Instytut Teleinformatyki

Wydział Inżynierii Elektrycznej i Komputerowej
Politechnika Krakowska

programowanie usług sieciowych

„Libpcap”

laboratorium: 11
system operacyjny: Linux

Kraków, 2009

Spis treści

1. Wiadomości wstępne

Pierwsza część niniejszej instrukcji zawiera podstawowe wiadomości teoretyczne dotyczące biblioteki *libpcap*. Poznanie tych wiadomości umożliwi prawidłowe zrealizowanie praktycznej części laboratorium.

1.1. Tematyka laboratorium

Tematyką laboratorium jest programowanie aplikacji wykorzystujących bibliotekę *libpcap*. Biblioteka *libpcap* została opracowana w 1994 roku przez naukowców z Uniwersytetu Kalifornia w Berkley. Głównym celem autorów biblioteki było stworzenie niezależnego od platformy API, które umożliwiałoby przechwytywanie pakietów sieciowych. API *libpcap* eliminuje konieczność korzystania z niskopoziomowych mechanizmów dostarczanych przez systemy operacyjne. Dzięki temu, aplikacje korzystające z *libpcap* są przenośne.
API biblioteki umożliwia:

- przechwytywanie pakietów na poziomie warstwy łącza danych,
- zapisywanie przechwyconych pakietów do pliku,
- transmisję „surowych” pakietów (użytkownik jest odpowiedzialny za dostarczenie nagłówków: ramki Ethernet, protokołu IP, protokołu TCP, itp.),
- pobieranie informacji statystycznych na temat ruchu sieciowego,
- pobieranie informacji na temat dostępnych interfejsów sieciowych.

Biblioteka *libpcap* została zaimplementowana w języku C, ale wiele innych języków programowania – w tym Perl, Python, C# i Java – udostępnia własne interfejsy (tzw. *wrappery*), które umożliwiają korzystanie z jej funkcjonalności. Wersja biblioteki dla systemów UNIX jest rozwijana w ramach projektu *Tcpdump*. Niezależnie od niej rozwijana jest wersja dla systemu Windows - *WinPcap*.
Następujące aplikacje korzystają z usług biblioteki *libpcap*:

- sniffery (np. *tcpdump*, *tshark*, *Wireshark*),
- systemy NIDS (np. *Snort*),
- skanery sieciowe (np. *Nmap*),
- generatory ruchu sieciowego,
- narzędzia do przechwytywania haseł (np. *L0phtCrack*, *dsniff*),
- narzędzia do przeprowadzania ataków sieciowych (np. *ettercap*),
- demony *port knocking* i SPA (ang. *Single Packet Authorization*, np. *Fwknop*),

- narzędzia identyfikujące systemy operacyjne i usługi, tzw. *fingerprinting*

1.2. Zagadnienia do przygotowania

Przed przystąpieniem do realizacji laboratorium należy zapoznać się z następującymi zagadnieniami: [1 – 7]

- budowa ramki Ethernet,
- budowa nagłówków protokołów IP, ICMP, ARP,
- zasada działania protokołu ARP,
- porządek bajtów (*little-endian*, *big-endian;* funkcje: `ntohs()`, `ntohl()`, `htons()`, `htonl()`),
- konwersja adresów IP (funkcje `inet_pton()`, `inet_ntop()`, `getnameinfo()`),
- konwersja adresów MAC (funkcje `ether_aton()` i `ether_ntoa()`),
- API biblioteki *libpcap*.

Ponadto, wymagana jest: [1, 8, 9]

- znajomość podstawowych opcji programów *iptables*, *tcpdump* oraz wyrażeń filtrujących *tcpdump*,
- umiejętność korzystania ze struktur adresowych:
  - `sockaddr_in` (`<netinet/in.h>`),
  - `sockaddr_in6` (`<netinet/in.h>`),
  - `sockaddr_ll` (`<netpacket/packet.h>`),
  - `sockaddr` (`<sys/socket.h>`).
- znajomość nagłówków protokołów zdefiniowanych w plikach:
  - `<net/ethernet.h>` - nagłówek ramki Ethernet,
  - `<netinet/if_ether.h>`, `<net/if_arp.h>` - nagłówek protokołu ARP,
  - `<netinet/ip.h>` - nagłówek protokołu IP,
  - `<netinet/ip_icmp.h>` - nagłówek protokołu ICMP.

Literatura:

[1] W.R. Stevens, *„Programowanie Usług Sieciowych”*, *„API: gniazda i XTI”*
[2] IETF (http://www.ietf.org/), RFC 826, *„An Ethernet Address Resolution Protocol”*
[3] IETF (http://www.ietf.org/), RFC 791, *„INTERNET PROTOCOL”*
[4] IETF (http://www.ietf.org/), RFC 792, *„INTERNET CONTROL MESSAGE PROTOCOL”*
[5] MAN (3) *„byteorder”*, *„inet_ntop”*, *„inet_pton”*, *„getnameinfo”*
[6] MAN (3) *„pcap”*, http://www.tcpdump.org/pcap3_man.html
[7] Luis Martin Garcia, *„Programming with Libpcap – Sniffing the Network From Our Own Application”*, Hakin9 2/2008 (15)
[8] MAN (8) *„tcpdump”*
[9] MAN (8) *„iptables”*

1.3. Opis laboratorium

1.3.1. Inicjalizacja biblioteki

Przed przystąpieniem do przechwytywania i analizy pakietów należy w odpowiedni sposób zainicjalizować bibliotekę *libpcap*. Biblioteka może zostać skonfigurowana do przechwytywanie pakietów z interfejsu sieciowego lub w trybie umożliwiającym odczyt pakietów z pliku. W pierwszym przypadku należy wywołać funkcję **`pcap_open_live()`**:

```
#include <pcap.h>

pcap_t *pcap_open_live(const char *device, int snaplen, int promisc,
                       int to_ms, char *errbuf);
```

