시스템프로그래밍

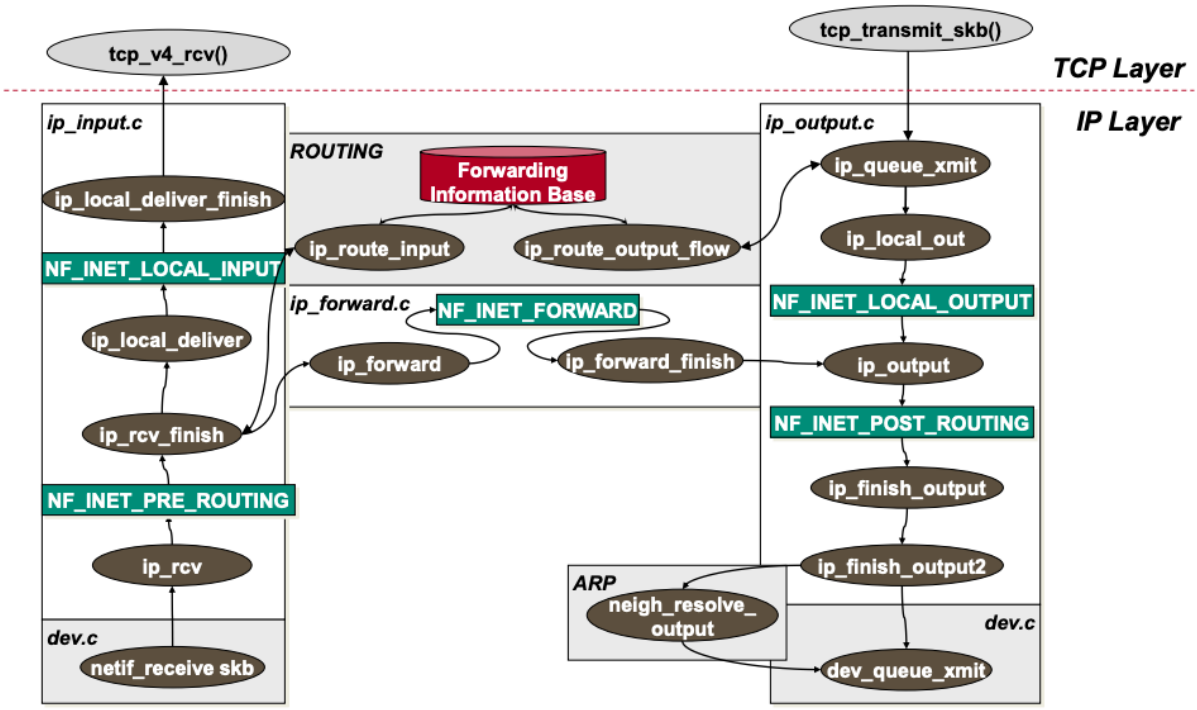
3 차 과제 보고서

목차

1. 배경지식

A. Packet Forwarding

OSI 모델에서 네트워크 계층이 주로 하는 일로, 네트워크에서 패킷을 다른 노드로 전달하는 과정이다. 패킷 포워딩은 주로 라우터나 스위치에서 동작한다. 라우터는 패킷의 목적지 IP 주소를 검사하고, 라우터의 host IP 주소가 아니라면 routing table 을 참고하여 목적지까지의 최적 경로를 결정하여 패킷을 포워딩한다. 다음을 통해 네트워크 계층의 패킷 이동 경로를 확인할 수 있다.



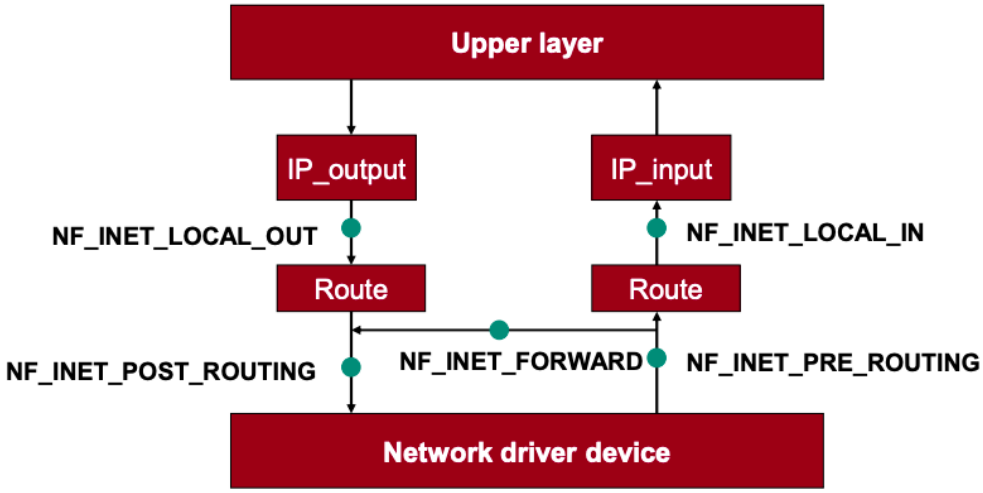
B. Netfilter

Netfilter 는 리눅스 커널에 존재하는 Network packet mangling framework 이다. 네트워크 프로토콜 스택과 관계 없이 독립적으로 존재한다. 간단한 메커니즘으로 동작하여 firewall 설정, 패킷 필터링, 네트워크 주소 변환과 패킷 mangling 이 용이하다.

C. Hook

Hook 는 Netfilter 가 사용하는 abstraction 으로, 갈고리처럼 패킷이 네트워크를 통과할 때 패킷 경로에 정의된 지점이다. 패킷이 Hook 에 도달하면 Netfilter 메커니즘이 동작 한다. Netfilter 메커니즘은 다음과 같다. 패킷이 Hook 에 도달하면 Hook 함수가 호출된다. Hook 함수는 패킷을 조작하거나 변환한다. Hook 함수의 패킷 처리가 완료되면, 네트워크 스택을 이어서 진행한다.

각 프로토콜 familiy 는 서로 다른 hook 들이 존재한다. IPv4 에는 총 5 개의 Hook 이 존재한다. 아래의 그림은 IPv4 의 Netfilter 구조를 보여준다.



- NF_INET_PRE_ROUTING
  외부에서 들어온 패킷의 sanity check 후, 라우팅을 결정하기 전에 존재한다.
- NF_INET_LOCAL_IN
  라우팅 결정 후, 패킷의 현재 호스트일 경우 패킷을 분석하기 전에 존재한다.
- NF_INET_FORWARD
  라우팅 결정 후, 패킷이 현재 호스트가 아닐 경우, 이 패킷을 다른 인터페이스로 포워딩하기 전에 존재한다.
- NF_INET_LOCAL_OUT
  패킷이 로컬 호스트에서 제작된 후, 라우팅 되기 전에 존재한다.
- NF_INET_POST_ROUTING
  라우팅 결정 후, 패킷이 현재 네트워크를 벗어나려고 할 때 존재한다.

Hook 함수는 일반적인 함수이고 다음의 구조를 갖는다.

- typedef unsiged int nf_hookfn(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
  패킷의 abstraction 인 sk_buff 를 인자로 전달 받는다.

이 함수는 Netfilter 패킷을 처리하기 위해 다음 다섯 개의 값 중 하나를 반환해야 한다.

- NF_DROP
  현재 패킷을 버린다.
- NF_ACCEPT
  현재 패킷을 다음 네트워크 스택 루틴으로 넘긴다.
- NF_STOLEN
  현재 패킷을 커널이 잊어버리고 뒤처리를 한다.

- NF_QUEUE
  현재 패킷을 네트워크 스택을 태우지 않고 사용자 공간에 올린다.
- NF_REPEAT
  현재 Hook 를 다시 호출한다.

이와 같은 Hook 함수가 동작하려면 후킹 지점에 이 함수들을 등록 해야하는데, 이때 다음의 구조체와 함수를 사용한다.

- struct nf_hook_ops
  후킹 포인트에 등록하려는 함수 포인터와 네트워크 family, 후킹포인트, 후킹의 우선순위 정보를 담고 있다.
- Int nf_register_hook(struct nf_hook_ops *reg)
  nf_hook_ops 를 Netfilter 에 등록한다.

사용이 끝난 nf_hook_ops 는 넷필터에서 해제한다. 이때 다음의 함수를 사용한다.

- Void nf_unregister_hook(struct nf_hook_ops *reg)

2. 소스코드 설명

A. group36.c

group36.c 는 Loadable Kernel Module 을 활용하기위한 스크립트이다. LKM 을 이용하여 커널 속 Netfilter 를 관리한다. Customized Hook function 을 LKM 을 통해 커널 함수로 등록할 수 있고, proc file 을 읽고 쓰는 동작을 필터링 관리 동작의 abstraction 이라 할 수 있다. 다음 소스코드는 총 세 지점에서의 hook function 과 이를 관리하기위한 LKM 들이 정의 돼있다. 소스코드를 자세히 살펴보도록한다.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#include <linux/netfilter.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <uapi/linux/netfilter_ipv4.h>

#include <linux/string.h>
```

프로그래밍에 필요한 라이브러리들을 추가한다.

