Scanner sieťových služieb - Varianta OMEGA

Tomáš Sasák

April 19, 2019

Contents

1 Úvod do problematiky

Zadanie je nasledujúce, implementujte sieťový TCP/UDP skener v jazyku C/C++. Program oskenuje IP adresu a porty. Na standartný výstup vypíše, v akom stave sa porty nachádzajú.

1.1 UDP skenovanie

Pri UDP skenovaní, očakávame pri zavretom porte správu protokolom ICMP typu 3. Všetko ostatné považujeme, ako otvorené.

1.2 TCP skenovanie

Pri TCP skenovaní, sa posiela SYN paket, ale neprebieha 3-way-handshake. Ak príde odpoveď typu RST, port sa považuje za zatvorený, ak odpoveď typu SYN-ACK považuje sa za otvorený a ak nepríde žiadna odpoveď, je nutné poslať nový SYN paket, slúžiaci pre overenie, či sa predchádzajúci SYN paket nestratil.

2 Implementácia

Vybraný jazyk pre implementáciu tohoto projektu je jazyk `C++`. Projekt bol implementovaný pomocou objektovo-orientovaného programovania. Celý projekt je zabalený do jednej triedy, táto trieda má meno `Scanner`.

Pre odosielanie skenovaích paketov boli použité BSD sockety. Tieto sockety sú nastavené na RAW a protokol `IPv4` alebo `IPv6`. Taktiež, pri socketoch patriace pod `IPv4` je použité nastavenie definované pre socket `IP_HDRINCL`, ktoré znamená, že `IPv4` hlavička je vytváraná aplikáciou a neni potrebné od kernelu, aby vytváral vlastnú. Pre `IPv6` sa mi nepodarilo nájsť podobné nastavenie, čo je dôsledok toho, že pri skenovaní `IPv6` adries je `IPv6` hlavička zostavená pomocou kernelu.

Z predchádzajúceho odseku vyplýva, že najskôr je zostavená `IPv4` paket hlavička a následovne za tým je zostavená `UDP` alebo `TCP` paket hlavička. Pri `IPv6`, je zostavená len `UDP` alebo `TCP` paket hlavička.

2.1 Súbory projektu

Skener sa skladá z nasledujúcich súborov.

- `ipk-scan.hpp` - Deklarácie triedy a metód skeneru, použité knižnice, definované číselné makrá a poriadne dokumentované elementy.
- `ipk-scan.cpp` - Srdce skeneru, implementácia spracovávania argumentov a následovne spúšťanie `TCP` a `UDP` skenovania.
- `tcp.cpp` - Implementácia `TCP` skenovania a výpis výsledkov `TCP` skenovania.
- `udp.cpp` - Implementácia `UDP` skenovania a výpis výsledkov `UDP` skenovania.
- `pseudo-headers.cpp` - Definícia štruktúry, použitá pre počítanie `TCP` checksum.

2.2 Použité knižnice

V nasledujúcom v liste, sa nachádza výpis použitých knižníc pre implementáciu tohoto projektu (základné knižnice pre prácu s `C++` a `C` sa v liste nenachádzajú).

- `pcap.h` - Knižnica použitá, pre odchytávanie odpovedí od skenovaného portu.
- `thread` - Knižnica použitá, pre viacvláknové programovanie (vysvetlenie neskôr).
- `mutex` - Knižnica použitá, pre viacvláknové programovanie a synchronizáciu vlákien.

- `netinet/ip.h, netinet/udp.h, netinet/tcp.h` - Knižnice použité, pre použitie už definovaných hlavičiek `UDP`, `TCP` a `IP` packetov.
- `netinet/if.h` - Knižnica použitá, pre vyhľadávanie zariadení (interface).
- `netdb.h` - Knižnica použitá, pre preklade domény na `IP` adresu.
- `regex.h` - Knižnica použitá, pre správne spracovávanie argumentov.

2.3 ipk-scan.cpp

Jadro skeneru, na počiatku spracuje parametre funkcia `void Scanner::parse_arguments` pomocou funkcie `getopt_long_only` a naplní atribúty inštancie objektu `Scanner`.