Funkcja `pcap_open_live()` zwraca deskryptor wykorzystywany przez pozostałe funkcje biblioteki lub `NULL` w przypadki błędu. Jeżeli wywołanie funkcji zakończy się niepowodzeniem, to bufor wskazywany przez `errbuf` powinien zawierać opis błędu w formie stringu (łańcucha znaków zakończonego przez `'\0'`). Za alokację pamięci dla bufora błędów odpowiada użytkownik. Przed wywołaniem funkcji należy upewnić się, że rozmiar bufora wynosi co najmniej `PCAP_ERRBUF_SIZE` bajtów – jest to reguła dla wszystkich funkcji *libpcap* przyjmujących jawnie wskaźnik na bufor błędów. W przypadku prawidłowego wywołania funkcji `pcap_open_live()`, bufor błędów może zawierać ostrzeżenie. Aby wykryć sytuację, w której funkcja zwraca ostrzeżenie, należy przed jej wywołaniem umieścić w buforze błędów string o długości zero, a po wywołaniu funkcji sprawdzić, czy długość stringu uległa zmianie.
Parametr `device` definiuje nazwę interfejsu sieciowego, który będzie używany do przechwytywania pakietów. W systemach Linux z jądrem 2.2 i nowszych, można użyć nazwy ”`any`” lub wartości `NULL` w celu przechwytywania na wszystkich interfejsach.
Parametr `snaplen` określa maksymalną liczbę bajtów do przechwycenia – tylko pierwsze `snaplen` bajtów pakietu zostanie przechwycone i przekazane bibliotece *libpcap*. Dla większości sieci, wartość 65535 powinna być wystarczająca.
Kolejny parametr, `promisc`, określa czy interfejs powinien działać w trybie *promiscuous*. Jeżeli tryb *promiscuous* jest wyłączony, to przechwytywane będą tylko pakiety kierowane na adres stacji przechwytującej lub na adres rozgłoszeniowy. Włączenie trybu *promiscuous* umożliwia przechwytywanie wszystkich pakietów, także kierowanych do innych stacji sieciowych. Jeżeli parametr `promisc` ma wartość zero, nie oznacza to, że tryb *promiscuous* jest wyłączony (interfejs może zostać skonfigurowany w trybie *promiscuous* niezależnie od biblioteki *libpcap*). Parametr `promisc` jest ignorowany, jeżeli interfejs, na którym odbywa się przechwytywanie pakietów to interfejs o nazwie ”`any`”.

Pakiet przechwycony na interfejsie sieciowym, nie musi być przekazany natychmiast bibliotece *libpcap*. Pakiety mogą być buforowane w jądrze systemu i po upłynięciu zdefiniowanego okresu (od chwili otrzymania pierwszego pakietu) kopiowane do bufora biblioteki *libpcap* w przestrzeni użytkownika. Kopiowanie kilku pakietów między domeną jądra, a domeną użytkownika za pomocą jednej operacji jest bardziej wydajne od kopiowania pojedynczych pakietów. Okres czasu przez jaki pakiety mogą być buforowane w jądrze systemu definiuje się w ms za pomocą parametru `tp_ms`. Podanie wartości 0 spowoduje, że pakiety będą buforowane w jądrze systemu, aż do chwili zapełnienia bufora. Dopiero wówczas zostaną skopiowane do bufora biblioteki *libpcap*.
Nie wszystkie systemy operacyjne wykorzystują parametr `tp_ms` (parametr ten jest ignorowany przez system Linux).

Nazwa interfejsu sieciowego w wywołaniu funkcji `pcap_open_live()` może zostać podana przez użytkownika (jako string) lub określona za pomocą jednej z dwóch funkcji *libpcap*: `pcap_lookupdev()`, `pcap_findalldevs()`.
Funkcja **`pcap_lookupdev()`** zwraca nazwę pierwszego dostępnego interfejsu, który może zostać wykorzystany do przechwytywania pakietów:

```
#include <pcap.h>

char *pcap_lookupdev(char *errbuf);
```

W przypadku wystąpienia błędu zwracana jest wartość `NULL`, a opis błędu (string) można odczytać z bufora `errbuf`. Jeżeli użytkownik aplikacji nie zdefiniuje konkretnego interfejsu, funkcja `pcap_lookupdev()` może zostać wykorzystana do określenia domyślnego interfejsu aplikacji – w sposób niezależny od platformy.
Druga z wymienionych funkcji, **`pcap_findalldevs()`**, zwraca listę interfejsów, które umożliwiają przechwytywanie pakietów lub -1 w przypadku błędu:

```
#include <pcap.h>

int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf);
```

Parametr `alldevsp` stanowi adres wskaźnika na pierwszy element listy interfejsów. Lista jest alokowana dynamicznie i należy ją zwolnić za pomocą funkcji **`pcap_freealldevs()`**:

```
#include <pcap.h>

void pcap_freealldevs(pcap_if_t *alldevs);
```


Jeżeli proces nie posiada odpowiednich uprawnień (`CAP_NET_RAW` dla systemu Linux), to lista interfejsów zwracana przez `pcap_findalldevs()` może być pusta, pomimo że wywołanie funkcji zakończy się powodzeniem.
Elementy listy interfejsów są reprezentowane przez strukturę `pcap_if`:

```
#include <pcap.h>

struct pcap_if {
        /* Wskaźnik na następny element listy: */
        struct pcap_if *next;

        /* Nazwa interfejsu - dla "pcap_open_live()": */
        char *name;

        /* Tekstowy opis interfejsu lub NULL */
        char *description;

        /* Lista adresów interfejsu: */
        struct pcap_addr *addresses;

        /* Flagi interfejsu: */
        bpf_u_int32 flags;
};

/*
 * Jedyna zdefiniowana flaga określa, czy interfejs jest interfejsem
 * loopback:
 */
#define PCAP_IF_LOOPBACK        0x00000001
```

