



# Implementing Routing Policy on Cisco IOS XR Software

---

A routing policy instructs the router to inspect routes, filter them, and potentially modify their attributes as they are accepted from a peer, advertised to a peer, or redistributed from one routing protocol to another. Routing protocols make decisions to advertise, aggregate, discard, distribute, export, hold, import, redistribute and otherwise modify routes based on configured routing policy.

The routing policy language (RPL) has been designed to provide a single, straightforward language in which all routing policy needs can be expressed. RPL was designed to support large-scale routing configurations. It greatly reduces the redundancy inherent in previous routing policy configuration methods. RPL has been designed to streamline routing policy configuration, to reduce system resources required to store and process these configurations, and to simplify troubleshooting.



## Note

For a complete description of the RPL commands used in this chapter, refer to the “Routing Policy Language Commands on Cisco IOS XR Software” module of the *Cisco IOS XR Routing Command Reference* publication. To locate documentation of other commands that appear in this chapter, use the command reference master index, or search online.

## Feature History for Implementing Routing Policy on Cisco IOS XR Software

Release	Modification
Release 2.0	This feature was introduced.

## Contents

- [Prerequisites for Implementing Routing Policy, page 146](#)
- [Restrictions for Implementing Routing Policy, page 146](#)
- [Information About Implementing Routing Policy, page 146](#)
- [How to Implement Routing Policy, page 170](#)
- [Configuration Examples for Implementing Routing Policy, page 174](#)
- [Additional References, page 183](#)

## Prerequisites for Implementing Routing Policy

The following are prerequisites for implementing Routing Policy on Cisco IOS XR Software:

- You must be a member of a user group associated with the proper task IDs for routing commands. Task IDs for commands are listed in the *Cisco IOS XR Task ID Reference Guide*.
- Border Gateway Protocol (BGP), integrated Intermediate System-to-Intermediate System (IS-IS), or Open Shortest Path First (OSPF) must be configured in your network.

## Restrictions for Implementing Routing Policy

IPv6 addresses and prefixes are supported only for the BGP protocol. Prefix sets may contain prefix specifications for both IPv4 and IPv6 using dotted-decimal and colon-separated hexadecimal formats, respectively. However, IPv6 matching on destination, source, and next hop and setting of IPv6 next hops is only supported at BGP attach points.

## Information About Implementing Routing Policy

To implement RPL, you need to understand the following concepts:

- [Comparison of Cisco IOS Route Maps and Cisco IOS XR Routing Policy Language, page 147](#)
- [Routing Policy Language, page 147](#)
- [Routing Policy Configuration Basics, page 152](#)
- [Policy Definitions, page 152](#)
- [Parameterization, page 153](#)
- [Semantics of Policy Application, page 154](#)
- [Policy Statements, page 158](#)
- [Attach Points, page 162](#)
- [Attached Policy Modification, page 169](#)
- [Nonattached Policy Modification, page 169](#)

## Comparison of Cisco IOS Route Maps and Cisco IOS XR Routing Policy Language

Route maps and RPL are both mechanisms to define routing policy so that routes are filtered and their attributes are potentially modified. Route maps and RPL differ in the ways that they allow routing policy to be expressed. Some of these key differences are:

- Traditionally, route maps consider each clause in order until a successful set of matching criteria occur. When a match happens, the corresponding set actions associated with that clause take effect. If no matches successfully occur, then the route is dropped.

In contrast, RPL has no distinct match clauses. Statements are processed in order from beginning to end. All statements are executed unless a drop statement is reached, which indicates that the route should be explicitly discarded and execution of the policy may stop. If a drop statement has not been processed by the end of the policy, and no attempt has been made to modify the route, the route is dropped.

- In a route policy, it is possible to selectively and conditionally set an attribute to one of several values. Thus it becomes possible, for example, to set the default local preference value at the beginning of a route policy and then later override that value under a selective set of conditions.

In route maps, each clause maps to one specific set of match criteria and each clause allows only one set statement for each attribute type.

- Set statements in RPL can behave in an additive manner, for example, set med +5.
- Cisco IOS XR software uses microemacs as an alternative editor to command line editing, which allows a user to edit an existing RPL object by name or create a new one.

## Routing Policy Language

This section contains the following information:

- [Routing Policy Language Overview, page 147](#)
- [Routing Policy Language Structure, page 148](#)
- [Routing Policy Language Components, page 151](#)

## Routing Policy Language Overview

RPL was developed in an effort to support large-scale routing configurations. RPL has several fundamental capabilities that differ from those present in configurations oriented to traditional route maps, access lists, and prefix lists. The first of these capabilities is the ability to build policies in a modular form. Common blocks of policy can be defined and maintained independently. These common blocks of policy can then be applied from other blocks of policy to build complete policies. This capability reduces the amount of configuration information that needs to be maintained. In addition, these common blocks of policy can be parameterized. This parameterization allows for policies that share the same structure but differ in the specific values that are set or matched against to be maintained as independent blocks of policy. For example, three policies that are identical in every way except for the local preference value they set can be represented as one common parameterized policy that takes the varying local preference value as a parameter to the policy.

The policy language introduces the notion of sets. Sets are containers of similar data that can be used in route attribute matching and setting operations. There are four set types: prefix-sets, community-sets, as-path-sets, and extcommunity-sets. These sets hold groupings of IPv4 or IPv6 prefixes, community

values, AS-path regular expressions, and extended community values, respectively, and are analogous to prefix lists, community lists, AS-path lists, and extended community lists from traditional Cisco IOS configuration with one significant difference: Sets do not encapsulate the notion of accept and deny, which are present in their traditional counterparts. Sets are simply containers of data. Most sets also have an inline variant. An inline set allows for small enumerations of values to be used directly in a policy rather than having to refer to a named set. Prefix lists, community lists, and AS-path lists must be maintained even when only one or two items are in the list. An inline set in RPL allows the user to place small sets of values directly in the policy body without having to refer to a named list.

Decision making, such as accept and deny, is explicitly controlled by the policy definitions themselves. RPL combines matching operators, which may use set data, with the traditional boolean logic operators *and*, *or*, and *not* into complex conditional expressions. All matching operations return either a simple true or false result. The execution of these conditional expressions and their associated actions can then be controlled by using simple *if then*, *elseif*, and *else* structures, which allow the evaluation paths through the policy to be fully specified by the user.

## Routing Policy Language Structure

This section describes the basic structure of RPL.

### Names

The policy language provides two kinds of persistent, namable objects: sets and policies. Definition of these objects is bracketed by beginning and ending command lines. For example, to define a policy named *test*, the configuration syntax would look similar to the following:

```
route-policy test
  [ . . . policy statements . . . ]
end-policy
```

Legal names for policy objects can be any sequence of the upper- and lowercase alphabetic characters; the numerals 0 to 9; and the punctuation characters period, hyphen, and underscore. A name must begin with a letter or numeral.

### Sets

In this context, the term set is used in its mathematical sense to mean an unordered collection of unique elements. The policy language provides sets as a container for groups of values for matching purposes. Sets are used in conditional expressions. The elements of the set are separated by commas. Null (empty) sets are not allowed.

There are four kinds of sets: [as-path-set](#), [community-set](#), [extcommunity-set](#), and [prefix-set](#). You may want to perform comparisons against a small number of elements, such as two or three community values, for example. To allow for these comparisons, the user can enumerate these values directly. These enumerations are referred to as *inline sets*. Functionally, inline sets are equivalent to named sets, but allow for simple tests to be inline. Thus, comparisons do not require that a separate named set be maintained when only one or two elements are being compared. See the set types described in the following sections for the syntax. In general, the syntax for an inline set is a comma-separated list surrounded by parentheses as follows: (<element-entry>,<element-entry>,<element-entry>, ...<element-entry>), where <element-entry> is an entry of an item appropriate to the type of usage such as a prefix or a community value.

The following is an example using an inline community set:

```
route-policy sample-inline
  if community matches-any (10:100, 20:100) then
    set local-preference 100
```

```

    endif
end-policy

```

The following is an equivalent example using the named set test-communities:

```

community-set test-communities
    10:100,
    20:100
end-set

route-policy sample
    if community matches-any test-communities then
        set local-preference 100
    endif
end-policy

```

Both of these policies are functionally equivalent, but the inline form does not require the configuration of the community set just to store the two values. The user can choose the form appropriate to the configuration context. In the following sections, examples of both the named set version and the inline form are provided where appropriate.

## as-path-set

An as-path-set comprises operations for matching an AS-path attribute. The only matching operation is a regular expression match, compatible with the as-regexp provided by Cisco IOS software in the **ip as-path access-list** command.

### Named Set Form

The named set form uses the **ios-regex** keyword to indicate the type of regular expression, in this case one compatible with those provided by Cisco IOS software in the **ip as-path access-list** command, and requires single quotes around the regular expression.

The following is a sample definition of a named as-path-set:

```

as-path-set aset1
    ios-regex '_42$',
    ios-regex '_127$'
end-set

```

This is an as-path-set composed of two elements. When used in a matching operation, this as-path-set will match any route whose AS-path ends with either the autonomous system (AS) number 42 or the autonomous system number 127.

To remove the named as-path-set, use the **no as-path-set aset1** CLI command.

