VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÝCH TECHNOLÓGIÍ

IPK projekt
SNIFFER PACKETOV

Matúš Tvarožný
xtvaro00

24.4.2021

OBSAH

OBSAH 2
ÚVOD 3
IMPLEMENTÁCIA 3
  VSTUPNÉ ARGUMENTY 3
  PRÁCA SNIFFERU, POUŽITÉ FUNKCIE 4
TESTOVANIE 5
PRÍKLADY SPUSTENIA 8
ZDROJE 8

ÚVOD

Toto je dokumentácia k druhému projektu z predmetu IPK. Vybral som si zadanie ZETA, čiže analyzátor (ďalej ako sniffer) sieťových paketov, ktorý by mal fungovať podobne ako aplikácia WireShark. Sniffer dokáže zachytávať IPv4(ICMPv4, TCP, UDP), IPv6(ICMPv6) a ARP pakety, v prípade potreby len tie, ktoré prechádzajú zadaným portom. Jeho výstupom je vypísanie zachyteného paketu, čas jeho zachytenia vo formáte RFC3339 a oboch adries(v prípade ARP paketov MAC adries) a portov(v prípade TCP a UDP paketov), ktorými paket prešiel.

IMPLEMENTÁCIA

Projekt som implementoval v jazyku C++. Cela implementácia snifferu je obsiahnutá v súbore `ipk-sniffer.cpp`. Základ projektu je postavený na knižnici `pcap`, ktorej funkcie sa starajú o všetku prácu s paketmi a ich odchytávaním. Za zmienku tiež stoja knižnice ako `chrono` na výpis času v požadovanom formáte RFC3339, knižnica `netinet` napríklad na prácu so štruktúrami podporovaných paketov a par knižníc jazyka C++, ktoré mi poskytli nezbité funkcie pri práci na projekte.

VSTUPNÉ ARGUMENTY

Sniffer je možné spúšťať s viacerými vstupnými argumentami(prepínačmi). Na ich poradí nezáleží.

Prvým z nich je prepínač `-i/--interface` *`rozhranie`*, je to povinný argument ktorý programu hovorí o tom na akom rozhraní sa budú analyzovať pakety. Ak tento argument nie je zadaný alebo je zadaný bez rozhrania program vypíše všetky dostupné rozhrania na aktuálnom zariadení a skončí. Rovnako sa program chová ak nie je zadaný žiadny vstupný argument.

Argument `-p` *`port`* slúži na filtrovanie paketov podľa portu. Tento argument nie je povinný a ak nebude zadaný, analyzované budú pakety na všetkých portoch na danom zariadení.

Argument `-n` *`pocet_paketov`* určuje počet paketov, ktoré budú vypísane na `stdout`, opäť nie je povinný a ak nebude uvedený vypíše sa jeden paket.

Nasledujúca skupina prepínačov slúži na nastavenie filtra paketov podľa ich protokolov. Ak nie je zadaný žiadny. analyzované budú všetky pakety, naopak ak je zadaný aspoň jeden, budú analyzované práve pakety daného typu. Sú to nasledujúce:
`-t/--tcp`, `-u/--udp`, `--icmp`, `--arp`

PRÁCA SNIFFERU, POUŽITÉ FUNKCIE

Vo funkcii `main()` je po definícii pár potrebných premenných volaná funkcia `ArgumentParser()`, ktorá má na starosti prácu so vstupnými argumentami.

Hneď na jej začiatku je volaná funkcia `LoadInterfaces()`, ktorá pomocou funkcie z knižnice `pcap` `pcap_findalldevs()` načíta všetky dostupné rozhrania na aktuálnom zariadení a vracia ukazovateľ na prvé rozhranie v zozname. Tento ukazovateľ je uložený do premennej `list_of_interfaces`. V prípade potreby tieto zariadenia vypísať na `stdout` je definovaná funkcia `PrintInterfaces()`. Prvý prechod všetkými vstupnými argumentami je implementovaný z dôvodu, ak by bol prepínač `-i/--interface` *`rozhranie`* zadaný bez rozhrania, napríklad posledný. Vypísanie všetkých dostupných rozhraní má prednosť. Na kontrolu či je zadané rozhranie validné a nachádza sa v zozname dostupných rozhraní je definovaná funkcia `CheckInterface()`, ktorá v prípade, ak je rozhranie nevalidné vypíše na `stderr` chybovú hlášku a program je ukončený. V poslednom rade sú tu nastavené aj globálne premenné.