```
#define PROC_DIRNAME "group36"    // proc directory name
```

```
#define FILENAME_ADD "add"           // proc file to add black list
#define FILENAME_SHOW "show"        // proc file to show black list
#define FILENAME_DEL "del"          // proc file to delete black list
```

proc file system 프로그래밍에 필요한 매크로이다. Proc file directory 이름과 포트 블랙리스트 관리를 위한 proc file 이름들을 매크로로 정의한다. 블랙리스트에 포트를 추가하기 위한 proc file add, 블랙리스트를 출력하기 위한 proc file show, 블랙리스트에서 포트를 삭제하기위한 proc file del 이다. 그리고 이 파일들이 존재할 proc file directory 가 group36 이다.

```
#define WRITE_BUFSIZE 50             // command buffer : add/delete
#define SHOW_BUFSIZE 200             // show buffer
#define PORT_MAX 50                  // maximum black list number
```

포트 블랙리스트 관리를 위한 매크로이다. 블랙리스트 추가,삭제를 위해 명령어를 각 proc file 에 출력할 때 사용하는 커널 버퍼의 최대 크기 WRITE_BUFSIZE, 블랙리스트를 유저 공간에 출력하기 위해 사용하는 커널 버퍼의 최대 크기 SHOW_BUFSIZE, 블랙리스트의 최대 크기 PORT_MAX 이다.

```
static struct proc_dir_entry *proc_dir;
static struct proc_dir_entry *add_file;
static struct proc_dir_entry *show_file;
static struct proc_dir_entry *del_file;
```

각 proc file 구현을 위한 proc_dir_entry 들이다.

```
// struct to manage ports to be filtered
typedef struct _port
{
        char type;
        int num;
}port;

static port black_list[PORT_MAX];      // list of ports to be filtered
static int port_size;                  // size of black list
```

포트 블랙리스트를 정의하는 부분이다. port 는 블랙리스트의 원소 구조체이다. 포트 필터링 타입을 저장할 변수 type 과 포트 번호를 저장할 변수 num 으로 구성된다.
배열 black_list 는 포트 블랙리스트의 abstraction 이다. port_size 는 포트 블랙리스트를 관리하기 위한 변수로, 포트 블랙리스트에 포함된 포트의 수이다.

```
static bool finish_cat = true;          // flag variable to finish "cat"
```

유저 스페이스에서 cat을 통해 proc file을 읽을 때, read가 무한히 수행되는 상황을 방지하기 위한 flag 변수이다.

```
// convert string to struct iphdr->addr
```

```
unsigned int as_addr_to_net(char *str)
{
        unsigned char arr[4];
        sscanf(str, "%d.%d.%d.%d", &arr[0], &arr[1], &arr[2], &arr[3]);
        return *(unsigned int *)arr;
}
```

문자열로 정의된 IP 주소를 struct iphdr가 IP주소를 다루는 형식으로 변환한다.

```
// customized hook : monitor NF_INET_PRE_ROUTING
// packet sanity check, before routing
// specifically deal with proxy
static unsigned int hook_pre_routing(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
        // get ip header of skb
        struct iphdr *ih = ip_hdr(skb);

        // get tcp header of skb
        struct tcphdr *th = tcp_hdr(skb);

        int sport, dport;                       // source port, destination port
        char saddr[16], daddr[16];          // source ip address, destination ip address
        bool syn,fin,ack,rst;                   // tcp flags

        int i;                                          // iterator to manage black list

        snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
        snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
        sport = ntohs(th->source);                              // format source port info in tcp header
        dport = ntohs(th->dest);                                // format destination port info in tcp header

        //    get tcp flags from tcp header
        syn = th->syn;
        fin = th->fin;
        ack = th->ack;
        rst = th->rst;

        // filtering : whether this packet must be filterted or not : cannot come in local?
        for(i = 0 ; i < port_size ; i++)
        {
```

```
            // if source port of this packet is in the black list with flag I,
            // drop this packet before routing
            if((black_list[i].num==sport)&&(black_list[i].type=='I')){
                    // logging
                    printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","DROP(INBOUND)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);
                    return NF_DROP;
            }
            // if source port of this packet is in the black list with flag P,
            // this packet is for proxy
            else if((black_list[i].num==sport)&&(black_list[i].type=='P'))
            {

                    // proxy destination
                    ih->daddr = as_addr_to_net("131.1.1.1");
                    th->dest = th->source;

                    // logging
                    snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
                    snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
                    dport = (unsigned int)(th->dest);
                    printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","PROXY(INBOUND)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);

                    //    get this packet in local for routing
                    return NF_ACCEPT;
            }
        }

        // logging
        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-15s,%d,%d,%d,%d\n","INBOUND", ih->protocol,
sport, dport, saddr, daddr, syn, fin, ack, rst);

        //    get this packet in local
        return NF_ACCEPT;
}

// overload hook functions for pre-routing
static struct nf_hook_ops hook_pre_routing_ops = {
        .hook = hook_pre_routing,
```

```
        .pf = PF_INET,
        .hooknum = NF_INET_PRE_ROUTING,
        .priority = NF_IP_PRI_FIRST
};
```

위의 함수와 구조체는 Netfilter의 hook 함수를 위한 자료구조들이다. hook_pre_routing은 IPv4의 후킹 포인트 NF_INET_PRE_ROUTING에서 동작하는 함수로, 이 포인트에서 후킹 되는 패킷은 외부에서 현재 네트워크로 전송된 패킷이다. 이 후킹포인트에 함수를 implement하기 위한 구조체 hook_pre_routing_ops이다. 이제 코드를 자세히 살펴보도록한다.

```
// customized hook : monitor NF_INET_PRE_ROUTING
// packet sanity check, before routing
// specifically deal with proxy
static unsigned int hook_pre_routing(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
        // get ip header of skb
        struct iphdr *ih = ip_hdr(skb);

        // get tcp header of skb
        struct tcphdr *th = tcp_hdr(skb);

        int sport, dport;                       // source port, destination port
        char saddr[16], daddr[16];          // source ip address, destination ip address
        bool syn,fin,ack,rst;                   // tcp flags

        int i;                                          // iterator to manage black list
```

함수의 선언부이다. skb는 패킷의 abstraction이다. skb로부터 IP헤더와 TCP헤더를 추출한다. 함수 ip_hdr는 입력으로 받은 패킷 데이터에서 IP헤더가 시작하는 부분의 포인터를 찾아 반환한다. 이를 받은 ih는 IP헤더의 abstraction이다. 함수 tcp_hdr는 입력으로 받은 패킷 데이터에서 TCP헤더가 시작하는 부분의 포인터를 찾아 반환한다. 이를 받은 th는 TCP헤더의 abstraction이다. 패킷의 소스와 목적지 주소를 저장하기위한 변수들을 선언하고, TCP flag들을 저장할 변수들을 선언한다. 필터링을 위해 포트 블랙리스트를 관리하기위한 이터레이터 변수 i를 선언한다.

```
        snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
        snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
        sport = ntohs(th->source);                              // format source port info in tcp header
        dport = ntohs(th->dest);                                // format destination port info in tcp header

        //    get tcp flags from tcp header
        syn = th->syn;
        fin = th->fin;
```

```
        ack = th->ack;
        rst = th->rst;
```

패킷의 소스 주소와 목적지 주소 정보를 초기화한다. IP주소를 초기화하기 위해 snprintf함수를 사용했다. 이 함수는 첫번째 인자에 네번째 인자 값을 세번째 인자로 포맷팅하여 출력하는 함수이다. IP헤더에 저장된 IP주소를 문자열 형태로 저장한다. Port번호를 초기화하기 위해 ntohs함수를 사용했다. TCP헤더에 저장된 포트 번호의 자료형은 be16으로 이를 클라이언트의 byte order로 변환하기위함이다. 그리고 TCP헤더에서 통신 flag들을 읽어와 초기화한다.

```
// filtering : whether this packet must be filterted or not : cannot come in local?
        for(i = 0 ; i < port_size ; i++)
        {
                // if source port of this packet is in the black list with flag I,
                // drop this packet before routing
                if((black_list[i].num==sport)&&(black_list[i].type=='I')){
                        // logging
                        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","DROP(INBOUND)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);
                        return NF_DROP;
                }
                // if source port of this packet is in the black list with flag P,
                // this packet is for proxy
                else if((black_list[i].num==sport)&&(black_list[i].type=='P'))
                {

                        // proxy destination
                        ih->daddr = as_addr_to_net("131.1.1.1");
                        th->dest = th->source;

                        // logging
                        snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
                        snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
                        dport = (unsigned int)(th->dest);
                        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","PROXY(INBOUND)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);

                        //    get this packet in local for routing
                        return NF_ACCEPT;
                }
        }
```

패킷을 필터링하는 부분이다. 블랙리스트의 원소들을 확인하며 현재 패킷의 소스포트가 블랙리스트에 I type으로 등록 돼있는지 확인한다. 만약 그렇다면, 이 패킷은 로컬로 전송될 수 없는 패킷이므로, 커널에 필터링 정보를 출력하고 패킷을 버린다. 만약 그렇지 않고, 현재 패킷의 소스 포트가 블랙리스트에 P type으로 등록 돼 있는지 확인한다. 만약 그렇다면 이는 프록시 서버로 부터 온 패킷을 의미한다. 따라서 패킷의 목적지 주소를 변경하여 필터링 정보를 출력하고, 패킷이 라우팅 되도록 accept한다.

