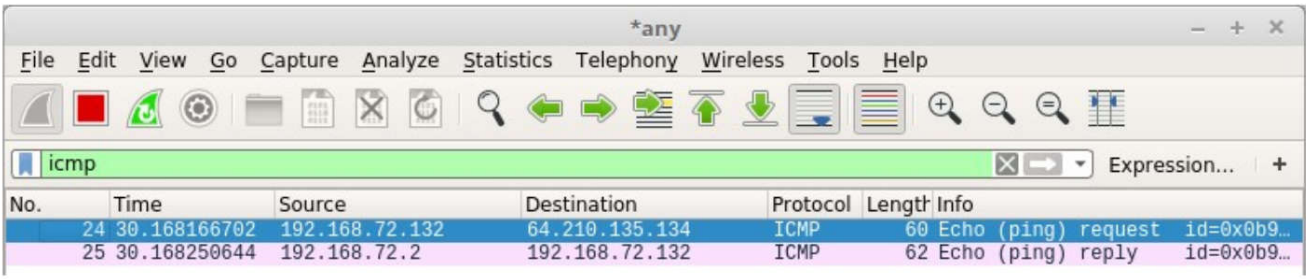
Programowanie usług sieciowych											
03	Temat:	***Opcje IP i gniazda surowe***	Zadania:								Data:
	Autor:	**Witold Karaś**	1	2	3	4	5				**2017-03-22**
	Autor:	**Łukasz Maj**	e	e	m	m	h				**14:30-16:00**

Zadanie 1. Wyznaczanie trasy przez nadawcę

Zadanie polega na analizie działania podanego kodu. Kod demonstruje możliwość wysyłania datagramu z opcją IP o nazwie SSRR za pomocą surowego gniazda.

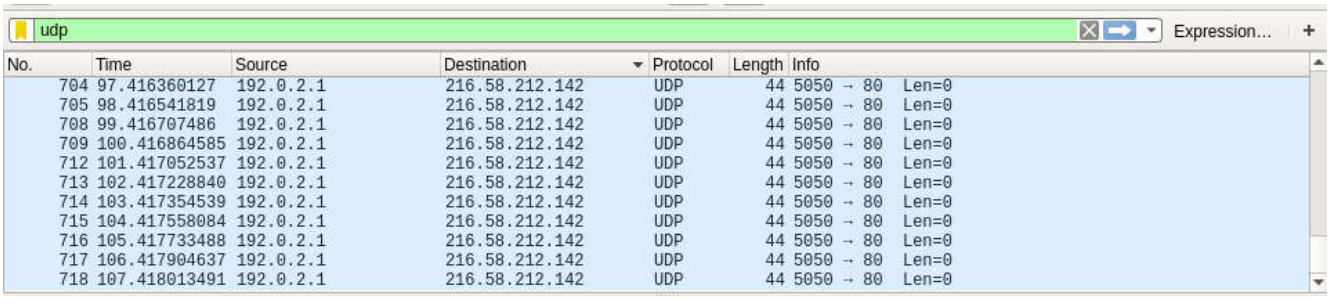
SSRR (ang. strict source and record route) - rygorystyczne trasowanie według nadawcy, służy do przesyłania pakietów wzdłuż zdefiniowanej trasy. Opcja ta wskazuje drogę pakietu do stacji docelowej. W przypadku wybrania tej opcji, datagram IP musi przejść przez wszystkie zdefiniowane węzły i co ważne TYLKO przez nie. Zatem kolejne węzły muszą być swoimi sąsiadami.

Program pomimo prawidłowego uruchomienia się nie dostawał wiadomości zwrotnej. Należało więc zmienić adresy w *ip_option* na adres bramy domyślnej. Mimo to, wireshark wychwytywał tylko pakiet *request*, natomiast pakiet *reply* nie docierał. Problem został zauważony w dystrybucji *Ubuntu*. Po sprawdzeniu działania programu w dystrybucji *Mint* problemu nie zauważono.