Pole `addresses` struktury `pcap_if` stanowi wskaźnik na pierwszy element listy adresów interfejsu. Definicja struktury adresowej interfejsu przedstawiona jest poniżej:

```
#include <pcap.h>

struct pcap_addr {
        /* Wskaźnik na następny element listy adresów: */
        struct pcap_addr *next;

        /* Adres interfejsu: */
        struct sockaddr *addr;

        /* Maska sieciowa: */
        struct sockaddr *netmask;
```



```
        /* Adres rozgłoszeniowy: */
        struct sockaddr *broadaddr;

        /* Adres docelowy dla P2P: */
        struct sockaddr *dstaddr;
};
```

Pola reprezentujące adresy oraz maskę interfejsu mają postać wskaźnika na strukturę `sockaddr`. Struktura `sockaddr` jest ogólną strukturę adresową złożoną z dwóch pól: rodziny adresowej oraz dopełnienia. Poniższy listing prezentuje definicję struktury `sockaddr`:

```
#include <sys/socket.h>

typedef unsigned short int sa_family_t;

struct sockaddr {
        sa_family_t  sa_family;
        char         sa_data[14];
};
```

Rodzina adresowa to pole typu `unsigned short int` o nazwie `sa_family`. Rodzina adresowa określa jakie informacje adresowe przenosi struktura i jaki jest jej właściwy rozmiar. Przykładowe rodziny adresowe to `AF_INET` dla protokołu IP oraz `AF_INET6` dla IPv6. Proszę zauważyć, że ogólna struktura adresowa (`sockaddr`) daje dostęp tylko do rodziny adresowej i dopełnienia. Znając rodzinę adresową, można jednak rzutować wskaźnik do ogólnej struktury `sockaddr`, na wskaźnik do struktury konkretnego protokołu, np. `sockaddr_in` dla IPv4, czy `sockaddr_in6` dla IPv6. Umożliwia to dostęp do właściwych informacji adresowych. Struktury adresowe dla protokołów IPv4 oraz IPv6 są zdefiniowane w pliku nagłówkowym `<netinet/in.h>`.
Specyficzną dla systemu Linux strukturą adresową jest `sockaddr_ll`. Struktura ta przenosi informacje adresowe dla rodziny `AF_PACKET`, tzn. adresy warstwy łącza danych (MAC dla Ethernetu). Struktura `sockaddr_ll` jest używana przez gniazda rodziny `PF_PACKET`, które służą do odbierania i wysyłania „surowych” pakietów w systemie Linux. Termin „surowe” pakiety odnosi się w tym przypadku do możliwości odbierania i wysyłania pełnych ramek Ethernet. Gniazda rodziny `PF_PACKET` są podstawowym mechanizmem wykorzystywanym przez bibliotekę *libpcap* do przechwytywania pakietów w systemie Linux. Więcej informacji na temat gniazd `PF_PACKET` można znaleźć na stronie podręcznika systemowego *„packet (7)”*. Definicja struktury `sockaddr_ll` ma następującą postać:


```
#include <netpacket/packet.h>
struct sockaddr_ll {
    unsigned short int sll_family;
    unsigned short int sll_protocol;
    int sll_ifindex;
    unsigned short int sll_hatype;
    unsigned char sll_pkttype;
    unsigned char sll_halen;
    unsigned char sll_addr[8];
};
```

Poniższa tabela wyjaśnia znaczenie poszczególnych pól struktury `sockaddr_ll`:

Pole	Opis
`sll_family`	rodzina adresowa – zawsze `AF_PACKET`
`sll_protocol`	identyfikator protokołu przenoszony w ramce Ethernet; pole w porządku sieciowym; identyfikatory protokołów są zdefiniowane w pliku nagłówkowym `<linux/if_ether.h>`; przykładowe wartości to `ETH_P_IP` dla IPv4 oraz `ETH_P_ARP` dla protokołu ARP
`sll_ifindex`	identyfikator interfejsu, na którym odebrany został pakiet (którym ma być wysłany pakiet – dla gniazd `PF_PACKET`)
`sll_hatype`	identyfikator protokołu sprzętowego protokołu ARP; wartości są zdefiniowane w pliku `<linux/if_arp.h>` lub `<net/if_arp.h>`; dla Ethernetu jest to `ARPHRD_ETHER` (wartość 1)
`sll_pkttype`	typ pakietu: • `PACKET_HOST` – adresowany do lokalnej stacji, • `PACKET_BROADCAST` – rozgłoszeniowy, • `PACKET_MULTICAST` – transmisji grupowej, • `PACKET_OTHERHOST` – pakiet dla innej stacji sieciowej przechwycony w trybie *promiscuous*, • `PACKET_OUTGOING` – wysyłany przez lokalną stację
`sll_halen`	długość adresu warstwy łącza danych
`sll_addr`	adres warstwy łącza danych

W przypadku struktury zwracanej przez funkcje `pcap_findalldevs()` znaczenie mają tylko pola: `sll_family`, `sll_hatype`, `sll_halen` oraz `sll_addr`.

