Uniwersytet w Białymstoku

Instytut Informatyki

Mateusz Maciejczuk

Policy routing oraz VPN na firewallu opartym o IPFilter

Praca dyplomowa napisana
pod kierunkiem
dr. Mariusza Żynela

Białystok 2021

Składam wyrazy uznania
oraz serdeczne podziękowania
dr. Mariuszowi Żynelowi
za ogromną pomoc
i poświęcony czas

Mateusz Maciejczuk

Spis treści

Wstęp

Bezpieczeństwo w świecie wirtualnym jest równie ważne, jak w świecie realnym. Wraz z burzliwym rozwojem technologii można zaobserwować, jak coraz więcej aspektów życia przechodzi w sferę wirtualną. Już dzięki jednemu kliknięciu smartfona można załatwić sprawę urzędową, unikając stania w kolejce do urzędnika. Upowszechniają się kryptowaluty. Należy znaleźć sposób, by z jednej strony zapewnić bezpieczeństwo w wirtualnym świecie, a z drugiej, aby dostęp do usług sieciowych był swobodny, nieograniczony.

Zapora sieciowa to niemal synonim bezpieczeństwa sieciowego, natomiast VPN to technologia ułatwiająca pracę zdalną - jakże ostatnio pożądaną. Policy routing z kolei można zastosować, aby lepiej wykorzystać dostępne zasoby sieciowe.

Celem tej pracy jest odnalezienie i naprawienie usterek w oprogramowaniu zapory sieciowej IPFilter, tak aby możliwe było jednoczesne korzystanie z VPN oraz policy routing. Do testowania wykorzystuje się ekosystem złożony z dwóch sieci LAN, które za pośrednictwem jednego routera z IPFilter łączą się z Internetem za pośrednictwem dwóch różnych dostawców ISP. Ekosystem ten został całkowicie zaimplementowany w środowisku wirtualnym VirtualBox. W sieciach LAN używa się połączeń VPN ze zdalnymi sieciami.

W pierwszym rozdziale, jako wprowadzenie do tematyki, zostaną opisane ogólne zagadnienia filtrowania, translacji adresów oraz specyficznego trasowania w sieciach TCP/IP. Drugi rozdział to krótki przegląd technologii VPN. W kolejnym, trzecim rozdziale opisane jest oprogramowanie IPFilter – jego historia, architektura oraz sposoby konfiguracji. Ostatni, czwarty rozdział to sprawozdanie z wykonanych prac praktycznych polegających na modyfikacji kodu IPFilter, kompilacji i testowaniu.

W formie dodatków, na końcu pracy, załączona została gramatyka reguł, parametry tuningowe oraz wykaz szczegółowych zmian dokonanych w kodzie zapory IPFilter.

Rozdział 1

Filtrowanie i trasowanie TCP/IP

Każdy chce chronić swoją prywatność, dlatego chronimy: zwykłe osoby, uczelnie, duże korporacje. Więc aby chronić sieci komputerowe, systemy, strony internetowe przed atakami osób z zewnątrz, bądź przed dostaniem się tych osób do naszych zasobów. Z początku taką funkcje pełniły routery, lecz wraz z rozwojem sieci Internet została stworzona technologia ściany ogniowej. Pierwsza wzmianka o stosowaniu firewall'a w sieciach komputerowych pojawia się pod koniec lat 80. W roku 1988 zostaje opublikowana pierwsza generacja ściany ogniowej opierająca się na filtrowaniu pakietów. Działało to na zasadzie filtrowania przesyłanych danych, jeżeli dane pasowały co do ustalonych wytycznych w filtrze, zostały przepuszczane, w przeciwnym wypadku były odrzucane i administrator był informowany o wykrytej nieprawidłowości. Standardowo dla tej generacji przesył danych odbywał się na portach TCP i UDP (por. [1]).

Druga generacja powstawała w latach 1989-1990 i przez twórców została określona jako circuit-level-gateways. W modelu OSI działała ona tylko do warstwy transportowej. Przez firewall przechodzą dane i są określane, czy są początkiem lub końcem połączenia, lub czy nie należą do żadnego połączenia.

Trzecia generacja datowana jest na rok 1993. Firewall ten działa na warstwie aplikacji i dzięki temu rozumie protokoły takie jak FTP, HTTP, DNS. Ściana ogniowa tej generacji wykrywa, czy któryś z protokołów nie jest szkodliwie nadużywany lub, czy dany protokół łączy się z prawidłowym portem. W roku 2012 powstała tzw. zapora nowej generacji (next-generation firewall), która rozszerza kompetencje ściany ogniowej w warstwie aplikacji. Firewalle wykorzystują różne techniki ochrony dostępu między innymi: filtrowanie pakietów, translację adresów, stateful filters. Do ochrony służy również oprogramowanie Intrusion Detection System (IDS). Jest to zbiór różnych oprogramowań bądź sprzętu, który służy do wykrycia ataku na sieć, informując administratora.

Możemy znaleźć wiele pakietów oprogramowania, które pozwalają nam stworzyć firewall. Jednym z takich pakietów jest IPFilter. IPFilter jest wie-

loplatformowym pakietem typu Open Source, który zawiera w sobie funkcję ściany ogniowej, translację adresów (NAT) oraz stateful firewall.

1.1 Filtrowanie pakietów

Funkcję filtrowania pakietów wykorzystywaną w firewallu można w pewnym stopniu porównać do działania routera. Po odpowiednim skonfigurowaniu dane z zewnątrz trafiają do punktu docelowego. Jednak gdy przyjrzymy się działaniu filtra pakietów, można dostrzec różnice. Ściana ogniowa pierwszej generacji została oparta na funkcji filtrowania pakietów. Filtry te są ustawiane najczęściej na routerach, mostach lub hostach. Służą one do przeprowadzania wybiórczej kontroli na danych, które są odbierane z sieci lub do niej wysyłane. Kontrola przesyłanych danych zależy od tego, jak skonfigurujemy dany filtr. Możemy np. ustalić, z jakich adresów IP dane będą do nas przesyłane lub odrzucane, ustalić które usługi korzystające z portów TCP lub UDP mają zostać odrzucone. Ogólne działanie najprostszych filtrów polega na porównaniu różnych nagłówków ramki danych w warstwach 3 i 4 modelu OSI. Kiedy przesyłany pakiet nie będzie pasował do konfiguracji bazowej i konfiguracji ustalonej w filtrze zostanie on odrzucony. Podczas badania pakietu filtr analizuje między innymi:

- adres źródłowy,
- adres docelowy,
- protokół (TCP, UDP, ICMP),
- rozmiar pakietu,
- port źródłowy (TCP, UDP),
- port docelowy (TCP, UDP).

Filtrowanie możemy podzielić na filtrowanie bezstanowe i filtrowanie stanowe. Pierwszy przypadek jest najpowszechniej spotykanym sposobem filtrowania i był stosowany przy pierwszych firewallach. W ścianach ogniowych bezstanowych każdy protokół badany był indywidualnie. W standardowych ustawieniach badany był tylko nagłówek tego protokołu, lecz dało się ten filtr skonfigurować tak, by badał każdy z elementów protokołu. Problemem bezstanowej filtracji jest fakt, że nie zapamiętuje ona pakietów, które zostały przepuszczone w przeszłości, co może skutkować ominięciem firewall'a poprzez podszycie się pod inny adres IP w nagłówku pakietu. Filtrowanie stanowe natomiast nie traci informacji o ruchu danych przechodzących przez ścianę ogniową, pozwala to odrzucić pakiet, który wcześniej został uznany za niebezpieczny. W odróżnieniu od filtracji bezstanowej filtrowanie stanowe działa

w warstwie 3, 4 i 5 modelu OSI co pozwala na pobranie informacji na temat przepływającej sesji danych. Zasada działania polega na zapamiętaniu wcześniejszego połączenia pomiędzy dwoma stronami, zapisania tego połączenia w tablicy stanów i przy próbie następnego połączenia porównania, czy dane zawarte w pakiecie zgadzają się ze sobą. Jeżeli przy następnej próbie połączenia przyjdzie zgodna odpowiedź, pakiet zostanie przepuszczony, w przeciwnym wypadku zostaje on odrzucany i nie dopuszcza do sytuacji podszycia się tak jak w filtracji bezstanowej. Zapisane dane z tablicy stanów zostaną usunięte, kiedy sesja zostanie zamknięta lub jej czas wygaśnie (por. [1, 5]).

1.2 Translacja adresów

W prostych słowach translację adresów NAT możemy opisać jako zamianę wielu adresów IP w jeden. Na przykład w sieci prywatnej przedsiębiorstwa dysponujemy dużą pulą adresów IP, podczas gdy każdy host będzie się łączył z siecią zewnętrzną, widziane będzie tylko jedno IP. Jednak translacja adresów jest bardziej skomplikowana. Ideą przyświecającą powstaniu NAT była obawa wykorzystania wszystkich adresów IP w technologii IPv4 oraz skalowalność sieci HTCP/IP. NAT zostaje wprowadzony do użytku na początku 1990 roku i staje się tymczasowym środkiem na uporanie się ze zmniejszającą się pulą adresów w protokole IPv4. Translacja adresów miała odgrywać rolę środka tymczasowego do czasu wprowadzenia protokołu IPv6. W najprostszej konfiguracji NAT działa na zasadzie zmiany informacji o IP w nagłówku tego pakietu podczas przesyłania tego pakietu. Powoduje to, że nasz adres, który należy do puli prywatnej, zostanie zamieniony na przypisany nam adres z puli publicznej. Jednak nas interesuje zastosowanie translacji adresów do zwiększenia bezpieczeństwa. Dzięki NAT ukrywamy adres wewnętrzny, co utrudnia dla osoby z zewnątrz np. zmapowanie naszej sieci wewnętrznej. NAT ukrywa naszą tożsamość poprzez ukrycie informacji TCP/IP o naszym hoście. Oznacza to, że wszystkie hosty pracujące w tej samej sieci co my po zastosowaniu NAT mają na zewnątrz jeden adres publiczny. Firewall oparty na NAT, po wykonaniu translacji nadpisuje adres hosta docelowego lub źródłowego w nagłówku IP przetłumaczonymi adresami. Rozróżnia się dwa rodzaje translacji adresów (por. [1, 5]):

- Source Network Adress Trasnlation (SNAT),
- Destination Network Adress Tanslation (DNAT).

Oba rodzaje są wykorzystywane w IPFilter.

1.3 Filtrowanie pakietów z pamięcią stanów

Filtrowanie z pamięcią stanu powstało po to, aby usprawnić zwykłe filtrowanie pakietów. Poprawia ono bezpieczeństwo i zwiększa moc filtrowania zwykłego filtra pakietu. Filtrowanie pakietów z pamięcią stanów działa na zasadzie śledzenia połączeń protokołu sterowania transmisji TCP, zaczynając od procedury three-way handshake, następnie przez transmisje danych i kończąc na ostatnim pakiecie sesji. Firewall oparty o filtrowanie stanowe sprawdza najpierw, czy w nagłówku TCP pakietu danych, ustawiona jest flaga synchronize i jednocześnie, czy nie są ustawione inne flagi. Taki pakiet protokołu TCP rozpoczyna procedurą three-way handshake, który otwiera sesje. Fakt ten zapisywany jest w tablicy stanów (state table). Zapamiętywane zostają

- adres źródłowy,
- adres docelowy,

oraz

- port źródłowy,
- port docelowy.

O tym, czy dany pakiet pasuje do sesji, decydują właśnie adres źródłowy i docelowy, porty źródłowy i docelowy oraz numer sekwencyjny TCP. Kolejne pakiety danych przesyłane pomiędzy klientem a serwerem, pasujące do zapamiętanej sesji, będą przepuszczane. Istotne jest tutaj to, że filtr przepuszcza zarówno ruch wchodzący (inboard), jak i wychodzący (outboard) w ramach zapamiętanej sesji. Dobry filtr będzie przepuszczał nie tylko pakiety TCP, ale również ICMP lub inne pakiety, związane z daną sesją. IPFilter może działać zarówno jako filtr bezstanowy, jak i filtr z pamięcią stanów. Wszystko zależy od tego, jak zostaną sformułowane konkretne reguły w jego konfiguracji. Użycie opcji flags S oraz keep state w regule powoduje, że będzie dla niej używana tablica stanów. W IPFilter opcja keep state może być także użyta w odniesieniu do protokołów UDP i ICMP. Dla nich jednak nie ma tak dobrze określonego pojęcia sesji, jak w przypadku TCP, więc stanowość filtra ma także uproszczony sens i ogranicza się do przepuszczania pakietów będących odpowiedziami serwera. Dzięki stanowemu filtrowaniu pakietów nie dopuścimy między innymi do podszycia się pod inne połączenie. Dzięki wykorzystaniu tablicy stanów filtr pakietów wie, że dane przechodzące w konkretnej sesji są już częścią istniejącego połączenia. W filtrowaniu bezstanowym filtr "tylko zakłada", że dany pakiet przesyłany podczas sesji należy do istniejącego połączenia. W filtrowaniu stanowym mamy również możliwość skanowania portów, co zwiększa szanse na wykrycie ataków i zablokowanie ich.

Podsumowując filtrowanie pakietów z pamięcią stanów, sprawdza każdy datagram przechodzący przez filtr, wchodzący bądź wychodzący z hosta do

sieci. Każdy pakiet, który przeszedł przez filtr, zostanie zapisany np. z datą i stanem połączenia pomiędzy serwerem a klientem, dzięki temu sprawdzanie będzie dość restrykcyjne, co zwiększy nasze bezpieczeństwo (por. [5]).

1.4 Policy routing

Przy połączeniach pomiędzy hostami w sieci lokalnej nie potrzebujemy skomplikowanej konfiguracji lub urządzeń innych niż przełączniki. Drobne problemy zaczną się, gdy będziemy próbować przesłać dane do hosta znajdującego się w innej sieci (zewnętrznej). Do tego połączenia będziemy potrzebować węzła pośredniego np. routera. Proces przesyłania danych pomiędzy dwoma hostami w różnych sieciach określa się trasowaniem (routing). Ideą routingu jest znalezienie odpowiedniej trasy, która połączy dwie lub więcej sieci, tak by hosty znajdujące się w każdej sieci mogły się między sobą komunikować. Trasowanie najczęściej przekazywane jest w formie tablic routingu. W tablicach tych określane są miejsca docelowe, do jakich pakiet danych ma dotrzeć. Tablice mogą być zbudowane przez administratora sieci lub przez któryś protokół routingu.

W skrócie, działanie routingu można przedstawić tak, że dane wchodzące do np. routera są sprawdzane, gdzie jest ich adres docelowy. Następnie router na podstawie tego adresu sprawdza tablice i podejmuje decyzję dokąd wysłać dany pakiet.

Policy routing (polityka routingu) to technika, która w odróżnieniu od zwykłego trasowania oprócz adresu docelowego bierze pod uwagę również między innymi: adres źródłowy, porty, a czasem zawartość pakietu.

W przypadku prostych sieci gdzie konkretny pakiet musi dotrzeć do jakiegoś konkretnego miejsca, wystarcza zwykły routing IPv4, który określa trasę dzięki adresowi docelowemu. W przypadku sieci, gdzie połączeń jest wiele, a nie każde połączenie pomiędzy użytkownikami jest potrzebne, sytuacja się zmienia. Pakiet danych nie może zostać przechwycony lub skierowany po innej trasie, by trafił w inne miejsce. Są również przypadki, gdy na przykład wykorzystujemy protokół RIP i mamy dwie trasy, wtedy pakiety będą trasowane tylko po jednej, co jest nieopłacalne. Z tego względu powstało policy routing. Jednym z pionierów powstania policy routing był Quality of Service (QoS). Są to wymagania, jakie muszą być spełnione dla uzyskania odpowiedniej jakości użytkowej danej usługi. W sieciach komputerowych QoS nie jest dosłowne związane z jakością usługi. Quality of Service w tym przypadku określa między innymi:

- sterowanie przepustowością,
- równy dostęp do zasobów,
- nadawanie priorytetu pakietom danych,
- kontrolę i zarządzanie opóźnieniami,

- unikanie przeciążeń.

