- Access Control Lists -

Access Control Lists (ACLs)

Access control lists (**ACLs**) can be used for two purposes on Cisco devices:

- To **filter** traffic
- To **identify** traffic

Access lists are a set of rules, organized in a rule table. Each rule or line in an access-list provides a condition, either **permit** or **deny**:

- When using an access-list to filter traffic, a *permit* statement is used to "allow" traffic, while a *deny* statement is used to "block" traffic.
- Similarly, when using an access list to identify traffic, a *permit* statement is used to "include" traffic, while a *deny* statement states that the traffic should "not" be included. It is thus interpreted as a **true/false** statement.

Filtering traffic is the primary use of access lists. However, there are several instances when it is necessary to identify traffic using ACLs, including:

- Identifying interesting traffic to bring up an ISDN link or VPN tunnel
- Identifying routes to filter or allow in routing updates
- Identifying traffic for QoS purposes

When filtering traffic, access lists are applied on interfaces. As a packet passes through a router, the top line of the rule list is checked first, and the router continues to go down the list until a match is made. Once a match is made, the packet is either permitted or denied.

There is an implicit 'deny all' at the end of all access lists. You don't create it, and you can't delete it. Thus, access lists that contain **only deny statements** will **prevent all traffic**.

Access lists are applied either inbound (packets received on an interface, before routing), or outbound (packets leaving an interface, *after* routing). Only one access list **per interface**, **per protocol**, **per direction** is allowed.

More specific and frequently used rules should be at the top of your access list, to optimize CPU usage. New entries to an access list are added to the bottom. You **cannot remove individual lines** from a numbered access list. You must delete and recreate the access to truly make changes. Best practice is to use a text editor to manage your access-lists.

Types of Access Lists

There are two categories of access lists: **numbered** and **named**.

Numbered access lists are broken down into several ranges, each dedicated to a specific protocol:

1–99	IP standard access list
100-199	IP extended access list
200-299	Protocol type-code access list
300-399	DECnet access list
400-499	XNS standard access list
500-599	XNS extended access list
600-699	Appletalk access list
700-799	48-bit MAC address access list
800-899	IPX standard access list
900-999	IPX extended access list
1000-1099	IPX SAP access list
1100-1199	Extended 48-bit MAC address access list
1200-1299	IPX summary address access list
1300-1999	IP standard access list (expanded range)
2000-2699	IP extended access list (expanded range

Remember, individual lines *cannot* be removed from a numbered access list. The entire access list must be deleted and recreated. All new entries to a numbered access list are added to the bottom.

Named access lists provide a bit more flexibility. Descriptive names can be used to identify your access-lists. Additionally, individual lines *can* be removed from a named access-list. However, like numbered lists, all new entries are still added to the bottom of the access list.

There are two common types of named access lists:
- IP standard named access lists
- IP extended named access lists

Configuration of both numbered and named access-lists is covered later in this section.

Wild Card Masks

IP access-lists use **wildcard masks** to determine two things:

1. Which part of an address must match exactly
2. Which part of an address can match any number

This is as opposed to a **subnet mask**, which tells us what part of an address is the network (subnet), and what part of an address is the host. Wildcard masks look like inversed subnet masks.

Consider the following address and wildcard mask:

Address:	172.16.0.0
Wild Card Mask:	0.0.255.255

The above would match any address that begins "172.16." The last two octets could be anything. How do I know this?

Two Golden Rules of Access Lists:

1. If a bit is set to **0** in a wild-card mask, the corresponding bit in the address must be **matched exactly.**
2. If a bit is set to **1** in a wild-card mask, the corresponding bit in the address can **match any number.** In other words, we "don't care" what number it matches.

To see this more clearly, we'll convert both the address and the wildcard mask into binary:

Address:	10101100.00010000.00000000.00000000
Wild Card Mask:	00000000.00000000.11111111.11111111

Any **0** bits in the wildcard mask, indicates that the corresponding bits in the address must be matched exactly. Thus, looking at the above example, we must exactly match the following in the first two octets:

10101100.00010000 = 172.16

Any **1** bits in the wildcard mask indicates that the corresponding bits can be anything. Thus, the last two octets can be any number, and it will still match this access-list entry.

Wild Card Masks (continued)

If wanted to match a **specific address** with a wildcard mask (we'll use an example of 172.16.1.1), how would we do it?

Address:	172.16.1.1
Wild Card Mask:	0.0.0.0

Written out in binary, that looks like:

Address:	10101100.00010000.00000001.00000001
Wild Card Mask:	00000000.00000000.00000000.00000000

Remember what a wildcard mask is doing. A **0** indicates it must match exactly, a **1** indicates it can match anything. The above wildcard mask has all bits set to 0, which means we must match all four octets exactly.

