TP1 :

Attaques passives: sniffing passif

Les sniffers (appelé aussi « analyseurs de protocoles » ou « analyseurs de réseau ») sont des outils logiciels qui peuvent capturer les trames circulant sur un réseau local et afficher leurs contenus (entêtes des protocoles, identités des utilisateurs, mot de passe non cryptés, etc). Ces outils sont utilisés par les administrateurs pour analyser leurs réseaux et localiser les problèmes dans ces derniers. Ils sont aussi utilisés par les attaquants pour espionner les données circulant dans un réseau local.

Le tableau 1 présente une comparaison de quelques sniffers (GUI : graphical User Interface, CLI : Command Line Interface)

sniffer	User interface	Software license	Microsoft Windows	OS X	Linux	BSDs	Solaris
Cain and Abel	GUI	Freeware	Yes	No	No	No	No
CommView	GUI	Proprietary	Yes	No	No	No	No
dSniff	CLI	BSD License	?	Yes	Yes	Yes	Yes
EtherApe	GUI	GNU General Public License	No	Yes	Yes	Yes	Yes
Ettercap	Both	GNU General Public License	Yes	Yes	Yes	Yes	Yes
netsniff-ng	CLI	BSD-style	No	No	Yes	No	No
tcpdump	CLI	BSD License	Yes WinDump	Yes	Yes	Yes	Yes
Wireshark (formerly Ethereal)	Both	GNU General Public License	Yes	Yes	Yes	Yes	Yes

Tableau 1 : comparaison de quelques sniffers

1- **Objectifs de ce TP:**

- Implémenter un sniffer passif simple
- Manipuler des logiciels de sniffing

2- **Outils logiciels:**

Linux, wireshark, compilateur cc ou gcc

3- **Informations utiles :**

- Les cartes réseau fonctionnent en deux modes
  - mode normal (mode par défaut) : permet à une carte de filtrer les trames reçus en fonction de l'adresse MAC destination
  - mode promiscuous : consiste à accepter toutes les trames circulant dans un réseau, même ceux qui ne sont pas destinés à la carte.
- Sous Unix, la commande # ifconfig promisc permet d'activer le mode promiscuous.
- La plupart des logiciels sniffers permettent d'activer le mode promoscuous lors de leur lancement.
- Dans un réseau commuté, le sniffing **passif** de toutes les trames qui circulent dans le réseau est impossible à réaliser puisqu'un nœud ne peut recevoir que les trames qui lui sont destinées.
- Le sniffing **actif** (qui sera traité au niveau du TP2) permet de faire du sniffing sur un réseau même s'il est commuté.
- Le sniffer doit être sur le même réseau à sniffer. Sinon, il doit faire du « **remote sniffing** » en contrôlant à distance une machine qui se trouve sur le réseau à sniffer.

4- **Partie 1 : Implémentation d'un sniffer passif**

L'annexe 1 présente le code source d'un sniffer passif qui permet de récupérer les trames reçu par une interface réseau (exemple ETHERNET). Ce code source est écrit en langage C et peut être compilé et exécuté sur une machine Linux. Les fonctions les plus importantes de ce code sont (voir contenu de la fonction main):

- La fonction recvfrom qui permet de récupérer les trames reçues sur l'interface réseau.
- La fonction PrintPacketInHex qui permet d'afficher la trame sous format hexadécimal
- La fonction ParseEthernetHeader qui permet d'afficher quelques champs de l'entête ETHRNET
- La fonction ParseIpHeader qui permet d'afficher quelques champs de l'entête IP
- La fonction ParseTcpHeader qui permet d'afficher quelques champs de l'entête TCP
- La fonction ParseData qui permet d'afficher les données sous format hexadécimal

Manipulations à faire (sous linux):