W sieciach komputerowych jednym z głównych znaczników jakości była szybkość przepływu danych pomiędzy jedną a drugą siecią, za co przede wszystkim odpowiadało trasowanie i kolejkowanie pakietów danych zależnych od ToS (Type of Service), który jest powiązany z QoS. Policy routing oparty na QoS umożliwia dodanie reguł, które pozwolą nam uzyskać najlepsze trasy pakietu danych oraz uzyskać gwarancje przepustowości. Można stwierdzić, że QoS jest integralną częścią policy routing. QoS i technologie powstałe na podstawie QoS zapewniają nadawanie priorytetu pakietom danych, poprzez klasyfikacje i kolejkowanie, a następnie przesłany po odpowiedniej trasie – tak jak w source routing.

Termin policy routing możemy rozważyć w dwóch aspektach. Pierwszy z nich mówi nam o zasadach i normach, które są wymagane, aby spełnić politykę jakości. Natomiast drugi aspekt pokazuje praktyczne zaimplementowanie polityki routingu w przesyle danych w sieci.

Polityka jakości sieci w policy routing nie jest czystym policy routing. Ideą policy routing jest to, że trasa, którą ma przebyć pakiet danych, jest oparta na informacji znajdującej się w jednej lub w każdej części pakietu danych. Ma to do siebie, że informacje co do trasy są w całym pakiecie, a nie tylko w nagłówku. Głównymi przesłankami wprowadzającymi tę technologię, było zapewnienie jak największego bezpieczeństwa przepływu danych. W czasach kiedy Internet rozrósł się globalnie, sam przesył informacji oparty o miejsce docelowe zapewniał dość mierne bezpieczeństwo. Dlatego początkowo bazując na QoS, wprowadzono zestaw reguł, które opisywały i zakazywały różnych czynności w operacji trasowania sieci. Podstawowym urządzeniem spełniającym policy routing jest każdy odpowiednio skonfigurowany router. Policy routing jest wykorzystywane między innymi w sytuacjach, kiedy mamy kilka sieci wewnętrznych i każda odpowiada za coś innego (w moim przypadku biuro i serwis) oraz posiadamy połączenie do sieci zewnętrznej z wykorzystaniem np. dwóch ISP (Internet Serice Provider). Za cały proces połączenia pomiędzy wszystkimi segmentami odpowiada router rdzeniowy (który może być również komputerem). Policy routing (odpowiednio skonfigurowane) w tym przypadku będzie odpowiadało za to, aby wiadomość wysłana np. z biura nie dotarła do serwisu i nie przez innego niż docelowego dostawcę Internetu. Zastosowanie policy routing w zaporze sieciowej opartej o IPFilter zwiększy znacznie bezpieczeństwo, zniweluje ruch z zewnątrz sieci oraz zwiększy prędkość przesyłu danych (por. [2]).

1.5 Source routing

Jednym ze sposobów routingu, będący praktycznie szczególnym przypadkiem szerszego pojęcia policy routing, jest tak zwany routing źródłowy (source routing). Jak nazwa wskazuje, główną rolę odgrywa tutaj informacja w nagłówku

IP o adresie źródłowym oraz w nagłówku TCP o porcie źródłowym, które zostały podane przy wysyłaniu pakietu danych przez urządzenie końcowe. W source routing droga, którą ma przebyć pakiet, jest już zdeterminowana przez hosta wysyłającego dane. Gdy pakiet danych po wysłaniu trafi do routera, decyzja o przekazaniu pakietu dalej podejmowana jest na podstawie o adres i port źródłowy. Router przekazuje taki pakiet danych do odpowiedniego interfejsu, który prześle pakiet dalej. W odróżnieniu od zwykłego trasowania, w source routing można powiedzieć, że urządzenia końcowe uczestniczą w całym procesie.

Jednym z założeń w source routing jest to, że router zna wszystkie możliwe połączenia pomiędzy hostami, innymi słowy zna całą topologię danej sieci. W source routing na podstawie adresu źródłowego wymuszona jest inna niż domyślna trasa dla pakietów.

Rozdział 2

Sieci VPN

2.1 Idea i zastosowanie

W dzisiejszych czasach urządzenia takie jak komputery, smartfony itp. pozbawione dostępu do sieci, praktycznie są bezużyteczne. Do sieci Internet podłączamy telewizor, radio, sprzęt hi-fi, ale także lodówkę i pralkę. Sieć przewodowa i bezprzewodowa jest wszechobecna. Urządzenia podłączone do sieci oraz same sieci można klasyfikować na różne sposoby. Nas będzie interesował podział na sieci publiczne i prywatne.

Sieć publiczna jest dużym zbiorem niepowiązanych ze sobą użytkowników, którzy w większy lub mniejszy sposób prowadzą swobodną wymianę informacji między sobą. Użytkownicy w sieci publicznej mogą mieć jakieś powiązania ze sobą, ale mogą także nie wiedzieć o istnieniu znacznej części innych użytkowników.

Sieć prywatna składa się z użytkowników posiadających swoje komputery np. w jakiejś organizacji, która udostępnia informacje tylko w swoim zasięgu. Dla sieci prywatnych utrwaliła się nazwa LAN, czyli Local Area Network. Komunikacja pomiędzy siecią prywatną a publiczną odbywa się poprzez router. Właściciel sieci prywatnej po swojej stronie uruchamia firewall (ścianę ogniową), który ma uniemożliwić użytkownikom z sieci publicznej dostęp do zasobów sieci prywatnej, bądź uniemożliwić użytkownikom sieci prywatnej dostęp do sieci publicznej.

Ideą VPN jest zatarcie granicy pomiędzy siecią publiczną a prywatną. VPN jest sposobem na stworzenie sieci prywatnej w sieci publicznej takiej jak np. Internet, bez wykorzystywania żadnych połączeń fizycznych. Bezpieczeństwo i niewidoczność dla użytkowników niebędących w tej sieci wirtualnej zapewnia szyfrowanie, uwierzytelnianie, tunelowanie pakietów oraz firewall'e. VPN pozwala np. skonsolidować połączenie internetowe i WAN w jednym routerze i na jednej linii co pozwoli oszczędzić na sprzęcie i infrastrukturze telekomunikacyjnej. Wirtualna sieć prywatna ma również zastosowanie między innymi na łączenie odległych od siebie oddziałów firmy oraz zdalny dostęp pracowników

do firmy (por. [4]).

2.2 Zasada działania

VPN jest to sposób zasymulowania sieci prywatnej poprzez pośrednictwo sieci publicznej. Sieć ta jest nazywana wirtualną, ponieważ jest zależna od połączeń wirtualnych tj. tymczasowych połączeń, których fizycznie nie ma, lecz są złożone z trasowanych pakietów na różnych komputerach w sieci Internet na zasadzie sieci typu ad-hoc. Bezpieczne połączenie jest tworzone poprzez połączenie dwóch komputerów, komputera i sieci, pomiędzy dwoma sieciami. Jak zostało wspominanie wyżej, VPN zapewnia bezpieczeństwo poprzez:

- uwierzytelnianie,
- szyfrowanie,
- tunelowanie.

2.2.1 Uwierzytelnianie

Uwierzytelnianie jest niezbędne do prawidłowego działania VPN, gdyż zapewnia ono pewność komunikacji z właściwym użytkownikiem, bądź hostem. Duża część systemów uwierzytelniania sieci wirtualnych jest oparta na kluczu wspólnym. Algorytm skrótu (hashing algorithm) uruchamia klucze i generuje również wartość skrótu (hash value). Strona, z którą się komunikujemy, posiada własne klucze, które wygenerują wartość skrótu i porównają ją z tą, która została wysłana. Wartość skrótu wysłana przez Internet jest nic nieznaczącą informacją dla postronnego obserwatora. Więc żadne urządzenie typu sniffer nie mogłoby przejąć hasła. Przykładem tego rodzaju uwierzytelniania jest CHAP (Challenge Handshake Authentication Protocol) .

2.2.2 Szyfrowanie

W technologii sieci wirtualnych istnieją dwie dość popularne techniki szyfrowania:

- szyfrowanie klucza prywatnego,
- szyfrowanie klucza publicznego.

W szyfrowaniu tajnym mamy dane wspólne hasło, bądź hasło znane wszystkim, którzy mają otrzymać dostęp do zaszyfrowanych informacji. Przykładem tej metody szyfrowania jest standard DES stosowany w UNIX. Jednym z problemów tego szyfrowania jest to, że każdy, kto ma uzyskać dostęp do danych, musi znać tajny klucz. Jest to nieco uciążliwe przy dużej ilości osób, ponieważ

w przypadku odebrania dostępu jednej osobie musimy zmieniać cały klucz od początku i informować resztę osób, które mają prawo dostępu.

W przypadku metody szyfrowania klucza publicznego potrzebne są nam dwa klucze publiczny i prywatny. Nasz klucz publiczny jest publikowany dla wszystkich członków, a prywatny znamy tylko my. Kiedy chcemy wysłać poufne dane, szyfrujemy je za pomocą naszego klucza prywatnego i klucza publicznego osób, którym chcemy je przekazać. W momencie otrzymania tych danych zostają one odszyfrowane za pomocą naszego klucza publicznego i klucza prywatnego odbiorcy. Szyfrowanie tą metodą jest dużo wolniejsze niż szyfrowanie metodą klucza prywatnego.

2.2.3 Tunelowanie

Tunelowanie sieci wirtualnych chroni nasze połączenie przed nieuprawnionym dostępem osób z zewnątrz. Jest to swego rodzaju tunel występujący w sieci publicznej i łączący między sobą dwie stacje robocze. W VPN dzięki tunelowaniu możemy przesyłać wiele protokołów, a nie tylko jeden tak jak w zwykłym tunelowaniu.

2.3 Protokoły wykorzystywane w VPN

2.3.1 PPTP

Jest to jedna z metod wspierania sieci VPN opracowana przez Microsoft. Ze względu, że jest to dość archaiczny protokół, znane są poważne problemy dotyczące bezpieczeństwa podczas korzystania z jego pomocy. Dokumentacja PPTP jako opcje zabezpieczeń wskazuje wyłącznie tunelowanie, nie ma w niej żadnej wzmianki o uwierzytelnianiu bądź szyfrowaniu. PPTP opiera swoje działanie głównie na używaniu protokołu TCP. Protokół ten połączony z GRE (Generic Routing Encapsulation) tworzył swego rodzaju VPN. PPTP wspiera jedynie tunel pomiędzy serwerem a klientem.

2.3.2 GRE

Generic Routing Encapsulation jest kolejnym protokołem tunelowania, który został opracowany przez firmę CISCO. GRE wspiera między innymi pracę protokołu PPTP. Ogólne działanie tego protokołu polega na zapakowaniu pakietów wewnętrznych do zewnętrznego pakietu IP, który będzie widoczny. Tunele stworzone przez GRE przetransportują ten pakiet z początku do końca tunelu. Routery które będą ustawione na trasie, będą widziały tylko ten zewnętrzny pakiet (zawartość wewnętrzna będzie niewidoczna). Kiedy punkt końcowy zostanie osiągnięty, zewnętrzny pakiet zostanie usunięty, a dane zostaną przesłane do miejsca docelowego. GRE zapewnia większe bezpieczeń-

stwo dzięki dodatkowemu polu klucza szyfrującego. Powoduje to dodatkowe uwierzytelnienie podczas tunelowania.

2.3.3 L2TP

Jest to kolejna z metod wspierania VPN. Protokół ten powstał z połączenia dwóch istniejących już protokołów PPTP i L2F łącząc najlepsze ich cechy. Ponieważ L2TP nie zapewnia poufności i silnego uwierzytelniania to, aby zwiększyć bezpieczeństwo L2TP, zalecane jest użycie protokołu szyfrującego IPSec. L2TP opiera swoje działanie na protokole UDP. L2TP w odróżnieniu od PPTP nie musi zawsze działać na IP, lecz aby działał, może zastosować szereg różnych protokołów. L2TP działający razem z protokołem IPsec, jest o wiele bezpieczniejszy niż PPTP.

2.3.4 IPsec

Jest to protokół bezpieczeństwa zawierający w sobie metody szyfrowania oraz uwierzytelniania i również jest jednym z elementów sieci wirtualnych. Protokół ten służy głównie to budowy sieci VPN w Internecie. Ze względu na to, że IPsec działa w trzeciej warstwie modelu OSI, czyli w warstwie sieci (podobnie jak protokół IP) potrafi zaszyfrować cały pakiet. Działa to dzięki dwóm mechanizmom AH i ESP. AH zapobiega manipulowaniu pakietem, ESP zapewnia, że informacje w pakiecie są szyfrowane i nie można ich odczytać. W VPN wykorzystujących IPSec możemy zastosować jeden z dwóch trybów. Tryb tunelowy służy przede wszystkim do komunikacji pomiędzy routerami, a tryb transportowy ma swoje zastosowanie w połączeniu VPN pomiędzy klientami.

2.3.5 OpenVPN

OpenVPN jest protokołem VPN i zarówno oprogramowaniem wykorzystującym techniki zabezpieczania połączeń point-to-point i site-to-site. Jest jednym z najbardziej popularnych pakietów oprogramowania do konfiguracji VPN. Protokół ten jest odpowiedzialny za obsługę połączenia w technologii klient-serwer. Obsługuje on tunelowanie, szyfrowanie oraz uwierzytelnianie. Do przesyłania danych OpenVPN może używać zarówno UDP i TCP, choć stosunkowo lepiej działa z UDP. OpenVPN nie korzysta z PPTP, L2TP, IPsec, lecz posiada własną warstwę szyfrowania opartą na SSL i TLS.

2.3.6 IKE

Internet Key Exchange jest to kolejny protokół wykorzystywany w sieciach VPN. Wykorzystuje on technologię tunelowania i jest oparty na IPSec. IKE tworzy klucz, który szyfruje i rozszyfrowuje dane, przesyłane za pomocą pakietów IP. Protokół ten działa w dwóch fazach. W fazie pierwszej przy wyko-

rzystaniu algorytmu Diffie-Hellmana'a tworzy klucz szyfrujący przesyłany za pomocą wcześniej utworzonego tunelu. Klucz ten jest wykorzystywany w fazie drugiej. W fazie drugiej jest wykorzystywany tunel utworzony w fazie pierwszej. W fazie tej powstaną minimum dwa skojarzenia bezpieczeństwa (przychodzące, wychodzące). Możemy wyróżnić dwie wersje protokołu: IKEv1 (opracowany w 1998 roku) oraz IKEv2 (opracowany w 2005 roku). Wersja druga wprowadziła między innymi ulepszenie translacji adresów NAT oraz obsługę połączeń przez użytkowników mobilnych.

2.3.7 SSTP

Secure Socet Tunneling Protocol to protokół tunelowania opracowany przez firmę Microsoft. Protokół ten zapewnia transport przez kanał SSL. Dzięki temu, że transport jest zapewniony przez SSL i również temu, że ruch przechodzi przez port TCP 443. SSTP może ominąć zabezpieczenia typu firewall, NAT i niektóre serwery proxy. Nazwa tego protokołu może być myląca, ponieważ SSTP zasadniczo nie obsługuje tuneli pomiędzy sieciami VPN. Kolejną różnicą w zabezpieczeniach sieci VPN dzięki temu protokołowi, jest to, że uwierzytelnianie dotyczy tylko użytkowników. SSTP został wymyślony raczej do łączenia się przez zdalnych, pojedynczych użytkowników do sieci niż połączeń typu site-to-site.