Przed przystąpieniem do przechwytywania pakietów, przydatne może okazać się określenie typu warstwy łącza danych interfejsu. W celu uzyskania tej informacji należy wywołać funkcję **`pcap_datalink()`**:

```
#include <pcap.h>
```



```
int pcap_datalink(pcap_t *p);
```

Funkcja `pcap_datalink()` przyjmuje deskryptor utworzony za pomocą `pcap_open_live()` i zwraca typ warstwy łącza danych w formie liczby całkowitej. Typy są zdefiniowane w pliku nagłówkowym `pcap-bpf.h` załączanym przez `pcap.h`. Dla Ethernetu, funkcja zwraca wartość `DTL_EN10MB`. Dysponowanie wiedzą na temat warstwy łącza danych interfejsu ułatwia analizę i przetwarzanie odbieranych pakietów – programista może poczynić założenia co do postaci nagłówka 2 warstwy modelu ISO/OSI. W systemie Linux pseudo-interfejs o nazwie *„any”* wykorzystuje własny nagłówek warstwy łącza danych (ang. *Linux cooked capture encapsulation*). Dla tego interfejsu funkcja `pcap_datalink()` zwraca wartość `DLT_LINUX_SLL`.

Oprócz możliwości przechwytywania pakietów za pomocą interfejsów sieciowych, biblioteka *libpcap* umożliwia odczyt pakietów z pliku. Format pliku jest określony prze stronę podręcznika systemowego *„tcpdump (8)”* – większość popularnych snifferów zapisuje dane w tym właśnie formacie. Aby skonfigurować bibliotekę do odczytu danych z pliku, należy wywołać funkcję **`pcap_open_offline()`**:

```
#include <pcap.h>

pcap_t *pcap_open_offline(const char *fname, char *errbuf);
```

Parametr `fname` określa nazwę pliku do otworzenia, a `errbuf` wskazuje na bufor wykorzystywany do zwrócenia opisu błędu.

1.3.2. Filtrowanie

Kolejnym etapem po utworzeniu deskryptora `pcap_t` za pomocą funkcji `pcap_open_live()`, jest opcjonalne zdefiniowanie programu filtrującego. Program filtrujący umożliwia określenie jakie pakiety będą przechwytywane przez bibliotekę *libpcap*. Tylko pakiety pasujące do reguły filtrującej będą kopiowane z bufora w jądrze systemu do bufora biblioteki, co w znaczący sposób może wpłynąć na wydajność systemu. Kopiowanie całego ruchu sieciowego (bez programu filtrującego) jest zadaniem pochłaniającym czas procesora i może doprowadzić do porzucania pakietów.
Jeżeli jądro systemu operacyjnego nie obsługuje filtrowania, to filtrowanie jest przeprowadzane przez bibliotekę *libpcap* w przestrzeni użytkownika.
Konfiguracja programu filtrującego wiąże się z:

- przygotowaniem reguły,
- kompilacją reguły do postaci programu filtrującego,
- przypisaniem programu filtrującego do deskryptora.

Programy filtrujące – określane jako BPF (ang. *BSD Packet Filter*) – są pisane w specjalnym języku, podobnym do asemblera. Dla wygody programistów i użytkowników,

biblioteka *libpcap* implementuje język wysokiego poziomu, który pozwala definiować programy filtrujące w dużo prostszy sposób – za pomocą reguł. Pełną specyfikację reguł filtrujących można odnaleźć na stronie podręcznika systemowego *„tcpdump (8)”*. Poniższa tabela przedstawia kilka przykładów reguł filtrujących.

Reguła	Opis
`src host 192.168.2.1`	pakiety o źródłowym adresie IP 192.168.2.1
`dst port 80`	pakiety TCP/UDP, w których docelowy port jest równy 80
`icmp[icmptype] == icmp-echoreplay`	pakiety, które są odpowiedzią na komunikaty *ICMP Echo*
`ip[8] == 64`	pakiety, dla których wartość pola TTL w nagłówku IP wynosi 64
`ether dst 00:01:de:f3:4a:a7`	pakiety o docelowym adresie MAC równym 00:01:de:f3:4a:a7

Dysponując regułą w postaci stringu, można przystąpić do jej kompilacji. Proces kompilacji transformuje regułę do postaci programu filtrującego. Kompilację reguły przeprowadza się za pomocą funkcji **`pcap_compile()`**.

```
#include <pcap.h>

int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str,
                 int optimize, bpf_u_int32 netmask);
```

Parametr	Opis
`p`	deskryptor utworzony za pomocą `pcap_open_live()`
`fp`	wskaźnik do struktury `bpf_program` zaalokowanej przez programistę; po wywołaniu funkcji, struktura będzie zawierać program filtrujący odpowiadający regule `str`
`str`	reguła filtrująca w formie stringu, np.: `”dst port 80”`
`optimize`	flaga określająca, czy kod wynikowy powinien być optymalizowany pod kątem zwiększenia wydajności
`netmask`	maska sieciowa wykorzystywana przez program filtrujący do testowania, czy dany adres jest adresem rozgłoszeniowym; jeżeli test sprawdzający czy adres jest rozgłoszeniowy wchodzi w skład reguły, a argument `netmask` przyjmuje wartość 0, to test ten nie będzie przeprowadzany; parametr `netmask` nie wpływa na inne testy definiowane przez regułę filtrującą

W przypadku wystąpienia błędu, funkcja `pcap_compile()` zwraca wartość -1. Opis błędu można uzyskać za pomocą funkcji `pcap_geterr()` lub `pcap_perror()`:


```
#include <pcap.h>

char* pcap_geterr(pcap_t *p);
void pcap_perror(pcap_t *p, char* prefix);
```

Funkcja **pcap_geterr()** zwraca tekstowy opis związany z ostatnim błędem biblioteki *libpcap*. Zadaniem funkcji **pcap_perror()** jest wypisanie informacji na temat ostatniego błędu *libpcap* na standardowe wyjście błędów. Opis błędu jest poprzedzony prefiksem `prefix` oraz znakami „: ” (dwukropek i spacja).
Jeżeli w wywołaniu `pcap_compile()` konieczne jest określenie maski sieciowej, można posłużyć się funkcją **pcap_lookupnet()**:

```
#include <pcap.h>

int pcap_lookupnet(const char *device, bpf_u_int32 *netp,
                   bpf_u_int32 *maskp, char *errbuf);
```

