\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Policy Based Routing

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Author*:

Gabriel Rosas



Purpose

Imagine a scenario where you decide to host a server on the internet. Perhaps a website, or maybe a file server. Potentially something else entirely. Without adding any traffic *rules* to your router, anyone from anywhere using any protocol would have access to the server with no restraints. *Policy-based Routing* (PBR) is a routing technique that forwards or denies traffic based on an implementation of rules and filters, known as *policies*. Suddenly, those unused ports on the network can be blocked, and only the necessary ones left open.

Background Information

Introduction

Two servers would run the same two protocols on the same network. However, only one type of traffic was to be permitted to each server from an external network. The goal of this project was to use *policy-based routing* to permit a certain protocol to each server. In this case, we chose to permit HTTP to one server and SSH to the other, with both servers running HTTP and SSH.

Access Lists

Routers and firewalls can use *Access-lists* (ACL) to permit or deny incoming or outgoing traffic. They function as a set of rules, read from the first to last rule. If a packet matches a rule before it is denied, then it “passes” the ACL and continues its path. If a packet does not match any rule in the ACL, then it is denied implicitly, a feature known as the *implicit deny*. There are two main types of ACL: standard and extended.

* Standard
  + Can only match source IP.
  + Cannot match port.
  + Doesn’t distinguish between TCP / UDP / other traffic.
  + Standard ACL identifiers: [*1-99*] & [*1300-1999*] or *named*.
  + *“ip access-list standard 10”  
    “permit 10.0.0.0 255.255.255.0”.*
* Extended
  + Can match traffic based on source IP, destination IP, source port, destination port.
  + Can distinguish between TCP / UDP / other traffic.
  + Extended ACL identifiers: [*100-199*] & [*2000-2699*] or *named*.
  + *“ip access-list extended PERMIT\_HTTP”  
    “permit tcp any host 10.0.0.1 eq www”.*

For example, let’s suppose I want to discard all ping (*icmp*) packets entering my network. A flood of ping requests can overwhelm a network device, denying users service. I could configure the following on a Cisco router:

*ip access-list extended DENY\_ICMP  
deny icmp any any  
permit ip any any*

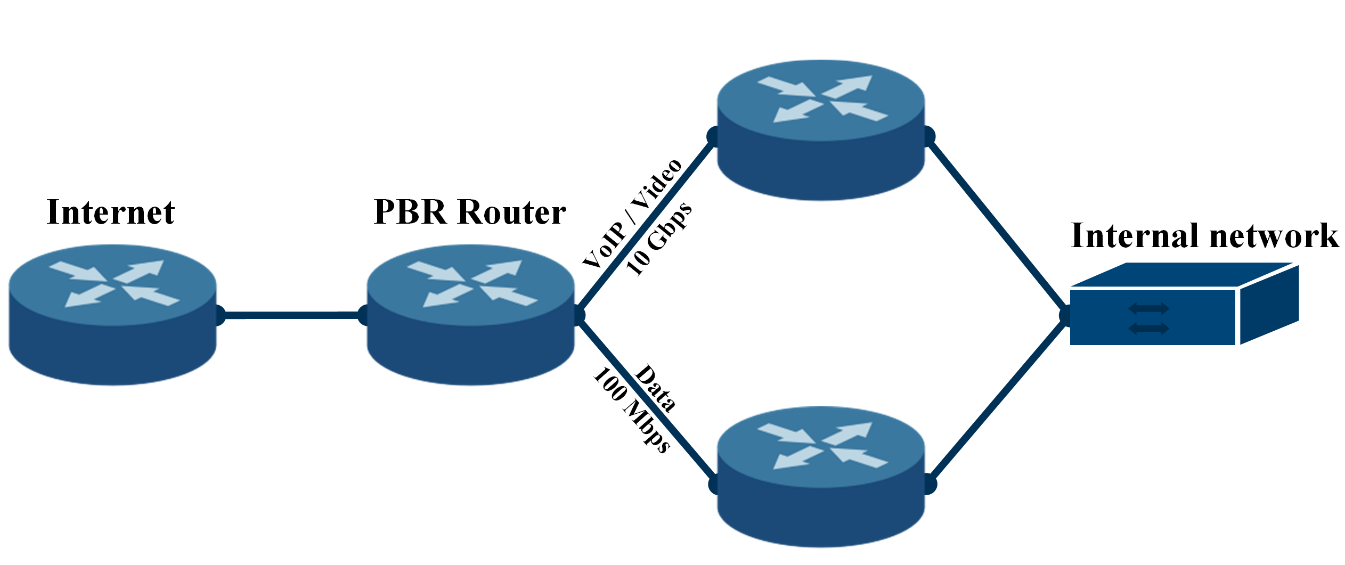
If applied on the correct interface, this access list will deny *icmp* packets entering the network but permit all other types of traffic. But what if an administrator from outside wants to ping our internal network? We would edit the ACL to permit the admin *before* we deny all icmp traffic. Since access lists are read from top to bottom, a permit line before the deny would save the admin’s traffic from being dropped. The admin’s ip address is *232.45.6.23*.