Hneď po skončení tejto funkcie je volaná ďalšia funkcia a to `Filter()`. Ta sa stará o nastavenie filtra na základe globálnych premenných. Filter je uložený v globálnej premennej `filter` ako obyčajný reťazec, ktorý ale zodpovedá požadovanej syntaxi funkcií `pcap_compile()`, ktorá je volaná neskôr.

Po dobehnutí funkcie `Filter()` sa na rad dostáva ďalšia funkcia z knižnice `pcap` a to `pcap_open_live()`, ktorá je základom celej implementácie, otvára dane rozhranie a vracia jeho popisovač.

Teraz potrebujeme získať masku a IP siete na to nám poslúži funkcia `pcap_lookupnet()`. Z tejto dvojice potrebujeme iba masku kvôli už spomínanej funkcií `pcap_compile()`, ktorá si ju pýta ako vstupný parameter a prichádza na rad ako ďalšia.

`Pcap_compile()` slúži na skompilovanie filtra z reťazca do potrebného formátu `struct bpf_program`. Ak kompilácia prebehne úspešne(funkcia nevráti `PCAP_ERROR`) prichádza na rad funkcia na nastavenie filtru `pcap_setfilter()`.

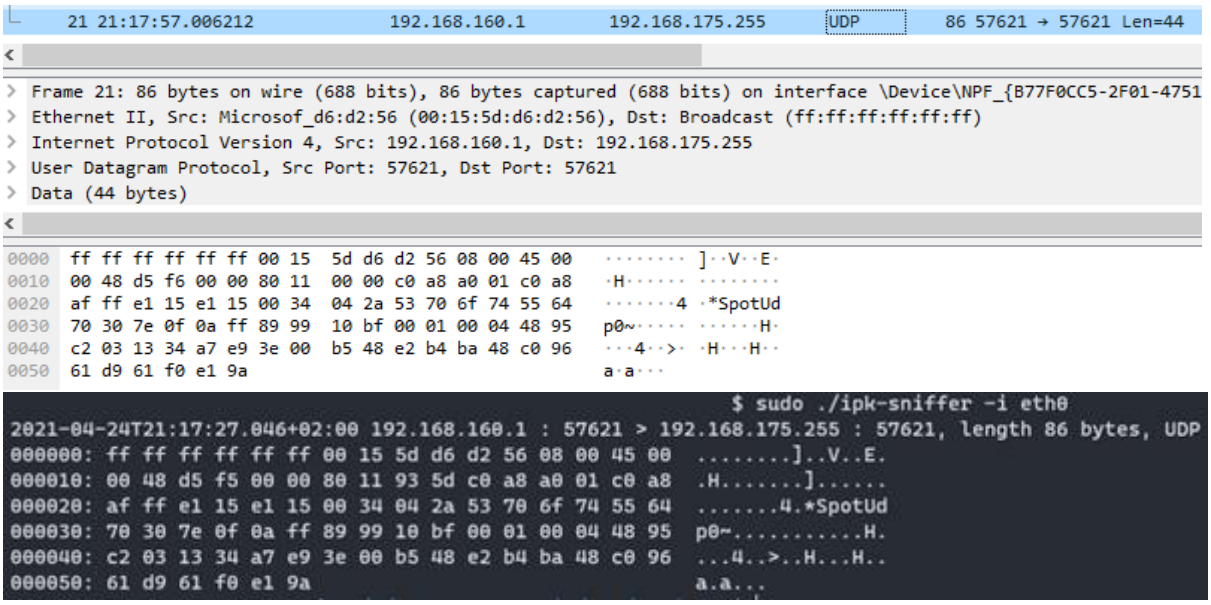
Posledná potrebná funkcia `pcap_loop()` je volaná až teraz, zachytáva pakety podľa nastaveného filtra, a prebehne toľkokrát (toľko paketov zachytí), koľko jej je zadané. Jej tretim parametrom je funkcia `PacketSniffer()` .