Parametr	Opis
`device`	nazwa interfejsu sieciowego
`netp`	wskaźnik do zmiennej typu `bpf_u_int32` zaalokowanej przez programistę; po wywołaniu funkcji, zmienna będzie zawierać adres sieci, do której należy interfejs
`maskp`	wskaźnik do zmiennej typu `bpf_u_int32` zaalokowanej przez programistę; po wywołaniu funkcji, zmienna będzie zawierać maskę sieciową interfejsu
`errbuf`	wskaźnik na bufor zaalokowany przez programistę; w przypadku błędu funkcja zwraca -1 i umieszcza opis błędu w buforze `errbuf`

Ostatnim etapem konfiguracji programu filtrującego jest przypisanie programu do deskryptora `pcap_t`. Za to zadanie, odpowiedzialna jest funkcja **pcap_setfilter()**.

```
#include <pcap.h>

int pcap_setfilter(pcap_t *p, struct bpf_program *fp);
```

Funkcja `pcap_setfilter()` przyjmuje deskryptor `p` oraz program filtrujący `fp`. W przypadku błędu funkcja zwraca wartość -1, a opis błędu można uzyskać za pomocą `pcap_geterr()` lub `pcap_perror()`.

Po wywołaniu funkcji `pcap_setfilter()`, program filtrujący jest kopiowany do przestrzeni jądra systemu operacyjnego. Jeżeli proces użytkownika nie będzie korzystał więcej ze swojej kopii, można wywołać funkcję **`pcap_freecode()`**:

```
#include <pcap.h>

void pcap_freecode(struct bpf_program *fp);
```

Funkcja `pcap_freecode()` jest odpowiedzialna za zwolnienie obszaru pamięci zaalokowanego dynamicznie na rzecz struktury `bpf_program` (jedno z pól struktury `bpf_program` jest wskaźnikiem na obszar pamięci alokowany dynamicznie przez `pcap_compile()`).

Oprócz określenia programu filtrującego, biblioteka *libpcap* umożliwia zdefiniowanie kierunku przechwytywania pakietów. Za pomocą funkcji **`pcap_setdirection()`** można wybrać jedną z trzech możliwości:

- `PCAP_D_IN` – przechwytywanie pakietów przychodzących,
- `PCAP_D_OUT` - przechwytywanie pakietów wysyłanych,
- `PCAP_D_INOUT` przechwytywanie pakietów przychodzących i wysyłanych.

Poniżej przedstawiona jest deklaracja funkcji `pcap_setdirection()`:

```
#include <pcap.h>

int pcap_setdirection(pcap_t *p, pcap_direction_t d);
```

W przypadku błędu, funkcja zwraca wartość -1.
Nie wszystkie systemy operacyjne umożliwiają określenie kierunku przechwytywania pakietów (Linux umożliwia). Domyślnym ustawieniem jest `PCAP_D_INOUT`.

1.3.3. Przechwytywanie pakietów

Po wybraniu interfejsu sieciowego i opcjonalnym skonfigurowaniu filtrowania można przystąpić do przechwytywania pakietów. Biblioteka *libcap* udostępnia cztery funkcje, które mogą zostać wykorzystane do tego celu:

- `pcap_next() i pcap_next_ex()`,
- `pcap_dispatch()`,
- `pcap_loop()`.

Funkcja **`pcap_next_ex()`** odczytuje kolejny pakiet z bufora w jądrze systemu operacyjnego.


```
#include <pcap.h>

int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header,
                 const u_char **pkt_data);
```

Funkcja przyjmuje deskryptor `p`, adres wskaźnika na strukturę `pcap_pkthdr` oraz adres wskaźnika na dane pakietu. Proszę zwrócić uwagę na fakt, że użytkownik dostarcza tylko adresy wskaźników, nie alokuje pamięci. Struktura `pcap_pkthdr` przechowuje metadane na temat odebranego pakietu:

```
#include <pcap.h>

struct pcap_pkthdr {
        /* Czas przechwycenia pakietu: */
        struct timeval ts;

        /* Rozmiar przechwyconych danych (pakiet lub jego część): */
        bpf_u_int32 caplen;

        /* Rozmiar pakietu odebranego przez kartę sieciową: */
        bpf_u_int32 len;
};
```

Przechwycone dane będą miały dokładnie `caplen` bajtów - może to być cały pakiet lub tylko jego fragment. Wpływ na liczbę przechwyconych bajtów ma parametr `snaplen` funkcji `pcap_open_live()`, omówionej w podrozdziale 1.3.1.
Funkcja `pcap_next_ex()` może zwrócić jedną z następujących wartości:

Wartość	Opis
1	pakiet został poprawnie odebrany
0	wartość 0 może zostać zwrócona w czterech przypadkach: • upłynął okres czasu zdefiniowany przez parametr `to_ms` funkcji `pcap_open_live()` i w zdefiniowanym okresie czasu nie odebrano pakietu; parametr `to_ms` jest ignorowany przez niektóre systemy operacyjne, w tym system Linux; część systemów uruchamia timer w chwili otrzymania pakietu, a część w chwili wywołania funkcji `pcap_next_ex()` • jądro systemu nie obsługuje filtrowania pakietów; biblioteka *libpcap* odrzuciła pakiet odebrany z jądra na podstawie filtru w przestrzeni użytkownika, • pakiet został odrzucony na podstawie zdefiniowanego kierunku przechwytywania pakietów (np. przechwytywane są tylko pa-

	kiety wychodzące, a odebrano pakiet przychodzący), • deskryptor używany do przechwytywania pakietów jest w trybie nieblokującym i podczas operacji odczytu nie było pakietów do odebrania; tryb deskryptora dla operacji I/O można zdefiniować za pomocą funkcji `pcap_setnonblock()`
`-1`	wystąpił błąd podczas odczytu pakietu
`-2`	w przypadku odczytu pakietów z pliku – brak pakietów do odczytania

Funkcja `pcap_next()` jest starszą wersją `pcap_next_ex()` i nie pozwala na szczegółowe rozróżnienie błędów.