*ip access-list extended DENY\_ICMP  
permit icmp host 232.45.6.23 any  
deny icmp any any  
permit ip any any*

Once the desired access list is created and written, we must apply it on an interface and specify whether we want the traffic coming in or going out to be examined. If I wanted to apply the icmp ACL on all packets *entering* the *GigabitEthernet0/0/0* interface of my router, then the command would be: *ip access-group DENY\_ICMP in* (in interface configuration mode).

Policy Based Routing

Now we know a little about access lists; so, how do they differ from *policy-based routing*? Policy-based routing uses *route maps* – containing access lists –to manipulate traffic like generic access lists. However, route maps provide the additional feature of editing and adding parts to traffic after it has passed or failed the access lists. Let’s look at an example.



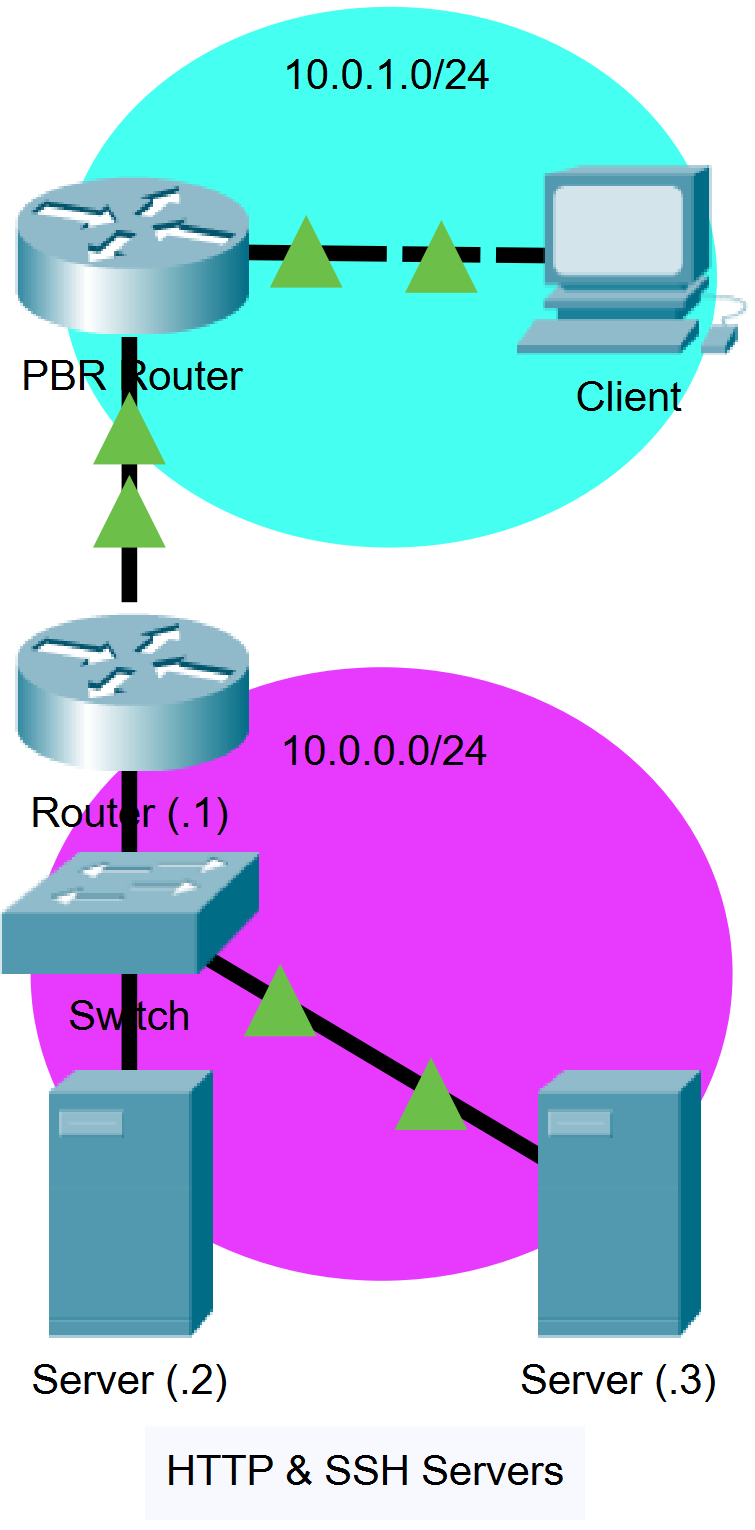
In this case, we have a router (*PBR Router*) that forwards traffic from the internet towards an internal network. The PBR Router has two choices of which to send the traffic: through a *10-gigabit* ethernet connection or through a *100-megabit* ethernet connection. Both routes eventually lead to the internal network. The network administrator wants to divide traffic so that VoIP and Video traffic is sent down the 10 *Gbps* link. With *policy-based routing*, we can *set* a *next-hop ip* based on the type of the source traffic (determined by an access list). The extra step of setting a next hop ip would not be possible with just access lists.