	PPTP	L2TP	IPsec	OpenVPN	SSTP	IKE
Wspierany system	Windows, Unix, Linux, iOS, Android, Mac OS X	Windows, Unix, Linux, iOS, Android, Mac OS X	Windows, Unix, Linux, iOS, Android, Mac OS X	Windows, Unix, Linux, iOS, Android, Mac OS X	Windows, Unix, Linux, BSD	Windows, Unix, Linux, iOS, Android, Mac OS X
Szyfrowanie	szyfrowanie podstawowe do 128bit	mocne szyfrowanie do 128bit	mocne szyfrowanie do 128bit	bardzo mocne szyfrowanie do 256bit	bardzo mocne szyfrowanie do 256bit	bardzo mocne szyfrowanie do 256bit
Szybkość	bardzo szybki	wymaga więcej procesora	wymaga więcej procesora	bardzo duża prędkość	bardzo duża prędkość	bardzo duża prędkość
Używane porty	TCP 1723	UDP 1701, UDP 500, UDP 4500	UDP 1701, UDP 500, UDP 4500	TCP 993, TCP 443	TCP 443	
Urządzenia	komputery, smartfony	komputery, smartfony	komputery, smartfony	komputery	komputery	komputery

Tabela 2.1: Porównanie protokołów VPN.

Rozdział 3

IPFilter

Ten rozdział opracowano wspólnie z Albertem Denkiewiczem, który pisze pracę [9] poświęconą również oprogramowaniu IPFilter.

3.1 Historia

IPFilter to oprogramowanie umożliwiające filtrowanie pakietów (ang. packet filtering), a dzięki temu, na konstruowanie zapór sieciowych (ang. firewall) nie tylko na platformie Solaris, ale także na FreeBSD, NetBSD, OpenBSD, AIX, HP-UX, IRIX, Linux oraz wielu innych systemach Unixowych. Wspiera protokoły IPv4 oraz IPv6. Jest filtrem pakietów z pamięcią stanów.

Autorem jest Darren Reed. Projekt od początku funkcjonował jako open source i był dość ściśle związany z platformą Solaris. Niestety został zaniechany przed kilku laty. Główną przyczyną było wykupienie Sun Microsystems przez Oracle i zamknięcie projektu OpenSolaris.

Na platformie Solaris, do wersji 9, nie było zintegrowanego oprogramowania do filtrowania pakietów. Jedyną alternatywą był IPFilter, ale należało go samodzielnie skompilować i zainstalować. W wersji 9 pojawił się firewall pod nazwą SunScreen, zaimplementowany przez inżynierów z Sun Microsystems. Architektura SunScreen przypominała współczesne IPTables z Linuxa, choć nie tak rozbudowane. Był to moduł jądra systemu z interaktywnym interfejsem do wprowadzania reguł. Przy bardziej rozbudowanych konfiguracjach każda drobna modyfikacja reguły, usunięcie reguły lub dodanie nowej urastało do bardzo poważnej operacji wymagającej dobrej znajomości interfejsu. Trudna była także diagnostyka. Wielu administratorów nie instalowało w ogóle SunScreen, tylko kompilowało i instalowało IPFilter.

W kolejnej, 10 wersji Solaris pojawił się IPFilter. Były to ciekawe czasy (2003-2005), gdy grupa użytkowników Solaris odwiodła Sun Microsystems od rezygnacji z platformy x86, a firma zaczęła dużo agresywniej inwestować w swój system operacyjny. W tej wersji pojawiło się całe mnóstwo nowatorskich rozwiązań: ZFS, D-Trace, Branded Zones, SMF, FM, Trusted Extensions, JDS

by wymienić tylko najciekawsze, które dopiero teraz trafiają na inne systemy, dzięki udostępnieniu kodu źródłowego jako OpenSolaris w 2008 roku. Wtedy Darren Reed podpisał kontrakt z Sun Microsystems. Pojawiły się konkretne motywacje finansowe, IPFilter oficjalnie stał się integralną częścią systemu Solaris i zyskał trwałe wsparcie ze strony świetnych inżynierów z Sun Microsystems.

IPFilter powstał na Solaris, a w zasadzie na SunOS, bo wówczas tak nazywał się system tworzony przez Sun Microsystems. Pierwsza wersja 1.0 została wypuszczona 22 kwietnia 1993 roku.

W Solaris 10 znalazła się wersja 4.1.9 w 2004. To jest ostatnia wersja IPFilter, jaka została zintegrowana z systemem Solaris. Przez wiele lat istnienia i wpierania Solaris 10 była ona wielokrotnie poprawiana i dostosowywana do nowszych wersji jądra systemu. IPFilter ewoluował dwutorowo: firma Sun Microsystem podtrzymywała wersję 4.1.9, natomiast niezależnie Darren Reed wypuszczał nowsze, udoskonalone wersje, często zawierające poprawki firmy Sun. Spadkobiercy Solaris 10, czyli Oracle Solaris 11, Illumos, OpenIndiana, SmartOS i pozostałe dalej używają wersji 4.1.9.

Wersja 5.1, która ujrzała światło dzienne 9 maja 2010 roku, była całkowitym przepisaniem kodu z licznymi zmianami, aczkolwiek wsteczna kompatybilność konfiguracji została zachowana.

Ostatnia wersja IPFilter, jaką można znaleźć, jest 5.1.2. Wyszła 22 czerwca 2012 roku i została włączona do FreeBSD. Jedynie w źródłach tego systemu można tę wersję znaleźć.

W tej chwili pierwotna strona IPFilter i związana z nim lista dyskusyjna nie działają. Utworzona przez autora IPFilter strona na `sourceforge.net` zawiera źródła wersji 5.0.5 oraz 4.1.32. Jej ostatnia aktualizacja była wykonana w 2013 roku. Na znajdującym się tam forum czasem ktoś coś zamieści, ale są to pojedyncze posty pozostające bez odzewu. Darren Reed nie odpisuje na wysłane wiadomości e-mail. Aktywni zostali chyba wyłącznie twórcy FreeBSD konserwujący wersję 5.1.2. Tak obecnie wygląda sytuacja z IPFilter.

Narzuca się w tym momencie pytanie, po co inwestować w IPFilter i wiązać z tym oprogramowaniem swoje rozwiązania? Odpowiedź jest prosta: tak długo, jak chcemy używać Solaris, jesteśmy skazani na IPFilter, bo to jedyny sensowny firewall dla tej platformy.

3.2 Architektura

IPFilter działa zgodnie z ogólną teorią filtra pakietów warstwy 3 i 4 modelu OSI i nie różni się zbytnio od innych implementacji jak IPTables, IPFW, czy PF. Różnice polegają na podejściu do sposobu konfiguracji i oczywiście składni reguł oraz na wewnętrznej realizacji filtra.

Z poziomu userland możemy manipulować stosem TCP/IP, ale w dość ograniczony sposób. Programy narzędziowe takie jak `ifconfig`, `route`, czy

`ndd` mogą w istotny sposób wpływać na przepływ pakietów przez system, ale filtrowanie pakietów i translacja adresów wymagają dostępu do pakietów TCP/IP w ich surowej postaci dostępnej jedynie na poziomie zarezerwowanym dla jądra systemu. Dlatego najważniejszą częścią IPFilter jest moduł jądra, zwany `ipf`, który odrabia całą czarną robotę. IPFilter można skompilować razem z jądrem, jako jego integralną część, ale robi się to w bardzo wyjątkowych sytuacjach, na ogół, na platformach z jądrem monolitycznym.

Firewall powinien mieć nieograniczony dostęp do pakietów TCP/IP i przetwarzać je możliwie sprawnie, nie wpływając na ogólną przepustowość sieci w istotny sposób. To jest drugi powód, aby umieścić go w jądrze systemu.

Poniżej zamieszczony jest fragment listy załadowanych modułów jądra w Solaris 10 uzyskanej poleceniem `modinfo`. Jak widać, pod ID 161 znajduje się moduł `ipf`.

```
 Id         Loadaddr   Size Info Rev Module Name
154 ffffffffefd2a000  33790   65   1  e1000g (Intel PRO/1000 Ethernet 5.2.27)
155 ffffffffef26f898    dc0    -   1  mac_ether (Ethernet MAC plugin 1.1)
157 ffffffffefa9a000   1d48    6   1  openeepr (OPENPROM/NVRAM Driver v1.20)
158 ffffffffefd5e000  a0930    8   1  zfs (ZFS filesystem version 15)
158 ffffffffefd5e000  a0930  181   1  zfs (ZFS storage pool)
159 ffffffffefcd4300    f80   24   1  pts (Slave Stream Pseudo Terminal dr)
160 ffffffffefcf2000  1d2e0  202   1  mpt_sas (MPTSAS HBA Driver 00.00.00.16)
161 ffffffffefdf6000  9b4d0  165   1  ipf (IP Filter: v5.1.2)
162 ffffffffef228748    b28   21   1  log (streams log driver)
163 ffffffffefa9c000   1b48  154   1  cryptoadm (Cryptographic Administrative In)
165 ffffffffefa9e000   16d0  185   1  power (power button driver v1.17)
166 ffffffffefc2c220    f30   90   1  kstat (kernel statistics driver 1.26)
167 ffffffffefd13000   5ab0   88   1  devinfo (DEVINFO Driver 1.71)
169 ffffffffefcab6b8    b28  104   1  objmgr (Object Manager 1.27)
170 ffffffffef9f3440    f38  113   1  xsvc (xserver svc)
```

W IPFilter można wyróżnić 3 następujące podsystemy:

1. podsystem filtra pakietów,
2. podsystem translacji adresów,
3. podsystem puli adresów.

Filtr pakietów analizuje pakiety TCP/IP porównując je do reguł umieszczonych na wewnętrznej liście. Jeśli pakiet pasuje do danej reguły, wykonywana jest jedna z dwóch możliwych akcji: pakiet jest przepuszczany lub blokowany. Dodatkowo może być przekazana informacja do monitora, przepuszczony pakiet może zostać zduplikowany albo przekierowany na inny interfejs niż ten, wyznaczony na podstawie tablicy routingu. Ostatnia akcja wykorzystywana jest do realizacji policy routing.

Jak sugeruje nazwa, zadaniem podsystemu translacji adresów jest zmiana adresu IP i portu TCP w pakiecie zgodnie z regułami translacji umieszczonymi na wewnętrznej liście. Są tutaj dwie klasy reguł: jedna odpowiada SNAT, druga DNAT.

Podsystem puli adresów pozwala na gromadzenie wielu adresów sieciowych pod wybraną nazwą. Jeśli na przykład mamy 5 adresów IP, z który blokujemy wszelki ruch, możemy je umieścić w jednej puli i zamiast pisać 5 niemal identycznych reguł różniących się jedynie adresem źródłowym pakietu, wystarczy napisać jedną regułę, używając nazwy puli w miejscu adresu.

Poza modułem `ipf` w skład oprogramowania IPFilter w wersji 5.1.2 (podobnie dla wersji wcześniejszych od 4.0) wchodzą następujące programy pozwalające kontrolować jego zachowanie:

`ipf` Interfejs użytkownika do podsystemu filtra pakietów. Modyfikuje i zarządza wewnętrzną listą reguł w jądrze systemu. Czyta plik konfiguracyjny, parsuje znajdujące się tam reguły i umieszcza je na wewnętrznej liście reguł. Przy jego pomocy można także usuwać reguły, wyłączyć i włączyć filtrowanie, podejrzeć lub zmodyfikować wewnętrzne zmienne sterujące filtrowaniem pakietów oraz translacją adresów.

`ipfs` Zapisuje i odtwarza tablicę stanów filtra oraz NAT.

`ipfstat` Raportuje statystyki filtra pakietów.

`ipftest` Testuje reguły filtra bez konieczności ładowania ich do wewnętrznej listy w jądrze systemu. Pracuje na plikach utworzonych przez różne sniffery sieciowe jak `libpcap`, `snoop`, `tcpdump` i dla każdej reguły zwraca: `pass`, `block` lub `nomatch`.

`ipmon` Monitoruje pakiety przeznaczone do umieszczenia w logach. Przetwarza pakiety z surowej postaci do postaci czytelnej dla człowieka i wynik umieszcza w pliku lub w logu systemowym `syslog`.

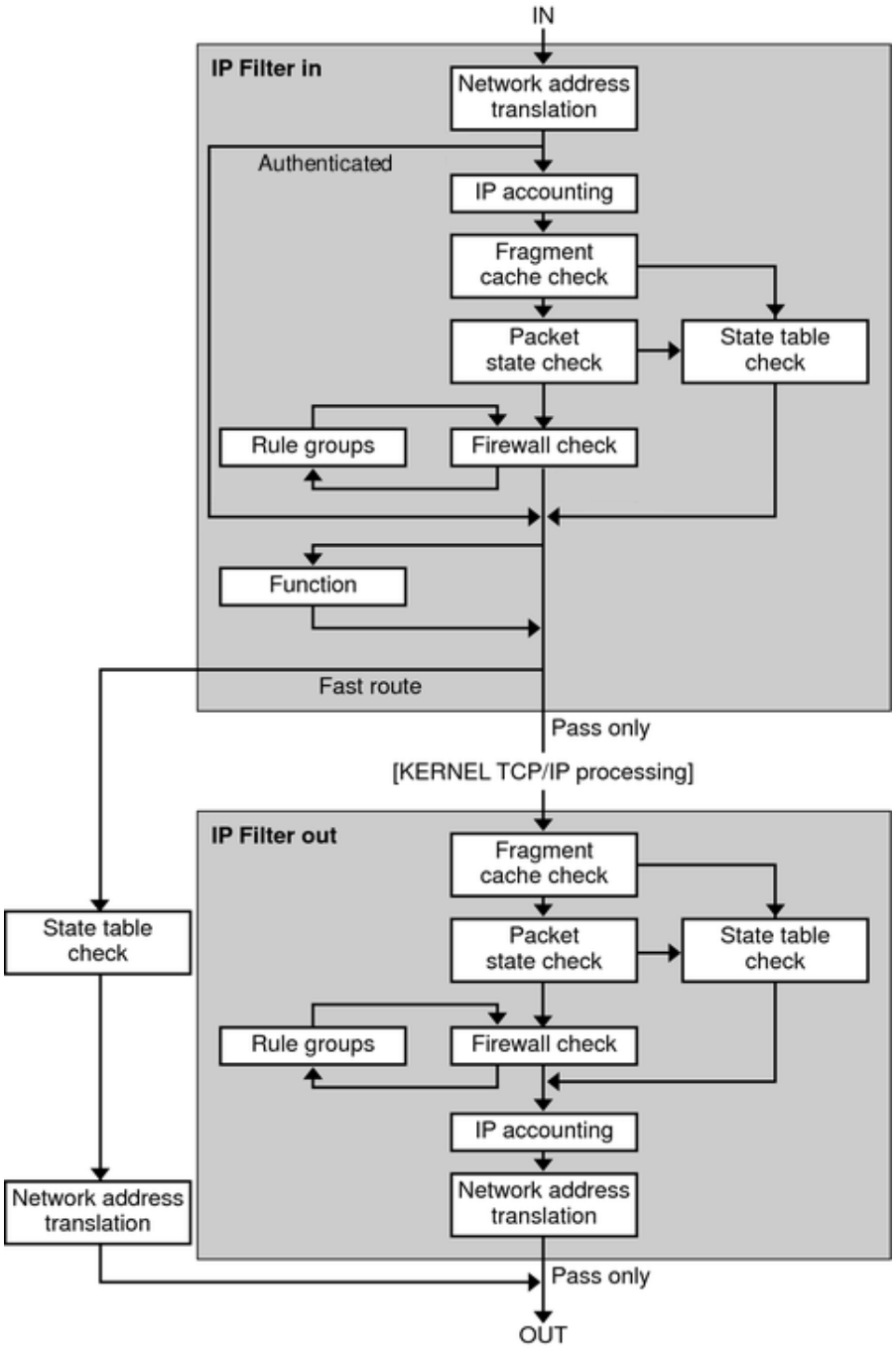
`ipnat` Interfejs użytkownika do podsystemu translacji adresów. Modyfikuje i zarządza wewnętrzną listą reguł translacji w jądrze systemu. Czyta plik konfiguracyjny, parsuje znajdujące się tam reguły translacji i umieszcza je na wewnętrznej liście reguł translacji. Przy jego pomocy można także usuwać reguły, obejrzeć aktualną listę reguł oraz statystyki NAT.