Funkcja **`pcap_dispatch()`** jest używana do przetwarza kolekcji pakietów:

```
#include <pcap.h>

int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback,
                  u_char *user);
```

Parametr `cnt` określa maksymalną liczbę pakietów do przetworzenia. Funkcja zazwyczaj kopiuje pakiety, które są aktualnie dostępne w buforze, nie czekając na nadejście nowych pakietów. Z tego powodu, przetworzonych może zostać mniej niż `cnt` pakietów. Jeżeli parametr `cnt` ma wartość -1, funkcja przetwarza wszystkie pakiety, które są aktualnie dostępne w buforze jądra systemu.
W systemie Linux parametr `cnt` jest ignorowany, co powoduje, że przetwarzany jest tylko jeden pakiet – tak jak w przypadku `pcap_next()`. Funkcja `pcap_next()` jest zresztą zaimplementowana za pomocą `pcap_dispatch()` – wywołuje `pcap_dispatch()` z argumentem `cnt` równym 1.
Dla każdego pakietu odebranego przez `pcap_dispatch()` wywoływana jest funkcja `callback`, do której przekazywane są dane określone za pomocą argumentu `user`. Wskaźnik na funkcję `callback` jest zdefiniowany jako typ `pcap_handler`:

```
#include <pcap.h>

typedef void (*pcap_handler)(u_char *, const struct pcap_pkthdr *,
                             const u_char *);
```

Funkcja `callback` przyjmuje dane określone w wywołaniu `pcap_dispatch()`, wskaźnik na strukturę z metadanymi na temat pakietu (`pcap_pkthdr`) oraz wskaźnik do danych odebranego pakietu (`const u_char*`). Za zdefiniowanie ciała funkcji `callback` odpowiedzialny jest programista.

Funkcja `pcap_dispatch()` zwraca liczbę odebranych pakietów lub jedną z następujących wartości:

Wartość	Opis
`0`	wartość 0 może zostać zwrócona w pięciu przypadkach: • upłynął okres czasu zdefiniowany przez parametr `to_ms` funkcji `pcap_open_live()` i w zdefiniowanym okresie czasu nie odebrano pakietu; parametr `to_ms` jest ignorowany przez niektóre systemy operacyjne, w tym system Linux; część systemów uruchamia timer w chwili otrzymania pakietu, a część w chwili wywołania funkcji `pcap_dispatch()`, • jądro systemu nie obsługuje filtrowania pakietów; biblioteka *libpcap* odrzuciła pakiet odebrany z jądra na podstawie filtru w przestrzeni użytkownika, • pakiet został odrzucony na podstawie zdefiniowanego kierunku przechwytywania pakietów (np. przechwytywane są tylko pakiety wychodzące, a odebrano pakiet przychodzący), • deskryptor używany do przechwytywania pakietów jest w trybie nieblokującym i podczas operacji odczytu nie było pakietów do odebrania; tryb deskryptora dla operacji I/O można zdefiniować za pomocą funkcji `pcap_setnonblock()`, • w przypadku odczytu pakietów z pliku – brak pakietów do odczytania
`-1`	wystąpił błąd podczas odczytu pakietu
`-2`	odczytywanie pakietów zostało przerwane za pomocą funkcji `pcap_breakloop()`, zanim jakikolwiek pakiet został przetworzony

Ostatnią z funkcji odpowiedzialnych za przechwytywanie pakietów jest **`pcap_loop()`**:

```
#include <pcap.h>

int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

Funkcja `pcap_loop()` jest podobna do `pcap_dispatch()` z jednym wyjątkiem: odczytuje pakiety, dopóki nie zostanie przetworzone `cnt` pakietów lub wystąpi błąd. Pętla może również zostać przerwana explicite - za pomocą funkcji `pcap_breakloop()`. Zdefiniowanie parametru `cnt` jako ujemnej liczby całkowitej spowoduje, że funkcja będzie odczytywać pakiety w nieskończonej pętli.
Funkcja `pcap_loop()` zwraca jedną z następujących wartości:

Wartość	Opis
0	wartość 0 może zostać zwrócona w dwóch przypadkach: • funkcja odczytała cnt pakietów, • w przypadku odczytu pakietów z pliku – brak pakietów do odczytania
-1	wystąpił błąd podczas odczytu pakietu
-2	pętla została przerwana za pomocą funkcji pcap_breakloop(), zanim jakikolwiek pakiet został odczytany/przetworzony

Poniżej przedstawiony jest przykład zliczania liczby odebranych pakietów za pomocą funkcji pcap_loop().

```
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>

void callback(u_char *arg, const struct pcap_pkthdr *pkt_header,
              const u_char *packet) {
      unsigned int *countp = (unsigned int*)arg;
      ++(*countp);
      fprintf(stdout, "Packet count: %u\n", *countp);
}

int main(int argc, char **argv) {

char              errbuf[PCAP_ERRBUF_SIZE];

/* Deskryptor wykorzystywany do przechwytywania pakietów: */
pcap_t            *descriptor;

/* Typ warstwy łącza danych: */
int               data_link_type;

/* Liczba pakietów: */
unsigned int      count = 0;

if (argc != 2) {
      fprintf(stderr, "Invocation: %s <interface>\n", argv[0]);
      exit(EXIT_FAILURE);
}

/* Utworzenie deskryptora do przechwytywania pakietów: */
descriptor = pcap_open_live(argv[1], 65565, 0, 10, errbuf);
if (descriptor == NULL) {
```