```
// logging
        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-15s,%d,%d,%d,%d\n","INBOUND", ih->protocol,
sport, dport, saddr, daddr, syn, fin, ack, rst);

        //    get this packet in local
        return NF_ACCEPT;
}
```

만약 현재 패킷의 소스 포트가 블랙리스트에 I type으로 등록 돼있지 않고, P type으로도 등록 돼 있지 않다면 패킷이 routing 되도록 accept한다.

```
// overload hook functions for pre-routing
static struct nf_hook_ops hook_pre_routing_ops = {
        .hook = hook_pre_routing,
        .pf = PF_INET,
        .hooknum = NF_INET_PRE_ROUTING,
        .priority = NF_IP_PRI_FIRST
};
```

위에서 정의한 함수를 NF_INET_PRE_ROUTING 후킹 포인트에 implement하기위해 구조체를 선언하고 구조체의 멤버들을 초기화한다. 후킹 함수의 함수포인터, 프로토콜 family, 후킹포인트, 후킹 우선순위 값들을 할당한다. 함수 포인터는 위에서 정의한 함수의 이름인 hook_pre_routing이고, 프토로콜 family는 TCP/IP이므로 PF_INET, 후킹포인트는 NF_INET_PRE_ROUTING, 우선 순위는 최우선이다.

```
// customized hook : monitor NF_INET_FORWARD
static unsigned int hook_forward(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
        // get ip header of skb
        struct iphdr *ih = ip_hdr(skb);

        // get tcp header of skb
        struct tcphdr *th = tcp_hdr(skb);
```

```
        int sport, dport;                       // source port, destination port
        char saddr[16], daddr[16];          // source ip address, destination ip address
        bool syn,fin,ack,rst;                   // tcp flags

        int i;                                          // iterator to manage black list

        snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
        snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
        sport = ntohs(th->source);                      // format source port info in tcp header
        dport = ntohs(th->dest);                        // format destination port info in tcp header

        //    get tcp flags from tcp header
        syn = th->syn;
        fin = th->fin;
        ack = th->ack;
        rst = th->rst;

        // filtering : whether this packet must be filterted or not : must be forwarded?
        for(i = 0 ; i < port_size ; i++)
        {
                // if source port of this packet is in the black list with flag F,
                // drop this packet before forwarding
                if((black_list[i].num==sport)&&(black_list[i].type=='F')){
                // logging
                        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","DROP(FORWARD)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);
                        return NF_DROP;
                }
        }

        // logging
        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-15s,%d,%d,%d,%d\n","FORWARD", ih->protocol,
sport, dport, saddr, daddr, syn, fin, ack, rst);

        //    forward this packet
        return NF_ACCEPT;
}

// overload hook functions for forwarding
static struct nf_hook_ops hook_forward_ops = {
```

```
        .hook = hook_forward,
        .pf = PF_INET,
        .hooknum = NF_INET_FORWARD,
        .priority = NF_IP_PRI_FIRST
};
```

위의 함수는 hook_forward은 IPv4의 후킹 포인트 NF_INET_FORWARD에서 동작하는 함수로, 이 포인트에서 후킹되는 패킷은 라우팅 결정 후, 패킷의 목적지가 현재 호스트가 아니며, 다른 인터페이스로 포워딩되는 패킷이다. 이 후킹포인트에 함수를 implement하기 위한 구조체 hook_forwad_ops이다. 이제 코드를 자세히 살펴보도록한다.

```
// customized hook : monitor NF_INET_FORWARD
static unsigned int hook_forward(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
        // get ip header of skb
        struct iphdr *ih = ip_hdr(skb);

        // get tcp header of skb
        struct tcphdr *th = tcp_hdr(skb);

        int sport, dport;                       // source port, destination port
        char saddr[16], daddr[16];          // source ip address, destination ip address
        bool syn,fin,ack,rst;                   // tcp flags

        int i;                                          // iterator to manage black list

        snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
        snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
        sport = ntohs(th->source);                              // format source port info in tcp header
        dport = ntohs(th->dest);                                // format destination port info in tcp header

        //    get tcp flags from tcp header
        syn = th->syn;
        fin = th->fin;
        ack = th->ack;
        rst = th->rst;
```

함수의 선언부이고 소스와 목적지 정보, TCP 패킷 flag들을 초기화하는 부분이다. 이 부분은 앞서 설명한 hook_pre_routing의 선언부와 소스와 목적지 정보, TCP 패킷 flag들을 초기화하는 부분과 동일하므로 설명을 생략한다.

```
// filtering : whether this packet must be filterted or not : must be forwarded?
```

```
	for(i = 0 ; i < port_size ; i++)
	{
		// if source port of this packet is in the black list with flag F,
		// drop this packet before forwarding
		if((black_list[i].num==sport)&&(black_list[i].type=='F')){
		// logging
			printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","DROP(FORWARD)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);
			return NF_DROP;
		}
	}
```

패킷을 필터링하는 부분이다. 블랙리스트의 원소들을 확인하며 현재 패킷의 소스포트가 블랙리스트에 F type 으로 등록 돼있는지 확인한다. 만약 그렇다면, 이 패킷은 포워딩될 수 없는 패킷이므로, 커널에 필터링 정보를 출력한고 패킷을 버린다.

```
// logging
	printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-15s,%d,%d,%d,%d\n","FORWARD", ih->protocol,
sport, dport, saddr, daddr, syn, fin, ack, rst);

	//	forward this packet
	return NF_ACCEPT;
}
```

만약 현재 패킷의 소스 포트가 블랙리스트에 F type 으로 등록 돼 있지 않다면 패킷이 포워딩되도록 accept 한다.

```
// overload hook functions for forwarding
static struct nf_hook_ops hook_forward_ops = {
	.hook = hook_forward,
	.pf = PF_INET,
	.hooknum = NF_INET_FORWARD,
	.priority = NF_IP_PRI_FIRST
};
```

위에서 정의한 함수를 NF_INET_FORWARD 후킹 포인트에 implement 하기위해 구조체를 선언하고 구조체의 멤버들을 초기화한다. 후킹 함수의 함수포인터, 프로토콜 family, 후킹포인트, 후킹 우선순위 값들을 할당한다. 함수 포인터는 위에서 정의한 함수의 이름인 hook_forward 이고, 프로토콜 family 는 TCP/IP 이므로 PF_INET, 후킹포인트는 NF_INET_forward, 우선 순위는 최우선이다.

```
// customized hook : monitor NF_INET_POST_ROUTING
static unsigned int hook_post_routing(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
	// get ip header of skb
	struct iphdr *ih = ip_hdr(skb);

	// get tcp header of skb
	struct tcphdr *th = tcp_hdr(skb);

	int sport, dport;			// source port, destination port
	char saddr[16], daddr[16];		// source ip address, destination ip address
	bool syn,fin,ack,rst;			// tcp flags

	int i;					// iterator to manage black list

	snprintf(saddr, 16, "%pI4", &ih->saddr);	// formatted print as IP
	snprintf(daddr, 16, "%pI4", &ih->daddr);	// formatted print as IP
	sport = ntohs(th->source);			// format source port info in tcp header
	dport = ntohs(th->dest);			// format destination port info in tcp header

	//	get tcp flags from tcp header
	syn = th->syn;
	fin = th->fin;
	ack = th->ack;
	rst = th->rst;

		// filtering : whether this packet must be filterted or not : safe destination?
	for(i = 0 ; i < port_size ; i++)
	{
		// if destination port of this packet is in the black list with flag O,
		// drop this packet as destination may not be safe
		if((black_list[i].num==dport)&&(black_list[i].type=='O')){
		// logging
			printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","DROP(OUTBOUND)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);
			return NF_DROP;
		}
	}

	// logging
```

```
        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-15s,%d,%d,%d,%d\n","OUTBOUND", ih->protocol,
sport, dport, saddr, daddr, syn, fin, ack, rst);
        return NF_ACCEPT;
}

// overload hook functions for post routing
static struct nf_hook_ops hook_post_routing_ops = {
        .hook = hook_post_routing,
        .pf = PF_INET,
        .hooknum = NF_INET_POST_ROUTING,
        .priority = NF_IP_PRI_FIRST
};
```

위의 함수는 hook_post_routing 은 IPv4 의 후킹 포인트 NF_INET_post_routing 에서 동작하는 함수로, 이 포인트에서 후킹되는 패킷은 라우팅 결정 후, 현재 네트워크를 벗어나려는 패킷이다.. 이 후킹포인트에 함수를 implement 하기 위한 구조체 hook_post_routing_ops 이다. 이제 코드를 자세히 살펴보도록한다.