### Inline Set Form

The inline set form is a parenthesized list of comma-separated expressions, as follows:

```
(ios-regex '_42$', ios-regex '_127$')
```

This set matches the same AS-paths as the above-named set, but does not require the extra effort of creating a named set separate from the policy that uses it.

## community-set

A community-set holds community values for matching against the BGP community attribute. A community is a 32-bit quantity. Integer community values *must* be split in half, and expressed as two unsigned decimal integers in the range from 0 to 65535, separated by a colon. Single 32-bit community values are not allowed. The following is the named set form:

**Named Set Form**

```
community-set cset1
  12:34,
  12:56,
  12:78,
  internet
end-set
```

**Inline Set Form**

```
(12:34, 12:56, 12:78)
($as:34, $as:$tag1, 12:78, internet)
```

The inline form of a community-set also supports parameterization. Each 16-bit portion of the community may be parameterized. See the [Parameterization](#) section for more information.

RPL provides symbolic names for the standard well-known community values: internet is 0:0, no-export is 65535:65281, no-advertise is 65535:65282, and local-as is 65535:65283.

RPL also provides a facility for using *wildcards* in community specifications. A wildcard is specified by inserting an asterisk (\*) in place of one of the 16-bit portions of the community specification; the wildcard indicates that any value for that portion of the community will match. Thus, the following policy matches all communities where the autonomous system part of the community is 123:

```
community-set cset3
  123:*
end-set
```

Every community set must contain at least one community value. Empty community sets are illegal and will be rejected.

**extcommunity-set**

An extended community-set is analogous to a community-set except that it contains extended community values instead of regular community values. It also supports named forms and inline forms. The following are syntactic examples:

**Named Form**

```
extcommunity-set extcomm-set1
  RT:1.2.3.4:666,
  RT:1234:666,
  SoO:1.2.3.4:777,
  SoO :4567:777
end-set
```

**Inline Form**

```
(RT:1.2.3.4:666, RT:1234:6667, SoO:1.2.3.4:777, SoO:45678:777)
(RT:$ipaddr:666, RT:1234:$tag, SoO:1.2.3.4:777, SoO:$tag2:777)
```

As with community sets, the inline form supports parameterization within parameterized policies. Either portion of the extended community value can be parameterized.

**Note**


---

Parameterization of the extended community type, RT (route-target) and SoO (site of origin), is not supported. Also, wildcarding of extended communities is not currently supported.

---

Every extended community-set must contain at least one extended community value. Empty extended community-sets are illegal and will be rejected.

## prefix-set

A prefix-set holds IPv4 or IPv6 prefix match specifications, each of which has four parts: an address, a mask length, a minimum matching length, and a maximum matching length. The address is required, but the other three parts are optional. The address is a standard dotted-numeric IPv4 or IPv6 address. The mask length, if present, is a nonnegative decimal integer in the range from 0 to 32 (0 to 128 for IPv6) following the address and separated from it by a slash mark. The optional minimum matching length follows the address and optional mask length and is expressed as the keyword **ge** (mnemonic for **g**reater than or **e**qual to), followed by a nonnegative decimal integer in the range from 0 to 32 (0 to 128 for IPv6). The optional maximum matching length follows the rest and is expressed by the keyword **le** (mnemonic for **l**ess than or **e**qual to), followed by yet another nonnegative decimal integer in the range from 0 to 32 (0 to 128 for IPv6). A syntactic shortcut for specifying an exact length for prefixes to match is the **eq** keyword (mnemonic for **e**qual to).

If a prefix match specification has no mask length, then the default mask length is 32 for IPv4 and 128 for IPv6. The default minimum matching length is the mask length. If a minimum matching length is specified, then the default maximum matching length is 32 for IPv4 and 128 for IPv6. Otherwise, if neither minimum nor maximum is specified, the default maximum is the mask length.

The prefix-set itself is a comma-separated list of prefix match specifications. The following is an example:

```
prefix-set legal-prefix-examples
  10.0.1.1,
  10.0.2.0/24,
  10.0.3.0/24 ge 28,
  10.0.4.0/24 le 28,
  10.0.5.0/24 ge 26 le 30,
  10.0.6.0/24 eq 28
end-set
```

The first element of the prefix-set will match only one possible value, 10.0.1.1/32 or the host address 10.0.1.1. The second element will match only one possible value, 10.0.2.0/24. The third element will match a range of prefix values, from 10.0.3.0/28 to 10.0.3.255/32. The fourth element will match a range of values, from 10.0.4.0/24 to 10.0.4.240/28. The fifth element matches prefixes in the range from 10.0.5.0/26 to 10.0.5.252/30. The sixth element will match any prefix of length 28 in the range from 10.0.6.0/28 through 10.0.6.240/28.

The following prefix-set consists entirely of illegal prefix match specifications:

```
prefix-set ILLEGAL-PREFIX-EXAMPLES
  10.1.1.1 ge 16,
  10.1.2.1 le 16,
  10.1.3.0/24 le 23,
  10.1.4.0/24 ge 33,
  10.1.5.0/25 ge 29 le 28
end-set
```

Neither the minimum length nor the maximum length is legal without a mask length. The maximum length must be at least the mask length. For IPv4, the minimum length must be less than 32, the maximum length of an IPv4 prefix. For IPv6, the minimum length must be less than 128, the maximum length of an IPv6 prefix. The maximum length must be equal to or greater than the minimum length.

## Routing Policy Language Components

Four main components in the routing policy language are involved in defining, modifying, and using policies: the configuration front end, the policy repository, the execution engine, and the policy clients themselves.

The configuration front end (CLI) is the mechanism to define and modify policies. This configuration is then stored on the router using the normal storage means and can be displayed using the normal configuration **show** commands.

The second component of the policy infrastructure, the policy repository, has several responsibilities. First, it compiles the user-entered configuration into a form that the execution engine can understand. Second, it carries out much of the verification of policies; it ensures that defined policies can actually be executed properly. Third, it tracks which attach points are using which policies so that when policies are modified the appropriate clients are properly updated with the new policies relevant to them.

The third component is the execution engine. This component is the piece that actually runs policies as the clients request. The process can be thought of as receiving a route from one of the policy clients and then executing the actual policy against the specific route data.

The fourth component is the policy clients (the routing protocols). This component calls the execution engine at the appropriate times to have a given policy be applied to a given route, and then carry out some number of actions. These actions may include deleting the route if policy indicated that it should be dropped, passing along the route to the protocol decision tree as a candidate for the best route, or advertising a policy modified route to a neighbor or peer as appropriate.

## Routing Policy Configuration Basics

Route policies comprise of a series of statements and expressions that are bracketed with the **route-policy** and **end-policy** keywords. Rather than a collection of individual commands (one per line), the statements within a route policy have context relative to each other. Thus, instead of each line being an individual command, each policy or set is an independent configuration object that can be used, entered, and manipulated as a unit.

Each line of a policy configuration is a logical subunit. At least one new line must follow the **then**, **else**, and **end-policy** keywords. A new line must also follow the closing parenthesis of a parameter list and the name string in a reference to an AS-path set, community set, extended community set, or prefix set. At least one new line must precede the definition of a route policy, AS-path set, community set, extended community set, or prefix set. One or more new lines can follow an action statement. One or more new lines can follow a comma separator in a named AS-path set, community set, extended community set, or prefix set. A new line must appear at the end of a logical unit of policy expression, and may not appear anywhere else.

## Policy Definitions

Policy definitions create named sequences of policy statements. A policy definition consists of the CLI **route-policy** keyword followed by a name, a sequence of policy statements, and the **end-policy** keyword. For example, the following policy will drop any route it encounters:

```
route-policy drop-everything
    drop
end-policy
```

The name serves as a handle for binding the policy to protocols. To remove a policy definition, issue the **no route-policy name** command.

Policies may also refer to other policies such that common blocks of policy can be reused. This reference to other policies is accomplished by using the **apply** statement, as shown in the following example:

```
route-policy check-as-1234
    if as-path passes-through '1234' then
        apply drop-everything
```

```
    else
      pass
    endif
  end-policy
```

The **apply** statement indicates that we should execute the policy drop-everything if the route under consideration passed through autonomous system 1234 before we received it. If a route that has autonomous system 1234 in its AS-path is received, the route is dropped; otherwise, the route is accepted without modification. This policy is an example of a hierarchical policy. Thus the semantics of the **apply** statement are just as if the applied policy were cut and pasted into the applying policy:

```
route-policy check-as-1234-prime
  if as-path passes-through '1234' then
    drop
  else
    pass
  endif
end-policy
```

You may have as many levels of hierarchy as desired. However, many levels may be difficult to maintain and understand.

## Parameterization

In addition to supporting reuse of policies using the **apply** statement, policies can be defined that allow for parameterization of some of the attributes. The following example defines a parameterized policy named param-example. In this case, the policy takes one parameter \$mytag. Parameters always begin with a dollar sign, and consist otherwise of any alphanumeric characters. Parameters can be substituted into any attribute that takes a parameter.