```
      fprintf(stderr, "pcap_open_live(): %s\n", errbuf);
      exit(EXIT_FAILURE);
}

/* Sprawdzenie typu warstwy łącza danych: */
data_link_type = pcap_datalink(descriptor);
if (data_link_type != DLT_EN10MB) {
      /* Jeżeli inny od ETHERNET: */
      exit(EXIT_FAILURE);
}

/*
 * Dla każdego pakietu wywoływana jest funkcja “callback” z adresem
 * zmiennej “count”:
 */
if (pcap_loop(descriptor, -1, callback, (u_char*)&count) == -1) {
      pcap_perror(descriptor, "pcap_loop()");
      exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}
```

Bardziej elegancki sposób pobierania informacji na temat liczby przechwyconych pakietów udostępnia funkcja **`pcap_stats()`**:

```
#include <pcap.h>

int pcap_stats(pcap_t *p, struct pcap_stat *ps);
```

W przypadku powodzenia, funkcja `pcap_stats()` zwraca zero i wypełnia strukturę wskazywaną przez parametr `ps`. Za alokację pamięci dla struktury `pcap_stat` odpowiedzialny jest programista. W przypadku wystąpienia błędu, funkcja `pcap_stats()` zwraca wartość -1. Opis błędu można uzyskać za pomocą funkcji `pcap_geterr()` lub `pcap_perror()`. Poniżej przedstawiona jest definicja struktury `pcap_stat`.

```
#include <pcap.h>

struct pcap_stat {
      u_int ps_recv;          /* Liczba odebranych pakietów */
      u_int ps_drop;          /* Liczba porzuconych pakietów */
      u_int ps_ifdrop;        /* Pole aktualnie nieobsługiwane */
```



```
#ifdef WIN32
      /* Liczba pakietów odebranych przez aplikację: */
      u_int bs_capt;
#endif /* WIN32 */
};
```

1.3.4. Wysyłanie pakietów

Wysyłanie pakietów może zostać zrealizowane za pomocą funkcji `pcap_inject()` lub `pcap_sendpacket()`.