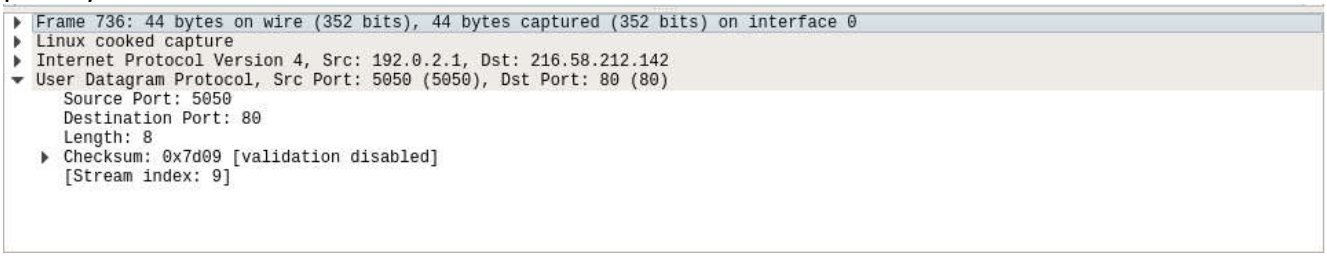


Zadanie 2. Gniazda surowe – protokół UDP

Zadanie polega na analizie działania podanego kodu. Program wykorzystuje gniazdo surowe dla protokołu UDP, co 1 sekundę wysyłając puste datagramy (nagłówek UDP bez danych) na adres i numer portu podanych jako argumenty wywołania programu. Zamiast adresu IP, argumentem wywołania może być nazwa domenowa. Numer portu i adres źródłowy zostały zdefiniowane jako stałe wartości.



Na załączonym screenie wykonanym podczas pracy programu *wireshark* widać przesyłane pakiety UDP.



Po rozwinięciu okna szczegółów dotyczącego datagramów UDP, możemy dostrzec, że te przesyłane są puste, bowiem składają się z minimalnej możliwej liczby bajtów tj. 8.

Zadanie 3. Gniazda surowe – protokół TCP

Celem zadania było zaimplementowanie programu wykorzystującego gniazdo surowe dla protokołu TCP.

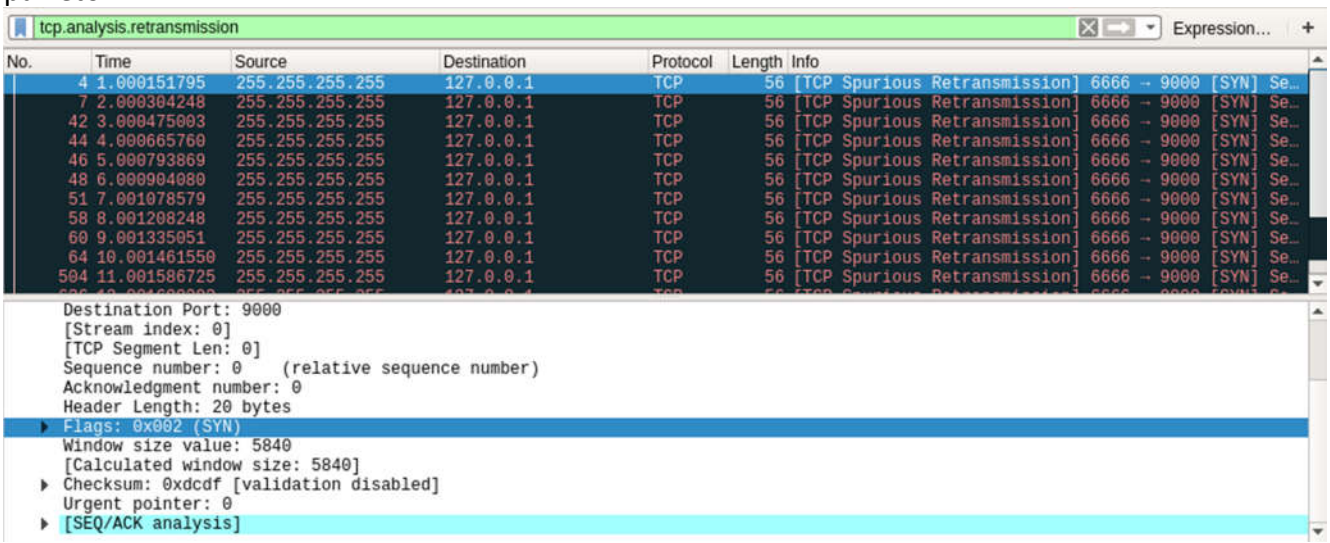
Nagłówek TCP został tak wypełniony, aby program SYN floodował zadany w argumencie adres IP na wybranym przez użytkownika porcie.