Táto funkcia je veľmi dôležitá a komplexná, prebehne zakaždým ako je zachytený vyhovujúci paket. Najprv zavolá funkciu `PrintTime()`, ktorá vypíše aktuálny čas (čiže čas kedy bol paket zachytený) v požadovanom formáte. Prvý „switch“ rozhodne podľa premennej `ether_type`, ktorá sa nachádza v štruktúre `ether_header` či sa jedná o paket IPv4, IPv6 alebo priamo ARP. V vnútri „casu“ pre IPv4 sa nachádza ďalší switch, ktorý rozhodne o tom k akému protokolu daný paket patrí, pre korektný prístup k údajom ako sú IP adresy a čísla portov, cez ktorý sa k nám paket dostal. V prípade IPv6 sa mi podarilo nasimulovať (teda aj implementovať) iba ICMPv6 pakety. Tie na výpis adresy volajú funkciu Ipv6Expander(). Ak je načítaný paket protokolu ARP sú vypísane obe jeho MAC adresy. Na výpis paketu ako takého sa v každom prípade používa funkcia `PrintPacket()`.

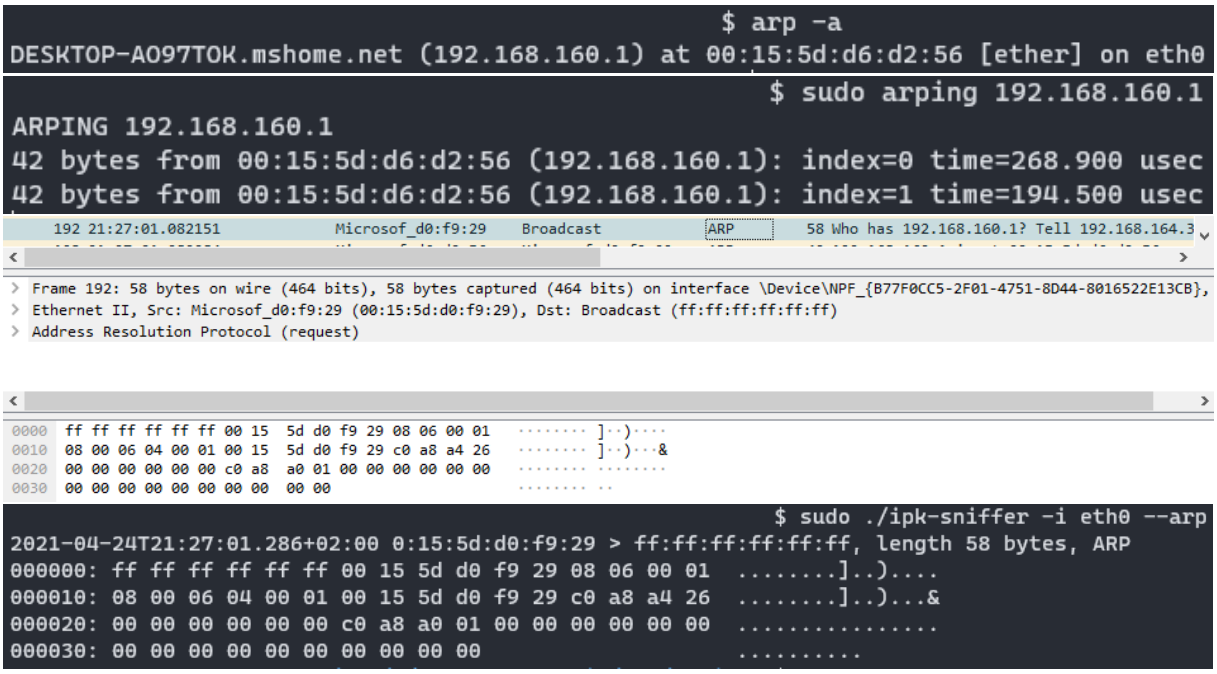
Na konci je zatvorené „zariadenie” na ktorom sa „sniffovalo” a uvoľnená alokovaná pamäť funkciou `pcap_close()` a `pcap_freecode()`.

