주소이다.

```
forward: FORWARD (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
```

FORWARD 후킹 포인트에서 변조된 패킷 정보가 출력된다. Source port와 destination port는 7777로 변경되었으며, source IP 주소는 server-side program이 실행되는 VM의 IP 주소이다. 또한 forwarding해주기 위하여 destination IP 주소는 앞서 라우팅 테이블에서 설정한 100.1.1.0으로 변경되었다. 해당 로그를 통해 패킷이 제대로 forwarding되었음을 확인할 수 있다.

```
forward: POST_ROUTING (6; 7777; 7777; 192.168.56.101; 100.1.1.0)
```

POST_ROUTING 후킹 포인트에서 변조된 패킷 정보가 출력된다. 출력되는 정보들은 FORWARD 후킹 포인트에서 출력된 정보와 같다. 해당 로그를 통해 패킷이 제대로 forwarding되었음을 확인할 수 있다.

Source port가 2222인 패킷들은 source port와 destination port가 모두 3333으로 바뀐 후 drop된다.

```
drop: PRE_ROUTING (6; 2222; 55090; 192.168.56.101; 192.168.56.102)
```

PRE_ROUTING 후킹 포인트에서 패킷 변조 전, source port가 2222인 패킷 정보가 출력된다. Source IP 주소는 server-side program이 실행되는 VM의 IP 주소이고, destination IP 주소는 이 VM(client)의 IP 주소이다.

```
drop: PRE_ROUTING (6; 3333; 3333; 192.168.56.101; 192.168.56.102)
```

PRE_ROUTING 후킹 포인트에서 패킷 변조 후, source port와 destination port가 모두 3333인 패킷 정보가 출력된다. source IP 주소는 server-side program이 실행되는 VM의 IP 주소이고, destination IP 주소는 이 VM(client)의 IP 주소이다.

이후 LOCAL_IN 후킹 포인트에서는 포트번호가 3333인 패킷이 존재하지 않는 것을 통해 해당 패킷이 제대로 drop된 것을 확인할 수 있다.

5. 과제 수행 시의 trouble과 troubleshooting 과정

-nf_hook_ops의 priority를 지정할 때 0이 아닌 NF_IP_PRI_FIRST를 사용하면 컴파일 시 에러가 나는 문제가 있었다. <linux/netfilter_ipv4.h> 헤더파일을 include하여 문제를 해결하였다.

-proc으로 forward/drop할 포트 번호를 받아올 때 앞 포트번호(forward port번호)가 잘 받아지지 않는 문제가 있었다.

```
sscanf(proc_buffer, "%u%u", &forward_port, &drop_port);
```

unsigned short일 때 사용하는 format specifier인 %hu대신 unsigned int에 맞는 %u를 사용하여 나타난 문제임을 확인하고 소스코드를 다음과 같이 변경하여 해결하였다.

```
sscanf(proc_buffer, "%hu%hu", &forward_port, &drop_port);
```