



41526: 네트워크보안

강문수

(mskang@chosun.ac.kr)

컴퓨터네트워크 연구실
(Computer Networks Lab.)

목차

4.1 개요

4.2 패킷 수신 방법

4.3 패킷 스니핑

4.4 패킷 스푸핑

4.5 스니핑 후 스푸핑하기

4.6 하이브리드 방식을 이용한 패킷 스푸핑

4.7 엔디안

4.8 검사합 계산

4.9 요약

4.1 개요

■ 스니핑(sniffing) 공격

- ➡ 공격자는 유선 또는 무선의 물리 네트워크를 도청해서 네트워크를 통해 전송되는 패킷을 캡처할 수 있다.
- ➡ 전화망에 대한 도청 공격과 유사하다

■ 스푸핑(spoofing) 공격

- ➡ 공격자는 거짓 ID로 패킷을 보낼 수 있다.
- ➡ 다른 컴퓨터에서 온 것처럼 위장하는 패킷을 보낼 수 있다

- 이 두 가지 공격은 인터넷에서 DNS 캐시 감염과 TCP 세션 하이재킹 공격과 같은 많은 공격의 기반이 된다.

- 패킷 스니핑과 스푸핑은 Wireshark, netwox 및 Scapy와 같은 도구를 사용하여 수행할 수 있다.