TESTOVANIE

Na väčšinu testov mi vystačilo WSL a ručná kontrola načítaných paketov s aplikáciou WireShark. Keďže môj poskytovateľ internetu nepodporuje IPv6 a počítač mam pripojený pomocou wi-fi dostával som len TCP a UDP pakety(IPv4). Tieto po zachytení vyzerali nejako takto:



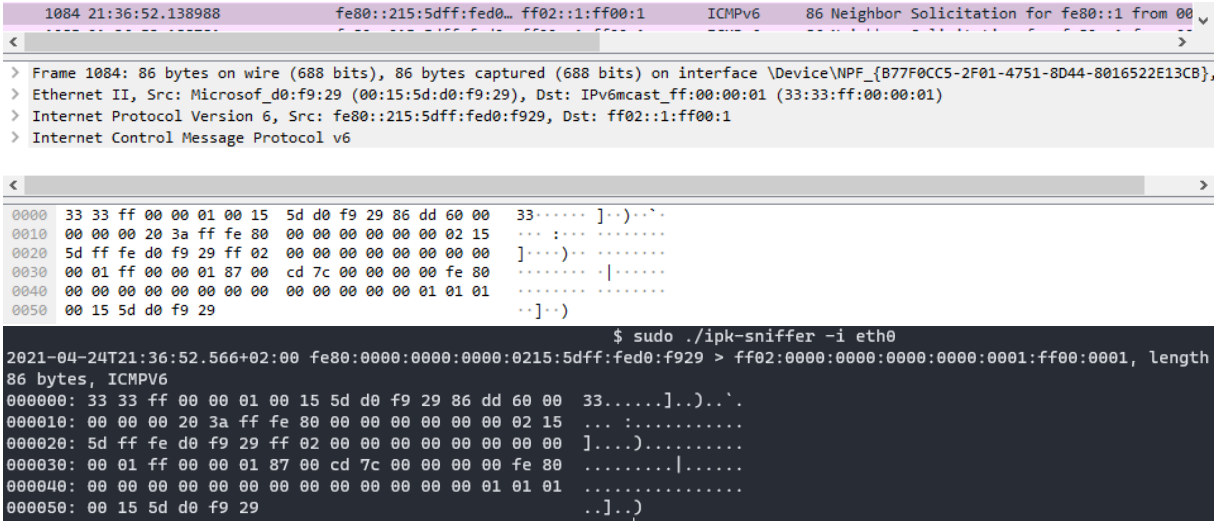
Na simuláciu ARP protokolov som používal nasledujúce:



Na simuláciu ICMPv6 protokolov:

```
                                                $ sudo ping6 fe80:0000:0000:0000:0000:0000:0000:1
PING fe80:0000:0000:0000:0000:0000:0000:1(fe80::1) 56 data bytes
From fe80::215:5dff:fed0:f929%eth0 icmp_seq=1 Destination unreachable: Address unreachable
From fe80::215:5dff:fed0:f929%eth0 icmp_seq=2 Destination unreachable: Address unreachable
From fe80::215:5dff:fed0:f929%eth0 icmp_seq=3 Destination unreachable: Address unreachable
```

Kde `fe80:0000:0000:0000:0000:0000:0000:1` je adresa nášho domáceho wi-fi routera.

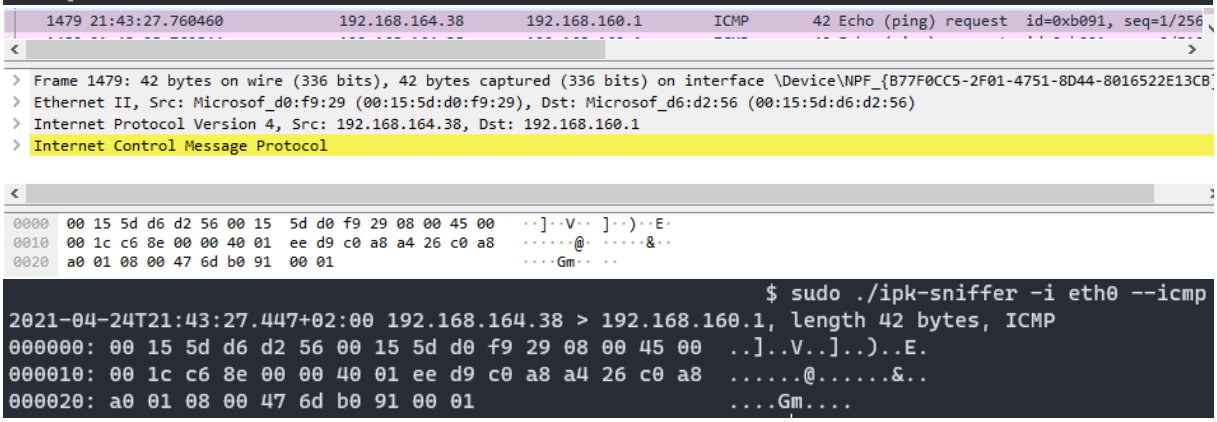



```
                                                $ sudo ./ipk-sniffer -i eth0
2021-04-24T21:36:52.566+02:00 fe80:0000:0000:0000:0215:5dff:fed0:f929 > ff02:0000:0000:0000:0000:0001:ff00:0001, length
86 bytes, ICMPV6
000000: 33 33 ff 00 00 01 00 15 5d d0 f9 29 86 dd 60 00  33......]..)..`.
000010: 00 00 00 20 3a ff fe 80 00 00 00 00 00 00 02 15  ... :...........
000020: 5d ff fe d0 f9 29 ff 02 00 00 00 00 00 00 00 00  ]....).........
000030: 00 01 ff 00 00 01 87 00 cd 7c 00 00 00 00 fe 80  .........|......
000040: 00 00 00 00 00 00 00 00 00 00 00 00 01 01 01  ................
000050: 00 15 5d d0 f9 29                                ..]..)
```