Druhý krok, je získanie `IP` adresy skenovaného cielu. Ak bolo špecifikované zariadenie, vyhľadáva sa explicitne `IP` adresa typu založeného na tom, akú `IP` adresu toto zariadenie vlastní. Ak zariadenie nebolo zadané, vyberá sa prvá nalezená adresa. To znamená, že prednosť má vždy `IPv6` adresa pred `IPv4` adresou. Adresa sa vyhľadáva pomocou funkcie `getaddrinfo` a preferovaná adresa sa predáva pomocou štruktúry `hints`.

Ako nasledujúci krok, je načítavanie lokálnej adresy a interface (funkcia `void Scanner::fetch_local_ip`). Ak bolo zadané zariadenie od uživatela (interface), tak funkcia končí, pretože spracovávanie zadaného zariadenia má na starosti funkcia pre spracovávanie argumentov. Ak zariadenie nebolo zadané, pomocou funkcie `getifaddrs`, sa skener pýta kernelu pre list existujúcich zariadení a vyberá sa prvé zariadenie ktoré funguje, nemá loopback adresu a jeho `IP` adresa má zhodujúcu skupinu (4 alebo 6). Adresa zariadenia je ešte prekladaná pomocu `getnameinfo`, aby skener mohol `IP` paket hlavičku vyplniť správnymi údajmi.

Týmito krokmi, je práca tejto časti skenera hotová a prechádza úloha skenovania portov. Tieto vlastnosti sú naimplementované v nasledujúcich súboroch.

2.4 Vypĺňanie paket hlavičiek

Pri skenovaných portoch na adresách `IPv4`, je nutné vyplniť `IPv4` hlavičku. Pri nastavení socketu použitím `IP_HDRINCL` je zaručené, že kernel bude brať hlavičku poskytnutú aplikáciou. Táto hlavička je vyplnená zdrojovou adresou, cielovou adresou, počtom hopov, veľkosťou, typom ďalšieho protokolu v datagrame a typom tejto `IP` (4/6). Checksum nieje potrebné rátať pri IPv4, toto zaručuje kernel ak checksum hodnota je nastavená na hodnotu 0. (viz. [2])

Pri skenovaných portoch na adresách `IPv6`, neni potrebné vyplniť `IPv6` hlavičku. A je automaticky vyplnená kernelom.

Pri skenovaných portoch pomocou protokolu `UDP`, je nutné vyplniť `UDP` hlavičku. Hlavička je vyplnená zdrojovým portom, cielovým portom, velkosťou a checksum pri IPv4 je nastavený na 0 (IPv4 checksum je vyrátaný, nieje potrebné rátať UDP checksum). Pri `IPv6` je už potrebné vyrátať `UDP` checksum, pretože `IPv6` hlavička neobsahuje checksum. Toto je vyriešené pomocou socketového nastavenia `IPV6_CHECKSUM`, ktorému pri zadanom offsete (tam kde sa nachádza

v hlavičke hodnota checksum), dokáže vyrátať checksum sám a vloží túto hodnotu do hlavičky na daný offset. (viz. [3])

Pri skenovaných portoch pomocou protokolu `TCP`, je nutné vyplniť `TCP` hlavičku. Hlavička je vyplnená zdrojovým portom, cielovým portom, veľkosťou, sekvenčným číslom, predchádzajúcim sekvenčným číslom, offsetom, potrebnými flagmi, veľkosťou okna, a checksumom. Pri `IPv4` je nutné tento TCP checksum vypočítať, toto je vykonané pomocou pseudo-hlavičky a checksum funkcie (viz. [4], [5], [6]). Pri `IPv6` je opäť možné, požiadať kernel pomocou offsetu a nastavenia `IPV6_CHECKSUM` o výpočet checksum automaticky (viz. [1]).

2.5 udp.cpp

Vyžiadané skenované porty, sú uložené v atribúte triedy `Scanner` a to vo vektore `udpTargetPorts`. Ak je vektor prázdny, skenovanie neprebieha a funkcia končí.

Ako prvé, sa vytvorí `ICMP (UDP)` paketový odchytávač. Toto je realizované pomocou knižnice `libpcap`. Vytvorí sa odchytávač, buď na zadanom zariadení, alebo na zariadení ktoré si skener vybral sám. Následovne sa na odchytávač, prichytí nasledujúci filter:
Pre IPv4:

```
icmp[icmpcode] = 3 and src ...
```

Pre IPv6:

```
icmp6 && ip6[40] == 1 and src ...
```

Tri bodky značí adresa cielu skenu. Tento filter odchytáva `ICMP (UDP)` pakety, typu port unreachable.