In the following example, a 16-bit community tag is used as a parameter:

```
route-policy param-example ($mytag)
  set community (1234:$mytag) additive
end-policy
```

This parameterized policy can then be reused with different parameterizations as shown in the following example. In this manner, policies that share a common structure but use different values in some of their individual statements can be modularized. For details on which attributes can be parameterized, see the individual attribute sections.

```
route-policy origin-10
  if as-path originates-from '10' then
    apply param-example(10)
  else
    pass
  endif
end-policy

route-policy origin-20
  if as-path originates-from '20' then
    apply param-example(20)
  else
    pass
  endif
end-policy
```

The parameterized policy `param-example` provides a policy definition that is expanded with the values provided as the parameters in the `apply` statement. Note that the policy hierarchy is always maintained. Thus, if the definition of `param-example` changes, then the behavior of `origin_10` and `origin_20` will change to match.

The effect of the `origin-10` policy is that it adds the community `1234:10` to all routes that pass through this policy and have an AS-path indicating the route originated from autonomous system 10. The `origin-20` policy is similar except that it adds to community `1234:20` for routes originating from autonomous system 20.

## Semantics of Policy Application

This section discusses how routing policies are evaluated and applied. The following concepts are discussed:

- [Boolean Operator Precedence, page 154](#)
- [Multiple Modifications of the Same Attribute, page 154](#)
- [When Attributes Are Modified, page 155](#)
- [Default Drop Disposition, page 156](#)
- [Control Flow, page 156](#)
- [Policy Verification, page 157](#)

### Boolean Operator Precedence

Boolean expressions are evaluated in order of operator precedence, from left to right. The highest precedence operator is *not*, followed by *and*, and then *or*. The following expression:

```
med eq 10 and not destination in (10.1.3.0/24) or community matches-any ([10..25]:35)
```

if fully parenthesized to display the order of evaluation would look like this:

```
(med eq 10 and (not destination in (10.1.3.0/24))) or community matches-any ([10..25]:35)
```

The inner *not* applies only to the destination test; the *and* combines the result of the *not* expression with the Multi Exit Discriminator (MED) test; and the *or* combines that result with the community test. If the order of operations are rearranged:

```
not med eq 10 and destination in (10.1.3.0/24) or community matches-any ([10..25]:35)
```

then the expression, fully parenthesized, would look like the following:

```
((not med eq 10) and destination in (10.1.3.0/24)) or community matches-any ([10..25]:35)
```

### Multiple Modifications of the Same Attribute

When a policy replaces the value of an attribute multiple times, the last assignment wins because all actions are executed. Because the MED attribute in BGP is one unique value, the last value it gets set to wins. Therefore, the following policy results in a route with a MED value of 12:

```
set med 9
set med 10
set med 11
set med 12
```

This example is trivial, but the feature is not. It is possible to write a policy that effectively changes the value for an attribute. For example:

```
set med 8
if community matches-any cs1 then
  set local-preference 122
  if community matches-any cs2 then
    set med 12
  endif
endif
```

The result is a route with a MED of 8, unless the community list of the route matches both cs1 and cs2, in which case the result is a route with a MED of 12.

In the case where the attribute being modified can contain only one value, it is easy to think of this case as the last statement wins. However, there are a few attributes that can contain multiple values and the result of multiple actions on the attribute is accumulative rather than a replacement. The first of these cases is the use of the **additive** option on community and extended community evaluation. Consider a policy of the form:

```
route-policy community-add
  set community (10:23)
  set community (10:24) additive
  set community (10:25) additive
end-policy
```

This policy sets the community string on the route to contain all three community values: 10:23, 10:24, and 10:25.

The second of these cases is AS-path prepending. Consider a policy of the form:

```
route-policy prepend-example
  prepend as-path 2 3
  prepend as-path 666 2
end-policy
```

This policy prepends the following to the AS-path (666 666 2 2 2). This prepending is a result of all actions being taken and to AS-path being an attribute that contains an array of values rather than a simple scalar value.

## When Attributes Are Modified

A policy does not modify route attribute values until all tests have completed. In other words, comparison operators always run on the initial data in the route. Intermediate modifications of the route attributes will not have a cascading effect on the evaluation of the policy. Take the following example:

```
if med eq 12 then
  set med 42
  if med eq 42 then
    drop
  endif
endif
```

This policy will never execute the drop statement because the second test (med eq 42) sees the original, unmodified value of the MED in the route. Because the MED had to be 12 to get to the second test, the second test will always return false.

## Default Drop Disposition

All route policies have a default action to drop the route under evaluation unless the route has been modified by a policy action or explicitly passed. Applied (nested) policies implement this disposition as though the applied policy were pasted into the point where it is applied.

Consider a policy to allow all the routes in the 10 net and set their local preference to 200 while dropping all other routes. You might write the policy as follows:

```
route-policy two
  if destination in (10.0.0.0/8 ge 8 le 32) then
    set local-preference 200
  endif
end-policy

route-policy one
  apply two
end-policy
```

It may appear that policy one will drop all routes, because it neither contains an explicit **pass** statement nor modifies a route attribute. However, the applied policy does set an attribute for some routes and this disposition is passed along to policy one. The result is that policy one passes routes with destinations in network 10, and drops all others.

## Control Flow

Policy statements are processed sequentially in the order in which they appear in the configuration. Policies that hierarchically reference other policy blocks are processed as if the referenced policy blocks had been directly substituted inline. For example, if the following policies are defined:

```
route-policy one
  set weight 100
end-policy

route-policy two
  set med 200
end-policy

route-policy three
  apply two
  set community (2:666) additive
end-policy

route-policy four
  apply one
  apply three
  pass
end-policy
```

Policy four could be rewritten in an equivalent way as follows:

```
route-policy four-equivalent
  set weight 100
  set med 200
  set community (2:666) additive
  pass
end-policy
```



### Note

The **pass** statement is not required and can be removed to represent the equivalent policy in another way.

## Policy Verification

Several different types of verification occur when policies are being defined and used.

### Range Checking

As policies are being defined, some simple verifications, such as range checking of values, is done. For example, the MED that is being set is checked to verify that it is in a proper range for the MED attribute. However, this range checking cannot cover parameter specifications, because they may not have defined values yet. These parameter specifications are verified when a policy is attached to an attach point. The policy repository also verifies that there are no recursive definitions of policy, and that parameter numbers are correct. At attach time, all policies must be well-formed. All sets and policies that they reference must be defined and have valid values. Likewise, any parameter values must also be in the proper ranges.

### Incomplete Policy and Set References

As long as a given policy is not attached at an attach point, the policy is allowed to refer to nonexistent sets and policies, which allows for freedom of workflow. You can build configurations that reference sets or policy blocks that are not yet defined, and then can later fill in those undefined policies and sets, thereby achieving much greater flexibility in policy definition. Every piece of policy you want to reference while defining a policy need not exist in the configuration. Thus, a user can define a policy sample that references the policy bar via an **apply** statement even if the policy bar does not exist. Similarly, a user can enter a policy statement that refers to a nonexistent set.

However, the existence of all referenced policies and sets is enforced when a policy is attached. If you attempt to attach the policy sample with the reference to an undefined policy bar at an inbound BGP policy using the **neighbor 1.2.3.4 address-family ipv4 unicast policy sample in** command, the configuration attempt will be rejected because the policy bar does not exist.

Likewise, you cannot remove a route policy or set that is currently in use at an attach point because this removal would result in an undefined reference. An attempt to remove a route policy or set that is currently in use will result in an error message to the user.

There is a condition referred to as a null policy where the policy bar could exist but have no statements, actions, or dispositions in it. In other words, policy bar could exist as follows:

```
route-policy bar
end-policy
```

This is a valid policy block. It would effectively force all routes to be dropped, because it is a policy block that will never modify a route, nor does it include the pass statement. Thus the default action of drop for the policy block will be followed.

### Attached Policy Modification

Policies that are in use will, on occasion, need to be modified. Traditionally, configuration changes were done by completely removing the relevant configuration and then reentering it. However, this allows for a window of time in which no policy is attached and the default action takes place. RPL provides a mechanism for an atomic change so that if a policy is redeclared, or edited using the emacs editor, the new configuration is applied immediately, which allows for policies that are in use to be changed without having a window of time where no policy is applied at the given attach point.

## Verification of Attribute Comparisons and Actions

The policy repository knows which attributes, actions, and comparisons are valid at each attach point. When a policy is attached, these actions and comparisons are verified against the capabilities of that particular attach point. Take, for example, the following policy definition:

```
route-policy bad
  set med 100
  set level level-1-2
  set cost 200
end-policy
```

This policy attempts to perform actions to set the BGP attribute med, the IS-IS attribute level, and the OSPF attribute cost. The system will allow you to define such a policy, but it will not allow you to attach such a policy. If you had defined the policy bad and then attempted to attach it as an inbound BGP policy using the BGP configuration statement **neighbor 1.2.3.4 address-family ipv4 unicast policy bad** in the system would reject this configuration attempt. This rejection is the result of the verification process checking the policy and realizing that while BGP could set the MED, it has no way of setting the level or the cost as those are attributes of IS-IS and OSPF, respectively. Instead of silently omitting the actions that cannot be done, the system generates an error to the user. Likewise, a valid policy in use at an attach point cannot be modified in such a way as to introduce an attempt to modify a nonexistent attribute or to compare against a nonexistent attribute. The verifiers test for nonexistent attributes and will reject such a configuration attempt.