실습 환경 설정

■ 우리 교재의 홈페이지



실습 환경 설정

Packet Sniffing and Spoofing Lab

Overview

Sniffing Spoofing

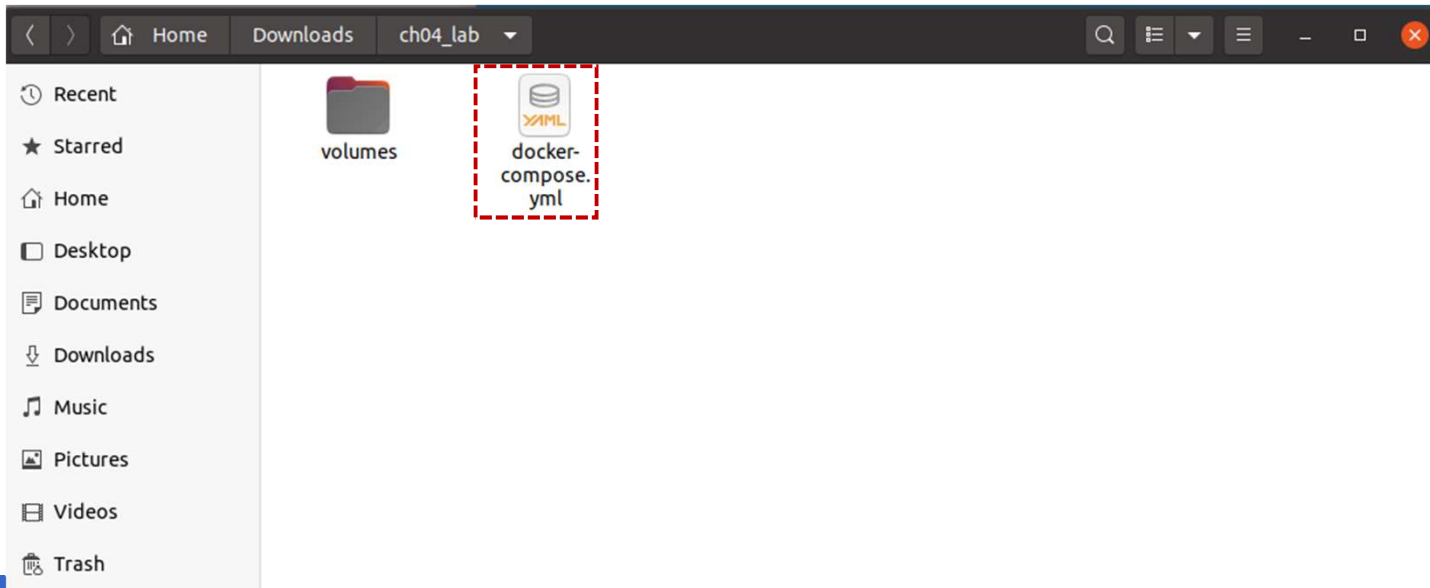
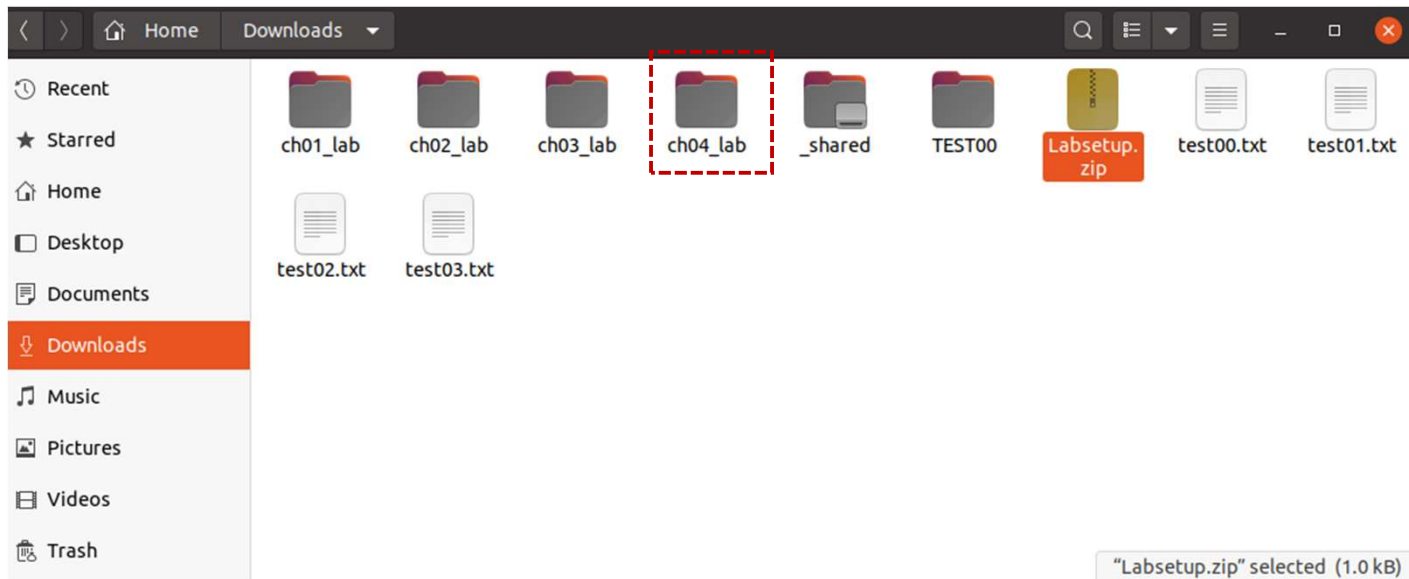
Packet sniffing and spoofing are the two important concepts in network security; they are two major threats in network communication. Being able to understand these two threats is essential for understanding security measures in networking. There are many packet sniffing and spoofing tools, such as Wireshark, Tcpdump, Netwox, etc. Some of these tools are widely used by security experts, as well as by attackers. Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this lab is for students to master the technologies underlying most of the sniffing and spoofing tools. Students will play with some simple sniffer and spoofing programs, read their source code, modify them, and eventually gain an in-depth understanding on the technical aspects of these programs. At the end of this lab, students should be able to write their own sniffing and spoofing programs.