`ippool` Interfejs użytkownika do podsystemu pul adresów. Czyta plik konfiguracyjny i ładuje pule adresów do jądra systemu. Umożliwia także inspekcję pul w jądrze, dodawanie i usuwanie pojedynczych adresów.

`ipresend` Wysyła pakiety TCP/IP przechwycone wcześniej snifferem sieciowym. Pozwala testować i diagnozować sieć TCP/IP.

`ipsend` Generuje pakiet TCP/IP i wysyła go poprzez wskazany interfejs sieciowy. Pozwala testować i diagnozować sieć TCP/IP.

`iptest` Wysyła serię pakietów TCP/IP w celu testowania stosu TCP/IP, filtra pakietów lub NAT.



Rysunek 3.1: Diagram przepływu pakietów przez jądro systemu i IPFilter.

W przepływie pakietu TCP/IP przez system z IPFilter (rys. 3.1) wyróżniamy następujące, istotne etapy przetwarzania:

Network Address Translation Zamiana prywatnego, źródłowego adresu IP na adres publiczny albo mapowanie wielu prywatnych, źródłowych adresów na jeden adres publiczny w przypadku SNAT. W przypadku DNAT zamiana publicznego adresu docelowego na adres prywatny w sieci lokalnej.

IP accounting Gromadzenie danych statystycznych na potrzeby raportowania użycia filtra pakietów i translatora adresów.

Fragment cache check Jeśli pakiet w kolejce jest fragmentem, a poprzedni fragment został przepuszczony, to ten fragment jest również przepuszczany z pominięciem tablicy stanów i sprawdzania reguł.

Packet state check Jeśli reguła zawiera `keep state`, to wszystkie pakiety w określonej sesji są automatycznie przepuszczane albo blokowane, w zależności od tego, czy ta reguła przepuszcza, czy blokuje pakiety.

Firewall check Sprawdzenie czy dany pakiet ma zostać przepuszczony do dalszego przetwarzania w jądrze systemu lub do wyjścia przez interfejs do sieci.

Groups Grupy pozwalają pisać reguły w sposób przypominający drzewo.

Function Akcja jaka ma być podjęta. Możliwe akcje to przepuszczenie, zablokowanie lub wysłanie komunikatu ICMP.

Fast-route Specjalne reguły pozwalają przekazać pakiet bezpośrednio na podany interfejs wyjściowy z pominięciem stosu TCP/IP w jądrze systemu, który wyznacza trasę pakietu na podstawie tablicy routingu i powoduje pomniejszenie TTL pakietu.

Authenticated Pakiety, które zostały poddane autentykacji, przechodzą przez filtr tylko raz, aby uniknąć wielokrotnego przetwarzania.

3.3 Konfiguracja

Wdrożenie filtra pakietów wymaga precyzyjnego sformułowania polityki, określenia kto ma dostęp do jakich zasobów i usług. Na podstawie takiej polityki opracowuje się plan reguł filtra i translacji adresów. Ostatni etap to zapisanie planu reguł zgodnie ze składnią wdrażanego systemu, czy oprogramowania.

Domyślnie, do konfiguracji IPFiltra używane są trzy następujące pliki:

- `/etc/ipf/ipf.conf`,

- `/etc/ipf/ipnat.conf`,
- `/etc/ipf/ippool.conf`.

Każdy z nich odpowiada jednemu z trzech podsystemów: filtra, translatora adresów i puli adresów.

3.3.1 `ipf.conf`

W pliku `/etc/ipf/ipf.conf` znajdują się reguły opisujące zachowanie filtra pakietów. Plik ten jest czytany przez program `ipf`, który ładuje je do pamięci wewnętrznej modułu jądra. Jest to domyślna, standardowa lokalizacja tego pliku, ale można ją zmienić i podać pełną ścieżkę używając opcji `-f`. Źródłem reguł może być także standardowe wejście, gdy jako ścieżkę do pliku podamy znak `-`. Może to być użyteczne na przykład wtedy, gdy potrzebujemy usunąć wszystkie reguły wejściowe:

```
ipfstat -i | ipf -rf -
```

Program `ipf` dopisuje reguły do listy wewnętrznej kolejno, tak jak zostały umieszczone w pliku konfiguracyjnym. Dopisanie sekwencji `@<num>` przed regułą powoduje wstawienie tej reguły na pozycji `<num>` aktualnej listy wewnętrznej. Jest to szczególnie wygodne podczas testowania aktywnego zestawu reguł.

Reguły sprawdzane są w takiej kolejności, w jakiej znajdują się na liście wewnętrznej. Jeden pakiet może pasować do wielu reguł. Ostatnia z nich jest decydująca, chyba że zastosowano opcję `quick`.

Z punktu widzenia zapory sieciowej istotne są dwa rodzaje reguł: te, które blokują i odrzucają pakiety (reguły typu `block`) oraz te, które przepuszczają pakiety (reguły typu `pass`). Obok decyzji co zrobić z danym pakietem znajdują się deklaracje określające warunki podjęcia tej decyzji i sposobu jej egzekucji. Najprostsze, poprawne reguły, jakie można sformułować, mogłyby wyglądać następująco:

```
block in all
pass out all
```

ale w tej postaci są zupełnie bezużyteczne, chyba że chcemy całkowicie odciąć się od sieci.

Każda reguła filtra składa się z co najmniej trzech następujących komponentów:

1. słowo kluczowe decyzji (`pass`, `block`, itp.),
2. kierunek pakietu (`in` albo `out`),
3. wzorzec adresu lub słowo kluczowe `all`.

Długie linie w pliku konfiguracyjnym można łamać excplicite używając znaku \, albo implicite pisząc jedną regułę w kilku wierszach. Komentarze zaczynają się od znaku `#` i kończą znakiem końca linii.

Dalej opisywane są zasady jak konstruować poprawne reguły. Jakie są dobre i złe praktyki, co należy uznać za bezpieczne, a co nie, wykracza jednak poza zakres tego opisu. Ograniczyliśmy się także do opisu tylko tych konstrukcji i deklaracji, które wykorzystujemy dalej w przykładach i testach, pomijając wiele pozostałych. Gramatyka pliku `ipf.conf` znajduje się w dodatku A. Tekst opracowano na podstawie [3].

Słowa kluczowe filtra

Pierwsze słowo kluczowe każdej reguły określa, co zostanie zrobione, gdy pakiet zostanie dopasowany do tej reguły. Możliwe są następujące słowa kluczowe:

`pass` Pakiet zostaje oznaczony jako przepuszczony w kierunku jakim płynie.

`block` Pakiet zostaje oznaczony jako zablokowany. Zablokowane pakiety płynące w kierunku `in` nie pojawią się w stosie TCP/IP jądra systemu, natomiast pakiety płynące w kierunku `out` nigdy nie wyjdą poza interfejs i nie pojawią się w na kablu.

`log` W wyniku tej reguły tworzony jest rekord informacyjny dla programu `ipmon`, który zapisuje go w określonym dzienniku systemowym. Reguły tego typu nie wpływają na to, czy ostatecznie pakiet ma być przepuszczony, czy zablokowany.

`count` Reguły tego typu pozwalają administratorowi zliczać pakiety i ilość przesyłanych bajtów. Dla ruchu wchodzącego (inbound) reguły takie są aplikowane po translacji adresów i filtrze, natomiast dla ruchu wychodzącego (outbound) są aplikowane przed translacją adresów, zanim pakiet zostanie odrzucony. Dlatego reguły te nie powinny być traktowane jako rzetelny wskaźnik.

`auth` Pakiety dopasowane taką regułą są kolejkowane do przetworzenia w programie z warstwy użytkowej systemu. Program ten zwraca werdykt co dalej zrobić z pakiem, przepuścić, czy zablokować. Jeśli kolejka się zapełni, pakiety zostaną odrzucone.

`call` Reguły takie daję dostęp do dodatkowych funkcji wbudowanych w IP-Filter, które pozwalają podejmować bardziej skomplikowane akcje, aby podjąć ostateczną decyzję.

Dopasowanie interfejsu sieciowego

W systemach wyposażonych w kilka interfejsów sieciowych może być konieczne określenie, którego z tych interfejsów dotyczy dana reguła. Przyjęta polityka filtra pakietów może wymagać rozróżniania pakietów według interfejsu, na jakim się znajduje.

W niektórych maszynach obecność interfejsu sieciowego może być dynamiczna. Na przykład programowe interfejsy sieci komutowanych opartych o Point-to-Point Protocol pojawiają się w momencie nawiązania połączenia na porcie szeregowym. Aby umożliwić działanie filtra na takich systemach, interfejs występujący w regułach może nie być obecny w systemie. Trzeba uważać, bo może to prowadzić do cichych błędów, gdy nazwa interfejsu została przekręcona podczas wpisywania.

Poniższe reguły dotyczą interfejsów odpowiednio `e1000g0` oraz `e1000g1`:

```
block in on e1000g0 all
pass out on e1000g1 all
```

Dopasowanie adresu

Najprostszy i najbardziej podstawowy sposób dopasowania w regułach filtra jest określenie adresu IP oraz portu TCP albo UDP. W nagłówku warstwy drugiej modelu TCP/IP mamy dwa adresy IP: źródłowy (source) oraz docelowy (destination). Konstruując regułę, adres źródłowy podajemy, poprzedzając go słówkiem `from`, a docelowy słówkiem `to`.

Adresy IP podajemy zgodnie z notacją CIDR, gdzie znak `/` oddziela adres IP od liczby określającej długość maski w bitach. W ten sposób możemy określić całą podsieć. Pojedynczy host określamy, wpisując `/32` lub nie wpisując długości maski wcale. Słówko `any` oznacza dowolny adres IP. Na przykład:

```
block in on e1000g0 from 192.168.0.0/24 to any
pass out on e1000g1 from any to 172.16.0.10
```

Nie jest możliwe podanie zakresu adresów, który nie da się wyrazić za pomocą notacji CIDR, czyli nie jest podsiecią.

Ogólnie rzecz biorąc, zamiast adresów IP można używać adresów domenowych DNS. Należy jednak uważać, bo jeśli jednej nazwie odpowiada więcej niż jeden adres IP, to brany jest tylko pierwszy z nich. Natomiast w przypadku, gdy pobranie adresu z DNS trwa długo, albo jest blokowane przez inną część reguł, załadowanie i sprawdzenie reguły z adresem DNS może być opóźnione, jeśli w ogóle nie zakończy się błędem.

W sytuacji, gdy aplikujemy tę samą regułę do wielu adresów IP, zamiast tworzyć osobne reguły, można skonstruować jedną regułę, która zamiast konkretnego adresu IP używa tak zwanej puli adresów. Na przykład:

```
pass in on e1000g0 from pool/trusted to 172.16.0.10
```

Pule adresów tworzone są programem `ippool`.

W systemach o wielu interfejsach, albo gdy adresy interfejsów są dynamiczne (na przykład uzyskiwane są przez DHCP), wygodne jest używanie nazw interfejsów zamiast ich adresów. W regułach skojarzonych z konkretnym interfejsem, poprzez użycie słówka `on` albo `via`, zamiast adresu IP możemy użyć nazwy interfejsu. Na przykład, gdy podstawowy adres interfejsu `e1000g0` to `192.168.0.1`, a maska to `255.255.255.0`, wówczas następujące reguły są równoważne:

```
pass in on e1000g0 from any to 192.168.0.1/32
pass in on e1000g0 from any to e1000g0/0
```

Warto tu zwrócić uwagę na odstępstwo od zasady, że sufiks `/32` oznacza konkretnego hosta w notacji CIDR. Podanie takiego sufiksu i całkowite opuszczenie sufiksu po nazwie interfejsu nie daje spodziewanych rezultatów.

Dodatkowo słówko `netmasked` pomaga zastosować dynamicznie przydzielaną maskę, tak aby zaadresować całą podsieć:

```
pass in on e1000g0 from any to 192.168.1.0/24
pass in on e1000g0 from any to e1000g0/netmasked
```

W regułach, które nie są skojarzone z żadnym interfejsem, nazwę interfejsu podajemy w nawiasach okrągłych. Poniższe reguły są równoważne:

```
pass in from any to 192.168.0.1/32
pass in from any to (e1000g0)/0
pass in from any to (e1000g0)
```

Dopasowanie protokołu

Aby móc dopasować pakiet według portu TCP/UDP musimy najpierw określić protokół pakietu. Robi się to za pomocą słówka `proto`, po którym podajemy nazwę protokołu albo jego numer (z pliku `/etc/protocols`). Na przykład:

```
block in on e1000g0 proto tcp from 192.168.0.0/24 to any
pass out on e1000g1 proto udp from any to 172.16.0.10
pass in on bge0 proto icmp from any to 192.168.0.0/16
```

Dopasowanie portu TCP/UDP

Gdy określimy w regule protokół, możemy wskazać numer portu, jaki ma być dopasowany. Ponieważ numery portów używane są inaczej niż adresy IP, dopasowując port, możemy użyć jednej z następujących relacji logicznych w ich całkiem naturalnym sensie: `< x`, `<= x`, `> x`, `>= x`, `= x`, `!= x`, gdzie `x` to numer portu. Dodatkowo można stosować zakresy portów:

`x <> y` numer portu jest mniejszy niż `x` i większy niż `y`,

`x >< y` numer portu jest większy niż `x` i mniejszy niż `y` oraz

`x:y` numer portu jest większy lub równy `x` i mniejszy lub równy `y`.

Na przykład:

```
block in on e1000g0 proto tcp from any port >= 1024 to any port < 1024
pass in on e1000g1 proto tcp from any to 172.16.10 port = 22
block out proto udp from any to 10.1.1.1 port = 135
pass in proto udp from 1.1.1.1 port = 53 to 10.1.1.1 port = 53
pass in proto tcp from 127.0.0.0/8 to any port = 6000:6009
```

Rozszerzone dopasowanie

W nagłówkach pakietów TCP znajdują się flagi, które wskazują na to, czy dany pakiet to nawiązanie, kontynuacja lub zakończenie połączenia itp. W połączeniu z dopasowaniem według portów, dopasowanie według flag TCP daje możliwość konstruowania bardzo precyzyjnych reguł.

Najbardziej podstawowe flagi TCP to:

`S SYN` Ustawiana jest jako jedyna w pierwszym pakiecie inicjującym połączenie ze strony klienta. W odpowiedzi serwer odsyła pakiet a flagami `SYN` i `ACK` (tak zwany three-way handshake).