There are actually two ways we can match a host:
- Using a wildcard mask with all bits set to 0 – **172.16.1.1 0.0.0.0**
- Using the keyword "host" – **host 172.16.1.1**

How would we match **all addresses** with a wildcard mask?

Address:	0.0.0.0
Wild Card Mask:	255.255.255.255

Written out in binary, that looks like:

Address:	00000000.00000000.00000000.00000000
Wild Card Mask:	11111111.11111111.11111111.11111111

Notice that the above wildcard mask has all bits set to 1. Thus, each bit can match anything – resulting in the above address and wildcard mask matching all possible addresses.

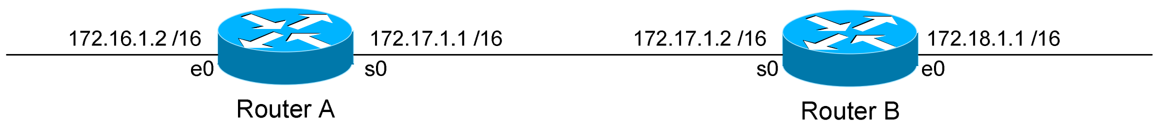
There are actually two ways we can match all addresses:
- Using a wildcard mask with all bits set to 1 – **0.0.0.0 255.255.255.255**
- Using the keyword "any" – **any**

Standard IP Access List

access-list *[1-99] [permit | deny] [source address] [wildcard mask] [log]*

Standard IP access-lists are based upon the source host or network IP address, and should be placed closest to the destination network.

Consider the following example:



In order to block network 172.18.0.0 from accessing the 172.16.0.0 network, we would create the following access-list on Router A:

```
Router(config)#  access-list 10 deny 172.18.0.0 0.0.255.255
Router(config)#  access-list 10 permit any
```

Notice the wildcard mask of 0.0.255.255 on the first line. This will match (*deny*) all hosts on the 172.18.x.x network.

The second line uses a keyword of *any*, which will match (*permit)* any other address. Remember that you must have at least one permit statement in your access list.

To apply this access list, we would configure the following on Router A:

```
Router(config)#  int s0
Router(config-if)#  ip access-group 10 in
```

To view all IP access lists configured on the router:

```
Router#  show ip access-list
```

To view what interface an access-list is configured on:

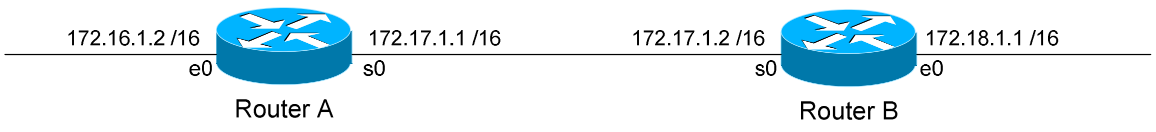
```
Router#  show ip interface
Router#  show running-config
```

Extended IP Access List

access-list *[100-199] [permit | deny] [protocol] [source address] [wildcard mask] [destination address] [wildcard mask] [operator [port]] [log]*

Extended IP access-lists block based upon the source IP address, destination IP address, and TCP or UDP port number. Extended access-lists should be placed closest to the source network.

Consider the following example:



Assume there is a webserver on the 172.16.x.x network with an IP address of 172.16.10.10. In order to block network 172.18.0.0 from accessing anything on the 172.16.0.0 network, EXCEPT for the HTTP port on the web server, we would create the following access-list on Router B:

Router(config)# *access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80*
Router(config)# *access-list 101 deny ip 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255*
Router(config)# *access-list 101 permit ip any any*

The first line allows the 172.18.x.x network access only to port 80 on the web server. The second line blocks 172.18.x.x from accessing anything else on the 172.16.x.x network. The third line allows 172.18.x.x access to anything else.

We could have identified the web server in one of two ways:

Router(config)# *access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80*
Router(config)# *access-list 101 permit tcp 172.18.0.0 0.0.255.255 172.16.10.10 0.0.0.0 eq 80*

To apply this access list, we would configure the following on Router B:

Router(config)# *int e0*
Router(config-if)# *ip access-group 101 in*

Extended IP Access List Port Operators

In the preceding example, we identified TCP port 80 on a specific host use the following syntax:

Router(config)# *access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80*

We accomplished this using an operator of *eq*, which is short for **equals**. Thus, we are identifying host *172.16.10.10* with a port that *eq*uals 80.