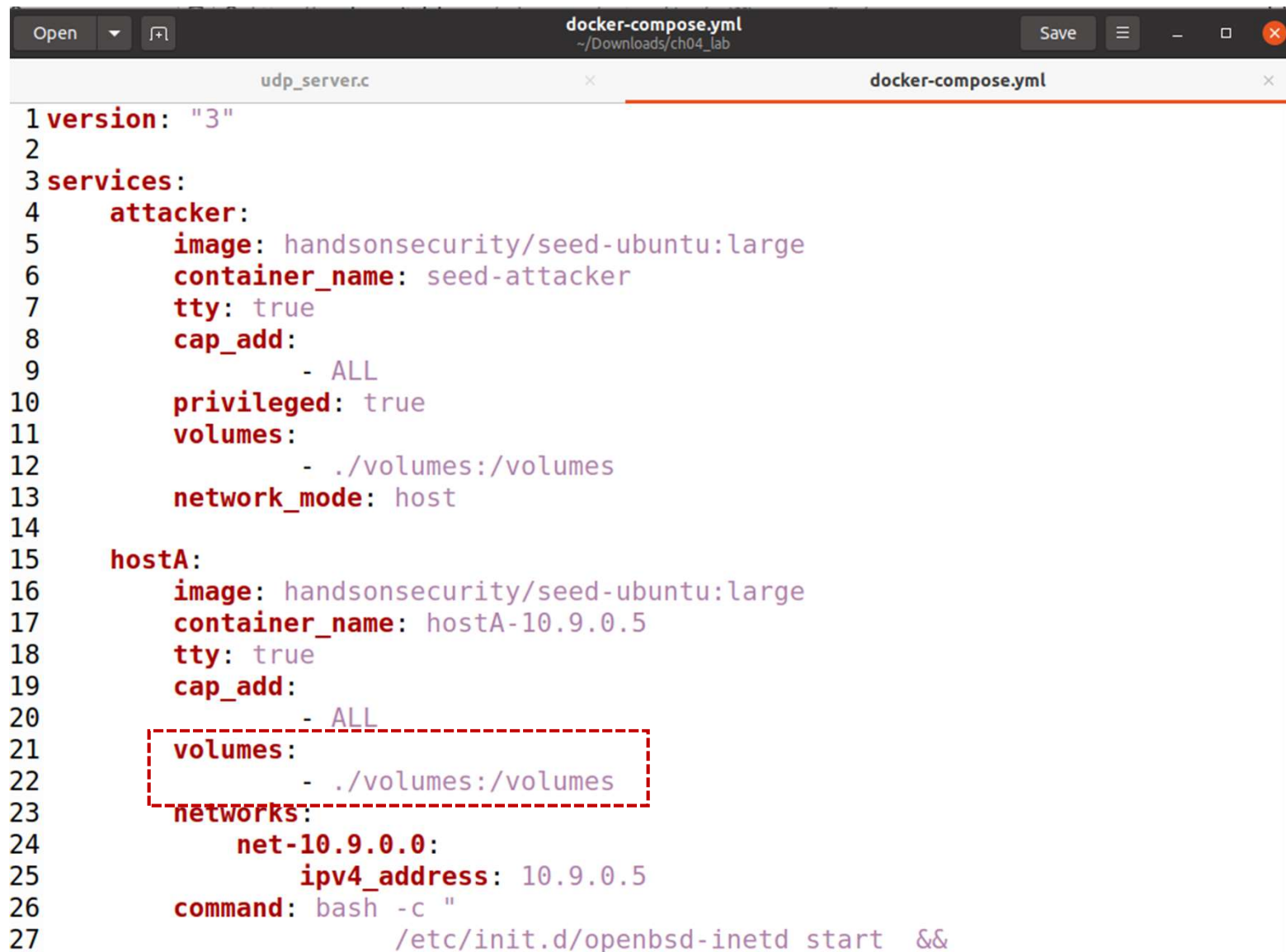
Tasks (English) (Spanish)

- **VM version:** This lab has been tested on our [SEED Ubuntu-20.04 VM](#)
- **Lab setup files**
 - [Labsetup.zip](#)
 - [Labsetup-arm.zip](#) (for Apple Silicon machines)
- **Manual:** [Docker manual](#)