Wartość pola *checksum* nagłówka TCP została wypełniona sumą kontrolną obliczoną na podstawie pseudo-nagłówka (określonych pól nagłówka IP oraz całego segmentu TCP).

Wypełnienie pseudo-nagłówka:

```
void pseudo_create()
{
        pseudo_header.ip_src.s_addr = ip_header->ip_src.s_addr;
        pseudo_header.ip_dst.s_addr = ip_header->ip_dst.s_addr;
        pseudo_header.unused = 0;
        pseudo_header.protocol = IPPROTO_TCP;
        pseudo_header.length = htons(20);
}
```

Przykładowe działanie programu na podstawie widoku przechwyconych przez wireshark pakietów:



W programie “na sztywno” ustawiony jest zarówno źródłowy adres IP jak i źródłowy numer portu.

Zadanie 4. Gniazda surowe – protokół IPv6

Zadanie polega na zaimplementowaniu programu wykorzystującego gniazdo surowe dla protokołu IPv6.

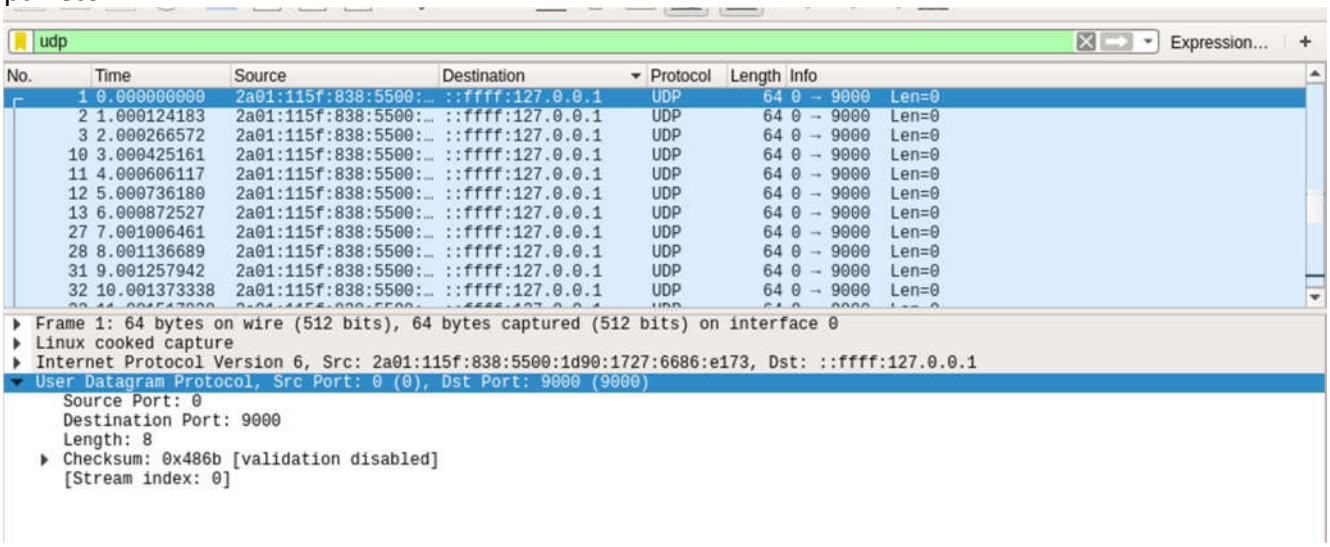
Program co sekundę wysyła puste datagramy UDP na adres (IPv6) i numer portu podany przez użytkownika w polach argumentu przy uruchamianiu programu.

W tym zadaniu, w przeciwieństwie do zadania 12, to system operacyjny, a nie programista, jest odpowiedzialny za wypełnienie pól nagłówka IP. Jądro systemu jest odpowiedzialne za obliczanie i weryfikację sumy kontrolnej w nagłówku UDP (*IPV6_CHECKSUM*). Opcja ta wymaga określenia położenia (*offsetu*) pola sumy kontrolnej nagłówka UDP. Dla protokołu UDP *offset* wynosi 6 bajtów.