```
#include <pcap.h>

int pcap_inject(pcap_t     *p, const void   *buf, size_t size);
int pcap_sendpacket(pcap_t *p, const u_char *buf, int    size);
```

Parametr	Opis
`p`	deskryptor
`buf`	wskaźnik na bufor zawierający pakiet do wysłania; pakiet musi zawierać nagłówek warstwy łącza danych
`size`	liczba bajtów do wysłania

Funkcja `pcap_inject()` zwraca liczbę poprawnie zapisanych bajtów lub -1 w przypadku wystąpienia błędu. Funkcja `pcap_sendpacket()` zwraca -1 w przypadku błędu, 0 w przypadku powodzenia i jest kompatybilna z `WinPcap`.

1.4. Cel laboratorium

Celem laboratorium jest zapoznanie się z API biblioteki *libpcap*. Laboratorium obejmuje aspekty związane z:

- pobieraniem informacji na temat interfejsów sieciowych,
- przechwytywaniem i przetwarzaniem pakietów sieciowych,
- wysyłaniem pakietów za pomocą *libpcap*,
- zasadą działania protokołu ARP i atakami *ARP Spoofing*.

2. Przebieg laboratorium

Druga część instrukcji zawiera zadania do praktycznej realizacji, które demonstrują zastosowanie technik z omawianego zagadnienia.

2.1. Zadanie 1. Pobieranie informacji konfiguracyjnych interfejsów sieciowych

Zadanie polega na analizie kodu przykładowego programu. Program wykorzystuje funkcję `pcap_findalldevs()` w celu wypisania listy interfejsów, które mogą zostać wykorzystane do przechwytywania pakietów. Dla każdego interfejsu wypisywana jest lista adresów IP oraz adres MAC. Proszę zwrócić szczególną uwagę na struktury wykorzystywane do przechowywania informacji adresowych.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc devices.c –lpcap –o devices`
3. Uruchomić program. Do prawidłowego działania, program wymaga uprawnienia `CAP_NET_RAW` (użytkownik *root* posiada wszystkie uprawnienia):
   `$ ./devices`
4. Czy da się zastąpić wywołanie funkcji `getnameinfo()` za pomocą `inet_ntop()`?

2.2. Zadanie 2. Przechwytywanie komunikatów ICMP

Celem zadania jest analiza kodu przykładowego programu. Program przechwytuje pakiety na wybranym interfejsie sieciowym, analizuje pakiety i wypisuje informacje tylko na temat komunikatów ICMP Echo. Do odbierania pakietów wykorzystywana jest funkcja `pcap_next_ex()`.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc icmp_sniff.c –lpcap –o icmp_sniff`
3. Uruchomić program podając jako argument nazwę interfejsu sieciowego, na którym będą przechwytywane pakiety. Do prawidłowego działania, program

wymaga uprawnienia `CAP_NET_RAW` (użytkownik *root* posiada wszystkie uprawnienia):
```
$ ./icmp_sniff <nazwa interfejsu>
```
4. Za pomocą programu `ping` wysłać komunikat ICMP Echo na adres interfejsu, na którym przechwytywane są pakiety.
```
$ ping <adres IP>
```
5. Zakończyć działanie programu za pomocą kombinacji CTRL + C. Wciśnięcie CTRL + C spowoduje wypisanie informacji statystycznych na temat liczby przechwyconych i porzuconych pakietów. Proszę porównać liczbę przechwyconych pakietów z liczbą komunikatów ICMP Echo, na temat których program wyświetlił informacje.
6. Zdefiniować regułę *iptables* dla łańcucha INPUT porzucającą wszystkie odebrane pakiety (wymagane uprawnienia *roota*):
```
$ iptables –I INPUT 1 –j DROP
```
7. Powtórzyć kroki 3-5. W jaki sposób reguła *iptables* wpływa na zachowanie programów `icmp_sniff` i `ping`?
8. Proszę usunąć zdefiniowaną wcześniej regułę *iptables*:
```
$ iptables –D INPUT 1
```

2.3. Zadanie 3. Filtrowanie przechwytywanych pakietów

Proszę zmodyfikować program z zadania 2 w taki sposób, aby przechwytywane były tylko komunikaty ICMP Echo (program z zadania 2 wypisywał informacje na temat komunikatów ICMP Echo, ale przechwytywał wszystkie pakiety). W tym celu konieczne jest zdefiniowanie odpowiedniego wyrażenia filtrującego (strony podręcznika systemowego *tcpdump*), kompilacja wyrażenia i przypisanie utworzonego programu filtrującego do deskryptora *pcap*.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Skompilować program źródłowy do postaci binarnej:
```
$ gcc icmp_filter.c –lpcap –o icmp_filter
```
2. Uruchomić program podając jako argument nazwę interfejsu sieciowego, na którym będą przechwytywane pakiety.
```
$ ./icmp_filter <nazwa interfejsu>
```
3. Za pomocą programu `ping` wysłać komunikat ICMP Echo na adres interfejsu, na którym przechwytywane są pakiety.
```
$ ping <adres IP>
```
4. Zakończyć działanie programu za pomocą kombinacji CTRL + C. Proszę porównać liczbę przechwyconych pakietów z liczbą komunikatów ICMP Echo, na temat których program wyświetlił informacje.

2.4. Zadanie 4. Przechwytywanie pakietów ARP

Celem zadania jest analiza kodu programu, który przechwytuje i wypisuje informacje na temat pakietów ARP. Za przetwarzanie pakietów odpowiedzialna jest funkcja *callback*, wywoływana dla każdego odebranego pakietu ARP.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc arp_sniff.c –lpcap –o arp_sniff`
3. Uruchomić program podając jako argument nazwę interfejsu sieciowego, na którym będą przechwytywane pakiety. Do prawidłowego działania, program wymaga uprawnienia `CAP_NET_RAW` (użytkownik *root* posiada wszystkie uprawnienia):
   `$ ./arp_sniff <nazwa interfejsu>`
4. Za pomocą programu `ping` wysłać komunikat ICMP Echo na adres dowolnej stacji w sieci lokalnej.
   `$ ping <adres IP>`
   Jeżeli nie spowoduje to wysłania pakietu ARP, można wyczyścić wpis w tablicy ARP odpowiadający danej stacji sieciowej (`arp –d`) lub wykorzystać program `arping`:
   `$ arping –I <nazwa interfejsu> <adres IP>`

2.5. Zadanie 5. ARP Spoofing

Proszę zmodyfikować program z zadania 4 w taki sposób, aby umożliwiał on przeprowadzanie ataku ARP Spoofing. Dla każdego odebranego pakietu ARP REQUEST, program powinien wysyłać pakiet ARP REPLAY, w którym:

- źródłowy adres MAC jest adresem interfejsu, na którym przechwytywane są pakiety,
- źródłowy adres IP jest docelowym adresem IP z pakietu ARP REQUEST,
- docelowy adres MAC jest źródłowym adresem MAC z pakietu ARP REQUEST,
- docelowy adres IP jest źródłowym adresem IP z pakietu ARP REQUEST.

Źródłowy adres MAC w pakiecie ARP REPLAY można zdefiniować w kodzie programu jako string i skonwertować do postaci binarnej za pomocą funkcji `ether_aton()`.
Proszę pamiętać, że w celu wysłania pakietu za pomocą funkcji `pcap_sendpacket()` konieczne jest zdefiniowanie nagłówka ramki Ethernet. Odpowiedź ARP, w przeciwieństwie do zapytania, musi zostać wysłana na adres MAC typu *unicast*.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Skompilować program źródłowy do postaci binarnej:
   `$ gcc arp_spoof.c –lpcap –o arp_spoof`

2. Uruchomić program podając jako argument nazwę interfejsu sieciowego, na którym będą przechwytywane i wysyłane pakiety ARP.
   `$ ./arp_spoof <nazwa interfejsu>`
3. Za pomocą sniffera *tcpdump* lub *tshark* zaobserwować pakiety ARP na wybranym interfejsie:
   `$ tcpdump –n –e –i <nazwa interfejsu> arp`
   lub
   `$ tshark –n –i <nazwa interfejsu> arp`
4. Aby wymusić wysłanie pakietu ARP można wyczyścić tablicę ARP (`arp -d`) i wysłać komunikat ICMP Echo na adres IP stacji w sieci lokalnej.

3. Opracowanie i sprawozdanie

Realizacja laboratorium pt. *„Libpcap”* polega na wykonaniu wszystkich zadań programistycznych podanych w drugiej części tej instrukcji. Wynikiem wykonania powinno być sprawozdanie w formie wydruku papierowego dostarczonego na kolejne zajęcia licząc od daty laboratorium, kiedy zadania zostały zadane.

Sprawozdanie powinno zawierać:

- opis metodyki realizacji zadań (system operacyjny, język programowania, biblioteki, itp.),
- algorytmy wykorzystane w zadaniach (zwłaszcza, jeśli zastosowane zostały rozwiązania nietypowe),
- opisy napisanych programów wraz z opcjami,
- trudniejsze kawałki kodu, opisane tak, jak w niniejszej instrukcji,
- uwagi oceniające ćwiczenie: trudne/łatwe, nie/realizowalne, nie/wymagające wcześniejszej znajomości zagadnień (wymienić jakich),
- wskazówki dotyczące ewentualnej poprawy instrukcji celem lepszego zrozumienia sensu oraz treści zadań.