실습 환경 설정

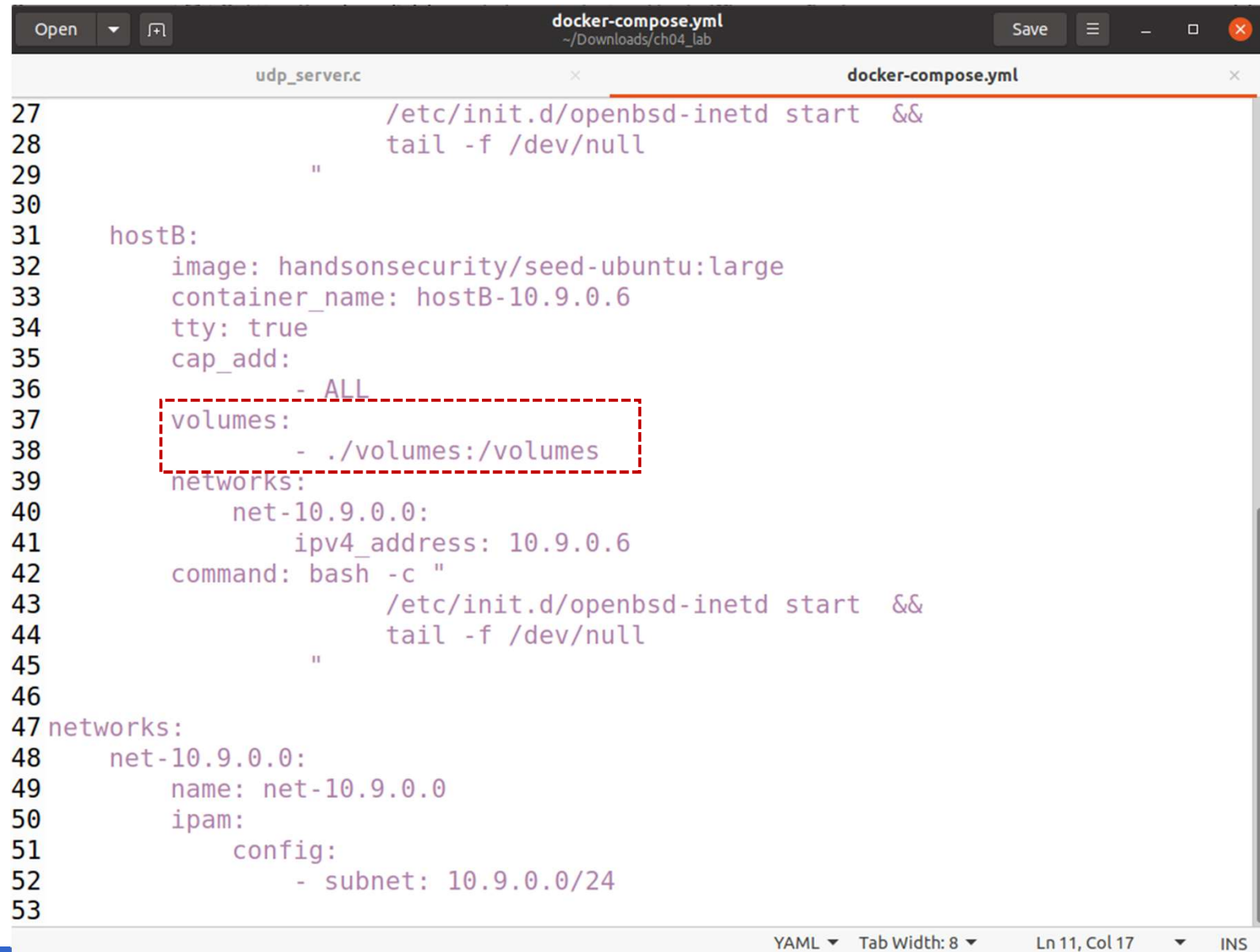


실습 환경 설정



```
1 version: "3"
2
3 services:
4   attacker:
5     image: handsonsecurity/seed-ubuntu:large
6     container_name: seed-attacker
7     tty: true
8     cap_add:
9       - ALL
10    privileged: true
11    volumes:
12      - ./volumes:/volumes
13    network_mode: host
14
15   hostA:
16     image: handsonsecurity/seed-ubuntu:large
17     container_name: hostA-10.9.0.5
18     tty: true
19     cap_add:
20       - ALL
21     volumes:
22       - ./volumes:/volumes
23     networks:
24       net-10.9.0.0:
25         ipv4_address: 10.9.0.5
26     command: bash -c "
27               /etc/init.d/openbsd-inetd start &&
```