1) Compiler (cc -c sniffer_eth_ip_tcp_data.c) le code source et générer l'exécutable (cc sniffer_eth_ip_tcp_data.c – o sniffer).
2) Exécuter (en mode root : « **sudo commande** » sous ubuntu) le sniffer (exemple : pour sniffer les 100 premières trames reçu sur l'interface eth0 tapez **./sniffer eth0 100**). Vous pouvez utiliser **wlan0** à la place de **eth0** si vous êtes connecté en sans fil (les trames Wifi sont automatiquement traduites en ETHERNET) ou **lo** (loopback) si vous n'avez aucune connexion. Si rien ne s'affiche, cela veut dire que vous n'êtes pas en train de recevoir des paquets, exécuter alors (dans une nouvelle fenêtre) un ping vers une autre machine et consulter de nouveau le résultat du sniffer.
3) Dans la manipulation précédente, les trames sont affichées sous format hexadécimal. Pour afficher le contenu de l'entête ETHERNET, il faut enlever le commentaire de la fonction ParseEthernetHeader, recompiler, régénérer l'exécutable et refaire l'étape 2).
4) Pour Afficher le contenu des entêtes des protocoles des niveaux supérieurs, enlevez les commentaires des fonctions correspondantes (au niveau de la fonction main), recompiler, régénérer l'exécutable et exécuter de nouveau le sniffer. Pensez à faire un échange de trafic TCP (en utilisant par exemple le serveur vsftpd ou en se connectant à Internet).
5) Ecrire une fonction qui permet d'afficher l'entête UDP et l'intégrer dans le code source du sniffer (localiser udp.h dans /usr/include/netinet).

5- **Partie2 : manipulation de sniffers**

Dans cette partie, nous nous intéressons à la manipulation de quelques sniffers existants.

1) Lancer le logiciel wireshark en arrière plan (wireshark &) et commencez la capture sur l'interface ETHERNET ou sans fil.
2) Lancez des applications d'échange de trafic entre d'autres machines et la votre. Observez les paquets capturés.
3) Est-ce que vous pouvez capturer les trafics échangés entre les machines du reste du réseau?
4) Configurer le filtre de wireshark pour (voir annexe 2).
    a. n'afficher que les trames concernant un protocole particulier : bootp, tcp, icmp,etc
    b. n'afficher que les trames dont l'adresse MAC destination est celle de votre machine
    c. n'afficher que les trames echangé entre deux machines d'adresse @IP1 et @IP2
    d. n'afficher que les trames dont la taille est supérieure à une taille donnée

6- **Partie3 : remote sniffing**

Dans cette section, nous nous intéressons l'utilisation d'un sniffer à distance « remote sniffing » pour obtenir les données circulant sur un autre réseau que celui sur lequel nous sommes. Supposons que nous sommes sur le réseau RES1 et nous voulons sniffer le réseau voisin RES2 (nous sommes séparés par un routeur). Nous utilisons alors un **client sniffer** sur une machine du réseau RES2 qui va sniffer ce dernier et envoyer les données capturées à notre **serveur sniffer** sur le réseau RES1. Le réseau B qui, en principe, était impossible à sniffer est devenu donc très accessible.

Nous utilisons le démon Rpcapd qui capture le trafic sur une machine, et est capable d'envoyer les données récupérées à un sniffer comme wireshark qui facilite ainsi la lecture en différenciant les trames et les protocoles. Notons qu'il est utile d'exclure le trafic entre la machine locale et la machine distante en utilisant les filtres de wireshark.

Manipulations à faire (sous linux):

1) Sur votre machine, lancer wireshark et accéder aux options de capture puis taper dans interface « **rpcap://@ipmachine distante/eth0 »** et taper dans capture filter « **not host votre adresse IP »**
2) Etudier les paquets capturés.