## Policy Statements

There are four types of policy statements: remark, disposition (drop and pass), action (set), and if (comparator).

### Remark

A remark is text attached to policy configuration but otherwise ignored by the policy language parser. Remarks can be useful for documenting parts of a policy. The syntax for a remark is text prepended with pound signs (#):

```
# This is a simple one-line remark.

# This
# is a remark
# comprising multiple
# lines.
```

In general, remarks are used between complete statements or elements of a set. Remarks are not supported in the middle of statements or within an inline set definition.

### Disposition

By default, a route is **dropped** at the end of policy processing unless either the policy **modifies** a route attribute or it passes the route by means of an explicit **pass** statement. For example, the following policy drops all routes because it neither modifies the attribute of any route nor explicitly passes it.

```
route-policy EMPTY
end-policy
```

Whereas the following policies pass all routes that they evaluate.

```
route-policy PASS-ALL
  pass
end-policy
```

```
route-policy SET-LPREF
  set local-preference 200
end-policy
```

In addition to being implicitly dropped, a route may be dropped by means of an **explicit drop** statement. **Drop** statements cause a route to be dropped immediately so that no further policy processing is done. Note also, that a **drop** statement overrides any previously processed **pass** statements or attribute modifications. For example, the following policy will drop all routes. The first **pass** statement is executed, but is then immediately overridden by the **drop** statement. The second **pass** statement never gets executed.

```
route-policy DROP-EXAMPLE
  pass
  drop
  pass
end-policy
```

When one policy applies another, it is as if the applied policy were copied into the right place in the applying policy, and then the same drop-and-pass semantics are put into effect. For example, policies ONE and TWO are equivalent to policy ONE-PRIME:

```
route-policy ONE
  apply route-policy two
  if as-path neighbor-is '123' then
    pass
  endif
end-policy

route-policy TWO
  if destination in (10.0.0.0/16 le 32) then
    drop
  endif
end-policy

route-policy ONE-PRIME
  if destination in (10.0.0.0/16 le 32) then
    drop
  endif
  if as-path neighbor-is '123' then
    pass
  endif
end-policy
```

Because the effect of an **explicit drop** statement is immediate, routes in 10.0.0.0/16 le 32 are dropped without any further policy processing. Other routes are then considered to see if they were advertised by Autonomous System 123. If they were advertised, they are passed; otherwise, they are implicitly dropped at the end of all the policy processing.

## Action

An action is a sequence of primitive operations that modify a route. Most actions, but not all, are distinguished by the **set** keyword. In a route policy, actions can be grouped together. For example, the following is a route policy comprising three actions:

```
route-policy actions
  set med 217
```

```

    set community (12:34) additive
    delete community in (12:56)
end-policy

```

## If

In its simplest form, an *if* statement uses a conditional expression to decide which actions or dispositions should be taken for the given route. For example:

```

if as-path in as-path-set-1 then
    drop
endif

```

The example indicates that any routes whose AS-path is in the set as-path-set-1 will be dropped. The contents of the *then* clause may be an arbitrary sequence of policy statements.

The following example contains two action statements:

```

if origin is igp then
    set med 42
    prepend as-path 73 5
endif

```

The *if* statement also permits an *else* clause, which is executed if the if-condition is false:

```

if med eq 8 then
    set community (12:34) additive
else
    set community (12:56) additive
endif

```

The policy language also provides syntax, using the **elseif** keyword, to string together a sequence of tests:

```

if med eq 150 then
    set local-preference 10
elseif med eq 200 then
    set local-preference 60
elseif med eq 250 then
    set local-preference 110
else
    set local-preference 0
endif

```

The statements within an *if* statement may themselves be *if* statements, as shown in the following example:

```

if community matches-any ([12..34]:[56..78]) then
    if med eq 150 then
        drop
    endif
    set local-preference 100
endif

```

This policy example sets the value of the local preference attribute to 100 on any route that has a community value of 12:34 or 56:78 associated with it. However, if any of these routes has a MED value of 150, then these routes with either the community value of 12:34 or 56:78 and a MED of 150 are dropped.

## Boolean Conditions

In the previous section describing the *if* statement, all of the examples used simple Boolean conditions that evaluated to either true or false. RPL also provides a way to build compound conditions from simple conditions by means of Boolean operators.

There are three Boolean operators: negation (*not*), conjunction (*and*), and disjunction (*or*). In the policy language, negation has the highest precedence, followed by conjunction, and then by disjunction. Parentheses may be used to group compound conditions to override precedence or to improve readability.

The following simple condition:

```
med eq 42
```

will be true only if the value of the MED in the route is 42, otherwise it will be false.

A simple condition may also be negated using the *not* operator:

```
not next-hop in (10.0.2.2)
```

Any Boolean condition enclosed in parentheses is itself a Boolean condition:

```
(destination in prefix-list-1)
```

A compound condition takes either of two forms. It can be a simple expression followed by the *and* operator, itself followed by a simple condition:

```
med eq 42 and next-hop in (10.0.2.2)
```

A compound condition may also be a simpler expression followed by the *or* operator and then another simple condition:

```
origin is igp or origin is incomplete
```

An entire compound condition may be enclosed in parentheses:

```
(med eq 42 and next-hop in (10.0.2.2))
```

The parentheses may serve to make the grouping of subconditions more readable, or they may force the evaluation of a subcondition as a unit.

In the following example, the highest-precedence *not* operator applies only to the destination test, the *and* combines the result of the *not* expression with the community test, and the *or* combines that result with the MED test.

```
med eq 10 or not destination in (10.1.3.0/24) and community matches-any  
([12..34]:[56..78])
```

With a set of parentheses to express the precedence, the result is the following:

```
med eq 10 or ((not destination in (10.1.3.0/24)) and community matches-any  
([12..34]:[56..78]))
```

The following is another example of a complex expression:

```
(origin is igp or origin is incomplete or not med eq 42) and next-hop in (10.0.2.2)
```

The left conjunction is a compound condition enclosed in parentheses. The first simple condition of the inner compound condition tests the value of the origin attribute; if it is Interior Gateway Protocol (IGP), then the inner compound condition is true. Otherwise, the evaluation moves on to test the value of the origin attribute again, and if it is incomplete, then the inner compound condition is true. Otherwise, the evaluation moves to check the next component condition, which is a negation of a simple condition.

## apply

As discussed in the sections on policy definitions and parameterization of policies, the **apply** command is used to execute another policy (either parameterized or unparameterized) from within another policy, which allows for the reuse of common blocks of policy. When combined with the ability to parameterize common blocks of policy, the **apply** command becomes a powerful tool for reducing repetitive configuration.

## Attach Points

Policies do not become useful until they are applied to routes, and for policies to be applied to routes they need to be made known to routing protocols. In BGP, for example, there are several situations where policies can be used, the most common of these is defining import and export policy. The policy attach point is the point where an association is formed between a specific protocol entity, in this case a BGP neighbor, and a specific named policy. It is important to note that a verification step happens at this point. Each time a policy is attached, the given policy, and any policies it may apply, is checked to ensure that the policy can be validly used at that attach point. For example, if a user defines a policy that sets the IS-IS level attribute and then attempts to attach this policy as an inbound BGP policy, the attempt would be rejected because BGP routes do not carry IS-IS attributes. Likewise, when policies are modified that are in use, the attempt to modify the policy is verified against all the current uses of the policy to ensure that the modification is compatible with the current uses.

## BGP Policy Attach Points

This section describes each of the BGP policy attach points.

### Aggregation

The aggregation attach point generates an aggregate route to be advertised based on the conditional presence of subcomponents of that aggregate. Policies attached at this attach point are also able to set any of the valid BGP attributes on the aggregated routes. For example, the policy could set a community value or a MED on the aggregate that is generated. The specified aggregate will be generated if any routes evaluated by the named policy pass the policy. More specifics of the aggregate can be filtered using the **suppress-route** keyword. Any actions taken to set attributes in the route will affect attributes on the aggregate.

In traditional Cisco IOS route map configuration, the configuration was controlled using three route maps: the advertise-map, the suppress-map, and the attribute-map. The advertise-map was used to select the component routes used to build various attributes of the aggregate. In the policy language the configuration is controlled by which routes pass the policy. The suppress map was used to selectively filter or suppress specific components of the aggregate when the summary-only flag is not set. In other words, when the aggregate and more specific components are being sent, some of the more specific components can be filtered out using a suppress map. In the policy language, this is controlled by selecting the route and setting the suppress flag. The attribute-map allowed the user to set specific attributes on the aggregated route. In the policy language, setting attributes on the aggregated route is controlled by normal action operations.

In the following example, the aggregate address 10.0.0.0/8 will be generated if there are any component routes in the range 10.0.0.0/8 ge 8 le 25 except for 10.2.0.0/24. Because summary-only is not set, all components of the aggregate will be advertised. However, the specific component 10.1.0.0 will be suppressed.

```
route-policy sample
```

```

    if destination in (10.0.0.0/8 ge 8 le 25) then
        set community (10:33)
    endif
    if destination in (10.2.0.0/24) then
        drop
    endif
    if destination in (10.1.0.0/24) then
        suppress-route
    endif
end-policy

router bgp 2
address-family ipv4
    aggregate-address 10.0.0.0/8 policy sample
    .
    .
    .