Linux

Like Windows and MacOS, Linux is an operating system which can run applications, but particularly excels at server-based services since it is much more lightweight than other software. In computing, something “lightweight” refers to software designed to have a small memory footprint (RAM), low CPU usage, and low overall usage of system resources. Perfect for something like a server, but less friendly to the average user. Users will typically navigate Linux through the command-line with less focus on graphical applications.

Since Linux is an open-source OS, where anyone can take the base code and manufacture it to their liking, there are many *distributions* designed for specific purposes. For example, in previous projects I worked with *Kali* Linux, a distribution focused on network penetration testing; and *Raspbian*, the official operating system of the Raspberry Pi. In this project, I hopped back on Raspbian to host an Apache Web Server as an end device.

Network Diagram



Lab Summary

I began by designing a topology on packet tracer. Originally, this topology consisted of one router which became two because of later problems. While the main focus was directed toward setting up routing policies, we would first need to configure web servers to give the policies a purpose.

Setting up Apache on Raspbian

We opted with Apache on a Linux-based system for our *http* service. I prepared Apache on my Raspberry Pi while my partner, Harsha Bhat, set up the other on an Ubuntu virtual machine. Installing Apache was straightforward on both machines: *sudo apt-get install apache2*. I followed an official guide by *ubuntu.com*, which provided the minimum configuration necessary to get the service up and running. During the rest of the day, we experimented with *nginx*, an alternative to Apache, installing it on the Raspberry Pi because it was recommended by an acquaintance. After messing around with nginx, we decided to remain with Apache because of its lesser complexity.

When I came back the next day to start up the Apache service on my Raspberry Pi, I encountered an odd error that denied me access when starting the web server: *"job for apache2.service failed because the control process exited with error code"*. I assumed that installing nginx caused some compatibility issues with Apache since Apache worked perfectly prior to the install of nginx. I uninstalled nginx with the command *sudo apt-get --purge remove nginx*, yet the error persisted. After a while, I came across a command that listed open ports currently on the Raspberry Pi: *sudo netstat -ltnp | grep :80*. Much like the *netstat* command on Windows, the output displayed current TCP sockets listening on *port 80* (hence the *grep :80*). A service that was not Apache popped up in the output log. The significance here was that *nothing should have been listening on port 80*.

The Apache server was attempting to listen on port 80 but was being denied because of a service already running on port 80. When Apache could not claim the socket, it threw the *apache2.service failed* error. So, what was listening on the http port? It turns out nginx was still haunting my system, because nginx popped up as the service hogging port 80. I could not figure out how to properly remove nginx, so instead I terminated the socket using the command *sudo kill -9 [id]*. Then I was able to start Apache.

Configuring Policy-Based Routing

With the servers working, it was time to implement the policies. I began by creating an access list rule that permitted all hosts using http to my Raspberry Pi and another permitting ssh to the Ubuntu VM, implicitly denying all other traffic. I applied this access list to the interface of the router connected to the servers for all the traffic exiting. This was all we needed to achieve the purpose of the lab – permitting certain protocols to different servers – however it was not truly *policy-based routing*. For policy-based routing, we needed a route-map to “set” attributes on packets that pass or fail the access-list. The attribute we would set is the *next-hop ip*. To accommodate passing a next-hop ip, we needed to add another router to our topology.

The “internal router” would bridge the policy router and the internal network. We could set the next-hop ip for packets that pass the access list of our policy router as this internal router. After implementing the relevant configuration in both routers, the network was working with a *route-map*.

Improvements

While in the process of writing, I realized there may have been an easier way to implement the policies. This change would only have required one of the routers, instead of both the “Policy Router” and the “Internal Router”. Instead of setting a *next-hop ip*, I could have set the *destination address* to the specified servers. This would theoretically eliminate the need for two routers in the topology.