==========

Annexe 1 :

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<features.h>
#include<linux/if_packet.h>
#include<linux/if_ether.h>
#include<errno.h>
#include<sys/ioctl.h>
#include<net/if.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <string.h>
//#include<linux/ip.h>
//#include<linux/tcp.h>
#include<netinet/in.h>

// protocol_to_sniff est le protocole niveau liaison. ça sera le protocole
// ETHERNET  car nous allons faire du sniffing sur un réseau ETHERNET
```

```
int CreateRawSocket(int protocol_to_sniff)
{
      int rawsock;
      if((rawsock=socket(PF_PACKET,SOCK_RAW,htons(protocol_to_sniff)))==-1)
      {
           perror("Error creating raw socket: ");
           exit(-1);
      }
      return rawsock;
}

int BindRawSocketToInterface(char *device, int rawsock, int protocol)
{
      struct sockaddr_ll sll;
      struct ifreq ifr;
      bzero(&sll, sizeof(sll));
      bzero(&ifr, sizeof(ifr));
      /* First Get the Interface Index  */
      strncpy((char *)ifr.ifr_name, device, IFNAMSIZ);
      if((ioctl(rawsock, SIOCGIFINDEX, &ifr)) == -1)
      {
          printf("Error getting Interface index !\n");
          exit(-1);
      }
      /* Bind our raw socket to this interface */
      sll.sll_family = AF_PACKET;
      sll.sll_ifindex = ifr.ifr_ifindex;
      sll.sll_protocol = htons(protocol);
      if((bind(rawsock, (struct sockaddr *)&sll,sizeof(sll)))== -1)
      {
          perror("Error binding raw socket to interface\n");
          exit(-1);
      }
      return 1;
}

void PrintPacketInHex(unsigned char *packet, int len)
{
      unsigned char *p = packet;
      printf("\n\n---------Packet---Starts----\n\n");
      while(len--)
      {
            printf("%.2x ", *p);
            p++;
      }
      printf("\n\n--------Packet---Ends-----\n\n");
}

PrintInHex(char *mesg, unsigned char *p, int len)
{
      printf("%s",mesg);
      while(len--)
      {
            printf("%.2X ", *p);
            //printf("%c",*p);
            p++;
      }
}
ParseEthernetHeader(unsigned char *packet, int len)
{
```

```
	struct ethhdr *ethernet_header;
	if(len > sizeof(struct ethhdr))
	{
		ethernet_header = (struct ethhdr *)packet;
		/* First set of 6 bytes are Destination MAC */
		PrintInHex("Destination MAC: ",ethernet_header->h_dest, 6);
		printf("\n");
		/* Second set of 6 bytes are Source MAC */
		PrintInHex("Source MAC: ",ethernet_header->h_source, 6);
		printf("\n");
		/* Last 2 bytes in the Ethernet header are the protocol */
		PrintInHex("Protocol: ",(void*)&ethernet_header->h_proto, 2);
		printf("\n");
	}
	else
	{
		printf("Packet size too small !\n");
	}
}

ParseIpHeader(unsigned char *packet, int len)
{
	struct ethhdr *ethernet_header;
	struct iphdr *ip_header;
	/* First Check if the packet contains an IP header using Ethernet */
	ethernet_header = (struct ethhdr *)packet;
	if(ntohs(ethernet_header->h_proto) == ETH_P_IP)
	{
		/* The IP header is after the Ethernet header  */
		if(len >= (sizeof(struct ethhdr) + sizeof(struct iphdr)))
		{
		ip_header = (struct iphdr*)(packet + sizeof(struct ethhdr));
		/* print the Source and Destination IP address */
		printf("TTL: %d \n",ip_header->ttl);
		printf("Dest IP address: %s\n", inet_ntoa( *(struct
			in_addr*)&ip_header->daddr));
		printf("Source IP address: %s\n", inet_ntoa( *(struct
			in_addr*)&ip_header->saddr));
		}
		else
		{
		      printf("IP packet does not have full header\n");
		}
		}
		else
		{
		      /* Not an IP packet */
		}
}

ParseTcpHeader(unsigned char *packet , int len)
{
	struct ethhdr *ethernet_header;
	struct iphdr *ip_header;
	struct tcphdr *tcp_header;
	/* Check if enough bytes are there for TCP Header */
	if(len >= (sizeof(struct ethhdr) + sizeof(struct iphdr) +
		  sizeof(struct tcphdr)))
	{
		/* Do all the checks: 1. Is it an IP pkt ? 2. is it TCP ? */
```