```
// customized hook : monitor NF_INET_POST_ROUTING
static unsigned int hook_post_routing(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
        // get ip header of skb
        struct iphdr *ih = ip_hdr(skb);

        // get tcp header of skb
        struct tcphdr *th = tcp_hdr(skb);

        int sport, dport;                       // source port, destination port
        char saddr[16], daddr[16];          // source ip address, destination ip address
        bool syn,fin,ack,rst;                   // tcp flags

        int i;                                          // iterator to manage black list

        snprintf(saddr, 16, "%pI4", &ih->saddr);        // formatted print as IP
        snprintf(daddr, 16, "%pI4", &ih->daddr);        // formatted print as IP
        sport = ntohs(th->source);                              // format source port info in tcp header
        dport = ntohs(th->dest);                                // format destination port info in tcp header

        //    get tcp flags from tcp header
        syn = th->syn;
```

```
        fin = th->fin;
        ack = th->ack;
        rst = th->rst;
```

함수의 선언부이고 소스와 목적지 정보, TCP 패킷 flag들을 초기화하는 부분이다. 이 부분은 앞서 설명한 hook_pre_routing의 선언부와 소스와 목적지 정보, TCP 패킷 flag들을 초기화하는 부분과 동일하므로 설명을 생략한다.

```
// filtering : whether this packet must be filterted or not : safe destination?
        for(i = 0 ; i < port_size ; i++)
        {
                // if destination port of this packet is in the black list with flag O,
                // drop this packet as destination may not be safe
                if((black_list[i].num==dport)&&(black_list[i].type=='O')){
                // logging
                        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-
15s,%d,%d,%d,%d\n","DROP(OUTBOUND)", ih->protocol, sport, dport, saddr, daddr, syn, fin, ack, rst);
                        return NF_DROP;
                }
        }
```

패킷을 필터링하는 부분이다. 블랙리스트의 원소들을 확인하며 현재 패킷의 목적지 포트가 블랙리스트에 O type 으로 등록 돼있는지 확인한다. 만약 그렇다면, 이 패킷은 현재 네트워크를 벗어날 수 없는 패킷이므로, 커널에 필터링 정보를 출력한고 패킷을 버린다.

```
        // logging
        printk(KERN_ALERT "%-15s:%2u,%5d,%5d,%-15s,%-15s,%d,%d,%d,%d\n","OUTBOUND", ih->protocol,
sport, dport, saddr, daddr, syn, fin, ack, rst);
        return NF_ACCEPT;
}
```

만약 현재 패킷의 소스 포트가 블랙리스트에 O type 으로 등록 돼 있지 않다면 패킷이 현재 네트워크를 떠나도록 accept 한다.

```
// overload hook functions for post routing
static struct nf_hook_ops hook_post_routing_ops = {
        .hook = hook_post_routing,
        .pf = PF_INET,
        .hooknum = NF_INET_POST_ROUTING,
        .priority = NF_IP_PRI_FIRST
};
```

위에서 정의한 함수를 NF_INET_POST_ROUTING 후킹 포인트에 implement 하기위해 구조체를 선언하고 구조체의 멤버들을 초기화한다. 후킹 함수의 함수포인터, 프로토콜 family, 후킹포인트, 후킹 우선순위 값들을 할당한다. 함수 포인터는 위에서 정의한 함수의 이름인 hook_post_routing 이고, 프로토콜 family 는 TCP/IP 이므로 PF_INET, 후킹포인트는 NF_INET_POST_ROUTING, 우선 순위는 최우선이다.

```
// customized open : to manage the black list
static int as_open(struct inode *inode, struct file *file)
{
	// action : show? add? delete? the black list
	char const* const str = file->f_path.dentry->d_name.name;
	printk(KERN_INFO "proc file open : %s.\n",str);

	return 0;
}
```

위 함수는 Netfilter 동작을 proc file system 을 활용하여 관리하기위해 proc file 을 open 하는 함수이다. 인수로 전달되는 file 는 포트 블랙리스트를 관리하기위한 명령어로 생각할 수 있다.

```
// customized read : to check black list information
static ssize_t as_show(struct file *file, char __user *ubuf, size_t size, loff_t *ppos)
{
	int len = 0;				// legth of black list info : rules
	char buf[SHOW_BUFSIZE];		// temp buffer for deliver rules to user

	int i;					// iterator to manage the rules

	// to finish "cat"
	if(!finish_cat){
		finish_cat = true;
	}
	else{
		finish_cat = false;
		return 0;
	}

	// print all the rules to temp buffer
	for(i = 0 ; i < port_size ; i++)
	{
		char port_info[20]; // info of one rule
```

```
            int port_info_len;    // length of info of one rule
            sprintf(port_info, "%d(%c) %d\n", i, black_list[i].type, black_list[i].num);

            // concate one rule to temp buffer
                sprintf(buf+len, "%s", port_info);

            port_info_len = strlen(port_info);

            len += port_info_len;

        }

        // print temp buffer to user space
        if(copy_to_user(ubuf, buf, len))
        {
            char* err_msg = "fail to copy to user\n";
                printk(KERN_INFO "                %s", err_msg);
            return 0;
        }

        // move file pointer
        *ppos = len;

        return len;
}
```

Netfilter 를 통해 필터링할 포트 블랙리스트를 유저 공간에 출력하는 함수이다. 함수를 자세히 살펴보자.

```
// customized read : to check black list information
static ssize_t as_show(struct file *file, char __user *ubuf, size_t size, loff_t *ppos)
{
        int len = 0;                            // legth of black list info : rules
        char buf[SHOW_BUFSIZE];                 // temp buffer for deliver rules to user

        int i;                                  // iterator to manage the rules
```

함수의 선언부이다. 커널 데이터를 유저 공간에 출력하는 것은 read(2)와 동일한 동작이다. 따라서 read(2)를 overload 하기위해 반환형과 인자들을 read(2)와 동일하게 설정한다. 유저 공간에 출력할 글자 수를 저장할 변수와 유저공간에 커널 데이터를 복사하기위해 사용할

버퍼를 정의한다. 포트 블랙리스트를 모두 출력해야하므로 이를 순환하기위한 이터레이터를 정의한다.

```
// to finish "cat"
	if(!finish_cat){
		finish_cat = true;
	}
	else{
		finish_cat = false;
		return 0;
	}
```

유저공간에서 cat 을 통해 proc file 을 read 한다. cat 은 read(2)를 통해 file 을 읽는데, read 가 0 을 반환해야만 read 를 멈춘다. 이 반환값만큼 ubuf 에서 데이터를 읽어서 유저 공간에 출력하기때문에, 딜레마가 발생한다. Read 를 통해 읽은 글자수를 반환하면 cat 이 무한히 read 를 호출하고, 0 을 반환하면 유저 공간에 커널 데이터가 출력되지 않는다. 따라서 이 구문을 추가하여 finish_cat flag 가 거짓일 때, cat 이 한 번 read 를 호출한 후, flag 를 참으로 바꾸고, 다음 이터레이션에서 flag 를 다시 거짓으로 바꾸고 read 가 0 을 반환하여 cat 을 read finish 하도록 하는 구문이다. 이를 통해 유저공간에서 cat 으로 proc file 을 호출하는 것에 문제가 발생하지 않는다.

```
// print all the rules to temp buffer
	for(i = 0 ; i < port_size ; i++)
	{
		char port_info[20]; // info of one rule
		int port_info_len;    // length of info of one rule
		sprintf(port_info, "%d(%c) %d\n", i, black_list[i].type, black_list[i].num);

		// concate one rule to temp buffer
		sprintf(buf+len, "%s", port_info);

		port_info_len = strlen(port_info);

		len += port_info_len;

	}
```

커널 공간의 데이터인 포트블랙리스트 정보를 중간 버퍼인 buf 에 출력하는 부분이다. 모든 포트블랙리스트를 한 번에 출력하기 위해 각 포트 정보를 이어 붙인다. port_info 버퍼는 한 포트에 대한 정보이고, 이를 buf 에 이어 붙인다. len 은 현재까지 읽은 글자수이다.

```
// print temp buffer to user space
```

```
    if(copy_to_user(ubuf, buf, len))
    {
        char* err_msg = "fail to copy to user\n";
            printk(KERN_INFO "                %s", err_msg);
        return 0;
    }

    // move file pointer
    *ppos = len;

    return len;
}
```

모든 포트블랙리스트 정보를 유저 공간의 버퍼에 복사한다. 읽은 글자수를 offset 변수를 관리하는 *ppos 에 할당하고, as_show 는 읽은 글자수를 반환한다.

```
// overloading : show as read operation as it reads the black list
static const struct file_operations show_fops = {
    .owner = THIS_MODULE,
    .open = &as_open,
    .read = &as_show,
};
```

Proc file show 의 file operation 을 위에서 정의한 함수들로 overloading 하기 위한 구조체이다. 앞서 언급했듯, as_show 는 read 를 overloading 한다.