Na simuláciu ICMPv4 protokolov:

```
                                        $ sudo nping --icmp -e eth0 192.168.160.1

Starting Nping 0.7.80 ( https://nmap.org/nping ) at 2021-04-24 23:42 CEST
SENT (0.0050s) ICMP [192.168.164.38 > 192.168.160.1 Echo request (type=8/code=0) id=38754 seq=1] IP [ttl=64 id=45355 ipl
en=28 ]
SENT (1.0055s) ICMP [192.168.164.38 > 192.168.160.1 Echo request (type=8/code=0) id=38754 seq=2] IP [ttl=64 id=45355 ipl
en=28 ]
```

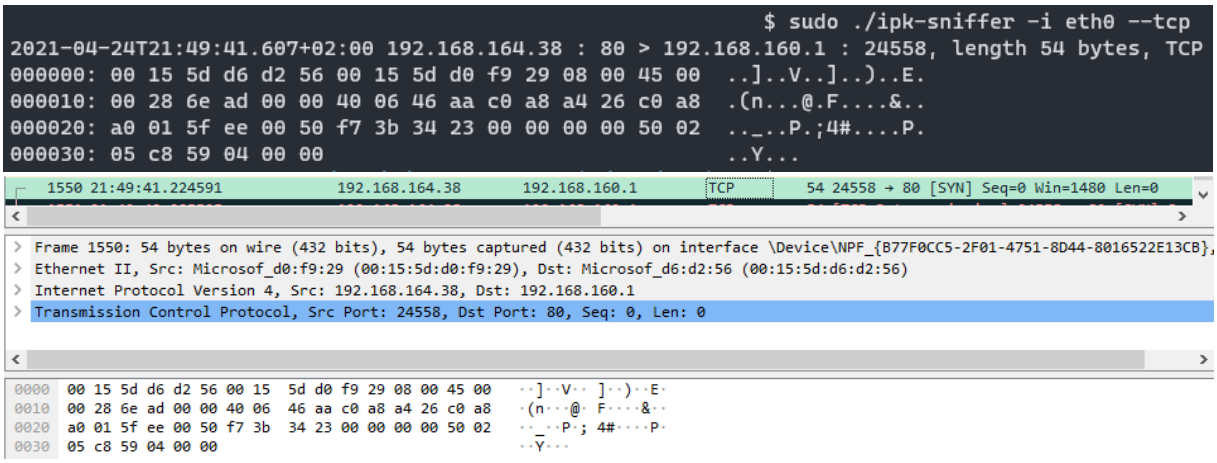



```
                                                        $ sudo ./ipk-sniffer -i eth0 --icmp
2021-04-24T21:43:27.447+02:00 192.168.164.38 > 192.168.160.1, length 42 bytes, ICMP
000000: 00 15 5d d6 d2 56 00 15 5d d0 f9 29 08 00 45 00  ..]..V..]..)..E.
000010: 00 1c c6 8e 00 00 40 01 ee d9 c0 a8 a4 26 c0 a8  ......@......&..
000020: a0 01 08 00 47 6d b0 91 00 01                    ....Gm....
```