실습 환경 설정



```
27         /etc/init.d/openbsd-inetd start  &&
28         tail -f /dev/null
29     "
30
31     hostB:
32         image: handsonsecurity/seed-ubuntu:large
33         container_name: hostB-10.9.0.6
34         tty: true
35         cap_add:
36             - ALL
37         volumes:
38             - ./volumes:/volumes
39         networks:
40             net-10.9.0.0:
41                 ipv4_address: 10.9.0.6
42         command: bash -c "
43             /etc/init.d/openbsd-inetd start  &&
44             tail -f /dev/null
45         "
46
47     networks:
48         net-10.9.0.0:
49             name: net-10.9.0.0
50             ipam:
51                 config:
52                     - subnet: 10.9.0.0/24
53
```


실습 환경 설정

■ 컨테이너 네트워크 실행

```
[05/16/24] seed@VM:~/.../ch04_lab$ docker-compose up
Creating network "net-10.9.0.0" with the default driver
Creating hostB-10.9.0.6 ... done
Creating seed-attacker ... done
Creating hostA-10.9.0.5 ... done
Attaching to seed-attacker, hostB-10.9.0.6, hostA-10.9.0.5
hostB-10.9.0.6 | * Starting internet superserver inetd      [ OK ]
hostA-10.9.0.5 | * Starting internet superserver inetd      [ OK ]
```

■ 컨테이너 호스트 확인

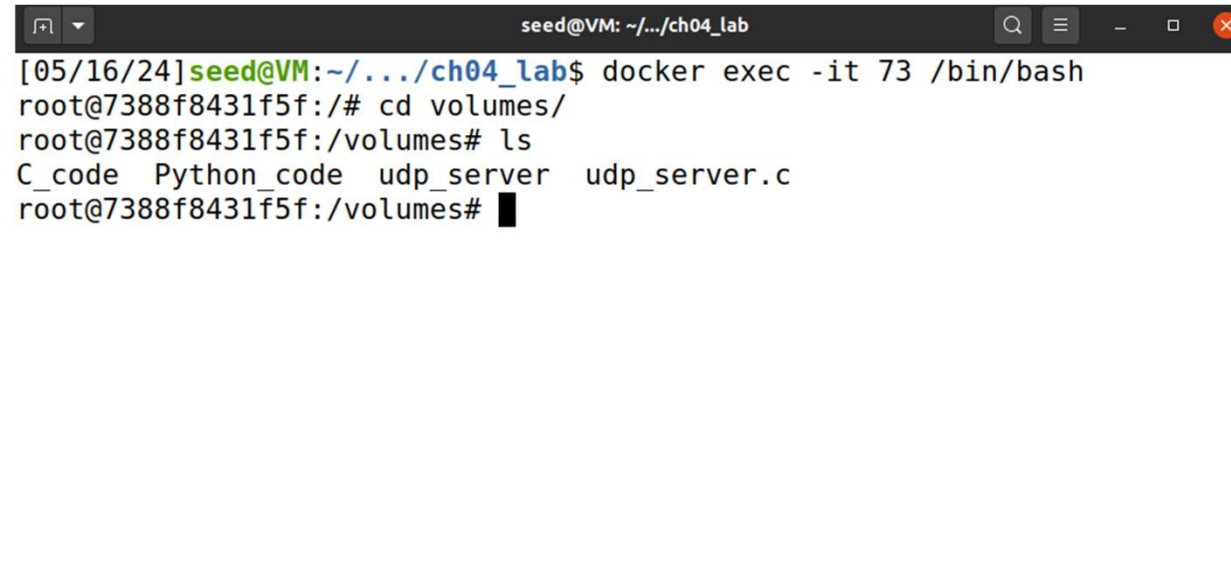
CONTAINER ID	IMAGE	COMMAND
7388f8431f5f	handsonsecurity/seed-ubuntu:large	"bash -c ' /etc/init..."
e98e0780e581	handsonsecurity/seed-ubuntu:large	"/bin/sh -c /bin/bash"
9cd4ffa7807c	handsonsecurity/seed-ubuntu:large	"bash -c ' /etc/init..."

```
[05/16/24] seed@VM:~/.../volumes$
```