```
// customized write : to add filtering ports
static ssize_t as_add(struct file *file, const char __user *ubuf, size_t size, loff_t *ppos)
{
    int len = 0;                    // length of command

    char buf[WRITE_BUFSIZE];

    char add_port_type;         // filtering type
    int add_port_num;            // port to be filterted

    // copy commands from user space to kernel
    if(copy_from_user(buf, ubuf, size))
    {
        char* err_msg = "fail to copy from user";
        printk(KERN_INFO "          %s\n",err_msg);
```

```
            return 0;
        }

        // analyze the command, and get a rule
        sscanf(buf, "%c %d", &add_port_type, &add_port_num);

        // add the rule to black list
        black_list[port_size].type = add_port_type;
        black_list[port_size].num = add_port_num;


        // increase length of the black list
        port_size++;

        len = strlen(buf);
        *ppos = len;

        return len;
}
```

Netfilter 를 통해 필터링할 포트 블랙리스트에 원소를 추가하기위한 함수이다. 함수를 자세히 살펴보자.

```
// customized write : to add filtering ports
static ssize_t as_add(struct file *file, const char __user *ubuf, size_t size, loff_t *ppos)
{
        int len = 0;                    // length of command

        char buf[WRITE_BUFSIZE];

        char add_port_type;         // filtering type
        int add_port_num;            // port to be filterted
```

함수의 선언부이다. 유저 공간의 데이터를 커널 공간에 출력하는 것은 write(2)와 동일한 동작이다. 따라서 write(2)를 overload 하기위해 반환형과 인자들을 write(2)와 동일하게 설정한다. 유저 공간에서 읽어들인 글자수를 저장할 변수와 커널에 유저 데이터를 복사하기위해 사용할 버퍼를 정의한다. 유저 데이터로부터 추가될 port 정보를 추출하여 저장할 변수들을 선언한다.

```
// copy commands from user space to kernel
        if(copy_from_user(buf, ubuf, size))
        {
            char* err_msg = "fail to copy from user";
```

```
            printk(KERN_INFO "            %s\n",err_msg);
            return 0;
        }
```

유저가 포트블랙리스트에 추가할 버퍼 정보를 유저버퍼 ubuf 로부터 중간 버퍼인 buf 에 출력하는 부분이다.

```
// analyze the command, and get a rule
        sscanf(buf, "%c %d", &add_port_type, &add_port_num);
```

buf 에는 다음과 같은 형태로 블랙리스트에 추가할 버퍼 정보가 담겨있다.

"(추가될 포트의 필터링 타입) (추가될 포트번호)"

이 값들을 sscanf 를 통해 각 데이터에 맞는 변수에 출력한다.

```
// add the rule to black list
        black_list[port_size].type = add_port_type;
        black_list[port_size].num = add_port_num;


        // increase length of the black list
        port_size++;
```

읽어들인 정보를 포트블랙리스트에 추가한다. 포트블랙리스트의 순서는 FCFS 이다. 원소가 추가됐으므로 블랙리스트 크기를 증가시킨다.

```
        len = strlen(buf);
        *ppos = len;

        return len;
}
```

유저 공간으로부터 읽어들인 글자수를 저장한다, 읽은 글자수를 offset 변수를 관리하는 *ppos 에 할당하고, as_add 는 읽은 글자수를 반환한다.

```
// overloading : add as write operation as it modifies the black list
static const struct file_operations add_fops = {
        .owner = THIS_MODULE,
        .open = &as_open,
        .write = &as_add,
};
```

Proc file add 의 file operation 을 위에서 정의한 함수들로 overloading 하기 위한 구조체이다. 앞서 언급했듯, as_add 는 write 를 overloading 한다.

```
// customized write : to delete filtering ports
static ssize_t as_del(struct file *file, const char __user *ubuf, size_t size, loff_t *ppos)
{
	int len = 0;			// length of command

	char buf[WRITE_BUFSIZE];

	int del_port_idx;		// filtering ports to be deleted

	int i;				// iterator to manage the rules

	// copy commands from user space to kernel
	if(copy_from_user(buf, ubuf, size))
	{
		char* err_msg = "fail to copy from user";
		printk(KERN_INFO "		%s\n",err_msg);
		return 0;
	}

	// analyze the command, and get a rule : get index of rule to be deleted
	sscanf(buf, "%d", &del_port_idx);

	// decrease size of the black list
	port_size--;

	// erase the rule with the index
	for(i = del_port_idx ; i < port_size ; i++)
	{
		black_list[i] = black_list[i+1];
	}

	len = strlen(buf);
	*ppos = len;

	return len;
}
```

Netfilter 를 통해 필터링할 포트 블랙리스트에서 인덱스를 통해 원소를 제거하기위한 함수이다. 함수를 자세히 살펴보자.

```
// customized write : to delete filtering ports
```

```
static ssize_t as_del(struct file *file, const char __user *ubuf, size_t size, loff_t *ppos)
{
	int len = 0;			// length of command

	char buf[WRITE_BUFSIZE];

	int del_port_idx;		// filtering ports to be deleted

	int i;				// iterator to manage the rules
```

함수의 선언부이다. 유저 공간의 데이터를 커널 공간에 출력하는 것은 write(2)와 동일한 동작이다. 따라서 write(2)를 overload 하기위해 반환형과 인자들을 write(2)와 동일하게 설정한다. 유저 공간에서 읽어들인 글자수를 저장할 변수와 커널에 유저 데이터를 복사하기위해 사용할 버퍼를 정의한다. 유저 데이터로부터 블랙리스트에서 삭제될 포트에 접근하기위한 인덱스를 추출하여 저장할 변수를 선언한다. 배열로 블랙리스트를 관리하므로 원소 삭제 후, 원소의 이동이 필요하므로 이터레이터 변수를 선언한다.

```
// copy commands from user space to kernel
	if(copy_from_user(buf, ubuf, size))
	{
		char* err_msg = "fail to copy from user";
		printk(KERN_INFO "		%s\n",err_msg);
		return 0;
	}
```

유저가 포트블랙리스트에서 삭제할 버퍼 정보를 유저버퍼 ubuf 로부터 중간 버퍼인 buf 에 출력하는 부분이다.

```
// analyze the command, and get a rule : get index of rule to be deleted
	sscanf(buf, "%d", &del_port_idx);
```

buf 에는 다음과 같은 형태로 블랙리스트에 추가할 버퍼 정보가 담겨있다.

"(삭제할 포트가 저장된 인덱스)"

이 값을 sscanf 를 통해 데이터에 맞는 변수에 출력한다.

```
// decrease size of the black list
	port_size--;

	// erase the rule with the index
	for(i = del_port_idx ; i < port_size ; i++)
	{
		black_list[i] = black_list[i+1];
	}
```

블랙리스트에서 포트정보를 삭제하는 과정이다. 원소가 하나 삭제 됐으므로 Port_size 를 줄인다. 그리고 삭제된 원소보다 큰 인덱스를 갖는 원소들을 한 칸씩 앞당겨 저장한다.

```
    len = strlen(buf);
    *ppos = len;

    return len;
}
```

유저 공간으로부터 읽어들인 글자수를 저장한다, 읽은 글자수를 offset 변수를 관리하는 *ppos 에 할당하고, as_del 은 읽은 글자수를 반환한다.

```
// overloading : del as write operation as it modifies the black list
static const struct file_operations del_fops = {
    .owner = THIS_MODULE,
    .open = &as_open,
    .write = &as_del,
};
```

Proc file del 의 file operation 을 위에서 정의한 함수들로 overloading 하기 위한 구조체이다. 앞서 언급했듯, as_del 은 write 를 overloading 한다.

```
// initialize : make proc file
static int __init simple_init(void)
{
    printk(KERN_INFO "Simple Module Init!!\n");

    proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
    add_file = proc_create(FILENAME_ADD, 0777, proc_dir, &add_fops);
    show_file = proc_create(FILENAME_SHOW, 0777, proc_dir, &show_fops);
    del_file = proc_create(FILENAME_DEL, 0777, proc_dir, &del_fops);

    nf_register_hook(&hook_pre_routing_ops);
    nf_register_hook(&hook_forward_ops);
    nf_register_hook(&hook_post_routing_ops);

    return 0;
}
```

커널에 지금까지 작성한 모듈들을 삽입하기 위한 함수이다. Proc file directory 를 생성하고, 그 하위에 proc file 의 abstraction 인 proc file entry 들을 생성하고 각 파일의 목적에

맞게 file operation 을 overloading 한다. nf_register_hook 함수를 통해 Customized Hook 함수들을 커널에 등록한다.

```
// When dispatching this module, remove all this module made
static void __exit simple_exit(void)
{
	printk(KERN_INFO "Simple Module Exit!!\n");

	remove_proc_entry(FILENAME_ADD, proc_dir);
	remove_proc_entry(FILENAME_SHOW, proc_dir);
	remove_proc_entry(FILENAME_DEL, proc_dir);
	remove_proc_entry(PROC_DIRNAME, NULL);

	nf_unregister_hook(&hook_pre_routing_ops);
	nf_unregister_hook(&hook_forward_ops);
	nf_unregister_hook(&hook_post_routing_ops);

	return;
}
```

커널에서 LKM 모듈을 제거하기 위한 함수이다. simple_exit 생성된 proc file 공간을 반환하고 file 과 directory 를 삭제한다. nf_unregister_hook 함수를 통해 Customized Hook 함수들을 커널에서 제거한다.