Wypełnienie pseudo-nagłówka:

```
pseudo_header->ip_dst.s_addr = ((struct sockaddr_in*)rp->ai_addr)->sin_addr.s_addr;
pseudo_header->unused = 0;
pseudo_header->protocol = IPPROTO_UDP;
pseudo_header->length = udp_header->uh_ulen;
```

Przykładowe działanie programu na podstawie widoku przechwyconych przez wireshark pakietów:



Datagram UDP jest pusty, o czym świadczy jego rozmiar. Składa się zaledwie z 8 bajtów, co jest możliwie najmniejszą wartością.

Zadanie 5. Prosty program ping

Celem zadania było zaimplementowanie prostego programu ping wykorzystującego gniazda surowe do wysyłania i odbierania pakietów ICMP. Program wymaga uruchomienia z prawami roota, jako argument przyjmuje adres IP lub nazwę domenową pingowanego hosta.

Na początku programu następuje standardowe utworzenie socketa z użyciem funkcji *getaddrinfo()* i *socket(),* omawianych już wielokrotnie. Następnie utworzony zostaje nowy proces potomny:

```
auto ppid=fork();

   if(ppid==0)
       mainProcess();
   else if(ppid>0)
       childProcess();
   else
   {
       cerr<<"Fork error! "<<endl;
       perror("");
       exit(EXIT_FAILURE);
   }
```

Proces główny od tej chwili zajmuje się czterokrotnym wysyłaniem komunikatów, natomiast proces potomny odpowiada za odbieranie odpowiedzi.

Wynik działania sniffera dla uruchomionego programu:

```
 1 0.000000000 10.0.2.15   192.168.1.1    DNS      70 Standard query 0x764e  A google.com
 2 0.038479000 192.168.1.1 10.0.2.15      DNS      86 Standard query response 0x764e  A 216.58.214.78
 3 0.039049000 10.0.2.15   216.58.214.78  ICMP     74 Echo (ping) request  id=0x0b32, seq=1/256, ttl=128 (reply in 4)
 4 0.141603000 216.58.214. 10.0.2.15      ICMP     74 Echo (ping) reply    id=0x0b32, seq=1/256, ttl=55 (request in 3)
 5 1.039681000 10.0.2.15   216.58.214.78  ICMP     74 Echo (ping) request  id=0x0b32, seq=2/512, ttl=128
 6 1.165716000 216.58.214. 10.0.2.15      ICMP     74 Echo (ping) reply    id=0x0b32, seq=2/512, ttl=55 (request in 5)
 7 2.040792000 10.0.2.15   216.58.214.78  ICMP     74 Echo (ping) request  id=0x0b32, seq=3/768, ttl=128 (reply in 8)
 8 2.086361000 216.58.214. 10.0.2.15      ICMP     74 Echo (ping) reply    id=0x0b32, seq=3/768, ttl=55 (request in 7)
 9 3.041198000 10.0.2.15   216.58.214.78  ICMP     74 Echo (ping) request  id=0x0b32, seq=4/1024, ttl=128 (reply in 10)
10 3.086399000 216.58.214. 10.0.2.15      ICMP     74 Echo (ping) reply    id=0x0b32, seq=4/1024, ttl=55 (request in 9)
```

Na zrzucie (rekordy 1-2) widzimy zapytanie DNS wykonywane przez funkcję *getaddrinfo(),* następnie 4 pary request-reply pakietów ping. Zawartość przykładowego odebranego datagramu jest następująca i pokrywa się z informacjami wyświetlanymi przez program:

```
▾ Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0x154f [correct]
    Identifier (BE): 2866 (0x0b32)
    Identifier (LE): 12811 (0x320b)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [Request frame: 3]
    [Response time: 102,554 ms]
  ▾ Data (32 bytes)
      Data: 484f4146524e5444574d565a535045424b49574b5257584d...
      [Length: 32]
```

Wywołanie naszego programu razem z systemowym pingiem powoduje kolizje – program odbiera odpowiedzi na pakiety wysyłane nie tylko przez siebie, ale także przez program systemowy (widoczna niezgodność numerów ID będących PIDami dla dwóch odpowiedzi podczas jednego uruchomienia programu):

```
--IP details--
Source address: 216.58.214.78
TTL: 7
Header length: 5
Destination address: 10.0.2.15
--ICMP details--
Type: 0
Code: 0
ID: 27404
Sequence number: 256
```

```
--IP details--
Source address: 216.58.214.78
TTL: 7
Header length: 5
Destination address: 10.0.2.15
--ICMP details--
Type: 0
Code: 0
ID: 27148
Sequence number: 512
```