`A ACK` Ustawiana jest w pakiecie potwierdzającym otrzymanie danych.

`F FIN` Ustawiana, gdy jeden z uczestników połączenia kończy je.

`R RST` Ustawiana wyłącznie w pakiecie odpowiedzi na pakiet skierowany na port, który nie jest używany.

Gdy chcemy przepuścić wyłącznie pakiety inicjujące połączenie, możemy sformułować to, używając słówka `flags` w następujący sposób:

```
pass in on e1000g0 proto tcp from any to e1000g0/0 port = 80 flags S
```

Tak naprawdę, dopiero gdy zastosujemy pamięć stanów, filtrowanie na podstawie flag TCP pokaże swoją moc.

Filtrowanie na podstawie innych parametrów nie tylko nagłówka IP, możliwe jest nie tylko w protokołach TCP i UDP, ale także w ICMP. Słówko `icmp-type` pozwala precyzyjnie określać jakie komunikaty ICMP blokujemy lub przepuszczamy na podstawie typu ICMP. Na przykład żądanie echa to typ 8 (`echo`), natomiast odpowiedź to typ 0 (`echorep`). Poniżej wpuszczamy żądania echa i wypuszczamy odpowiedzi:

```
pass in on e1000g0 proto icmp from any to e1000g0/0 icmp-type echo
pass out on e1000g1 proto icmp from e1000g1/0 to any icmp-type echorep
```

Filtrowanie z pamięcią stanu

Filtrowanie z pamięcią stanu polega na tym, że IPFilter zapamiętuje pewne informacje z jednego bądź kilku pakietów, które widział i jest w stanie je później zaaplikować do pakietów, jakie pojawią się w przyszłości.

W różnych warstwach transportowych TCP/IP wygląda to inaczej. W TCP pierwszy pakiet inicjujący połączenie (z ustawioną flagą SYN) posiada dość informacji, aby na ich podstawie przepuścić kolejne pakiety z tego połączenia. IPFilter korzysta z portu TCP, flag, rozmiaru okienka oraz numerów sekwencyjnych, aby dopasować pakiety. W UDP wyłącznie numer portu może być wykorzystany. W ICMP typy komunikatów razem z identyfikatorem mogą być użyte. Dla pozostałych protokołów pozostaje jedynie adres IP i numer protokołu.

Pamięć stanu w regule włączamy za pomocą `keep state`. Pozwala to między innymi zredukować ilość reguł. Na przykład 4 reguły:

```
pass in on e1000g0 proto tcp from any to any port = 22
pass out on e1000g1 proto tcp from any to any port = 22
pass in on e1000g1 proto tcp from any port = 22 to any
pass out on e1000g0 proto tcp from any port = 22 to any
```

mogą być zastąpione jedną:

```
pass in on e1000g0 proto tcp from any to any port = 22 flags S keep state
```

Gdy używamy pamięci stanów dla protokołu TCP warto dodawać opcję `flags S`, aby tworzyć nowy stan wyłącznie podczas inicjacji połączenia, na samym jego początku, nie w środku połączenia, gdy mamy mniej informacji.

Policy routing

Zapory sieciowe z uwagi na swoje przeznaczenie, często lokowane są na styku kilku sieci o różnych parametrach. Często pożądane jest, aby przepływ pakietów był inny niż ten przewidziany tablicą routingu w jądrze systemu. Decyzje o zmianie trasy pakietu mogą być podejmowane na podstawie nie tylko adresu docelowego, co jest naturalne, ale także na podstawie adresu źródłowego albo portów. Dlatego też czasem policy routing zwane jest source based routing.

IPFilter wspiera policy routing i pozwala określić następny przeskok (next hop) w konstruowanych regułach. Przeskok deklaruje się podając nazwę interfejsu, który ma być użyty do wysłania pakietu oraz adres IP rutera bezpośrednio przyłączonego do danego interfejsu. Na przykład, gdy ruter przyłączony do interfejsu `e1000g1` ma adres `10.0.1.1` i chcemy, aby pakiety pochodzące z sieci `192.168.0.0/24`, pojawiające się na interfejsie `e1000g0`, wyszły do tego routera, to instruujemy IPFilter w następujący sposób:

```
pass out quick on e1000g0 to e1000g1:10.0.1.1 \
    from 192.168.0.0/24 to any keep state
```

Pakiet wychodzący z rutera można zduplikować, na przykład w celach diagnostycznych, używając słówka `dup-to`:

```
pass out quick on e1000g0 to e1000g1:10.0.1.1 dup-to e1000g2:172.16.0.1 \
    from 192.168.0.0/24 to any keep state
```

Rozważmy przykładową sytuację, gdy mamy dwóch dostawców usług sieciowych ISP. Router jednego dostawcy podłączony jest do interfejsu `e1000g0` i to jest trasa domyślna pakietów. Router drugiego podłączony jest do interfejsu `e1000g1`. Aby odpowiedzi na pakiety wchodzące przez `e1000g1` wychodziły przez `e1000g1` pomoże konstrukcja oparta o `reply-to`:

```
pass in on e1000g1 reply-to e1000g1:10.0.1.1 \
    proto tcp from any to 10.0.1.10/32 port = 22 flags S keep state
```

Odsyłanie informacji o błędzie

W sytuacji, gdy pakiet jest odrzucany przez reguły filtra, normalnie nie jest wysyłana żadna informacja zwrotna o tym fakcie. Z jednej strony nie chcemy generować dodatkowego ruchu w sieci, gdy blokujemy próby ataku. Z drugiej strony host próbujący połączyć się z nieużywanym portem TCP albo UDP, będzie ponawiał próby, sądząc że pakiet został zgubiony gdzieś po drodze.

W przypadku TCP można odesłać pakiet z flagą `RST`, aby uniknąć dalszych prób połączenia na dany port:

```
block return-rst in proto tcp from 10.0.0.0/8 to any
```

W przypadku pozostałych protokołów (także i TCP) można odesłać pakiet ICMP typu 3, czyli Destination Unreachable:

```
block return-icmp in proto tcp from 10.0.0.0/8 to any
```

Dodatkowo przy `return-icmp` można określić kod ICMP informujący dokładnie o rodzaju problemu. Może to być jeden z następujących kodów:

- `filter-prohib`,
- `net-prohib`,
- `host-unk`,
- `host-unr`,
- `net-unk`,
- `net-unr`,
- `port-unr`,

- `proto-unr`.

Ich nazwy są dość sugestywne. Który kod wybierzemy, zależy jaki efekt chcemy uzyskać i w jakim stopniu chcemy zdradzać istnienie zapory sieciowej na drodze do hosta docelowego.

```
block return-icmp(port-unr) in proto tcp from 10.0.0.0/8 to any
```

Powyżej wygenerowany pakiet ICMP jako adres źródłowy zawierać będzie adres interfejsu użytego do wysłania tego pakietu. Gdy chcemy ukryć istnienie zapory sieciowej i udawać, że odpowiada host docelowy, używamy następującej konstrukcji:

```
block return-icmp-as-des(port-unr) in proto tcp from 10.0.0.0/8 to any
```

Raportowanie

Są dwa sposoby raportowania informacji o przepływających przez IPFilter pakietach. Można użyć specjalnie do tego stworzonej reguły typy `log` albo do reguł pozostałych typów dodać specjalne słówko `log`. W ten drugi sposób można jednocześnie przepuścić lub zablokować pakiet i zapisać informację o tym w dzienniku:

```
pass in log quick proto tcp from any to any port = 22
```

Gdy stosujemy pamięć stanów, to do dziennika trafią informacje o wszystkich pakietach z danego połączenia. Może być tego trochę za dużo, a wystarczy informacja tylko o pierwszym pakiecie:

```
pass in log first quick proto tcp from any to any port = 22 \
    flags S keep state
```

Normalnie IPFilter zapisuje w dzienniku wyłącznie informacje z nagłówka pakietu. Gdy potrzebujemy zapisać dodatkowe informacje zawierające przesyłane dane, stosujemy słówko `body`:

```
block in log body proto icmp all
```

Pozwala to dodatkowo zapisać w dzienniku do 128 bajtów danych przesyłanych w pakiecie.

3.3.2 ipnat.conf

Plik /etc/ipf/ipnat.conf zawiera reguły opisujące zachowanie podsystemu translacji adresów. Do listy wewnętrznej reguły ładowane są za pomocą programu ipnat. Parametr -f, podobnie jak w ipf, pozwala wskazać inny plik lub standardowe wejście zamiast domyślnego pliku. Poniższy opis został opracowany na podstawie dokumentacji [3]. Przedstawione tutaj zostały tylko wybrane konstrukcje.

Reguły NAT zbudowane są w ten sposób, że najpierw występuje słowo kluczowe określające rodzaj mapowania adresów, za nim znajdują się deklaracje pozwalające dopasować pakiet, dalej jest symbol ->, a za nim deklaracje mówiące co ma być wpisane do pakietu.

Translacja adresów źródłowych

Ten rodzaj translacji w terminologii IPFiltra określa się jako *mapowanie*. W mapowaniu zarówno adres IP, jak i port mogą zostać zmienione. Najprostsza tego typu reguła może wyglądać następująco:

```
map e1000g0 0/0 -> 0/32
```

Mówi ona, aby przemapować źródłowe adresy IP wszystkich pakietów wychodzących (outbound) przez interfejs e1000g0, na adres IP, jaki w danym momencie ma ten interfejs. Wartość 0/0 po lewej stronie pasuje do wszystkich pakietów, natomiast 0/32 oznacza bieżący adres interfejsu e1000g0.

Gdy potrzebujemy przemapować tylko część ruchu wychodzącego przez e1000g0, na przykład z lokalnej podsieci 192.168.0.0/24 na konkretny adres zewnętrzny 10.0.0.10, możemy to wykonać tak:

```
map e1000g0 192.168.0.0/24 -> 10.0.0.10/32
```

To proste przemapowanie samego tylko adresu źródłowego szybko doprowadzi do problemów w protokołach TCP i UDP. Ponieważ numery portów nie są mapowane, więc gdy kilka hostów z sieci lokalnej będzie nawiązywać połączenia z tym samym źródłowym portem, to po mapowaniu pojawi się wiele takich samych par adres/port. Rozwiązuje się to, stosując dyrektywę portmap po prawej stronie:

```
map e1000g0 192.168.0.0/24 -> 10.0.0.10/32 portmap tcp/udp 10000:30000
```

W ten sposób źródłowe numery portów TCP/UDP będą mapowane na zakres [10000, 30000].

Przy niektórych usługach translacja adresów musi być wykonana w dość przemyślany sposób. Dotyczy to między innymi protokołu FTP, w którym do komunikacji używane są dwa porty 20 (do transferu danych) i 22 (do przekazywania komend). W tej sytuacji można zastosować tak zwane proxy:

```
map e1000g0 192.168.0.0/24 -> 10.0.0.10/32 proxy port ftp ftp/tcp
```

Translacja adresów docelowych

Translacja adresów docelowych zwana jest *redyrekcją*. Zasady konstruowania takich reguł są podobne jak przy mapowaniu. Najlepszym chyba przykładem, gdy potrzebujemy zastosować redyrekcję, jest przekierowanie portu 9122 routera na port 22 hosta o adresie `192.168.0.100` w sieci lokalnej:

```
rdr e1000g0 0/0 port 9122 -> 192.168.0.100 port 22 tcp
```

IPFilter dostarcza mnóstwo dodatkowych opcji wspomagających konstruowanie skomplikowanych reguł translacji adresów. Reguły typu `rewrite` pozwalają zmieniać jednocześnie adresy źródłowe i docelowe, a także porty.

Kolejność reguł

Ładowanie reguł do jądra systemu odbywa się w tej kolejności, w jakiej zostały umieszczone w pliku konfiguracyjnym. Dopasowanie jednak odbywa się według maski, od 32 do 0, to znaczy, że bardziej szczegółowe reguły zostaną dopasowane wcześniej. Następujące reguły:

```
rdr e1000g0 192.0.0.0/8 port 80 -> 127.0.0.1 3132 tcp
rdr e1000g0 192.2.0.0/16 port 80 -> 127.0.0.1 3131 tcp
rdr e1000g0 192.2.2.0/24 port 80 -> 127.0.0.1 3129 tcp
rdr e1000g0 192.2.2.1 port 80 -> 127.0.0.1 3128 tcp
```

zostaną dopasowane w tej kolejności:

```
rdr e1000g0 192.2.2.1 port 80 -> 127.0.0.1 3128 tcp
rdr e1000g0 192.2.2.0/24 port 80 -> 127.0.0.1 3129 tcp
rdr e1000g0 192.2.0.0/16 port 80 -> 127.0.0.1 3131 tcp
rdr e1000g0 192.0.0.0/8 port 80 -> 127.0.0.1 3132 tcp
```

3.3.3 `ippool.conf`

Plik `/etc/ipf/ippool.conf` zawiera konfigurację podsystemu puli adresów. Pule adresów ładowane są do jądra systemu za pomocą programu `ippool`. Opcja `-f` pozwala załadować konfigurację z innego, niestandardowego pliku.

W dokumentacji [3] znajdziemy pełny opis wielu możliwych scenariuszy, w których stosuje się różne rodzaje puli adresów. Tutaj ograniczymy się do jednego tylko rodzaju puli adresów używanych do dopasowywania adresów źródłowych lub docelowych w regułach filtra `ipf.conf`.

Typowa deklaracja puli adresów dla `ipf` w postaci drzewa (tree) o nazwie `trusted` może wyglądać tak:

```
pool ipf/tree (name trusted;) { 1.1.1.1/32; 2.2.0.0/16; !2.2.2.0/24; };
```

W nawiasach klamrowych podajemy listę adresów. Rozdzielamy je średnikami. Znak `!` to negacja. Możemy podawać adresy pojedynczych hostów lub całych podsieci zgodnie z notacją CIDR. Czasem wygodniej jest mieć tę listę adresów w osobnym pliku:

```
pool ipf/tree (name trusted;) { "file:///etc/ipf/trusted.pool"; };
```

3.3.4 Parametry tuningowe

Poza zestawem reguł, do swoich potrzeb można dostrajać wartości licznych parametrów IPFiltra, takich jak na przykład: rozmiar tablicy stanów `state_size`, rozmiar tablicy translacji adresów `nat_table_size`, rozmiar bufora dziennika `log_size`. Pełna lista parametrów, uzyskana poleceniem `ipf -T list`, znajduje się w dodatku B. Podane są tam kolejno: nazwa parametru, wartość minimalna, wartość maksymalna oraz wartość bieżąca.

Rozdział 4

Kompilacja i uruchomienie

Po rozpakowaniu paczki z kodem źródłowym i uruchomieniu głównego `Makefile` za pomocą programu `make` wyświetla się lista platform, na jakie można kompilować IPFilter 5.1.2:

```
Chose one of the following targets for making IP filter:

solaris     - auto-selects SunOS4.1.x/Solaris 2.3-6/Solaris2.4-6x86
netbsd      - compile for NetBSD
openbsd     - compile for OpenBSD
freebsd20   - compile for FreeBSD 2.0, 2.1 or earlier
freebsd22   - compile for FreeBSD-2.2 or greater
freebsd     - compile for all other versions of FreeBSD
bsd         - compile for generic 4.4BSD systems
bsdi        - compile for BSD/OS
irix        - compile for SGI IRIX
hpux        - compile for HP-UX 11.00
osf         - compile for OSF/Tru64 5.1
```

Domyślnie używany jest kompilator GCC. Do kompilacji na Solaris 10 używaliśmy kompilatora Sun Studio 5.13, więc konieczne były drobne zmiany w `Makefile` oraz w skrypcie `buildsunos`. W tym drugim konieczne było dodanie `-xmodel=kernel` w opcjach kompilacji na platformę x64:

```
XARCH64_i386="$XARCH32 -m64 -xmodel=kernel"
```

Kompilacja przebiega czysto. Po instalacji binariów moduł jądra ładuje się i IPFilter się uruchamia . Pozornie wszystko działa poprawnie, ale przy bardziej zaawansowanej konfiguracji ujawniają się problemy, które opisujemy w kolejnych podrozdziałach.