```
module_init(simple_init);
module_exit(simple_exit);
```

module_init 은 simple_init 을 실행한다. module_exit 은 simple_exit 을 실행한다.

```

MODULE_AUTHOR("YUNYURIM");
MODULE_DESCRIPTION("It's Simple!!");
MODULE_LICENSE("GPL");
MODULE_VERSION("NEW");
```

3. 실행 방법

Step0. Ip_forward 사용 설정 : sudo sysctl -w net.ipv4.ip_forward=1

Step1. 서버 vm 의 전원을 키고 server2 를 실행한다. Server port 로 사용할 포트 번호를 입력한다.

Step2. 클라이언트 vm 의 전원을 키고 2 차 과제의 결과물인 2017320256_client 를 실행한다. 서버의 ip 주소를 입력하고 client port 로 사용할 포트의 수와 포트 번호를 입력한다.

Step3. group36.c 를 컴파일하여 LKM 인 group36.ko 를 생성한다.

Step4. sudo insmod group36.ko 를 통해 LKM 모듈을 커널에 삽입한다.

Step5. /proc/group36 에서 다음의 명령어로 필터링할 포트정보를 관리한다.

cat show : 포트블랙리스트정보를 출력한다.

echo "(필터링타입) (포트번호)" > add : 포트블랙리스트에 포트를 추가한다.

I type : Inbound packet 에 대한 필터링으로, 소스 포트가 적절한지 평가한다.

F type : Forwarding packet 에 대한 필터링으로, 소스 포트가 적절한지 평가한다.

O type : Outbound packet 에 대한 필터링으로, 목적지 포트가 적절한지 평가한다.

P type : Proxy packet 에 대한 필터링으로, 패킷의 목적지 주소를 변환한다.

echo "(삭제할 포트가 저장된 인덱스)" > del : 포트블랙리스트에서 포트를 인덱스를 이용해 접근하여 삭제한다.

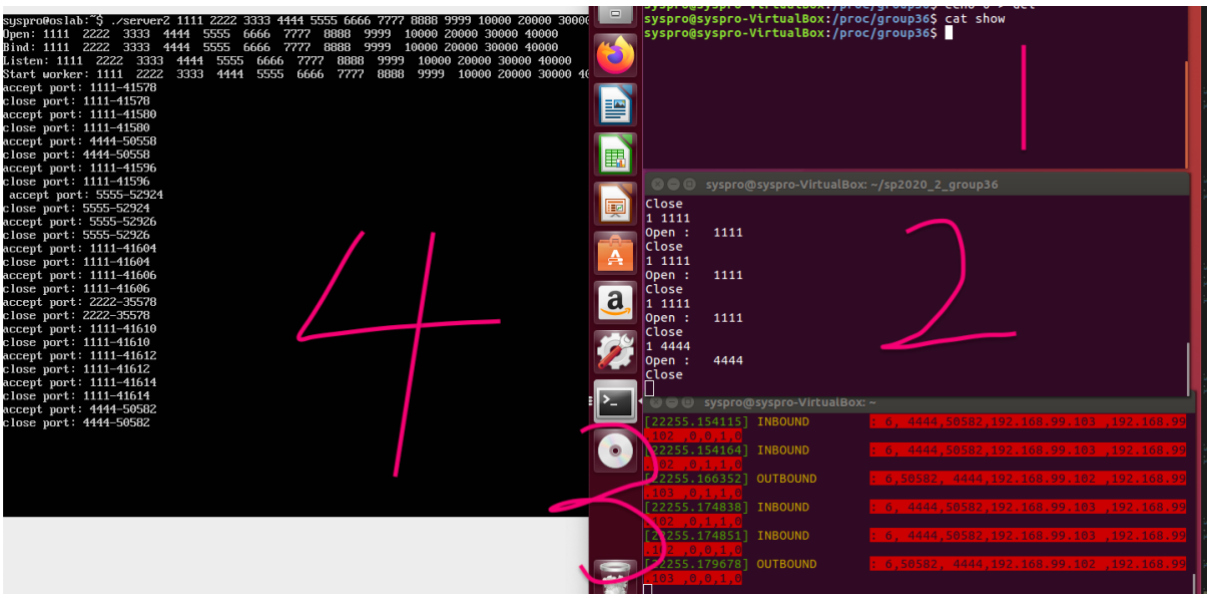
Step6. dmesg -w 를 통해 Netfilter 로그를 확인하며 패킷 이동을 관찰한다.

Step7. Sudo rmmod group36 을 통해 실험을 종료한다.

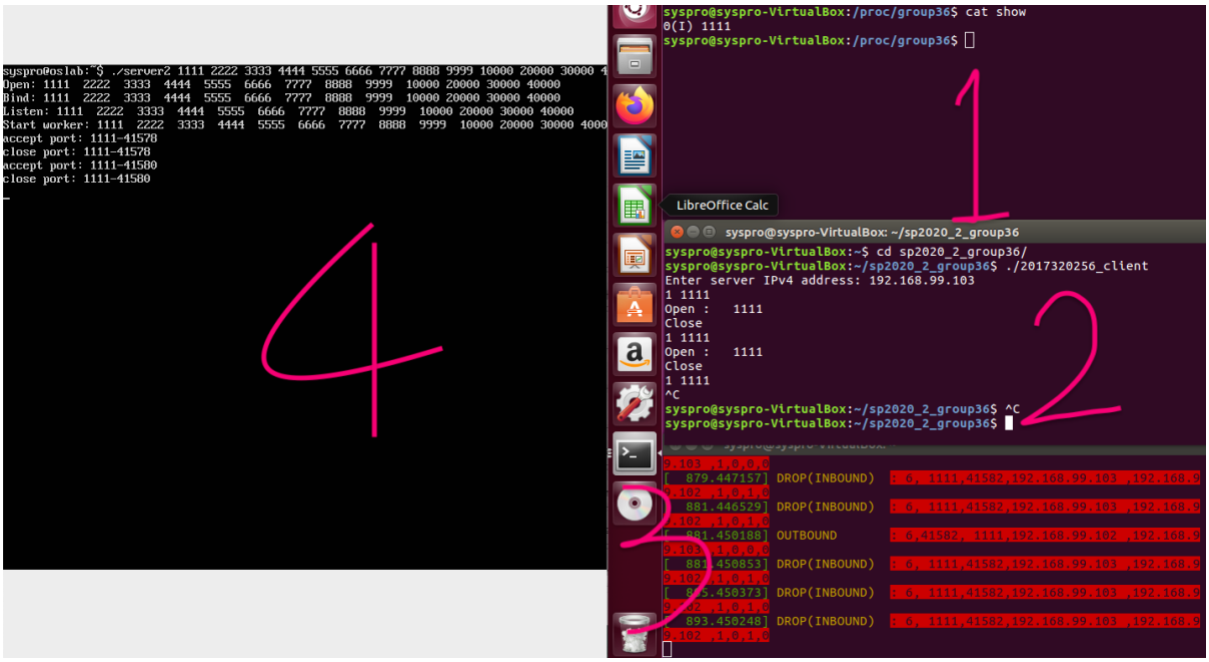
4. 실험 결과 및 분석

서버의 IP 주소 : 192.168.99.103

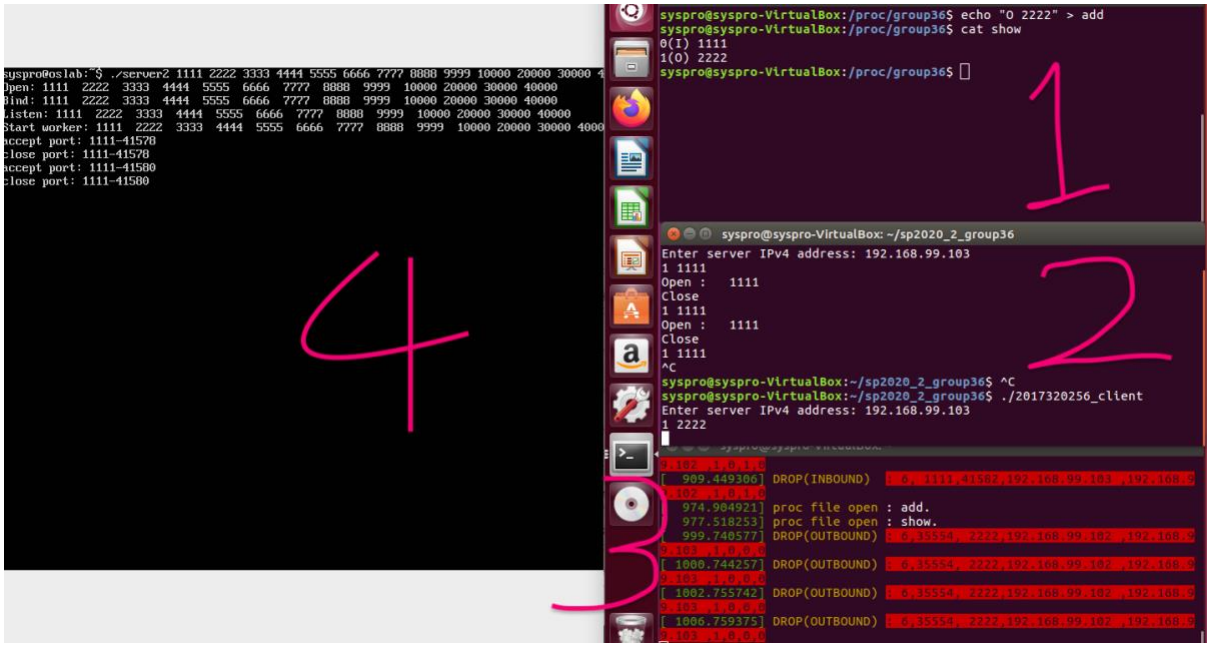
클라이언트의 IP 주소 : 192.168.99.102



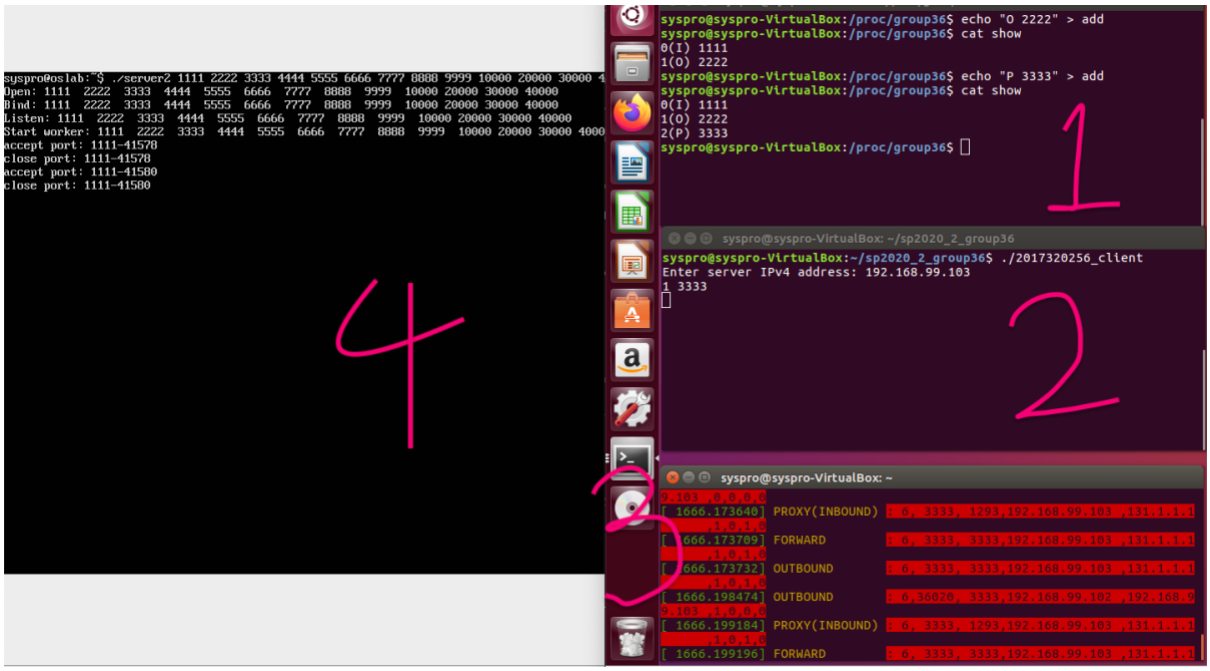
창 1 은 블랙리스트 관리를 위한 proc, 창 2 는 클라이언트, 창 3 은 netfilter 로그, 창 4 는 서버이다. 블랙리스트가 비어있을 때, 서버와 클라이언트의 통신을 보여준다. 창 3 의 로그를 확인해보면 각 라인의 첫번째 인자가 모두 6 임을 확인할 수 있는데, 이는 서버와 클라이언트가 TCP 패킷을 주고 받는 다는 것을 알 수 있다. 3 번째 패킷부터 차례로 살펴보면, 클라이언트의 포트 58582 가 소켓을 닫기 위해 서버의 포트 4444 에게 FIN+ACK 패킷을 보낸 것을 확인할 수 있고, 서버는 이에 대한 대답으로 FIN+ACK 패킷을 보냈다. 클라이언트는 이에 대한 대답으로 ACK 패킷을 서버에게 보냈고, 서버와 클라이언트의 연결이 안전하게 종료된 것을 창 2 와 4 를 통해 확인할 수 있다.



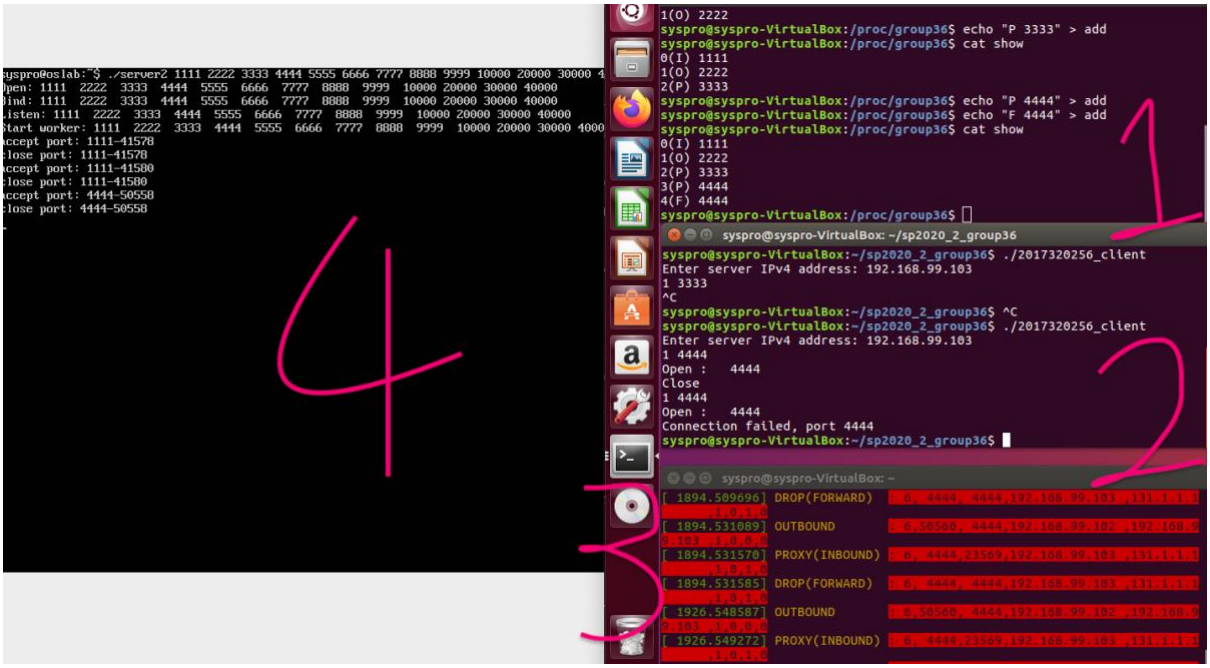