```
            ethernet_header = (struct ethhdr *)packet;

            if(ntohs(ethernet_header->h_proto) == ETH_P_IP)
            {
                  ip_header=(struct iphdr *)(packet+sizeof(struct ethhdr));
                  if(ip_header->protocol == IPPROTO_TCP)
                  {
                    printf("TCP num :%d)\n",ip_header->protocol);
                    tcp_header = (struct tcphdr*)(packet +
                    sizeof(struct ethhdr) + ip_header->ihl*4 );
                    /* Print the Dest and Src ports */
                    printf("Source Port: %d\n", ntohs(tcp_header->source));
                    printf("Dest Port: %d\n", ntohs(tcp_header->dest));
                  }
                  else
                  {
                    printf("Not a TCP packet\n");
                  }
            }
            else
            {
                  printf("Not an IP packet\n");
            }
      }
      else
      {
            printf("TCP Header not present \n");
      }
}

int ParseData(unsigned char *packet, int len)
{
      struct ethhdr *ethernet_header;
      struct iphdr *ip_header;
      struct tcphdr *tcp_header;
      unsigned char *data;
      int data_len;
      /* Check if any data is there */
      if(len > (sizeof(struct ethhdr) + sizeof(struct iphdr) +
                  sizeof(struct tcphdr)))
      {
            ip_header = (struct iphdr*)(packet + sizeof(struct ethhdr));
            data = (packet + sizeof(struct ethhdr) + ip_header->ihl*4
                  +sizeof(struct tcphdr));
            data_len = ntohs(ip_header->tot_len) - ip_header->ihl*4 -
                  sizeof(struct tcphdr);
            if(data_len)
            {
                  printf("Data Len : %d\n", data_len);
                  printf("------------------\n");
                  printf("%s\n",(char*)data);
                  printf("********************\n");
                  PrintInHex("Data : ", data, data_len);
                  printf("\n\n");
                  return 1;
            }
            else
            {
                  printf("No Data in packet\n");
                  return 0;
```

```
            }
      }
      else
      {
            printf("No Data in packet\n");
            return 0;
      }
}

int IsIpAndTcpPacket(unsigned char *packet, int len)
{
      struct ethhdr *ethernet_header;
      struct iphdr *ip_header;
      ethernet_header = (struct ethhdr *)packet;
      if(ntohs(ethernet_header->h_proto) == ETH_P_IP)
      {
            ip_header = (struct iphdr *)(packet + sizeof(struct ethhdr));
            if(ip_header->protocol == IPPROTO_TCP)
                  return 1;
            else
                  return -1;
      }
      else
      {
            return -1;
      }
}

main(int argc, char **argv)
{
      int raw;
      unsigned char packet_buffer[2048];
      int len;
      int packets_to_sniff;
      struct sockaddr_ll packet_info;
      int packet_info_size = sizeof(packet_info);
      if (argc < 3 )
      {
            printf("-------------------------------------------------\n");
            printf("Usage: sniffer <Interface> <Nbr of packets to sniff>\n");
            printf("-------------------------------------------------\n");
            return (EINVAL);
      }
      /* create the raw socket */
      raw = CreateRawSocket(ETH_P_IP);
      /* Bind socket to interface */
      BindRawSocketToInterface(argv[1], raw, ETH_P_IP);
      /* Get number of packets to sniff from user */
      packets_to_sniff = atoi(argv[2]);
      /* Start Sniffing and print Hex of every packet */
      while(packets_to_sniff--)
      {
            if((len = recvfrom(raw, packet_buffer, 2048, 0, (struct
sockaddr*)&packet_info, (socklen_t*)&packet_info_size)) == -1)
            {
                  perror("Recv from returned -1: ");
                  exit(-1);
            }
            else
            {
                  /* Packet has been received successfully !! */
```