실습 환경 설정

■ 3개의 터미널 준비

■ hostA

A terminal window titled 'seed@VM: ~/.../ch04_lab' with standard window controls. The terminal shows a sequence of commands and their outputs: a Docker exec command to start a bash shell in container 73, a 'cd volumes/' command, and an 'ls' command showing files 'C_code', 'Python_code', 'udp_server', and 'udp_server.c'.

```
seed@VM: ~/.../ch04_lab  
[05/16/24]seed@VM:~/.../ch04_lab$ docker exec -it 73 /bin/bash  
root@7388f8431f5f:/# cd volumes/  
root@7388f8431f5f:/volumes# ls  
C_code  Python_code  udp_server  udp_server.c  
root@7388f8431f5f:/volumes#
```

■ hostB

■ attacker

4.2 패킷 수신 방법

- NIC(네트워크 인터페이스 카드)는 기기와 네트워크 사이의 물리적 또는 논리적 연결
- 각 NIC는 자신의 MAC 주소 소유
- 네트워크상의 모든 NIC는 네트워크 상의 모든 프레임을 수신
- NIC는 각 패킷의 목적지 주소를 확인하며, 그 주소가 카드의 MAC 주소와 일치하면 패킷을 커널의 버퍼로 복사
- 지정된 NIC를 대상으로 하지 않는 프레임은 드랍.
- 무차별 모드로 동작할 때, NIC는 네트워크로부터 수신한 모든 프레임을 커널로 전달
- 커널에 스니퍼 프로그램이 등록되어 있으면, 모든 패킷을 수신.
- Wi-Fi에서는 모니터 모드.

BSD Packet Filter (BPF)

- BPF를 사용하여 사용자 프로그램이 소켓에 필터를 부착하여 커널에 원치 않는 패킷을 버리도록 지시.(빠르게 캡처 또는 폐기)
- 컴파일된 BPF 코드의 예 → 일종의 mask처럼 동작