창 1 은 블랙리스트 관리를 위한 proc, 창 2 는 클라이언트, 창 3 은 netfilter 로그, 창 4 는 서버이다. 클라이언트가 1111 포트를 사용하는 소스로 부터 온 패킷을 버리도록 rule 을 설정하고 블랙리스트에 등록한다. 창 3 의 세번째 로그부터 확인하면, 모든 패킷들의 Type 이 6 인 것을 보아 TCP 패킷들임을 알 수 있다. 클라이언트의 포트 41582 에서 서버 포트 1111 에 tcp 연결을 위해 SYN 패킷을 보냈다. 서버는 포트 1111 을 통해 SYN+ACK 패킷을 전송하지만, netfilter hook(NF_INET_PRE_ROUTING)은 블랙리스트의 0 번째 rule 에 따라 해당 패킷을 버린다. ACK 를 받지 못한 서버가 계속해서 클라이언트로 패킷을 보내지만, 이 패킷들 역시 모두 버려진다. 창 2 와 4 를 확인하면 TCP 연결이 성립되지 못한 것을 확인할 수 있다.



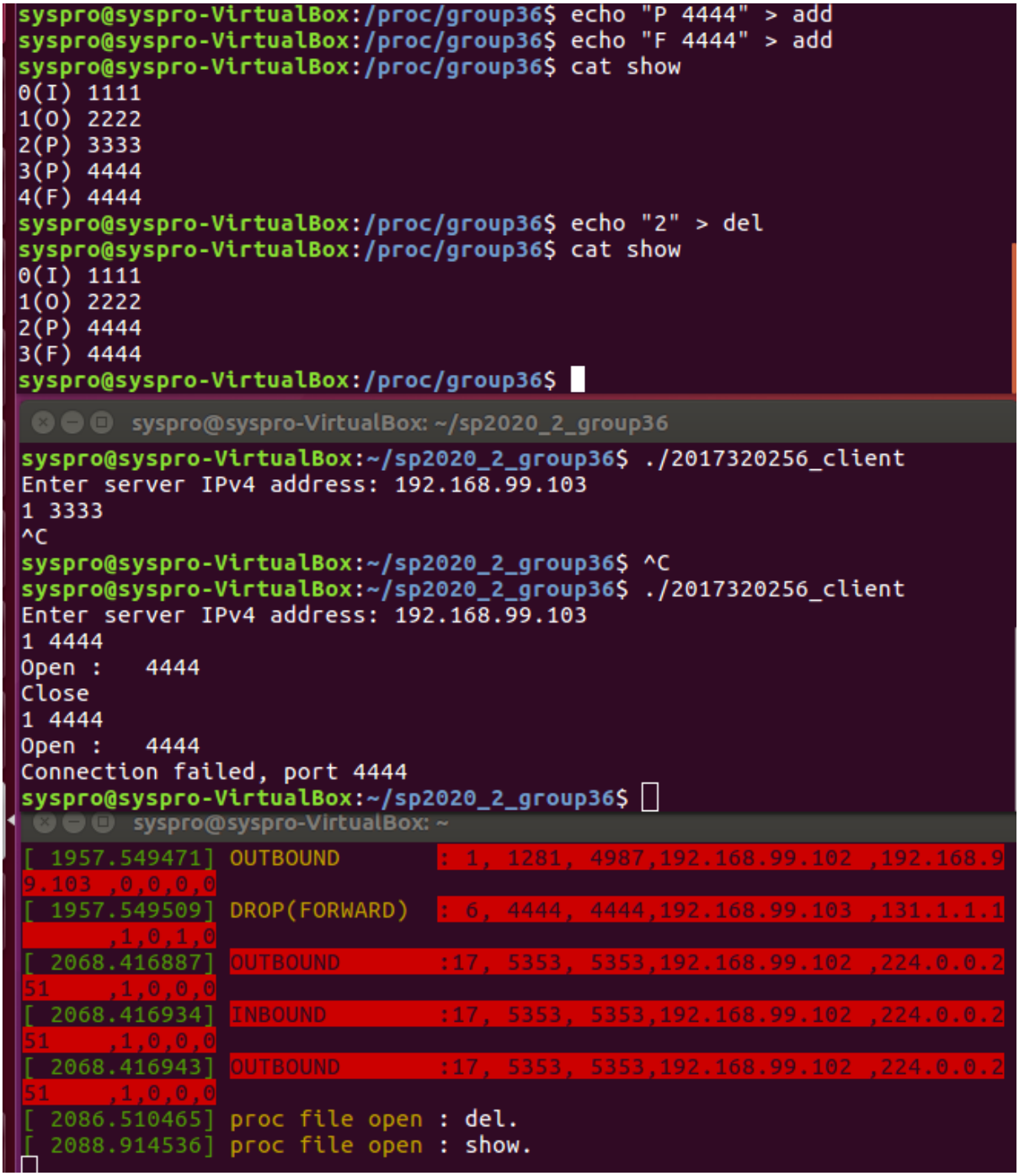
창 1 은 블랙리스트 관리를 위한 proc, 창 2 는 클라이언트, 창 3 은 netfilter 로그, 창 4 는 서버이다. 블랙리스트에 클라이언트의 패킷의 목적지 포트가 2222 라면 패킷을 버리도록 하는 rule 을 추가한다. 창 3 의 네번째 로그부터 확인하면, 모든 패킷들의 Type 이 6 인 것을 보아 TCP 패킷들임을 알 수 있다. 클라이언트의 포트 35554 에서 서버 포트 2222 에 tcp 연결을 위해 SYN 패킷을 보낸다. 하지만 netfilter hook(NF_INET_POST_ROUTING)은 블랙리스트의 1 번째 rule 에 따라 해당 패킷을 버린다. SYN+ACK 를 받지 못한 클라이언트가 계속해서 서버로 패킷을 보내지만, 이 패킷들 역시 모두 버려진다. 창 2 와 4 를 확인하면 TCP 연결이 성립되지 못한 것을 확인할 수 있다.



창 1 은 블랙리스트 관리를 위한 proc, 창 2 는 클라이언트, 창 3 은 netfilter 로그, 창 4 는 서버이다. 블랙리스트에 클라이언트의 패킷의 소스 포트가 3333 이라면 프록시 패킷으로 간주하여 패킷의 목적지를 수정하여 포워딩하는 rule 을 추가한다. 창 3 의 첫번째 로그부터 확인하면, 모든 패킷들의 Type 이 6 인 것을 보아 TCP 패킷들임을 알 수 있다. 서버 포트 3333 에서 클라이언트 포트 1293 에 패킷을 보냈으나, netfilter hook(NF_INET_PRE_ROUTING)은 블랙리스트의 2 번째 rule 에 따라 해당 패킷의 목적지 주소를 프록시패킷을 처리할 주소로 변경하여 accept 한다. 라우터는 변경된 목적지 주소에 따라 패킷을 포워딩한다. 이때 프록시의 목적지 주소가 local 이 아니므로 패킷은 local 로 들어오지 않고 Outbound 로 현재 네트워크를 나간다.



창 1 은 블랙리스트 관리를 위한 proc, 창 2 는 클라이언트, 창 3 은 netfilter 로그, 창 4 는 서버이다. 블랙리스트에 클라이언트의 패킷의 소스 포트가 4444 이라면 프록시 패킷으로 간주하여 패킷의 목적지를 수정하여 포워딩하는 rule 을 추가한다. 그리고 포워딩될 패킷의 소스 포트가 4444 라면 패킷을 버리기위한 rule 을 추가한다. 창 3 의 세번째 로그부터 확인하면, 모든 패킷들의 Type 이 6 인 것을 보아 TCP 패킷들임을 알 수 있다. 서버 포트 4444 에서 클라이언트 포트 12569 에 패킷을 보냈으나, netfilter hook(NF_INET_PRE_ROUTING)은 블랙리스트의 3 번째 rule 에 따라 해당 패킷의 목적지 주소를 프록시패킷을 처리할 주소로 변경하여 accept 한다. 라우터는 변경된 목적지 주소에 따라 패킷을 포워딩한다. 이때 netfilter hook(NF_INET_FORWARD) 지점에서 블랙리스트의 4 번째 rule 에 따라 패킷이 포워딩 되지 않고 버려진다.



위의 그림은 블랙리스트 인덱스를 이용하여 rule 을 삭제하는 과정을 보여준다. 가장 위의 창에서 echo "2" > del 을 통해 블랙리스트의 2 번째 rule 을 삭제한 것을 확인할 수 있다.

5. 과제 수행 시 어려웠던 부분과 해결 방법

A. cat 의 read(2) 무한 호출

proc file 을 활용하여 블랙리스트 rule 을 출력할 때 다음과 같은 명령어를 사용한다.

cat > show

이때 show 는 모든 rule 을 한 번만 출력해야 한다. 하지만 위의 명령어를 실행 했을 때, 다음과 같은 일이 발생했다.