```
                    PrintPacketInHex(packet_buffer, len);
                    /*Parse Ethernet Header */
                    //ParseEthernetHeader(packet_buffer, len);
                    /* Parse IP Header */
                    //ParseIpHeader(packet_buffer, len);
                    /* Parse TCP Header */
                    //ParseTcpHeader(packet_buffer, len);
                    /* Parse TCP Header */
                    //ParseUdpHeader(packet_buffer, len);
                    /*if(IsIpAndTcpPacket(packet_buffer, len))
                      {
                          if(!ParseData(packet_buffer, len))
                                packets_to_sniff++;
                      }*/
                }
        }
        return 0;
}
```

Annexe 2 : (site officiel de wireshark)

Display Filter comparison operators

English	C-like	Description and example
eq	==	Equal. ip.src==10.0.0.5
ne	!=	Not equal. ip.src!=10.0.0.5
gt	>	Greater than. frame.len > 10
lt	<	Less than. frame.len < 128
ge	>=	Greater than or equal to. frame.len ge 0x100
le	<=	Less than or equal to. frame.len <= 0x20

Display Filter Field Types

Type	Example
Unsigned integer (8-bit, 16-bit, 24-bit, 32-bit)	You can express integers in decimal, octal, or hexadecimal. The following display filters are equivalent: ip.len le 1500 ip.len le 02734 ip.len le 0x436
Signed integer (8-bit, 16-bit, 24-bit, 32-bit)	
Boolean	A boolean field is present in the protocol decode only if its value is true. For example, *tcp.flags.syn* is present, and thus true, only if the SYN flag is present in a TCP segment header. Thus the filter expression *tcp.flags.syn* will select only those packets for which this flag exists, that is, TCP segments where the segment header contains the SYN flag. Similarly,

	to find source-routed token ring packets, use a filter expression of *tr.sr*.
Ethernet address (6 bytes)	Separators can be a colon (:), dot (.) or dash (-) and can have one or two bytes between separators: eth.dst == ff:ff:ff:ff:ff:ff eth.dst == ff-ff-ff-ff-ff-ff eth.dst == ffff.ffff.ffff
IPv4 address	ip.addr == 192.168.0.1 Classless InterDomain Routing (CIDR) notation can be used to test if an IPv4 address is in a certain subnet. For example, this display filter will find all packets in the 129.111 Class-B network: ip.addr == 129.111.0.0/16
IPv6 address	ipv6.addr == ::1
String (text)	http.request.uri == "https://www.wireshark.org/"

Display Filter Logical Operations

English	C-like	Description and example
and	&&	Logical AND. `ip.src==10.0.0.5 and tcp.flags.fin`
or	||	Logical OR. `ip.scr==10.0.0.5 or ip.src==192.1.1.1`
xor	^^	Logical XOR. `tr.dst[0:3] == 0.6.29 xor tr.src[0:3] == 0.6.29`
not	!	Logical NOT. `not llc`
		Substring Operator. Wireshark allows you to select subsequences of a sequence in rather elaborate ways. After a label you can place a pair of brackets [] containing a comma separated list of range specifiers. eth.src[0:3] == 00:00:83 The example above uses the n:m format to specify a single range. In this case n is the beginning offset and m is the length of the range being specified. eth.src[1-2] == 00:83 The example above uses the n-m format to specify a single range. In this case n is the beginning offset and m is the ending offset. eth.src[:4] == 00:00:83:00 The example above uses the :m format, which takes everything from the beginning of a sequence to offset m. It is equivalent to 0:m eth.src[4:] == 20:20 The example above uses the n: format, which takes everything from offset n to the end of the sequence.

		eth.src[2] == 83 The example above uses the n format to specify a single range. In this case the element in the sequence at offset n is selected. This is equivalent to n:1. eth.src[0:3,1-2,:4,4:,2] == 00:00:83:00:83:00:00:83:00:20:20:83 Wireshark allows you to string together single ranges in a comma separated list to form compound ranges as shown above.