Tento vytvorený odchytávač, je uložený do inštancie triedy `Scanner`. Následuje odosielanie skenovacích paketov a odchytávanie odpovede. Toto je implementované pomocou viacvláknového programovania. V tomto okamihu sa vytvára, nové vlákno ktoré odosiela pakety a hlavné vlákno na ktorom odchytáva `ICMP` pakety daný odchytávač. Priebeh je následovný.

Hlavné (odchytávacie) vlákno pomocou `pcap_loop` a daného odchytávača začne odchytávať pakety, pomocou callback funkcie `callback_udp` dokáže informovať, odosielacie vlákno, že daný port je zatvorený. Existuje tu globálna premenná, menom `bool wasClosed`. Ktorú ak odchytávač, zavolá callback, zmení na hodnotu `True`. Medzitým v odosielaciom vlákne sa vytvorí socket, nastaví sa a odosielač, si zostaví paket (v tomto prípade UDP) a odosiela paket po jednom na port daný indexom v vektore `udpTargetPorts` a čaká štandartne (viz. Implementované rozšírenia) 2 sekundy na odpoveď. Po danom čase, nazrie do premennej `wasClosed` a na základe jej hodnoty vypíše, či daný port je zatvorený alebo otvorený. Ak nastené prípad, že port je otvorený, štandartne odosielač odosiela znova 1 `UDP` packet (viz. Implementované rozšírenia) aby sa overilo, či sa daný paket nestratil a port je skutočne otvorený.

Tu sú vidieť, príznaky zlého synchronizovania a to presnejšie príznaku "data race". Toto je ale ošetrené pomocou `mutex udpLock`, čím je zaistené že len jedno

vlákno môže pristupovať zaráz ku premennej `wasClosed`. Ak port neodpovedal `ICMP`

Po dokončení odosielania a prímania paketov, odosielacie vlákno ukončuje odchytávanie filtru pomocu `pcap_breakloop` a hlavné vlákno čaká na dokončenie odosielacieho vlákna pomocou `join`.

2.6 tcp.cpp

Vyžiadné skenované porty(`TCP`), sú uložené v atribúte triedy `Scanner` a to vo vektore `tcpTargetPorts`. Ak je vektor prázdny, skenovanie neprebieha a funkcia končí.

Postup je velmi podobný, ako v prípade `UDP` skenovania. Ako prvé, sa vytvoria dva odchatývače, `TCP RST` odchytávač a `TCP SYN-ACK` odchytávač. Na odchytávače, sa umiestnia nasledujúce filtre:
Pre IPv4 a RST:

```
tcp[tcpflags] & (tcp-rst) != 0 and src ...
```

Pre IPv6 a RST:

```
((ip6[6] == 6 && ip6[53] & 0x04 == 0x04) || (ip6[6] == 6 &&
tcp[13] & 0x04 == 0x04)) and src ...
```

Pre IPv4 a SYN-ACK:

```
tcp[tcpflags] & (tcp-syn|tcp-ack) != 0 or tcp[tcpflags] &
(tcp-syn) != 0 and tcp[tcpflags] & (tcp-rst) = 0 and src
```

Pre IPv6 a SYN-ACK:

```
((tcp[13] & 0x12 == 0x12) || (ip6[6] == 6 && ip6[53] & 0x12 == 0x12))
|| ((tcp[13] & 0x02 == 0x02) || (ip6[6] == 6 && ip6[53] & 0x02 == 0x02))
and src ...
```

Tri bodky značia adresu ciela skenu.

Tieto vytvorené odchytávače, sú uložené do inštancie triedy `Scanner`. Následuje opäť odosielanie paketov a odchýtávanie odpovedí. Toto je velmi podobne naimplementované, ako pri `UDP`. Ale s rozdielom, že odchytávač nieje v hlavnom vlákne, ale odchytávače majú svoje vlastné vlákno. Pribeh je následovný.