We can use several other operators for port numbers:

eq	Matches a specific port
gt	Matches all ports greater than the port specified
lt	Matches all ports less than the port specified
neq	Matches all ports except for the port specified
range	Match a specific inclusive range of ports

The following will match all ports *greater* than *100*:

Router(config)# *access-list 101 permit tcp any host 172.16.10.10 gt 100*

The following will match all ports *less* than *1024*:

Router(config)# *access-list 101 permit tcp any host 172.16.10.10 lt 1024*

The following will match all ports that do *not equal 443*:

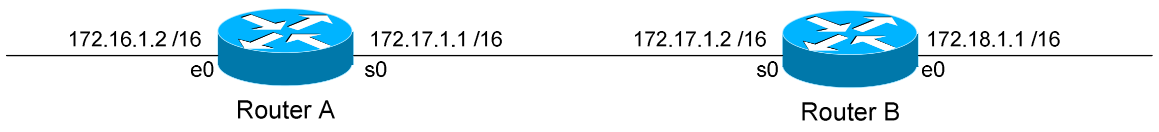
Router(config)# *access-list 101 permit tcp any host 172.16.10.10 neq 443*

The following will match all ports between *80* and *88*:

Router(config)# *access-list 101 permit tcp any host 172.16.10.10 range 80 88*

Access List Logging

Consider again the following example:



Assume there is a webserver on the 172.16.x.x network with an IP address of 172.16.10.10.

We wish to keep track of the number of packets permitted or denied by each line of an access-list. Access-lists have a built-in logging mechanism for such a purpose:

Router(config)# *access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80 log*
Router(config)# *access-list 101 deny ip 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255 log*
Router(config)# *access-list 101 permit ip any any log*

Notice we added an additional keyword *log* to each line of the access-list. When viewing an access-list using the following command:

Router# *show access-list 101*

We will now have a counter on each line of the access-list, indicating the number of packets that were permitted or denied by that line. This information can be sent to a syslog server:

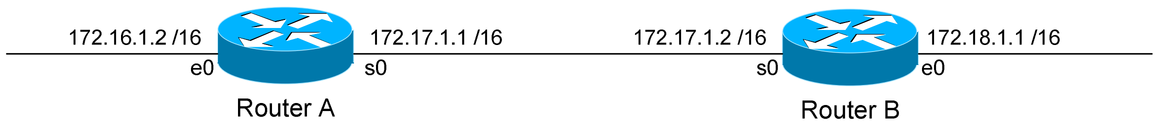
Router(config)# *logging on*
Router(config)# *logging 172.18.1.50*

The *logging on* command enables logging. The second *logging* command points to a syslog host at *172.18.1.50.*

We can include more detailed logging information, including the source MAC address of the packet, and what interface that packet was received on. To accomplish this, use the *log-input* argument:

Router(config)# *access-list 101 permit ip any any log-input*

ICMP Access List



Consider this scenario. You've been asked to block anyone from the 172.18.x.x network from "pinging" anyone on the 172.16.x.x network. You want to allow everything else, including all other ICMP packets.

The specific ICMP port that a "ping" uses is **echo**. To block specific ICMP parameters, use an extended IP access list. On Router B, we would configure:

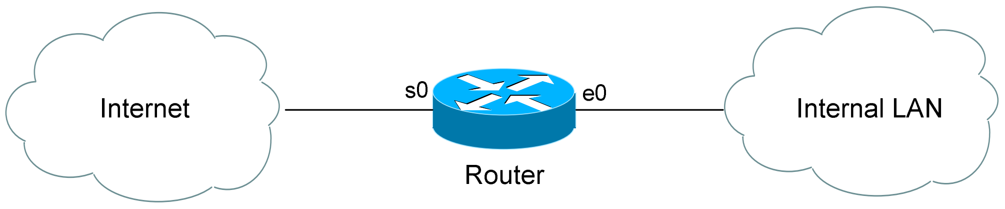
Router(config)# access-list 102 deny icmp 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255 echo
Router(config)# access-list 102 permit icmp 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255
Router(config)# access-list 102 permit ip any any

The first line blocks only ICMP echo requests (pings). The second line allows all other ICMP traffic. The third line allows all other IP traffic.