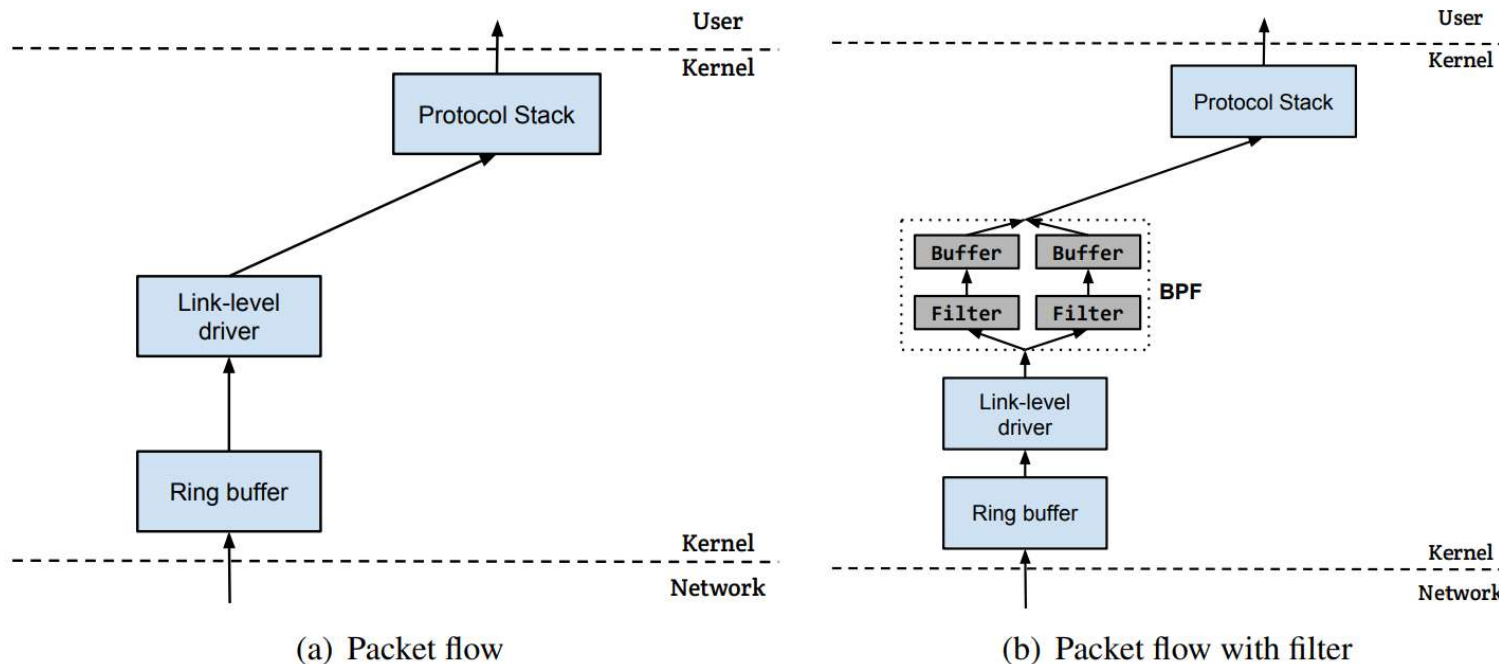
```
struct sock_filter code[] = {
    { 0x28, 0, 0, 0x0000000c }, { 0x15, 0, 8, 0x000086dd },
    { 0x30, 0, 0, 0x00000014 }, { 0x15, 2, 0, 0x00000084 },
    { 0x15, 1, 0, 0x00000006 }, { 0x15, 0, 17, 0x00000011 },
    { 0x28, 0, 0, 0x00000036 }, { 0x15, 14, 0, 0x00000016 },
    { 0x28, 0, 0, 0x00000038 }, { 0x15, 12, 13, 0x00000016 },
    { 0x15, 0, 12, 0x00000800 }, { 0x30, 0, 0, 0x00000017 },
    { 0x15, 2, 0, 0x00000084 }, { 0x15, 1, 0, 0x00000006 },
    { 0x15, 0, 8, 0x00000011 }, { 0x28, 0, 0, 0x00000014 },
    { 0x45, 6, 0, 0x00001fff }, { 0xb1, 0, 0, 0x0000000e },
    { 0x48, 0, 0, 0x0000000e }, { 0x15, 2, 0, 0x00000016 },
    { 0x48, 0, 0, 0x00000010 }, { 0x15, 0, 1, 0x00000016 },
    { 0x06, 0, 0, 0x0000ffff }, { 0x06, 0, 0, 0x00000000 },
};

struct sock_fprog bpf = {
    .len = ARRAY_SIZE(code),
    .filter = code,
};
```

BSD Packet Filter(BPF)

```
setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf))
```

- A compiled BPF pseudo-code can be attached to a socket through `setsockopt()`.
- When a packet is received by kernel, BPF will be invoked
- An accepted packet is pushed up the protocol stack.



4.3 패킷 스니핑(Packet Sniffing)

■ 일반적인 패킷 수신 방법 (udp_server.c)

⇒ 응용프로그램, 사용자의 패킷들을 수신



```
Open  udp_server.c  Save
~/Downloads/ch04_lab/volumes

1#include <unistd.h>
2#include <stdio.h>
3#include <string.h>
4#include <sys/socket.h>
5#include <netinet/ip.h>
6
7void main()
8{
9    struct sockaddr_in server;
10   struct sockaddr_in client;
11
12   int clientlen = sizeof(client);
13   char buf[1500];
14
15   // step 1
16   /*
17   이 줄은 인터넷 도메인에서 사용할 수 있는 UDP 소켓을 생성.
18   AF_INET은 IPv4 인터넷 프로토콜을 사용한다는 것을 의미,
19   SOCK_DGRAM은 데이터그램 기반의 비연결형 소켓을 생성한다는 것을 의미.
20   IPPROTO_UDP는 UDP 프로토콜을 사용함을 지정.
21   */
22
23   int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
24
25
```