4.1 Blokowanie protokołu GRE

IPFilter w wersji 4.9.1 dostarczany z Solaris 10 i 11 oraz z OpenIndiana działa niezawodnie, ale nie przepuszcza protokołu GRE używanego w kilku wariantach sieci VPN. To spore ograniczenie, gdy z sieci lokalnej za routerem w

postaci Solaris plus IPFilter chcemy łączyć się na zewnątrz, poprzez VPN. Próba przejścia na nowszą wersję IPFiltra podyktowana była tym właśnie problemem. O ile wersja 5.1.2 bezproblemowo przepuszcza protokół GRE, to niestety zaczynają się inne kłopoty.

4.2 Reguły typu fastroute

Na ruterach podłączonych do kilku dostawców usług sieciowych nieodzowne jest stosowanie policy routing, czyli reguł typu `to` oraz `replay-to`. Te reguły w wersji 5.1.2 po prostu nie zadziałały. Konieczna okazała się ingerencja w kod źródłowy. Ostatecznie zmian nie jest dużo, ale znalezienie i usunięcie usterki było nie lada wyzwaniem.

Za policy routing odpowiada funkcja `ipf_fastroute()` zdefiniowana w `ip_fil_solaris.c`. Tutaj zmiany są najpoważniejsze.

Technicznie policy routing polega na pominięciu tablicy routingu w jądrze systemu i przerzuceniu pakietu na odpowiedni interfejs wyjściowy. Napisana do tego została funkcja `ipf_sendpkt()`, ale nie do końca poprawnie. W Solaris 10 jest funkcja systemowa `net_inject()`, która to potrafi i należało ją wykorzystać. Ostatecznie usunęliśmy `ipf_sendpkt()`.

W `ipf_fastroute()` przeliczane są parametry nagłówka IP za pomocą `htons()`, która zamienia kolejność bajtów hosta (host byte order) na kolejność sieciową (network byte order). Niepotrzebnie, bo robią to funkcje wołane dalej z `ipf_fastroute()` i w efekcie kolejność bajtów jest niepoprawna. Objawiało się to bardzo dziwnym zachowaniem stosu TCP/IP. Widać było, że pakiety przechodzą, ale są poważnie poprzekręcane.

Poza tym należało poprawić wyliczanie sum kontrolnych oraz interakcję z podsystemem NAT. Większość zmian została dokonana na podstawie wielokrotnie poprawianego kodu źródłowego wersji 4.9.1 rozpowszechnianej z jądrem systemu Illumos [6].

4.3 Fragmentacja IP

Kolejna przypadłość wersji 5.1.2 to problem z obsługą pofragmentowanej transmisji TCP. Objawem był paskudny kernel panic:

```
ipf:ipf_frag_delete+55 ()
ipf:ipf_frag_expire+ad ()
ipf:ipf_slowtimer+28 ()
ipf:ipf_call_slow_timer+44 ()
ipf:ipf_timer_func+1a ()
```

Ponieważ wersja 5.1.2 jest używana we FreeBSD, na jednym z forów udało się namierzyć opis pluskwy i na szczęście banalną poprawkę [8]. W linii 477 pliku `ip_frag.c` należało zamienić `KFREE(fra)` na `KFREE(fran)` – prawdopodobnie literówka. To załatwiało problem.

4.4 Czyszczenia tablicy stanów

Po wdrożeniu wersji 5.1.2 na jednym serwerze produkcyjnym po pewnym czasie pojawiały się dziwne problemy prowadzące do dość szybkiego przepełnienia tablicy stanów. W konsekwencji częściowo przestawał działać NAT, reguły `reply-to` wykorzystujące opcję `keep state` zachowywały się tak, jakby ich nie było. W raportach `ipfstat` pokazywał wiszące, zakończone połączenia. Trudno było znaleźć przyczynę w kodzie IPFiltra. Małe śledztwo, porównanie kodu 4.9.1 oraz 5.1.2 w systemie Subversion SVN dla FreeBSD [7] wykazało problemy w `ip_nat.c` polegające na tym, że pewne wartości (`nss->ns_bucketlen[bkt]`) były dekrementowane bez wcześniejszego sprawdzenia, czy są dodatnie.

4.5 Sumy kontrolne MD5

Wydawało się, że wszystkie pluskwy w wersji 5.1.2 zostały wyłapane i IPFilter wdrożono na 3 systemach produkcyjnych. Na dwóch z nich co jakiś czas wyskakiwał kernel panic o bardzo podobnym przebiegu:

```
ipf:MD5Update+b1 ()
ipf:ipf_rand_push+bd ()
ipf:ipf_hk_v4_in+c1 ()
hook:hook_run+6c ()
ip:ipf_input+3bb ()
dls:i_dls_link_rx+32e ()
mac:mac_rx+71 ()
e1000g:e1000g_intr_work+c8 ()
```

To co łączyło te dwie maszyny to karty sieciowe Intel PRO/1000 Gigabit Ethernet działające ze sterownikiem `e10000g`. Na trzeciej używane były karty Broadcom i tam takich problemów nie było. Z drugiej strony `e10000g` niezawodnie obsługuje dość dużą rodzinę bardzo popularnych interfejsów sieciowych marki Intel. Podejrzenie padło na funkcję `MD5Update()` z modułu `ipf`. Używana ona jest, gdy należy losowo wygenerować numer portu po translacji adresu źródłowego, albo przy emisji pakietu TCP RST lub ICMP z informacją zwrotną. Po dłuższym grzebaniu w dokumentacji Solaris okazało się, że jest moduł `md5` dostarczający taką samą funkcję:

```
51 ffffffffefa29000   2c68   -   1  md5 (MD5 Message-Digest Algorithm)
51 ffffffffefa29000   2c68   -   1  md5 (MD5 Kernel SW Provider 1.1)
```

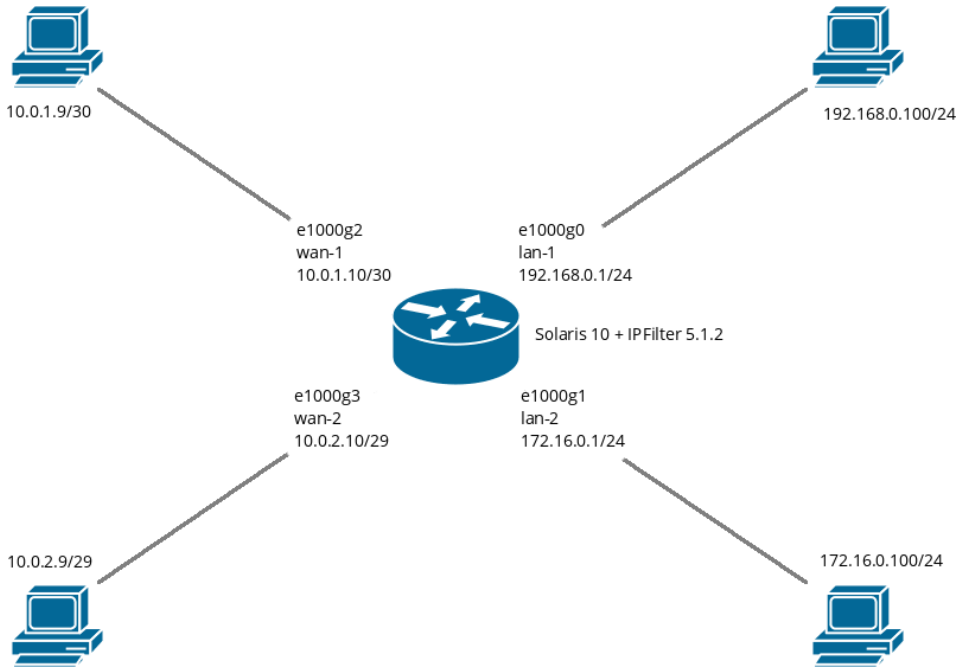
Prawdopodobnie w momencie pisania IPFiltra w Solaris jeszcze takiego modułu nie było i konieczna była implementacja niezbędnej funkcji. Poprawka okazała się banalna. Przekręcone były stałe preprocesora. Należało w pliku `arc4random.c` zmienić `_SYS_MD5_H` na `HAS_SYS_MD5_H`.

Od połowy kwietnia 2020 obie maszyny z poprawką działają bez kernel panic, ale nie chwalmy dnia przed zachodem słońca.

4.6 Testowanie

Za każdym razem, po kompilacji oprogramowania wypada przeprowadzić testy i sprawdzić, czy zmiany w kodzie dają oczekiwane rezultaty. W naszym przypadku zdarzało się wielokrotnie wracać do poprzedniej wersji i zaczynać od nowa. Na każdym etapie proste testy pozwalały ustalić, czy dana wersja działa w zadowalający sposób.

Testy polegały na sprawdzeniu tych typów reguł, które były modyfikowane. W tym celu stworzyliśmy wirtualną sieć opartą na VirtualBox. Centralną rolę w tej sieci gra oczywiście Solaris 10 z testowanym IPFilter jako router. Poza nim w sieci znajdują się 4 maszyny: dwie z nich symulują komputery w sieci WAN (Internet), pozostałe dwie udają komputery w dwóch różnych podsieciach lokalnych LAN. Diagram tej sieci przedstawiono na rysunku 4.1.



Rysunek 4.1: Diagram sieci do testowania routera opartego o IPFilter.

Istotne są dwie różne sieci WAN udające dwóch różnych dostawców ISP, aby można zweryfikować policy routing, czyli reguły `reply-to` oraz `to`. Dwie podsieci LAN symulują sytuację, gdzie nasz router obsługuje nie tylko więcej niż jednego ISP, ale i więcej niż jedną sieć LAN. Sieć LAN-1 to sieć biurowa, bardziej uprzywilejowana i zaufana, natomiast sieć LAN-2 to sieć serwisowa, znacznie mniej zaufana, o większych ograniczeniach. W praktyce takich podsieci może być więcej, ale nasz scenariusz obejmuje wszelkie możliwe połączenia.

W naszym routerze mamy 4 interfejsy sieciowe. Nazywają się one `e1000gN`,

gdzie N zmienia się od 0 do 3. Domyślna trasa prowadzi do dostawcy ISP-1 w sieci WAN-1, czyli przez interfejs `e1000g2`. Pokazuje to program `netstat`:

```
Routing Table: IPv4
  Destination           Gateway           Flags  Ref     Use     Interface
-------------------- -------------------- ----- ----- ---------- ---------
default              10.0.1.9             UG        1       6332
10.0.1.8             10.0.1.10            U         1        450 e1000g2
10.0.2.8             10.0.2.10            U         1        187 e1000g3
172.16.0.0           172.16.0.1           U         1        323 e1000g1
192.168.0.0          192.168.0.1          U         1        576 e1000g0
```

Na routerze włączone jest przekazywanie pakietów. Oznacza to, że pakiet wchodzący jednym interfejsem może wyjść drugim. Jest to podstawa działania routera i ta funkcjonalność odróżnia właśnie router od zwykłego hosta. Pokazuje to program `routeadm`:

```
              Configuration   Current             Current
                     Option   Configuration       System State
-------------------------------------------------------------
               IPv4 routing   disabled            disabled
               IPv6 routing   disabled            disabled
             IPv4 forwarding  enabled             enabled
             IPv6 forwarding  disabled            disabled
```

Podsieć LAN-1 korzysta z sieci Internet za pośrednictwem dostawcy ISP-1, natomiast podsieć LAN-2 za pośrednictwem ISP-2. Takie rozwiązanie wymaga zastosowania policy routing. W LAN-1 można korzystać z VPN. Poniżej prezentujemy plik konfiguracyjny `/etc/ipf/ipf.conf` używany w naszych testach.

```
##############################################################################
#
# LAN-1 : e1000g0 : 192.168.0.1/24 : biuro
#
### Inbound traffic
#
# Spoofing
#
block in quick on e1000g0 from 172.16.0.0/12 to any
```

Na interfejsie `e1000g0`, podłączonym do sieci LAN-1, blokujemy jedynie ruch wchodzący z sieci LAN-2, gdyby na przykład ktoś nieprawidłowo podłączył komputery do przełącznika.

```
##############################################################################
#
# LAN-2 : e1000g1 : 172.16.0.1/24 : serwis
#
### Inbound traffic
```

```
#
# Spoofing
#
block in quick on e1000g1 from 192.168.0.0/16 to any
block in quick on e1000g1 from any to 192.168.0.0/16
#
### Outbound traffic
#
# LAN-1 biuro to LAN-2 serwis
#
pass out quick on e1000g1 proto tcp \
    from 192.168.0.0/16 to 172.16.0.0/12 flags S keep state
pass out quick on e1000g1 proto udp \
    from 192.168.0.0/16 to 172.16.0.0/12 keep state
pass out quick on e1000g1 proto icmp \
    from 192.168.0.0/16 to 172.16.0.0/12 keep state
```

Na interfejsie `e1000g1`, podłączonym do sieci LAN-2, blokujemy ruch wchodzący z oraz skierowany do sieci LAN-1. Przepuszczamy natomiast ruch wychodzący pochodzący z zaufanej sieci biurowej LAN-1. Poszczególne reguły `pass out` dotyczą kolejno protokołów TCP, UDP i ICMP.

```
##############################################################################
#
# WAN-1 : e1000g2 : 10.0.1.10/30 : ISP-1
#
### Inbound traffic
#
# Restrictive policy
#
block in on e1000g2 all
#
# ICMP ping
#
pass in on e1000g2 proto icmp \
    from any to 10.0.1.10/32 icmp-type 8 keep state
#
# SSH
#
block in log quick on e1000g2 proto tcp \
    from any to 10.0.1.10/32 port = 22 flags S keep state
#
# HTTP
#
pass in on e1000g2 proto tcp \
    from any to 10.0.1.10/32 port = 80 flags S keep state
pass in on e1000g2 proto tcp \
    from any to 10.0.1.10/32 port = 443 flags S keep state
#
### Outbound traffic
#
# Policy routing
```

```
#
pass out quick on e1000g2 to e1000g3:10.0.2.9 proto tcp \
    from 172.16.0.0/24 to any flags S keep state
pass out quick on e1000g2 to e1000g3:10.0.2.9 proto udp \
    from 172.16.0.0/24 to any keep state
pass out quick on e1000g2 to e1000g3:10.0.2.9 proto icmp \
    from 172.16.0.0/24 to any keep state
pass out quick on e1000g2 to e1000g3:10.0.2.9 proto gre \
    from 172.16.0.0/24 to any keep state
#
# Locally generated traffic
#
pass out quick on e1000g2 proto tcp all flags S keep state
pass out quick on e1000g2 proto udp all keep state
pass out quick on e1000g2 proto icmp all keep state
pass out quick on e1000g2 proto gre all keep state
#
### NAT traffic
#
pass in on e1000g2 proto tcp \
    from pool/trusted to 192.168.0.100/32 port = 22 flags S keep state
pass in on e1000g2 proto tcp \
    from pool/trusted to 192.168.0.100/32 port = 80 flags S keep state
```

Na interfejsie `e1000g2`, podłączonym do sieci WAN-1, jeśli chodzi o ruch wchodzący do serwera, wpuszczamy żądania echa ICMP, protokoły SSH i HTTP.