Odchytávačom sa vytvoria samostatné procesy, ktoré sú vždy dva. Jeden slúžiaci pre odchytávanie `RST TCP` paketov a druhý pre odchytávanie `SYN-ACK TCP` paketov. Tieto odchytávače začnú odchytávať dané pakety pomocu `pcap_loop` a cez callback funkcie, oznamujú odosielaciemu vláknu aký paket dostali. Táto komunikácia je opäť vyriešená, pomocou globálnej premennej `int tcpPortStates` a rovnako v prevencií "data race" je využitý `mutex tcpLock`. Vzhľadom na to, že v tomto prípade môže byť viac stavov portu, ako len otvorený a zatvorený, používajú sa tu číselné makrá vytvorené v súbore `ipk-scan.hpp`, sú to makrá `TCP_CLOSED`, `TCP_OPEN` a `TCP_FILTERED`.

Odosielacie (hlavné) vlákno, opäť si vytvorí socket, nastaví socket a odosielač si zostaví paket (v tomto prípade `SYN TCP`) a odosiela paket po jednom na port daný indexom v vektore `tcpTargetPorts` a čaká štandartne (viz. Implementované rozšírenia) 2 sekundy na odpoveď. Po danom čase, nazrie do premennej `tcpPortStates` a na základe jej hodnoty vypíše, či daný port je zatvorený, otvorený alebo filtrovaný. Ak nastane prípad, že port je filtrovaný, štandartne odosielač odosiela znova 1 `SYN TCP` packet (viz. Implementované rozšírenia) aby sa overilo, či sa daný paket nestratil a port je skutočne filtrovaný.

Po dokončení odosielania a prímania paketov, odosielacie vlákno ukončuje odchytávanie filtru pomocou `pcap_breakloop` a čaká na dokončenie odchytávacích vlákien pomocou `join`.

3 Implementované rozšírenia

Počas implementácie hlavných funkcií skeneru, ma napadlo implementovať zopár rozšírení, ktoré mi prišli dosť užitočné pre pohodlné a pokročilejšie skenovanie portov.

3.1 Počet opätovných zasielaní TCP/UDP paketu

Parameter: `-rt <nasobok-opakovania-int>` a `-ru <nasobok-opakovania-int>`

Toto rozšírenie poskytuje uživatelovi zadať, počet kolko krát má skener opakovať odosielanie buď `TCP-SYN` alebo `UDP` paketu, ak port je filtrovaný alebo otvorený.
Príklad použitia:

```
sudo ./ipk-scan -pt 1-100 -pu 1-100 merlin.fit.vutbr.cz -rt 3 -ru 2
```

Výsledok: Ak jeden z portov TCP bude filtrovaný, odosielanie `SYN TCP` packetu sa bude opakovať maximálne 3 krát. Ak jeden z portov `UDP` bude otvorený, odosielanie `UDP` paketu sa bude opakovať maximálne 2 krát. Štandartne je táto hodnota inicializovaná na hodnotu 1 opakovanie.

3.2 Doba čakania na odpoveď od portu

Parameter: `-wt <doba-cakania-sekundy>` a `-wu <doba-cakania-sekundy>`

Toto rozšírenie poskytuje uživatelovi zadať dĺžku doby (v sekundách), kolko má skener (odosielacie vlákno) čakať na odpoveď od portu.
Príklad použitia:

```
sudo ./ipk-scan -pt 1-100 -pu 1-100 merlin.fit.vutbr.cz -wt 3 -wu 2.5
```

Výsledok: Skener (odosielacie vlákno) bude čakať na odpoveď pri `TCP` skenovaní 3 sekundy a pri `UDP` skenovaní 2 sekundy. Štandartne je táto hodnota inicializovaná na hodnotu 2.5 sekundy. [1]

4 Použité materiály

References

[1] RFC 793 - Transmission Control Protocol, https://tools.ietf.org/html/rfc793

[2] RFC 791 - Internet Protocol https://tools.ietf.org/html/rfc791

[3] RFC 768 - User Datagram Protocol https://tools.ietf.org/html/rfc768

[4] RFC 1071 - Computing the Internet Checksum https://tools.ietf.org/html/rfc1071

[5] How is TCP UDP Checksum Calculated? https://www.slashroot.in/how-is-tcp-and-udp-checksum-calculated

[6] Raw socket examples, taken checksum https://www.tenouk.com/Module43a.html

[7] A brief programming tutorial in C for raw sockets http://www.cs.binghamton.edu/~steflik/cs455/rawip.txt

[8] RAW, SOCKET, IP, TCP, UDP man pages https://linux.die.net/man