→ 실습
코드(udp_server.c)
참조 설명

실습

■ 1. UDP 서버 프로그램 확인

➡ 수업 자료 홈페이지 → udp_server.c 다운로드

➡ 코드 확인 → 코드 컴파일 (컨테이너 컴퓨터가 아닌, 호스트 컴퓨터에서)

■ 2. hostA에서 udp_server 실행

```
seed@VM: ~/.../ch04_lab
[05/16/24] seed@VM:~/.../ch04_lab$ docker exec -it 73 /bin/bash
root@7388f8431f5f:/# ip -br addr
lo                UNKNOWN          127.0.0.1/8
eth0@if231        UP                10.9.0.5/24
root@7388f8431f5f:/# cd volumes/
root@7388f8431f5f:/volumes# udp_server
Hello
```

■ 3. hostB에서 nc -u 10.9.0.5 9090 실행

```
seed@VM: ~/.../ch04_lab
[05/16/24] seed@VM:~/.../ch04_lab$ docker exec -it 9c /bin/bash
root@9cd4ffa7807c:/# ip -br addr
lo                UNKNOWN          127.0.0.1/8
eth0@if233        UP                10.9.0.6/24
root@9cd4ffa7807c:/# cd volumes/
root@9cd4ffa7807c:/volumes# ls
C_code  Python_code  udp_server  udp_server.c
root@9cd4ffa7807c:/volumes# nc -u 10.9.0.5 9090
Hello
```


원시 소켓을 이용한 패킷 수신

```
sniff_raw.c
~/Downloads/ch04_lab/volumes

Open Save

1 // POSIX 운영 체제 API를 위한 기본적인 시스템 서비스와 API 함수를 제공
2 #include <unistd.h>
3
4 // 입출력 관련 함수를 포함
5 #include <stdio.h>
6
7 // 소켓 프로그래밍을 위한 기본 헤더 파일
8 #include <sys/socket.h>
9 #include <arpa/inet.h>
10
11 // 로우 소켓을 사용하기 위한 구조체 및 상수 정의를 제공
12 #include <linux/if_packet.h>
13 #include <net/ethernet.h>
14
15
16 int main()
17 {
18     int PACKET_LEN = 32;
19     char buffer[PACKET_LEN];
20
21     // 네트워크 주소를 저장하는 데 사용되는 일반적인 데이터 구조
22     struct sockaddr saddr;
23
24     // 리눅스의 패킷 소켓에서 멀티캐스트 그룹 멤버십이나
25     // 프로미스큐어스 모드(promiscuous mode)와 같은 기능을 설정할 때 사용
26     struct packet_mreq mr;
27 }
```

→ 실습 코드(sniff_raw.c)
참조 설명

실습

■ 1. 원시 소켓 수신 코드 확인

➡ 수업 자료 홈페이지 → sniff_raw.c 다운로드

➡ 코드 확인

➡ 코드 컴파일 (컨테이너 컴퓨터가 아닌, 호스트 컴퓨터에서)

◆ `gcc -o sniff_raw sniff_raw.c`

■ 2. hostA에서 sniff_raw 실행

■ 3. hostB에서 `nc -u 10.9.0.5 9090` 실행

원시 소켓 패킷 수신의 문제점

- 다른 운영 체제로 이식 불가
- 필터 설정이 쉽지 않음.
- 성능 향상을 위한 최적화 안됨
- PCAP 라이브러리의 필요성
 - ➡ 내부적으로는 여전히 원시 소켓을 사용하지만, PCAP API는 모든 플랫폼에서 표준.
 - ➡ PCAP의 구현에 의해 운영체제별 세부 사항은 숨겨져 있음.
 - ➡ 프로그래머가 사람이 읽을 수 있는 부울 표현식을 사용해 필터링 규칙을 지정.