Jeśli natomiast chodzi o ruch wychodzący, to bardzo istotny jest tutaj zestaw reguł typu `to` oznaczony jako policy routing. Reguła typu `to` mówi, że jeśli na interfejsie `e1000g2` pojawi się pakiet z sieci LAN-2 skierowany dokądkolwiek to należy go przepuścić, ale nie przez `e1000g2` tylko przez `e1000g3` kierując go bezpośrednio na router `10.0.2.9` dostawcy ISP-2. Są 4 reguły typu `to`, po jednej dla każdego z protokołów TCP, UDP, ICMP oraz GRE. Ten ostatni związany jest z VPN i to jest drugi bardzo ważny test, bo na maszynie w LAN-1 korzystamy z VPN.

Dalej przepuszczamy ruch wychodzący w świat wygenerowany lokalnie na serwerze.

Na `e1000g2` mamy jeszcze do czynienia z ruchem wchodzącym wykorzystującym DNAT. Jest on skierowany do maszyny `192.168.0.100` w sieci LAN-1, której porty SSH i HTTP zostały przekierowane przez podsystem NAT.

```
########################################################################
#
# WAN-2 : e1000g3 : 10.0.2.10/29 : ISP-2
#
### Inbound traffic
#
# Restrictive policy
#
block in on e1000g3 all
#
```

```
# ICMP ping
#
pass in on e1000g3 reply-to e1000g3:10.0.2.9 proto icmp \
    from any to 10.0.2.10/32 icmp-type 8 keep state
#
# SSH
#
block in log quick on e1000g3 proto tcp from any to any port = 22
#
# HTTP
#
pass in on e1000g3 reply-to e1000g3:10.0.2.9 proto tcp \
    from any to 10.0.2.10/32 port = 80 flags S keep state
pass in on e1000g3 reply-to e1000g3:10.0.2.9 proto tcp \
    from any to 10.0.2.10/32 port = 443 flags S keep state
#
### Outbound traffic
#
### NAT traffic
#
pass in on e1000g3 reply-to e1000g3:10.0.2.9 proto tcp \
    from pool/trusted to 172.16.0.100/32 port = 3389 flags S keep state
```

Interfejs `e1000g3` podłączony jest do sieci WAN-2. Podobnie jak na `e1000g2` wpuszczamy do serwera żądania echa ICMP, SSH i HTTP, ale musimy zastosować policy routing, bo domyślna trasa biegnie przez `e1000g2` i w konsekwencji odpowiedzi na echo, SSH, HTTP wyjdą nie przez `e1000g3`, ale przez `e1000g2` z adresem źródłowym tego interfejsu i nigdy nie trafią do pytającego klienta. Aby tego uniknąć stosujemy deklarację `reply-to e1000g3:10.0.2.9`, która mówi, że odpowiedzi na odpowiednie żądania kierujemy poprzez `e1000g3` bezpośrednio do routera `10.0.2.9` dostawcy ISP-2. Poza tym wpuszczamy pakiety protokołu RDP (Remote Desktop) z puli adresów zaufanych skierowany do maszyny `172.16.0.100` w sieci LAN-2.

Poniżej prezentujemy plik konfiguracyjny `/etc/ipf/ipnat.conf` używany w testach.

```
##############################################################################
#
# WAN-1 : e1000g2 : 10.0.1.10/30 : ISP-1
#
### Destination NAT

rdr e1000g2 0/0 port 9022 -> 192.168.0.100 port 22 tcp
rdr e1000g2 0/0 port 9080 -> 192.168.0.100 port 80 tcp

### Source NAT

map e1000g2 192.168.0.0/24 -> 0/32 proxy port ftp ftp/tcp
map e1000g2 192.168.0.0/24 -> 0/32 portmap tcp/udp 10000:30000
map e1000g2 192.168.0.0/24 -> 0/32
```

Na interfejsie `e1000g2`, za pomocą reguł `rdr`, przekierowujemy porty 9022 oraz 9080 serwera odpowiednio na porty 22 i 80 maszyny `192.168.0.100` w sieci LAN-1. Za pomocą reguł `map` adresy źródłowe pakietów wychodzących przez ten interfejs z sieci LAN-1 zostaną przemapowane na adres naszego routera w sieci WAN-1.

```
##############################################################################
#
# WAN-2 : e1000g3 : 10.0.2.10/29 : ISP-2
#
#
### Destination NAT

rdr e1000g3 0/0 port 9033 -> 172.16.0.100 port 3389 tcp

### Source NAT

map e1000g3 172.16.0.0/24 -> 0/32 proxy port ftp ftp/tcp
map e1000g3 172.16.0.0/24 -> 0/32 portmap tcp/udp 10000:30000
map e1000g3 172.16.0.0/24 -> 0/32
```

Na interfejsie `e1000g3`, za pmocą `rdr`, przekierowujemy port 9033 serwera na port 3389 maszyny `172.16.0.100` w sieci LAN-2. Za pomocą reguł `map` adresy źródłowe pakietów wychodzących przez ten interfejs z sieci LAN-2 zostaną przemapowane na adres naszego routera w sieci WAN-2.

Podsumowanie

Pierwszą trudnością napotkaną przy realizacji tematu był oczywiście sam IPFilter – mało popularny, wręcz niszowy aktualnie, relatywnie słabo udokumentowany. Należało, zrozumieć jak działa i co można przy jego pomocy osiągnąć, no i jakie ma ograniczenia. Stąd tak rozbudowany rozdział trzeci pracy.

Druga, jeszcze poważniejsza trudność to zapanowanie nad kodem źródłowym, kompilacją i instalacją zapory IPFilter. Na potrzeby testów zostało opracowane wirtualne środowisko sieciowe złożone z dwóch sieci LAN, dwóch sieci WAN oraz z Solaris i IPFilter w środku (por. rys. 4.1).

Korzystając z forów FreeBSD, podglądając mocno poprawiony kod wersji 4.9.1 oraz system Subversion SVN dla FreeBSD udało się wyłowić i usunąć podstawowe usterki w kodzie IPFilter 5.1.2. Nie tylko testy, ale praktyczne wdrożenia poprawionej wersji pokazują, że cel został osiągnięty – IPFilter można używać tam, gdzie jednocześnie konieczne są połączenia VPN oraz policy routing.

Dodatek A

Gramatyka reguł filtra pakietów

```
filter-rule = [ insert ] action in-out [ options ] [ tos ] [ ttl ]
      [ proto ] [ ip ] [ group ] [ tag ] [ pps ] .

insert = "@" decnumber .
action = block | "pass" | log | "count" | auth | call .
in-out = "in" | "out" .
options = [ log ] [ "quick" ] [ onif [ dup ] [ froute ] ] .
tos = "tos" decnumber | "tos" hexnumber .
ttl = "ttl" decnumber .
proto = "proto" protocol .
ip = srcdst [ flags ] [ with withopt ] [ icmp ] [ keep ] .
group = [ "head" decnumber ] [ "group" decnumber ] .
pps = "pps" decnumber .

onif = "on" interface-name [ "out-via" interface-name ] .
block = "block" [ return-icmp[return-code] | "return-rst" ] .
auth = "auth" | "preauth" .
log = "log" [ "body" ] [ "first" ] [ "or-block" ] [ "level" loglevel ] .
tag = "tag" tagid .
call = "call" [ "now" ] function-name "/" decnumber.
dup = "dup-to" interface-name[":"ipaddr] .
froute = "fastroute" | "to" interface-name .
replyto = "reply-to" interface-name [ ":" ipaddr ] .
protocol = "tcp/udp" | "udp" | "tcp" | "icmp" | decnumber .
srcdst = "all" | fromto .
fromto = "from" object "to" object .

return-icmp = "return-icmp" | "return-icmp-as-dest" .
loglevel = facility"."priority | priority .
object = addr [ port-comp | port-range ] .
addr = "any" | nummask | host-name [ "mask" ipaddr | "mask" hexnumber ] .
port-comp = "port" compare port-num .
port-range = "port" port-num range port-num .
flags = "flags" flag { flag } [ "/" flag { flag } ] .
with = "with" | "and" .
icmp = "icmp-type" icmp-type [ "code" decnumber ] .
return-code = "("icmp-code")" .
```

```
keep = "keep" "state" [ "limit" number ] | "keep" "frags" .

nummask = host-name [ "/" decnumber ] .
host-name = ipaddr | hostname | "any" .
ipaddr = host-num "." host-num "." host-num "." host-num .
host-num = digit [ digit [ digit ] ] .
port-num = service-name | decnumber .

withopt = [ "not" | "no" ] opttype [ [ "," ] withopt ] .
opttype = "ipopts" | "short" | "nat" | "bad-src" | "lowttl" | "frag" |
          "mbcast" | "opt" ipopts  .
optname = ipopts [ "," optname ] .
ipopts  = optlist | "sec-class" [ secname ] .
secname = seclvl [ "," secname ] .
seclvl  = "unclass" | "confid" | "reserv-1" | "reserv-2" | "reserv-3" |
  "reserv-4" | "secret" | "topsecret" .
icmp-type = "unreach" | "echo" | "echorep" | "squench" | "redir" |
    "timex" | "paramprob" | "timest" | "timestrep" | "inforeq" |
    "inforep" | "maskreq" | "maskrep"  | "routerad" |
    "routersol" | decnumber .
icmp-code = decumber | "net-unr" | "host-unr" | "proto-unr" | "port-unr" |
    "needfrag" | "srcfail" | "net-unk" | "host-unk" | "isolate" |
    "net-prohib" | "host-prohib" | "net-tos" | "host-tos" |
    "filter-prohib" | "host-preced" | "cutoff-preced" .
optlist = "nop" | "rr" | "zsu" | "mtup" | "mtur" | "encode" | "ts" | "tr" |
  "sec" | "lsrr" | "e-sec" | "cipso" | "satid" | "ssrr" | "addext" |
  "visa" | "imitd" | "eip" | "finn" .
facility = "kern" | "user" | "mail" | "daemon" | "auth" | "syslog" |
   "lpr" | "news" | "uucp" | "cron" | "ftp" | "authpriv" |
   "audit" | "logalert" | "local0" | "local1" | "local2" |
   "local3" | "local4" | "local5" | "local6" | "local7" .
priority = "emerg" | "alert" | "crit" | "err" | "warn" | "notice" |
   "info" | "debug" .

hexnumber = "0" "x" hexstring .
hexstring = hexdigit [ hexstring ] .
decnumber = digit [ decnumber ] .

compare = "=" | "!=" | "<" | ">" | "<=" | ">=" | "eq" | "ne" | "lt" | "gt" |
  "le" | "ge" .
range = "<>" | "><" .
hexdigit = digit | "a" | "b" | "c" | "d" | "e" | "f" .
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
flag = "F" | "S" | "R" | "P" | "A" | "U" | "C" | "W" .
```

Dodatek B

Parametry tuningowe

```
ipf_flags           min 0   max 4294967295  current 0
active              min 0   max 0           current 1
control_forwarding  min 0   max 1           current 0
update_ipid         min 0   max 1           current 0
chksrc              min 0   max 1           current 0
min_ttl             min 0   max 1           current 4
icmp_minfragmtu     min 0   max 1           current 68
default_pass        min 0   max 4294967295  current 134217730
tcp_idle_timeout    min 1   max 2147483647  current 864000
tcp_close_wait      min 1   max 2147483647  current 480
tcp_last_ack        min 1   max 2147483647  current 60
tcp_timeout         min 1   max 2147483647  current 480
tcp_syn_sent        min 1   max 2147483647  current 480
tcp_syn_received    min 1   max 2147483647  current 480
tcp_closed          min 1   max 2147483647  current 60
tcp_half_closed     min 1   max 2147483647  current 14400
tcp_time_wait       min 1   max 2147483647  current 480
udp_timeout         min 1   max 2147483647  current 240
udp_ack_timeout     min 1   max 2147483647  current 24
icmp_timeout        min 1   max 2147483647  current 120
icmp_ack_timeout    min 1   max 2147483647  current 12
ip_timeout          min 1   max 2147483647  current 120
intercept_loopback  min 0   max 1           current 0
log_suppress        min 0   max 1           current 1
log_all             min 0   max 1           current 0
log_size            min 0   max 524288      current 32768
state_max           min 1   max 2147483647  current 4013
state_size          min 1   max 2147483647  current 5737
state_lock          min 0   max 1           current 0
state_maxbucket     min 1   max 2147483647  current 26
state_logging       min 0   max 1           current 1
state_wm_high       min 2   max 100         current 99
state_wm_low        min 1   max 99          current 90
state_wm_freq       min 2   max 999999      current 20
nat_lock            min 0   max 1           current 0
nat_table_size      min 1   max 2147483647  current 2047
nat_table_max       min 1   max 2147483647  current 30000
nat_rules_size      min 1   max 2147483647  current 127
rdr_rules_size      min 1   max 2147483647  current 127
hostmap_size        min 1   max 2147483647  current 2047
nat_maxbucket       min 1   max 2147483647  current 22
nat_logging         min 0   max 1           current 1
nat_doflush         min 0   max 1           current 0
nat_table_wm_low    min 1   max 99          current 90
nat_table_wm_high   min 2   max 100         current 99
proxy_debug         min 0   max 31          current 0
ftp_debug           min 0   max 127         current 0
```

```
ftp_pasvonly        min 0   max 1           current 0
ftp_insecure        min 0   max 1           current 0
ftp_pasvrdr         min 0   max 1           current 0
ftp_forcepasv       min 0   max 1           current 1
ftp_single_xfer     min 0   max 1           current 0
tftp_read_only      min 0   max 1           current 1
ftp_debug           min 0   max 127         current 0
ftp_pasvonly        min 0   max 1           current 0
ftp_insecure        min 0   max 1           current 0
ftp_pasvrdr         min 0   max 1           current 0
ftp_forcepasv       min 0   max 1           current 1
ftp_single_xfer     min 0   max 1           current 0
```