```

## Dampening

The dampening attach point controls the default route-dampening behavior within BGP. Unless overridden by a more specific policy on the associate peer, all routes in BGP will apply the associated policy to set their dampening attributes.

The following policy sets dampening values for BGP IPv4 unicast routes. Those routes more specific than a /25 will take longer to recover once they have been dampened than routes that are less specific than /25.

```

route-policy sample_damp
    if destination in (0.0.0.0/0 ge 25) then
        set dampening halflife 30 others default
    else
        set dampening halflife 20 others default
    endif
end-policy

router bgp 2
    address-family ipv4 unicast
        bgp dampening policy sample_damp
    .
    .
    .

```

## Default Originate

The default originate attach point allows the default route (0.0.0.0/0) to be conditionally generated and advertised to a peer, based on the presence of other routes. It accomplishes this configuration by evaluating the associated policy against routes in the Routing Information Base (RIB). If any routes pass the policy, the default route is generated and sent to the relevant peer.

The following policy will generate and send a default-route to the BGP neighbor 10.0.0.1 if any routes that match 10.0.0.0/8 ge 8 le 32 are present in the RIB.

```

route-policy sample-originate
    if rib-has-route in (10.0.0.0/8 ge 8 le 32) then
        pass
    endif
end-policy

router bgp 2
    neighbor 10.0.0.1
        remote-as 3

```

```

address-family ipv4 unicast
default-originate policy sample-originate
.
.
.

```

**Note**


---

The current implementation of default origination policy permits matching only on destination address.

---

**Neighbor Export**

The neighbor export attach point is used to select the BGP routes to send to a given peer or group of peers. The routes are selected by running the set of possible BGP routes through the associated policy. Any routes that pass the policy are then sent as updates to the peer or group of peers. The routes that are sent may have had their BGP attributes altered by the policy that has been applied.

The following policy sends all BGP routes to neighbor 10.0.0.5. Routes that are tagged with the community 2:100 will be sent with a MED of 100 and a community of 2:666. The rest of the routes will be sent with a MED of 200 and a community of 2:200.

```

route-policy sample-export
  if community matches-any ([12..34]:[56..78]) then
    set med 100
    set community (2:666)
  else
    set med 200
    set community (2:200)
  endif
end-policy

router bgp 2
  neighbor 10.0.0.5
  remote-as 3
  address-family ipv4 unicast
    policy sample-export out
  .
  .
  .

```

**Neighbor Import**

The neighbor import attach point controls the reception of routes from a specific peer. All routes that are received by a peer are run through the attached policy. Any routes that pass the attached policy are passed to the BGP Routing Information Base (BRIB) as possible candidates for selection as best path routes.

When a BGP import policy is modified, it is necessary to rerun all the routes that have been received from that peer against the new policy. The modified policy may now discard routes that were previously allowed through, allow through previously discarded routes, or change the way the routes have been modified. There is a new configuration option in BGP (**bgp auto-policy-soft-reset**) that allows this modification to happen automatically in cases where either soft reconfiguration is configured or the BGP route-refresh capability has been negotiated.

The following example receives routes from neighbor 10.0.0.1. Any routes received with the community 3:100 will have their local preference set to 100 and a community tag of 2:666. All other routes received from this peer will have their local preference values set to 200 and their community values set to 2:200.

```

route-policy sample_import
  if community matches-any ([10..20]:35) then
    set local-preference 100
    set community (2:666)
  else

```

```

        set local-preference 200
        set community (2:200)
    endif
end-policy

router bgp 2
  neighbor 10.0.0.1
    remote-as 3
    address-family ipv4 unicast
      policy sample_import in
      .
      .
      .

```

## Network

The network attach point controls the injection of routes from the RIB into BGP. A route policy attached at this point is able to set any of the valid BGP attributes on the routes that are being injected.

The following is an example of a route policy attached at the network attach point that sets the well-known community no-export for any routes more specific than /24:

```

route-policy NetworkControl
  if destination in (0.0.0.0/0 ge 25) then
    set community (no-export) additive
  endif
end-policy

router bgp 2
  address-family ipv4 unicast
    network 172.16.0.5/27 policy NetworkControl

```

## Redistribute

The redistribute attach point allows routes from other sources to be advertised by BGP. The policy attached at this point is able to set any of the valid BGP attributes on the routes that are being redistributed. Likewise, selection operators allow a user to control what route sources are being redistributed and which routes from those sources.

The following example redistributes all routes from OSPF instance 12 into BGP. If OSPF was carrying a default route, it will be dropped. Routes carrying a tag of 10 will have their local preference set to 300 and the community values of 2:666 and no-advertise attached. All other routes will have their local preference set to 200 and a community value of 2:100 set.

```

route-policy sample_redist
  if destination in (0.0.0.0/0) then
    drop
  endif
  if tag eq 10 then
    set local-preference 300
    set community (2:666, no-advertise)
  else
    set local-preference 200
    set community (2:100)
  endif
end-policy

router bgp 2
  address-family ipv4 unicast
    redistribute ospf 12 policy sample_redistribute
    .
    .

```

## Show bgp

The `show bgp attach point` allows the user to display selected BGP routes that pass the given policy. Any routes that are not dropped by the attached policy will be displayed in a manner similar to the output of the `show ip bgp` command.

In the following example, the `show bgp route-policy` command is used to display any BGP routes carrying a MED of 5:

```
route-policy sample-display
  if med eq 5 then
    pass
  endif
end-policy
!
show bgp route-policy sample-display
```

There is also a `show bgp policy route-policy` command, which runs all routes in the RIB past the named policy as if the RIB were an outbound BGP policy. This command then displays what each route looked like before it was modified and after it was modified, as shown in the following example:

```
RP/0/0/CPU0:router# show rpl policy test2

route-policy test2
  if (destination in (10.0.0.0/8 ge 8 le 32)) then
    set med 333
  endif
end-policy
!
RP/0/0/CPU0:router# show bgp

BGP router identifier 10.0.0.1, local AS number 2
BGP main routing table version 11
BGP scan interval 60 secs
Status codes:s suppressed, d damped, h history, * valid, > best
              i - internal, S stale
Origin codes:i - IGP, e - EGP, ? - incomplete
   Network        Next Hop           Metric LocPrf Weight Path
*> 10.0.0.0        10.0.1.2             10                0 3 ?
*> 10.0.0.0/9     10.0.1.2             10                0 3 ?
*> 10.0.0.0/10    10.0.1.2             10                0 3 ?
*> 10.0.0.0/11    10.0.1.2             10                0 3 ?
*> 10.1.0.0/16    10.0.1.2             10                0 3 ?
*> 10.3.30.0/24   10.0.1.2             10                0 3 ?
*> 10.3.30.128/25 10.0.1.2             10                0 3 ?
*> 10.128.0.0/9   10.0.1.2             10                0 3 ?
*> 10.255.0.0/24  10.0.101.2           1000             555             0 100 e
*> 10.255.64.0/24 10.0.101.2           1000             555             0 100 e
....

RP/0/0/CPU0:router# show bgp policy route-policy test2

10.0.0.0/8 is advertised to 10.0.101.2

Path info:
  neighbor:10.0.1.2      neighbor router id:10.0.1.2
  valid external best
Attributes after inbound policy was applied:
  next hop:10.0.1.2
  MET ORG AS
  origin:incomplete neighbor as:3 metric:10
  aspath:3
Attributes after outbound policy was applied:
```

```

next hop:10.0.1.2
MET ORG AS
origin:incomplete neighbor as:3 metric:333
aspath:2 3
...

```

## Table Policy

The table policy attach point allows the user to configure traffic-index values on routes as they are installed into the global routing table. This attach point supports the BGP policy accounting feature. BGP policy accounting uses the traffic indexes that are set on the BGP routes to track various counters. This way, router operators can select different sets of BGP route attributes using the matching operations and then set different traffic indexes for each different class of route they are interested in tracking.

The following example sets the traffic index to 10 in IPv4 unicast routes that originated from autonomous system 10. Likewise, any IPv4 unicast routes that originated from autonomous system 11 will have their traffic index set to 11 when they are installed into the FIB. These traffic indexes can then be used to count traffic being forwarded on these routes in line cards by enabling the BGP policy accounting counters on the interfaces of interest.

```

route-policy sample-table
  if as-path originates-from '10' then
    set traffic-index 10
  elseif as-path originates-from '11' then
    set traffic-index 11
  endif
end-policy
router bgp 2
  address-family ipv4 unicast
    table-policy sample-table
  .
  .
  .

```

## OSPF Policy Attach Points

This section describes each of the OSPF policy attach points.

### Default Originate

The default originate attach point allows the user to conditionally inject the default route 0.0.0.0/0 into the OSPF link-state database, which is done by evaluating the attached policy. If any routes in the local RIB pass the policy, then the default route is inserted into the link-state database.

The following example generates a default route if any of the routes that match 10.0.0.0/8 ge 8 le 25 are present in the RIB:

```

route-policy ospf-originate
  if rib-has-route in (10.0.0.0/8 ge 8 le 25) then
    pass
  endif
end-policy

router ospf 1
  default-information originate policy ospf-originate
  .
  .
  .