Lab Commands

|  |  |
| --- | --- |
| **Command** | A statement necessary for a configuration to work, denoted in bold |
| **[*Argument*]** | An argument necessary for a command to function, denoted in bold italics. |
| *Optional-Statement*  *<Optional Argument>* | An optional argument or statement, not necessary for a command to function, denoted in italics |

// Static Route

Router(config)# **ip route [*network address*] [*subnet mask*] [*interface*] [*next-hop ip*]**

* Create a static route to a specified destination network

// Access List

Router(config)# **ip access-list [*type*] [*id*]**

* Enter access list configuration mode

*The type will typically be either a standard or extended access list. Ids range from [1-99] & [1300-1999] for standard ACLs and [100-199] & [2000-2699] for extended. Ids can just be a unique name.*

Router(config-ext-nacl)# <*sequence number*> **permit tcp [*source address*] [*wildcard mask*] [*destination address*] [*wildcard mask*]** *eq* <*protocol*>

* Add a rule to an access list permitting tcp traffic

*The source and destination addresses can be replaced by “host” to then specify a particular host or can be replaced by “any” for all ip addresses. An optional sequence number can be used to determine the order of rules in the access list. An optional protocol can be added to match traffic of the specified protocol.*

// Route Map

Router(config)# **route-map [*name*] permit** <*sequence number*>

* Create a route map and enter route map configuration mode

*Much like access lists, route maps are read from the first to last sequence number. Other route maps with the same name can do different set operations, like “if branches” in programming.*

Router(config-route-map)# **match ip address [*access list id*]**

* Get the route map to match traffic based on an access list

*A set operation will be applied to all traffic matched.*

Router(config-route-map)# **set ip next-hop [*ip address*]**

* Set a next hop ip for matched traffic

Configurations

PBR Router

PBR-Router#show running-config

service timestamps debug datetime msec

service timestamps log datetime msec

no platform punt-keepalive disable-kernel-core

hostname PBR-Router

boot-start-marker

boot-end-marker

vrf definition Mgmt-intf

address-family ipv4

exit-address-family

address-family ipv6

exit-address-family

no aaa new-model

subscriber templating

multilink bundle-name authenticated

license udi pid ISR4321/K9 sn FDO214421BU

spanning-tree extend system-id

redundancy

mode none

vlan internal allocation policy ascending

interface GigabitEthernet0/0/0

ip address 192.168.1.1 255.255.255.252

negotiation auto

interface GigabitEthernet0/0/1

ip address 10.0.1.1 255.255.255.0

ip policy route-map http-ssh-policy

negotiation auto

interface Serial0/1/0

no ip address

shutdown

interface Serial0/1/1

no ip address

shutdown

interface Service-Engine0/2/0

no ip address

shutdown

interface GigabitEthernet0

vrf forwarding Mgmt-intf

no ip address

shutdown

negotiation auto

interface Vlan1

no ip address

shutdown

ip forward-protocol nd

ip http server

ip http authentication local

ip http secure-server

ip tftp source-interface GigabitEthernet0

ip route 10.0.0.0 255.255.255.0 GigabitEthernet0/0/0 192.168.1.2

access-list 100 permit tcp any host 10.0.0.2 eq www

access-list 100 permit tcp any host 10.0.0.3 eq 22

route-map http-ssh-policy permit 10

match ip address 100

set ip next-hop 192.168.1.2

route-map http-ssh-policy permit 20

set interface Null0

control-plane

line con 0

stopbits 1

line aux 0

stopbits 1

line vty 0 4

login

end

Internal Router

Router#show running-config

service timestamps debug datetime msec

service timestamps log datetime msec

platform qfp utilization monitor load 80

platform punt-keepalive disable-kernel-core

hostname Router

boot-start-marker

boot-end-marker

vrf definition Mgmt-intf

address-family ipv4

exit-address-family

address-family ipv6

exit-address-family

no aaa new-model

login on-success log

subscriber templating

multilink bundle-name authenticated

no license smart enable

diagnostic bootup level minimal

spanning-tree extend system-id

redundancy

mode none

interface GigabitEthernet0/0/0

ip address 10.0.0.1 255.255.255.0

negotiation auto

interface GigabitEthernet0/0/1

ip address 192.168.1.2 255.255.255.252

negotiation auto

interface GigabitEthernet0

vrf forwarding Mgmt-intf

no ip address

shutdown

negotiation auto

ip forward-protocol nd

ip http server

ip http authentication local

ip http secure-server

ip tftp source-interface GigabitEthernet0

ip route 10.0.1.0 255.255.255.0 GigabitEthernet0/0/1 192.168.1.1

control-plane

line con 0

transport input none

stopbits 1

line aux 0

stopbits 1

line vty 0 4

login

end

Conclusion

In this lab, we configured *policy-based routing* to secure two servers in an isolated network. Access lists are a big part of modern security, so it was good to implement them ourselves. In the future I may try to do this lab again but with only one router.