```
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)    5555
0(e)^C
```

cat 을 한 번만 호출했음에도, 블랙리스트의 모든 내용이 무한히 출력되는 현상이다. 유저공간에서 cat 을 통해 proc file 을 read 한다. cat 은 read(2)를 통해 file 을 읽는데, read 가 0 을 반환해야만 read 를 멈춘다. 한편, cat 은 read 의 반환값만큼 유저 버퍼에서 데이터를 읽어서 유저 공간에 출력하기때문에, 딜레마가 발생한다. Read 를 통해 읽은 글자수를 반환하면 cat 이 무한히 read 를 호출하고, 0 을 반환하면 유저 공간에 커널 데이터가 출력되지 않는다. 따라서 아래의 구문을 customized read 인 as_show 함수에 추가하여 finish_cat flag 가 거짓일 때, cat 이 한 번 read 를 호출한 후, flag 를 참으로 바꾸고, 다음 이터레이션에서 flag 를 다시 거짓으로 바꾸고 read 가 0 을 반환하여 cat 을 read finish 하도록 하는 구문이다. 이를 통해 유저공간에서 cat 으로 proc file 을 호출하는 것에 문제가 발생하지 않는다.

```
// to finish "cat"
        if(!finish_cat){
```

```
        finish_cat = true;
    }
    else{
        finish_cat = false;
        return 0;
    }
```

B. TCP 헤더에서 정보 추출하기

TCP 헤더에서 port 번호를 추출하기위해 tcphdr 구조체의 멤버 source 와 dest 에 접근했다. 값을 저장할 때, unsinged short 로 type casting 을 했으나 원하는 값이 아닌 이상한 정수를 얻었다.

이를 해결하기위해 ntohs 함수를 사용했다. TCP 헤더에 저장된 포트 번호의 자료형은 be16 으로 이를 클라이언트의 byte order 로 변환하기위함이다. 이를 통해 원하는 값을 추출할 수 있었다.