```

## Redistribute

The redistribute attach point within OSPF injects routes from other routing protocol sources into the OSPF link-state database, which is done by selecting the route types it wants to import from each protocol. It then sets the OSPF parameters of cost and metric type. The policy can control how the routes are injected into OSPF by using the **set level** command.

The following example redistributes routes from IS-IS instance instance\_10 into OSPF instance 1 using the policy OSPF-redist. The policy sets the metric type to type-2 for all redistributed routes. IS-IS routes with a tag of 10 will have their cost set to 100, and IS-IS routes with a tag of 20 will have their OSPF cost set to 200. Any IS-IS routes not carrying a tag of either 10 or 20 will not be redistributed into the OSPF link-state database.

```
route-policy OSPF-redist
  set metric-type type-2
  if tag eq 10 then
    set cost 100
  elseif tag eq 20 then
    set cost 200
  else
    drop
  endif
end-policy

router ospf 1
  redistribute isis instance_10 policy OSPF-redist
  .
  .
  .
```

## IS-IS Policy Attach Points

This section describes each of the IS-IS policy attach points.

### Redistribute

The redistribute attach point within IS-IS allows routes from other protocols to be readvertised by IS-IS. The policy is a set of control structures for selecting the types of routes that a user wants to redistribute into IS-IS. The policy can also control which IS-IS level the routes are injected into and at what metric values.

The following example redistributes routes from IS-IS instance 1 into IS-IS instance instance\_10 using the policy ISIS-redist. This policy sets the level to level-1-2 for all redistributed routes. OSPF routes with a tag of 10 will have their metric set to 100, and IS-IS routes with a tag of 20 will have their IS-IS metric set to 200. Any IS-IS routes not carrying a tag of either 10 or 20 will not be redistributed into the IS-IS database.

```
route-policy ISIS-redist
  set level level-1-2
  if tag eq 10 then
    set metric 100
  elseif tag eq 20 then
    set metric 200
  else
    drop
  endif
end-policy

router isis instance_10
  address-family ipv4 unicast
```

```

redistribute ospf 1 policy ISIS-redist
.
.
.

```

## Attached Policy Modification

Policies that are in use will, on occasion, need to be modified. In the traditional configuration model, a policy modification would be done by completely removing the policy and reentering it. However, this model allows for a window of time in which no policy is attached and default actions to be used, which is an opportunity for inconsistencies to exist. To close this window of opportunity, you can modify a policy in use at an attach point by respecifying it, which allows for policies that are in use to be changed, without having a window of time where no policy is applied at the given attach point.



### Note

A route policy or set that is in use at an attach point cannot be removed, because this removal would result in an undefined reference. An attempt to remove a route policy or set that is in use at an attach point will result in an error message to the user.

## Nonattached Policy Modification

As long as a given policy is not attached at an attach point, the policy is allowed to refer to nonexistent sets and policies. Configurations can be built that reference sets or policy blocks that are not yet defined, and then later those undefined policies and sets can be filled in. This method of building configurations gives much greater flexibility in policy definition. Every piece of policy you want to reference while defining a policy need not exist in the configuration. Thus you can define a policy sample1 that references the policy sample2 via an apply statement even if the policy sample2 does not exist. Similarly, you can enter a policy statement that refers to a nonexistent set.

However, the existence of all referenced policies and sets is enforced when a policy is attached. Thus, if a user attempts to attach the policy sample1 with the reference to an undefined policy sample2 at an inbound BGP policy using the statement **neighbor 1.2.3.4 address-family ipv4 unicast policy sample1 in**, the configuration attempt will be rejected because the policy sample2 does not exist.

## Editing Routing Policy Configuration Elements

RPL is based on statements rather than on lines. That is, within the begin-end pair that brackets policy statements from the CLI, a new line is merely a separator, on a par with a space character.

The CLI provides the means to enter and delete route policy statements. RPL provides a means to edit the contents of the policy between the begin-end brackets using a microemacs editor.

To edit the contents of a routing policy, use the following CLI command in EXEC mode:

```

edit {route-policy | prefix-set | as-path-set | community-set | extended-community-set}
name

```

A copy of the route policy is copied to a temporary file and the editor is launched. After editing, save the changes by using the **save-buffer** command, ^X^S (Control-X Control-S). To exit the editor, use the **quit** command, ^X^C. When you quit the editor, the policy object will be parsed. If there are no parse errors, a disposition query is displayed:

```

Successful parse of edited config.
Commit configuration? ('yes' or 'no'):

```

If you answer **yes**, the configuration is committed to the router. If you answer **no**, you are asked whether editing should continue:

```
Continue editing? ('yes' or 'no'):
```

If you answer **yes**, the editor continues on the text buffer from where you left off. If you answer **no**, the running configuration is not changed and the editing session is ended.

If there is a syntax error in the policy object, the following query is displayed:

```
parse error in edited config.
Continue editing? ('yes' or 'no'):
```

If you answer **yes**, the editing process is resumed. If you answer **no**, the candidate configuration element is abandoned.

## How to Implement Routing Policy

This section contains the following procedures:

- [Defining a Route Policy, page 170](#) (required)
- [Attaching a Routing Policy to a BGP Neighbor, page 171](#) (required)
- [Modifying a Routing Policy Using the Microemacs Editor, page 173](#) (optional)

## Defining a Route Policy

This task explains how to define a route policy.



### Note

---

If you want to modify an existing routing policy using the command-line interface (CLI), you must redefine the policy by completing this task.

---

### SUMMARY STEPS

1. **configure**
2. **route-policy** *name*
3. **end-policy**
4. **end**  
or  
**commit**

## DETAILED STEPS

	Command or Action	Purpose
Step 1	<b>configure</b>  <b>Example:</b> RP/0/RP0/CPU0:router# configure	Enters global configuration mode.
Step 2	<b>route-policy</b> <i>name</i>  <b>Example:</b> RP/0/RP0/CPU0:router(config)# route-policy sample1	Configures a route policy and enters route-policy configuration mode. <ul style="list-style-type: none"> <li>Once the policy has been created, a group of commands can be entered to define the policy.</li> </ul>
Step 3	<b>end-policy</b>  <b>Example:</b> RP/0/RP0/CPU0:router(config-rpl)# end-policy	Ends the definition of a route policy and exits route-policy configuration mode.
Step 4	<b>end</b> OR <b>commit</b>  <b>Example:</b> RP/0/RP0/CPU0:router(config)# end OR RP/0/RP0/CPU0:router(config)# commit	Saves configuration changes. <ul style="list-style-type: none"> <li>When you issue the <b>end</b> command, the system will prompt you to commit changes: Uncommitted changes found. Commit them? <ul style="list-style-type: none"> <li>Entering <b>yes</b> will save configuration changes to the running configuration file, exit the configuration session, and return the router to EXEC mode.</li> <li>Entering <b>no</b> will exit the configuration session and return the router to EXEC mode without committing the configuration changes.</li> </ul> </li> <li>Use the <b>commit</b> command to save the configuration changes to the running configuration file and remain within the configuration session.</li> </ul>

## Attaching a Routing Policy to a BGP Neighbor

This task explains how to attach a routing policy to a BGP neighbor. The procedure to attach a routing policy to an IS-IS or OSPF neighbor is the same as BGP, except that the commands and applicable arguments will vary.

### Prerequisites

A routing policy must be preconfigured and well defined prior to it being applied at an attach point. If a policy is not predefined, an error message is generated stating that the policy is not defined.

### SUMMARY STEPS

- configure**
- router bgp** *as-number*

3. **address-family** {*ipv4* | *ipv6*} {*multicast* | *unicast*}
4. **neighbor** *ip-address*
5. **policy** *policy-name* {*in* | *out*}
6. **exit**
7. **end**  
or  
**commit**

## DETAILED STEPS

	Command or Action	Purpose
Step 1	<b>configure</b>  <b>Example:</b> RP/0/RP0/CPU0:router# configure	Enters global configuration mode.
Step 2	<b>router bgp</b> <i>as-number</i>  <b>Example:</b> RP/0/RP0/CPU0:router(config)# router bgp 125	Configures a BGP routing process and enters router configuration mode. <ul style="list-style-type: none"> <li>The <i>as-number</i> argument identifies the autonomous system in which the router resides. Valid values are from 0 to 65535. Private autonomous system numbers that can be used in internal networks range from 64512 to 65535.</li> </ul>
Step 3	<b>address-family</b> { <i>ipv4</i>   <i>ipv6</i> } { <i>multicast</i>   <i>unicast</i> }	Specifies the address family, the version of IP that is in use, and either multicast or unicast. <ul style="list-style-type: none"> <li>Enters address family configuration mode.</li> </ul>
Step 4	<b>neighbor</b> <i>ip-address</i>  <b>Example:</b> RP/0/RP0/CPU0:router(config-router-af)# neighbor 10.0.0.20	Specifies a neighbor IP address.
Step 5	<b>policy</b> <i>policy-name</i> { <i>in</i>   <i>out</i> }	Attaches the policy, which must be well formed and predefined.
	<b>Example:</b> RP/0/RP0/CPU0:router(config-router-af)# policy example1 in	

	Command or Action	Purpose
Step 6	<b>exit</b>  <b>Example:</b> RP/0/RP0/CPU0:router(config-router-af)# exit	Exits address family configuration mode.
Step 7	<b>end</b> OR <b>commit</b>  <b>Example:</b> RP/0/RP0/CPU0:router(config-router)# end OR RP/0/RP0/CPU0:router(config-router)# commit	Saves configuration changes. <ul style="list-style-type: none"> <li>When you issue the <b>end</b> command, the system will prompt you to commit changes:  Uncommitted changes found. Commit them? <ul style="list-style-type: none"> <li>Entering <b>yes</b> will save configuration changes to the running configuration file, exit the configuration session, and return the router to EXEC mode.</li> <li>Entering <b>no</b> will exit the configuration session and return the router to EXEC mode without committing the configuration changes.</li> </ul> </li> </ul> Use the <b>commit</b> command to save the configuration changes to the running configuration file and remain within the configuration session.