Don't forget to apply it to an interface on Router B:

Router(config)# *int e0*
Router(config-if)# *ip access-group 102 in*

Untrusted networks (such as the Internet) should usually be blocked from pinging an outside router or any internal hosts:

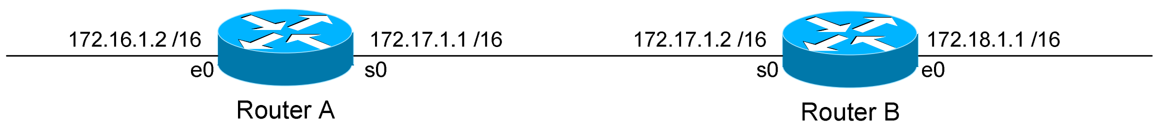


Router(config)# *access-list 102 deny icmp any any*
Router(config)# *access-list 102 permit ip any any*

Router(config)# *interface s0*
Router(config-if)# *ip access-group 102 in*

The above access-list completed disables ICMP on the serial interface. However, this would effectively disable ICMP traffic *in both directions* on the router. Any replies to pings initiated by the Internal LAN would be blocked on the way back in.

Telnet Access List



We can create access lists to restrict telnet access to our router. For this example, we'll create an access list that prevents anyone from the evil 172.18.x.x network from telneting into Router A, but allow all other networks telnet access.

First, we create the access-list on Router A:

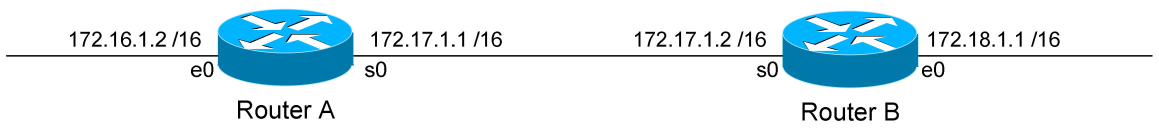
```
Router(config)#  access-list 50 deny 172.18.0.0 0.0.255.255
Router(config)#  access-list 50 permit any
```

The first line blocks the 172.18.x.x network. The second line allows all other networks.

To apply it to Router A's telnet ports:

```
Router(config)#  line vty 0 4
Router(config-line)#  access-class 50 in
```

Named Access Lists



Named access lists provide us with two advantages over numbered access lists. First, we can apply an identifiable name to an access list, for documentation purposes. Second, we can remove individual lines in a named access-list, which is not possible with numbered access lists.

Please note, though we can *remove* individual lines in a named access list, we cannot *insert* individual lines into that named access list. New entries are always placed at the bottom of a named access list.

To create a standard named access list, the syntax would be as follows:

```
Router(config)#  ip access-list standard NAME
Router(config-std-nacl)#  deny 172.18.0.0 0.0.255.255
Router(config-std-nacl)#  permit any
```

To create an extended named access list, the syntax would be as follows:

```
Router(config)#  ip access-list extended NAME
Router(config-ext-nacl)# permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80
Router(config-ext-nacl)#  deny ip 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255
Router(config-ext-nacl)#  permit ip any any
```

Notice that the actual configuration of the named access-list is performed in a separate router "mode":

```
Router(config-std-nacl)#

Router(config-ext-nacl)#
```

Time-Based Access-Lists

Beginning with IOS version 12.0, access-lists can be based on the time and the day of the week.

The first step to creating a time-based access-list, is to create a *time-range*:

```
Router(config)#  time-range BLOCKHTTP
```

The above command creates a *time-range* named *BLOCKHTTP*. Next, we must either specify an *absolute* time, or a *periodic* time:

```
Router(config)#  time-range BLOCKHTTP
Router(config-time-range)#  absolute start 08:00 23 May 2006 end 20:00 26 May 2006

Router(config)#  time-range BLOCKHTTP
Router(config-time-range)#  periodic weekdays 18:00 to 23:00
```

Notice the use of military time. The first *time-range* sets an *absolute* time that will *start* from May 23, 2006 at 8:00 a.m., and will *end* on May 26, 2006 at 8:00 p.m.

The second time-range sets a *periodic* time that is always in effect on *weekdays* from 6:00 p.m. to 11:00 p.m.

Only one *absolute* time statement is allowed per time-range, but multiple *periodic* time statements are allowed.

After we establish our time-range, we must reference it in an access-list:

```
Router(config)#  access-list 102 deny any any eq 80 time-range BLOCKHTTP
Router(config)#  access-list 102 permit ip any any
```

Notice the *time-range* argument at the end of the access-list line. This will result in HTTP traffic being blocked, but only during the time specified in the time-range.