Dodatek C

Zmiany w kodzie

```
diff -c /tmp/ip_fil5.1.2//arc4random.c ./arc4random.c
*** /tmp/ip_fil5.1.2//arc4random.c N gr 27 08:34:34 2009
--- ./arc4random.c Pt kw 17 12:17:01 2020
***************
*** 214,220 ****
   nsrc = src;
   mylen = length;

! #if defined(_SYS_MD5_H) && defined(SOLARIS2)
  # define buf buf_un.buf8
  #endif
   MUTEX_ENTER(&arc4_mtx);
--- 214,220 ----
   nsrc = src;
   mylen = length;

! #if defined(HAS_SYS_MD5_H) && defined(SOLARIS2)
  # define buf buf_un.buf8
  #endif
   MUTEX_ENTER(&arc4_mtx);
***************
*** 239,245 ****
   inpot += 64;
   }
   MUTEX_EXIT(&arc4_mtx);
! #if defined(_SYS_MD5_H) && defined(SOLARIS2)
  # undef buf
  #endif
  }
--- 239,245 ----
   inpot += 64;
   }
   MUTEX_EXIT(&arc4_mtx);
! #if defined(HAS_SYS_MD5_H) && defined(SOLARIS2)
  # undef buf
  #endif
  }
diff -c /tmp/ip_fil5.1.2//ip_compat.h ./ip_compat.h
*** /tmp/ip_fil5.1.2//ip_compat.h Pt lip 20 10:48:33 2012
--- ./ip_compat.h Pt kw 17 12:17:25 2020
***************
*** 171,176 ****
--- 171,177 ----
  # include <sys/sysmacros.h>
  # include <sys/kmem.h>
  # if SOLARIS2 >= 10
+ #  define       HAS_SYS_MD5_H 1
```

```
  #  include <sys/procset.h>
  #  include <sys/proc.h>
  #  include <sys/devops.h>
diff -c /tmp/ip_fil5.1.2//ip_fil_solaris.c ./ip_fil_solaris.c
*** /tmp/ip_fil5.1.2//ip_fil_solaris.c N lip 22 10:04:23 2012
--- ./ip_fil_solaris.c Cz maj 21 10:22:32 2020
***************
*** 63,70 ****
  static int ipf_send_ip __P((fr_info_t *fin, mblk_t *m));
  static void ipf_fixl4sum __P((fr_info_t *));
  static void *ipf_routeto __P((fr_info_t *, int, void *));
- static int ipf_sendpkt __P((ipf_main_softc_t *, int, void *, mblk_t *,
-   struct ip *, void *));
  static void ipf_call_slow_timer __P((ipf_main_softc_t *));
  #if (SOLARIS2 < 7)
  static void ipf_timer_func __P((void));
--- 63,68 ----
***************
*** 1039,1061 ****
   frdest_t *fdp;
  {
   ipf_main_softc_t *softc = fin->fin_main_soft;
-  struct in_addr dst;
   qpktinfo_t *qpi;
   frdest_t node;
   frentry_t *fr;
   frdest_t fd;
-  void *dstp;
   void *sifp;
   void *ifp;
   ip_t *ip;
! #ifndef sparc
!  u_short __iplen, __ipoff;
! #endif
  #ifdef USE_INET6
   ip6_t *ip6 = (ip6_t *)fin->fin_ip;
   struct in6_addr dst6;
  #endif

   fr = fin->fin_fr;
   ip = fin->fin_ip;
   qpi = fin->fin_qpi;
--- 1037,1062 ----
   frdest_t *fdp;
  {
   ipf_main_softc_t *softc = fin->fin_main_soft;
   qpktinfo_t *qpi;
   frdest_t node;
   frentry_t *fr;
   frdest_t fd;
   void *sifp;
   void *ifp;
   ip_t *ip;
!         struct sockaddr_in *sin;
!  struct sockaddr_in6 *sin6;
!  struct sockaddr *sinp;
!         net_inject_t *inj;
  #ifdef USE_INET6
   ip6_t *ip6 = (ip6_t *)fin->fin_ip;
   struct in6_addr dst6;
  #endif

+         inj = net_inject_alloc(NETINFO_VERSION);
+  if (inj == NULL)
+  return -1;
+
```

```
   fr = fin->fin_fr;
   ip = fin->fin_ip;
   qpi = fin->fin_qpi;
***************
*** 1070,1078 ****
    * If this is a duplicate mblk then we want ip to point at that
    * data, not the original, if and only if it is already pointing at
    * the current mblk data.
    */
!  if (ip == (ip_t *)qpi->qpi_m->b_rptr && qpi->qpi_m != mb)
   ip = (ip_t *)mb->b_rptr;

   /*
    * If there is another M_PROTO, we don't want it
--- 1071,1087 ----
    * If this is a duplicate mblk then we want ip to point at that
    * data, not the original, if and only if it is already pointing at
    * the current mblk data.
+   *
+   * Otherwise, if it's not a duplicate, and we're not already pointing
+   * at the current mblk data, then we want to ensure that the data
+   * points at ip.
    */
!  if ((ip == (ip_t *)qpi->qpi_m->b_rptr) && (qpi->qpi_m != mb)) {
   ip = (ip_t *)mb->b_rptr;
+  } else if ((qpi->qpi_m == mb) && (ip != (ip_t *)qpi->qpi_m->b_rptr)) {
+  qpi->qpi_m->b_rptr = (uchar_t *)ip;
+  qpi->qpi_off = 0;
+  }

   /*
    * If there is another M_PROTO, we don't want it
***************
*** 1085,1109 ****
   *mpp = mp;
   }

   /*
    * If the fdp is NULL then there is no set route for this packet.
    */
   if (fdp == NULL) {
!  ifp = fin->fin_ifp;
!
!  switch (fin->fin_v)
!  {
!  case 4 :
!  fd.fd_ip = ip->ip_dst;
!  ifp = ipf_routeto(fin, 4, &ip->ip_dst);
!  break;
! #ifdef USE_INET6
!  case 6 :
!  fd.fd_ip6.in6 = ip6->ip6_dst;
!  ifp = ipf_routeto(fin, 6, &ip6->ip6_dst);
!  break;
! #endif
   }
   fdp = &fd;
   } else {
   ifp = fdp->fd_ptr;
--- 1094,1118 ----
   *mpp = mp;
   }

+         sinp = (struct sockaddr *)&inj->ni_addr;
+         sin = (struct sockaddr_in *)sinp;
+         sin6 = (struct sockaddr_in6 *)sinp;
```

```
+  bzero((char *)&inj->ni_addr, sizeof (inj->ni_addr));
+  inj->ni_addr.ss_family = (fin->fin_v == 4) ? AF_INET : AF_INET6;
+  inj->ni_packet = mb;
+
   /*
    * If the fdp is NULL then there is no set route for this packet.
    */
   if (fdp == NULL) {
!  if (fin->fin_v == 4) {
!  sin->sin_addr = fd.fd_ip = ip->ip_dst;
!                          ifp = net_routeto(softc->ipf_nd_v4, sinp, NULL);
!  } else {
!  sin6->sin6_addr = fd.fd_ip6.in6 = ((ip6_t *)ip)->ip6_dst;
!                          ifp = net_routeto(softc->ipf_nd_v6, sinp, NULL);
   }
+
   fdp = &fd;
   } else {
   ifp = fdp->fd_ptr;
***************
*** 1111,1116 ****
--- 1120,1126 ----
   if (ifp == NULL || ifp == (void *)-1)
   goto bad_fastroute;
   }
+         inj->ni_physical = (phy_if_t)ifp;

   /*
    * In case we're here due to "to <if>" being used with
***************
*** 1120,1140 ****
   if ((fr != NULL) && (fin->fin_rev != 0)) {
   if ((ifp != NULL) && (fdp == &fr->fr_tif))
   return -1;
!  dst.s_addr = fin->fin_fi.fi_daddr;
   } else {
   if (fin->fin_v == 4) {
!  if (fdp->fd_ip.s_addr != 0)
!  dst = fdp->fd_ip;
!  else
!  dst.s_addr = fin->fin_fi.fi_daddr;
!  dstp = &dst;
   }
  #ifdef USE_INET6
   else if (fin->fin_v == 6) {
!  if (IP6_NOTZERO(&fdp->fd_ip))
!  dst6 = fdp->fd_ip6.in6;
!  else
!  dst6 = fin->fin_dst6.in6;
   }
  #endif
   }
--- 1130,1155 ----
   if ((fr != NULL) && (fin->fin_rev != 0)) {
   if ((ifp != NULL) && (fdp == &fr->fr_tif))
   return -1;
!                 if (fin->fin_v == 4) {
!  sin->sin_addr = fdp->fd_ip;
!
!                          /*
!                          int len=20;
!                          char buffer[len];
!                          inet_ntop(AF_INET, &(sin->sin_addr), buffer, len);
!                          printf("fdp->fd_ip:%s\n",buffer);
!                          */
!
```

```
!  } else {
!  sin6->sin6_addr = fdp->fd_ip6.in6;
!  }
   } else {
   if (fin->fin_v == 4) {
!  sin->sin_addr = fdp->fd_ip;
   }
  #ifdef USE_INET6
   else if (fin->fin_v == 6) {
!                          sin6->sin6_addr = fdp->fd_ip6.in6;
   }
  #endif
   }
***************
*** 1147,1152 ****
--- 1162,1173 ----
    * them through stateful checking, etc.
    */
   if ((fdp != &fr->fr_dif) && (fin->fin_out == 0)) {
+                 /*
+                  * Without it packet is not forwarded by kernel
+                  * between interfaces for reply-to + keep state rules.
+                  */
+                 mb->b_datap->db_struioun.cksum.flags = 0;
+
   sifp = fin->fin_ifp;
   fin->fin_ifp = ifp;
   fin->fin_out = 1;
***************
*** 1158,1175 ****
   (void) ipf_state_check(fin, &pass);
   }

!  switch (ipf_nat_checkout(fin, NULL))
!  {
!  case 0 :
!  break;
!  case 1 :
!  ip->ip_sum = 0;
!  break;
!  case -1 :
   goto bad_fastroute;
-  break;
-  }

   fin->fin_out = 0;
   fin->fin_ifp = sifp;
   } else if (fin->fin_out == 1) {
--- 1179,1197 ----
   (void) ipf_state_check(fin, &pass);
   }

!                 /*
!                  * Layer 2 (IP) header checskum becomes 0 here for nat, reply-to
!                  * and keep state mix when ipf_nat_checkout() returns 1.
!                  */
!  if (ipf_nat_checkout(fin, NULL) == -1)
   goto bad_fastroute;

+                 /*
+                  * Layer 3 (TCP/UDP) checksum is 0 without this fix.
+                  */
+                 mb->b_datap->db_struioflag = 0;
+                 ipf_fixl4sum(fin);
+
   fin->fin_out = 0;
```

```
   fin->fin_ifp = sifp;
   } else if (fin->fin_out == 1) {
***************
*** 1194,1218 ****
  #endif
   }

! #ifndef sparc
!  if (fin->fin_v == 4) {
!  __iplen = (u_short)ip->ip_len,
!  __ipoff = (u_short)ip->ip_off;
!
!  ip->ip_len = htons(__iplen);
!  ip->ip_off = htons(__ipoff);
!  }
! #endif
!  if (ipf_sendpkt(softc, 4, ifp, mb, ip, dstp) == 0) {
   ATOMIC_INCL(softc->ipf_frouteok[0]);
   } else {
   ATOMIC_INCL(softc->ipf_frouteok[1]);
   }
   return 0;

  bad_fastroute:
   ATOMIC_INCL(softc->ipf_frouteok[1]);
   freemsg(mb);
   return -1;
  }

--- 1216,1233 ----
  #endif
   }

!         if (net_inject(softc->ipf_nd_v4, NI_DIRECT_OUT, inj) == 0) {
   ATOMIC_INCL(softc->ipf_frouteok[0]);
   } else {
   ATOMIC_INCL(softc->ipf_frouteok[1]);
   }
+         net_inject_free(inj);
   return 0;

  bad_fastroute:
   ATOMIC_INCL(softc->ipf_frouteok[1]);
   freemsg(mb);
+         net_inject_free(inj);
   return -1;
  }

***************
*** 1363,1402 ****
   return net_inject(softc->ipf_nd_v6, NI_QUEUE_OUT, &inject);
  #endif
  }
-
-
- static int
- ipf_sendpkt(softc, v, ifp, mb, ip, dstp)
-  ipf_main_softc_t *softc;
-  int v;
-  void *ifp;
-  mblk_t *mb;
-  struct ip *ip;
-  void *dstp;
- {
- #if !defined(FW_HOOKS)
-         return pfil_sendbuf(ifp, mb, ip, dstp);
```

```
- #else
-  struct sockaddr_in6 *sin6;
-  struct sockaddr_in *sin;
-  net_inject_t inject;
-
-  inject.ni_physical = (phy_if_t)ifp;
-  inject.ni_packet = mb;
-
-  if (v == 4) {
-  sin = (struct sockaddr_in *)&inject.ni_addr;
-  sin->sin_family = AF_INET;
-  memcpy(&sin->sin_addr, dstp, sizeof(sin->sin_addr));
-  return net_inject(softc->ipf_nd_v4, NI_DIRECT_OUT, &inject);
-  }
-
-  sin6 = (struct sockaddr_in6 *)&inject.ni_addr;
-  sin6->sin6_family = AF_INET6;
-  memcpy(&sin6->sin6_addr, dstp, sizeof(sin6->sin6_addr));
-  return net_inject(softc->ipf_nd_v6, NI_DIRECT_OUT, &inject);
- #endif
- }


  static void
--- 1378,1383 ----
diff -c /tmp/ip_fil5.1.2//ip_frag.c ./ip_frag.c
*** /tmp/ip_fil5.1.2//ip_frag.c So lip  7 03:44:44 2012
--- ./ip_frag.c N lut 10 20:02:45 2019
***************
*** 474,480 ****
     IPFR_CMPSZ)) {
   RWLOCK_EXIT(lock);
   FBUMPD(ifs_exists);
!  KFREE(fra);
   return NULL;
   }

--- 474,480 ----
     IPFR_CMPSZ)) {
   RWLOCK_EXIT(lock);
   FBUMPD(ifs_exists);
!  KFREE(fran);
   return NULL;
   }

diff -c /tmp/ip_fil5.1.2//ip_nat.c ./ip_nat.c
*** /tmp/ip_fil5.1.2//ip_nat.c Pt lip 20 09:54:11 2012
--- ./ip_nat.c Cz maj 21 10:08:59 2020
***************
*** 2310,2316 ****

   bkt = nat->nat_hv[0] % softn->ipf_nat_table_sz;
   nss = &softn->ipf_nat_stats.ns_side[0];
!  nss->ns_bucketlen[bkt]--;
   if (nss->ns_bucketlen[bkt] == 0) {
   nss->ns_inuse--;
   }
--- 2310,2317 ----

   bkt = nat->nat_hv[0] % softn->ipf_nat_table_sz;
   nss = &softn->ipf_nat_stats.ns_side[0];
!  if (0 < nss->ns_bucketlen[bkt])
!  nss->ns_bucketlen[bkt]--;
   if (nss->ns_bucketlen[bkt] == 0) {
   nss->ns_inuse--;
   }
```

```
***************
*** 2317,2323 ****

   bkt = nat->nat_hv[1] % softn->ipf_nat_table_sz;
   nss = &softn->ipf_nat_stats.ns_side[1];
!  nss->ns_bucketlen[bkt]--;
   if (nss->ns_bucketlen[bkt] == 0) {
   nss->ns_inuse--;
   }
--- 2318,2325 ----

   bkt = nat->nat_hv[1] % softn->ipf_nat_table_sz;
   nss = &softn->ipf_nat_stats.ns_side[1];
!  if (0 < nss->ns_bucketlen[bkt])
!  nss->ns_bucketlen[bkt]--;
   if (nss->ns_bucketlen[bkt] == 0) {
   nss->ns_inuse--;
   }
```

Bibliografia

[1] Kevin Fall, Richard Stevens, *TCP/IP Illustrated*, Volume 1 The Protocols, Pearson Education Inc., 2012.

[2] Matthew Marsh, *Policy Routing Using Linux*, Wydawnictwo Sams Publishing, 2001.

[3] Darren Reed, Dokumentacja IPFilter, wersja 1.5.2, 2012.

[4] Charlie Scott, Paul Wolfe, Mike Erwin., *Virtual Private Networks*, Wydawnictwo O'Reilly, 1999.

[5] Elizabeth Zwicky, Simon Cooper, Brent Chapman, *Building Internet Firewalls*, Wydawnictwo O'Reilly, 2000.

[6] Unixowy system operacyjny Illumos,
`https://illumos.org/`
Dostęp na dzień 16.06.2020

[7] System Subversion SVN dla FreeBSD, Revision 338171,
`https://svnweb.freebsd.org/changeset/base/338171`
Dostęp na dzień 16.06.2020

[8] System Subversion SVN dla FreeBSD, Revision 317241,
`https://svnweb.freebsd.org/changeset/base/317241`
Dostęp na dzień 16.06.2020

[9] Albert Denkiewicz, *Failover oraz load balancing oparty na IPFilter*, Praca licencjacka, Instytut Informatyki, Uniwersytet w Białymstoku, 2020.