## Modifying a Routing Policy Using the Microemacs Editor

This task explains how to modify an existing routing policy using the microemacs editor.

### SUMMARY STEPS

1. **edit** {route-policy | prefix-set | as-path-set | community-set | extended-community-set} *name*
2. **show rpl policy** *name* [detail]
3. **show rpl prefix-set** *name*

## DETAILED STEPS

	Command or Action	Purpose
Step 1	<p><code>edit {route-policy   prefix-set   as-path-set   community-set   extended-community-set} name</code></p> <p><b>Example:</b> RP/0/RP0/CPU0:router# edit route-policy sample1</p>	<p>Identifies the route policy, prefix set, AS-path set, community set, or extended community set name to be modified.</p> <ul style="list-style-type: none"> <li>A copy of the route policy, prefix set, AS-path set, community set, or extended community set is copied to a temporary file and the microemacs editor is launched. When you finish editing the policy or set, save the changes by using the <b>save-buffer</b> command, ^X^S (Control-X Control-S). Follow the prompts to commit the changes to the router. To exit the editor, use the <b>quit</b> command, ^X^C.</li> </ul>
Step 2	<p><code>show rpl policy name [detail]</code></p> <p><b>Example:</b> RP/0/RP0/CPU0:router# show rpl policy sample2</p>	<p>(Optional) Displays the configuration of a specific named route policy.</p> <ul style="list-style-type: none"> <li>Use the <b>detail</b> keyword to display all policies and sets that a policy uses.</li> </ul>
Step 3	<p><code>show rpl prefix-set name</code></p> <p><b>Example:</b> RP/0/RP0/CPU0:router# show rpl prefix-set prefixset1</p>	<p>(Optional) Displays the contents of a named prefix set.</p> <ul style="list-style-type: none"> <li>To display the contents of a named AS-path set, community set, or extended community set, replace the <b>prefix-set</b> keyword with <b>as-path-set</b>, <b>community-set</b>, or <b>extcommunity-set</b>, respectively.</li> </ul>

## Configuration Examples for Implementing Routing Policy

This section provides the following configuration examples:

- [Routing Policy Definition: Example, page 174](#)
- [Simple Inbound Policy: Example, page 175](#)
- [Modular Inbound Policy: Example, page 176](#)
- [Translating Cisco IOS Route Maps to Cisco IOS XR Routing Policy Language: Example, page 177](#)

### Routing Policy Definition: Example

In the following example, a BGP route policy named sample1 is defined using the **route-policy name** command. The policy compares the network layer reachability information (NLRI) to the elements in the prefix set test. If it evaluates to true, the policy performs the operations in the *then* clause. If it evaluates to false, the policy performs the operations in the *else* clause, that is, sets the MED value to 200 and adds the community 2:100 to the route. The final steps of the example commit the configuration to the router, exit configuration mode, and display the contents of route policy sample1.

```
configure
  route-policy sample1
    if destination in test then
      drop
    else
      set med 200
      set community (2:100)
```

```
        endif
    end-policy
end
show config running route-policy sample1
```

```
Building configuration...
route-policy sample1
  if destination in test then
    drop
  else
    set metric 200
    set community (2:100)
  endif
end-policy
```

## Simple Inbound Policy: Example

The following policy discards any route whose network layer reachability information (NLRI) specifies a prefix longer than slash-24, and any route whose NLRI specifies a destination in the address space reserved by RFC 1918. For all remaining routes, it sets the MED and local preference, and adds a community to the list in the route.

For routes whose community lists include any of the values in the range from 101:202 to 106:202 that have a 16-bit tag portion containing the value 202, the policy prepends autonomous system number 2 twice, and adds the community 2:666 to the list in the route. Of these routes, if the MED is either 666 or 225, then the policy sets the origin of the route to incomplete, and otherwise sets the origin to IGP.

For routes whose community lists do not include any of the values in the range from 101:202 to 106:202, the policy adds the community 2:999 to the list in the route.

```
prefix-set too-specific
  0.0.0.0/0 ge 25 le 32
end-set

prefix-set rfc1918
  10.0.0.0/8 le 32,
  172.16.0.0/12 le 32,
  192.168.0.0/16 le 32
end-set

community-set prepend2
  100:202,
  101:202,
  102:202,
  103:202,
  104:202,
  105:202,
  106:202
end-set

route-policy inbound-tx(lpref)
  if destination in too-specific or destination in rfc1918 then
    drop
  endif
  set med 1000
  set local-preference $lpref
  set community (2:1001) additive
  if community matches-any prepend2 then
    prepend as-path 2 2
    set community (2:666) additive
    if med eq 666 or med eq 225 then
```

```

        set origin incomplete
    else
        set origin igp
    endif
else
    set community (2:999) additive
endif
end-policy

router bgp 2
    neighbor 10.0.1.2 address-family ipv4 unicast policy inbound-tx in

```

## Modular Inbound Policy: Example

The following policy example builds two inbound policies, in-100 and in-101, for two different peers. In building the specific policies for those peers, it reuses some common blocks of policy that may be common to multiple peers. It builds a few basic building blocks, the policies common-inbound, filter-bogons, and set-lpref-prepend.

The filter-bogons building block is a simple policy that will filter all undesirable routes such as those from the RFC 1918 address space. The policy set-lpref-prepend is a utility policy that can set the local preference and prepend the AS-path according to parameterized values that are passed in. The common-inbound policy uses these building block filter-bogons to build a common block of inbound policy. The common-inbound policy is used as a building block in the construction of in-100 and in-101 along with the set-lpref-prepend building block.

This is a simple example that illustrates the modular capabilities of the policy language.

```

prefix-set bogon
    10.0.0.0/8 ge 8 le 32,
    0.0.0.0,
    0.0.0.0/0 ge 27 le 32,
    192.168.0.0/16 ge 16 le 32
end-set
!
route-policy in-100
    apply common-inbound
    if community matches-any (100:2) then
        apply set-lpref-prepend (100,100,2)
        set community (2:1234) additive
    else
        set local-preference 110
    endif
    if community matches-any (100:666, 1000:999) then
        set med 444
        set local-preference 200
        set community (no-export) additive
    endif
end-policy
!
route-policy in-101
    apply common-inbound
    if community matches-any (101:2) then
        apply set-lpref-prepend(100,101,2)
        set community (2:1234) additive
    else
        set local-preference 125
    endif
end-policy
!
route-policy filter-bogons

```

```

    if destination in bogon then
        drop
    else
        pass
    endif
end-policy
!
route-policy common-inbound
    apply filter-bogons
    set origin igp
    set community (2:333)
end-policy
!
route-policy set-lpref-prepend($lpref,$as,$prependcnt)
    set local-preference $lpref
    prepend as-path $as $prependcnt
end-policy

```

## Translating Cisco IOS Route Maps to Cisco IOS XR Routing Policy Language: Example

Consider the following route maps, prefix lists, and community lists. We will show four different translations into the routing policy language, continually using more capabilities of the language to reduce the amount of configuration needed. This example steps you through using several of the features of the language to modularize the configuration. Decide what you should modularize and whether you should modularize specific portions in the context of how that particular piece of policy will be used.

You cannot use both RPL and old policy (including route maps and access control lists) at the same attach point.