Source:
(*http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t1/timerang.htm*)

Advanced Wildcard Masks

Earlier in this section, we discussed the basics of wildcard masks. The examples given previously matched one of three things:

- A specific host
- A specific octet(s)
- All possible hosts

It is also possible to match groups or ranges of hosts with wildcard masks. For example, assume we wanted a standard access-list that denied the following hosts:

172.16.1.4
172.16.1.5
172.16.1.6
172.16.1.7

We could create an access-list with four separate lines:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.5 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.6 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.7 0.0.0.0
```

However, it is also possible to match all four addresses in **one** line:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

How do I know this is correct? Let's write out the above four addresses, and my wildcard mask in binary:

```
172.16.1.4:          10101100.00010000.00000001.00000100
172.16.1.5:          10101100.00010000.00000001.00000101
172.16.1.6:          10101100.00010000.00000001.00000110
172.16.1.7:          10101100.00010000.00000001.00000111

Wild Card Mask:      00000000.00000000.00000000.00000011
```

Notice that the first 30 bits of each of the four addresses are identical. Each begin "*10101100.00010000.00000001.000001*". Since those bits must match exactly, the first 30 bits of our wildcard mask are set to **0**.

Advanced Wildcard Masks (continued)

Notice now that the *only* bits that are different between the four addresses are the last two bits. Not only that, but we use every computation of those last two bits: 00, 01, 10, 11.

Thus, since those last two bits can be anything, the last two bits of our wildcard mask are set to **1**.

The resulting access-list line:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

We also could have determined the appropriate address and wildcard mask by using AND/XOR logic.

To determine the address, we perform a logical **AND** operation:

1. If all bits in a column are set to **0**, the corresponding address bit is **0**
2. If all bits in a column are set to **1,** the corresponding address bit is **1**
3. If the bits in a column are a mix of **0**'s and **1's**, the corresponding address bit is a **0.**

Observe:

172.16.1.4:	10101100.00010000.00000001.00000100
172.16.1.5:	10101100.00010000.00000001.00000101
172.16.1.6:	10101100.00010000.00000001.00000110
172.16.1.7:	10101100.00010000.00000001.00000111
Result:	10101100.00010000.00000001.00000100

Our resulting address is **172.16.1.4**. This gets us half of what we need.

Advanced Wildcard Masks (continued)

To determine the wildcard mask, we perform a logical **XOR** (exclusive OR) operation:

1. If all bits in a column are set to **0**, the corresponding wildcard bit is **0**
2. If all bits in a column are set to **1,** the corresponding wildcard bit is **0**
3. If the bits in a column are a mix of **0**'s and **1's**, the corresponding wildcard bit is a 1**.**

Observe:

172.16.1.4:	10101100.00010000.00000001.00000100
172.16.1.5:	10101100.00010000.00000001.00000101
172.16.1.6:	10101100.00010000.00000001.00000110
172.16.1.7:	10101100.00010000.00000001.00000111
Result:	00000000.00000000.00000000.00000011

Our resulting wildcard mask is **0.0.0.3**. Put together, we have:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

Please Note: We can determine the number of addresses a wildcard mask will match by using a simple formula:

$$2^n$$

Where "n" is the number of bits set to **1** in the wildcard mask. In the above example, we have two bits set to 1, which matches exactly **four addresses** ($2^2 = 4$).

There *will* be occasions when we cannot match a range of addresses in one line. For example, if we wanted to deny 172.16.1.4-6, instead of 172.16.1.4-7, we would need two lines:

```
Router(config)#  access-list 10 permit 172.16.1.7 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

If we didn't include the first line, the second line would have denied the 172.16.1.7 address. Always remember to use the above formula (2^n) to ensure your wildcard mask doesn't match more addresses than you intended (often called overlap).

Advanced Wildcard Masks (continued)

Two more examples. How would we deny all **odd** addresses on the 10.1.1.x/24 subnet in one access-list line?

```
Router(config)#  access-list 10 deny 10.1.1.1 0.0.0.254
```

Written in binary:

10.1.1.1:	00001010.00000001.00000001.00000001
Wild Card Mask:	00000000.00000000.00000000.11111110

What would the result of the above wildcard mask be?

1. The first three octets must match exactly.
2. The last bit in the fourth octet must match exactly. Because we set this bit to **1** in our address, every number this matches will be **odd**.
3. All other bits in the fourth octet can match any number.

Simple, right? How would we deny all **even** addresses on the 10.1.1.x/24 subnet in one access-list line?

```
Router(config)#  access-list 10 deny 10.1.1.0 0.0.0.254
```

Written in binary:

10.1.1.0:	00001010.00000001.00000001.00000000
Wild Card Mask:	00000000.00000000.00000000.11111110

What would the result of the above wildcard mask be?

4. The first three octets must match exactly.
5. The last bit in the fourth octet must match exactly. Because we set this bit to **0** in our address, every number this matches will be **even**.
6. All other bits in the fourth octet can match any number.