Krátka ukážka, že všetko funguje ako ma aj s TCP protokolom:

```
                                        $ sudo nping --tcp -e eth0 192.168.160.1

Starting Nping 0.7.80 ( https://nmap.org/nping ) at 2021-04-24 23:49 CEST
SENT (0.0036s) TCP 192.168.164.38:24558 > 192.168.160.1:80 S ttl=64 id=28333 iplen=40  seq=4147852323 win=1480
SENT (1.0043s) TCP 192.168.164.38:24558 > 192.168.160.1:80 S ttl=64 id=28333 iplen=40  seq=4147852323 win=1480
```



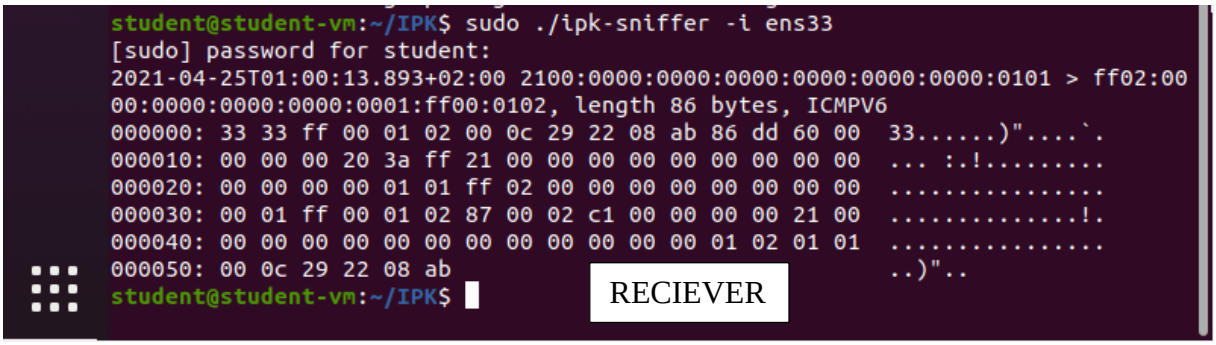
Pokusy o nasimulovanie TCP/UDP IPv6 boli nasledovné. Ako prvú som skúsil použiť funkciu `nping` s nasledovnými parametrami, bohužiaľ bez výsledku.

```
$ sudo nping --udp -6 -e eth0 fe80:0000:0000:0000:0000:0000:0000:1

Starting Nping 0.7.80 ( https://nmap.org/nping ) at 2021-04-25 00:23 CEST
sendto in send_packet: sendto(4, packet, 8, 0, fe80::1, 28) => Invalid argument
Offending packet: BOGUS!  Can't parse supposed IP packet
SENT (0.0030s) UDP :::53 > fe80::1:40125
sendto in send_packet: sendto(4, packet, 8, 0, fe80::1, 28) => Invalid argument
```

Vždy to bol iba ICMPv6 protokol.

Potom som nasimuloval UDP IPv6 paket na referenčných strojoch, ale „sniffer“ zachytil stále iba ICMPv6.



WireShark mi na referenčnom stroji nefungoval.

PRÍKLADY SPUSTENIA

```
$ sudo ./ipk-sniffer
$ sudo ./ipk-sniffer -i
$ sudo ./ipk-sniffer -i eth0
$ sudo ./ipk-sniffer -i eth0 -n 0
$ sudo ./ipk-sniffer -i eth0 -p 32652
$ sudo ./ipk-sniffer -i eth0 --tcp --udp
$ sudo ./ipk-sniffer -i eth0 -n 0 -t --icmp
$ sudo ./ipk-sniffer -i eth0 -n 5 -u -p 32652 --arp -t
```

ZDROJE

1. https://linuxos.sk/clanok/packet-capturing-s-libpcap-1/
2. https://linuxos.sk/clanok/packet-capturing-s-libpcap-2/
3. https://stackoverflow.com/questions/54325137/c-rfc3339-timestamp-with-milliseconds-using-stdchrono
4. https://linux.die.net/man/3/pcap
5. https://www.cplusplus.com/reference/
6. https://stackoverflow.com/questions/3727421/expand-an-ipv6-address-so-i-can-print-it-to-stdout
7. https://samsclass.info/124/proj11/P10xN-scapy-ipv6.html
8. https://www.tcpdump.org/manpages/
9. https://www.programcreek.com/cpp/?code=mq1n%2FNoMercy%2FNoMercy-master%2FSource%2FClient%2FNM_Engine%2FINetworkScanner.cpp