For example, the following configuration would be invalid:

```

router bgp 2
    neighbor 10.0.101.2
        address-family ipv4 unicast
            policy rp10 in
            route-map rm0 in

```

However, this configuration would be valid:

```

router bgp 2
    neighbor 10.0.101.2
        address-family ipv4 unicast
            policy rp10 in
            route-map rm0 out
    neighbor 10.0.101.3
        address-family ipv4 unicast
            policy rp11 out

```

In the following example, a route map is translated to the policy language while retaining the redundant operations:

### Original Route Map Configuration

```

ip prefix-list 101
    10 permit 10.48.0.0/16 le 32
    20 permit 172.48.0.0/19 le 32
    30 permit 172.10.10.0/24
    40 permit 172.11.1.0/24
    50 permit 192.168.3.0/24
    60 permit 192.168.8.0/21

```

```
70 permit 192.168.32.0/21

ip prefix-list 102
 10 permit 10.48.0.0/16 le 32
 20 permit 10.48.0.5/19 le 32
 30 permit 172.16.10.0/24
 40 permit 172.16.1.0/24
 50 permit 172.16.3.0/24
 60 permit 192.168.8.0/21
 70 permit 192.168.32.0/21

ip community-list 1
 10 permit 10:11

ip community-list 2
 10 permit 10:12

ip community-list 3
 10 permit 10:13

ip community-list 4
 10 permit 10:14

route-map sample1-translation-1 permit 10
 match ip address prefix-list 101
 match community 1
 set community 12:34 additive
 set metric 11
route-map sample1-translation-1 permit 20
 match ip address prefix-list 101
 match community 2
 set metric 12
 set community 12:34 additive
route-map sample1-translation-1 permit 30
 match ip address prefix-list 101
 match community 3
 set metric 13
 set community 12:34 additive
route-map sample1-translation-1 permit 40
 match ip address prefix-list 101
 match community 4
 set metric 14
 set community 12:34 additive
route-map sample1-translation-1 permit 50
 match ip address prefix-list 101
 set metric 100
 set community 12:34 additive

route-map sample2-translation-1 permit 10
 match ip address prefix-list 102
 match community 1
 set community 12:35 additive
 set metric 11
route-map sample2-translation-1 permit 20
 match ip address prefix-list 102
 match community 2
 set metric 12
 set community 12:35 additive
route-map sample2-translation-1 permit 30
 match ip address prefix-list 102
 match community 3
 set metric 13
 set community 12:35 additive
```

```
route-map sample2-translation-1 permit 40
  match ip address prefix-list 102
  match community 4
  set metric 14
  set community 12:35 additive
route-map sample2-translation-1 permit 50
  match ip address prefix-list 102
  set metric 100
  set community 12:35 additive
```

### A Simple Translation

A simple translation of this route map configuration to the policy language would retain the redundant operations, as shown in the following example:

```
prefix-set ps101
  10.48.0.0/16 le 32
  172.48.0.0/19 le 32
  172.10.10.0/24
  172.11.1.0/24
  192.168.3.0/24
  192.168.8.0/21
  192.168.32.0/21
end-set

prefix-set ps102
  10.48.0.0/16 le 32
  10.48.0.5/19 le 32
  172.16.10.0/24
  172.16.1.0/24
  172.16.3.0/24
  192.168.8.0/21
  192.168.32.0/21
end-set

community-set cs1
  10:11
end-set

community-set cs2
  10:12
end-set

community-set cs3
  10:13
end-set

community-set cs4
  10:14
end-set

route-policy sample1-translation-1a
  if destination in ps101 and community matches-any cs1 then
    set med 11
    set community 12:34 additive
  elseif destination in ps101 and community matches-any cs2 then
    set med 12
    set community 12:34 additive
  elseif destination in ps101 and community matches-any cs3 then
    set med 13
    set community 12:34 additive
  elseif destination in ps101 and community matches-any cs4 then
    set med 14
```

```

        set community 12:34 additive
    elseif destination in ps101
        set med 100
        set community 12:34 additive
    endif
end-policy

route-policy sample2-translation-1a
    if destination in ps102 and community matches-any cs1 then
        set med 11
        set community (12:35) additive
    elseif destination in ps102 and community matches-any cs2 then
        set med 12
        set community (12:35) additive
    elseif destination in ps102 and community matches-any cs3 then
        set med 13
        set community (12:35) additive
    elseif destination in ps102 and community matches-any cs4 then
        set med 14
        set community (12:35) additive
    elseif destination in ps102
        set med 100
        set community (12:35) additive
    endif
end-policy

```

### Nest Conditionals to Reduce Repetitive Comparisons

Common operations can be coalesced by nesting the conditionals, testing the destination address only once, and setting the community only once, as shown in the following example:

```

prefix-set ps101
    10.48.0.0/16 le 32
    172.48.0.0/19 le 32
    172.10.10.0/24
    172.11.1.0/24
    192.168.3.0/24
    192.168.8.0/21
    192.168.32.0/21
end-set

prefix-set ps102
    10.48.0.0/16 le 32
    10.48.0.5/19 le 32
    172.16.10.0/24
    172.16.1.0/24
    172.16.3.0/24
    192.168.8.0/21
    192.168.32.0/21
end-set

community-set cs1
    10:11
end-set

community-set cs2
    10:12
end-set

community-set cs3
    10:13
end-set

community-set cs4

```

```
10:14
end-set

route-policy sample1-translation-1b
  if destination in ps101 then
    set community (12:34) additive
    if community matches-any cs1 then
      set med 11
    elseif community matches-any cs2 then
      set med 12
    elseif community matches-any cs3 then
      set med 13
    elseif community matches-any cs4 then
      set med 14
    else
      set med 100
    endif
  endif
end-policy

route-policy sample2-translation-1b
  if destination in ps102 then
    set community (12:35) additive
    if community matches-any cs1 then
      set med 11
    elseif community matches-any cs2 then
      set med 12
    elseif community matches-any cs3 then
      set med 13
    elseif community matches-any cs4 then
      set med 14
    else
      set med 100
    endif
  endif
end-policy
```

### Use Inline Sets to Remove Small Indirect Set References

Because the community comparisons are quite simple, we can replace the named community set references with direct inline references, thus eliminating the need to define four community sets, each of which contains only one community value. These replacements leave two prefix sets and two policies, as follows:

```
prefix-set ps101
  10.48.0.0/16 le 32
  172.48.0.0/19 le 32
  172.10.10.0/24
  172.11.1.0/24
  192.168.3.0/24
  192.168.8.0/21
  192.168.32.0/21
end-set

prefix-set ps102
  10.48.0.0/16 le 32
  10.48.0.5/19 le 32
  172.16.10.0/24
  172.16.1.0/24
  172.16.3.0/24
  192.168.8.0/21
  192.168.32.0/21
end-set
```

```

route-policy sample1-translation-1c
  if destination in ps101 then
    set community (12:34) additive
    if community matches-any (10:11) then
      set med 11
    elseif community matches-any (10:12) then
      set med 12
    elseif community matches-any (10:13) then
      set med 13
    elseif community matches-any (10:14) then
      set med 14
    else
      set med 100
    endif
  endif
end-policy

```

```

route-policy sample2-translation-1c
  if destination in ps101 then
    set community (12:34) additive
    if community matches-any (10:11) then
      set med 11
    elseif community matches-any (10:12) then
      set med 12
    elseif community matches-any (10:13) then
      set med 13
    elseif community matches-any (10:14) then
      set med 14
    else
      set med 100
    endif
  endif
end-policy

```

### Take Advantage of Parameterization to Reuse Common Structures

The following example takes advantage of the ability to parameterize common structures and create a common parameterized policy (sample-translation-common) that is reused:

```

prefix-set ps101
  10.48.0.0/16 le 32
  172.48.0.0/19 le 32
  172.10.10.0/24
  172.11.1.0/24
  192.168.3.0/24
  192.168.8.0/21
  192.168.32.0/21
end-set

prefix-set ps102
  10.48.0.0/16 le 32
  10.48.0.5/19 le 32
  172.16.10.0/24
  172.16.1.0/24
  172.16.3.0/24
  192.168.8.0/21
  192.168.32.0/21
end-set

route-policy sample-translation-common(tag)
  set community (12:$tag) additive
  if community matches-any (10:11) then
    set med 11
  elseif community matches-any (10:12) then

```

```

        set med 12
    elseif community matches-any (10:13) then
        set med 13
    elseif community matches-any (10:14) then
        set med 14
    else
        set med 100
    endif
end-policy

route-policy sample1-translation-1d
    if destination in ps101 then
        apply sample-translation-common (34)
        pass
    endif
end-policy

route-policy sample2-translation-1d
    if destination in ps102 then
        apply sample-translation-common (35)
        pass
    endif
end-policy

```

## Additional References

The following sections provide references related to implementing RPL.

## Related Documents

Related Topic	Document Title
Routing policy language commands	<i>Routing Policy Language Commands on Cisco IOS XR Software</i>
Cisco CRS-1 Series Carrier Routing System Router Interface	<i>Cisco CRS-1 Series Carrier Routing System Router Interface Configuration Guide</i>
Cisco CRS-1 Series Carrier Routing System Craft Web Interface (CWI)	<i>Cisco CRS-1 Series Carrier Routing System Craft Web Interface (CWI) Configuration</i>
Regular expression syntax	<p>“Understanding Regular Expressions, Special Characters and Patterns” appendix in the <i>Cisco CRS-1 Series Carrier Routing System Getting Started Guide</i></p> <p>The appendix can be found at the following link:</p> <p><a href="http://www.cisco.com/univercd/cc/td/doc/product/core/crs/crsget/ppx_exp.htm">http://www.cisco.com/univercd/cc/td/doc/product/core/crs/crsget/ppx_exp.htm</a></p>

## RFCs

RFCs	Title
RFC 1771	<i>A Border Gateway Protocol 4 (BGP-4)</i>  This document is obsolete. Although it is the approved standard for BGP, and more recent drafts are not yet approved, the latter are more correct.

## Technical Assistance

Description	Link
Technical Assistance Center (TAC) home page, containing 30,000 pages of searchable technical content, including links to products, technologies, solutions, technical tips, and tools. Registered Cisco.com users can log in from this page to access even more content.	<a href="http://www.cisco.com/public/support/tac/home.shtml">http://www.cisco.com/public/support/tac/home.shtml</a>