

Building VLAN Interfaces in Linux and IOS

Michael J. Martin

In order to support 802.1Q interfaces on Linux, you first need to configure an Ethernet switch to support them. Configuring tagged and untagged VLAN ports on Cisco IOS-based switches is quite straightforward. Non-tagged ports are configured in the following manner:

Set the spanning tree method you want the switch to support:

```
outlan-rs01(config)#spanning-tree mode pvst
```

The command `<spanning-tree mode {mst|pvst|rapid-pvst}&rt;` can set the spanning tree protocol (STP) mode globally or on a per-interface basis. The "PVST" option is the Cisco proprietary per-VLAN spanning tree implementation. The "MST" spanning tree option enables 802.1s. The "Rapid-PVST" option enables a Cisco proprietary version of 802.1w rapid spanning tree. Next, create a VLAN using the `<vlan {1-4094}&rt;` command. Then assign the VLAN a human referable name using the VLAN configuration subcommand `<name {name_with_no_spaces}&rt;`:

```
outlan-rs01(config)#vlan 443
outlan-rs01(config-vlan)#name NTP_server_segment
```

Once the VLAN has been created, the ports can be assigned to the VLAN using the `<switchport&rt;` interface subcommand:

```
outlan-rs01(config-if)#switchport access vlan 67
outlan-rs01(config-if)#switchport mode access
outlan-rs01(config-if)#switchport nonegotiate
outlan-rs01(config-if)#switchport port-security maximum 1
outlan-rs01(config-if)#switchport port-security aging type inactivity
outlan-rs01(config-if)#switchport port-security aging time 120
```

The `<switchport access vlan {1-4094}&rt;` command assigns the port to a VLAN group. The `<switchport mode access&rt;` command hard-sets the port as an untagged access port. The alternative option is `<switchport mode trunk&rt;`, which hard-sets the port as a tagged access port. Cisco nomenclature refers to this as a "trunk port." The default setting of `<switchport mode&rt;` command is "dynamic," which allows the port to dynamically negotiate access or trunk mode. The `<switchport nonegotiate&rt;` command disables the port from sending dynamic trunking protocol (DTP) messages.

The last three commands set up port security, which sets the number of MAC addresses that be associated with a given switchport. This is a great security option that can minimize problems of oversubscription by limiting the number of devices that can send traffic through the switchport. When set to "1," only a single device can be connected to the port and forward traffic. Even if a hub or another switch is connected, still only one device can forward traffic. The command `< switchport port-security maximum {1-6272}` sets the number of MACs that can be associated with the port. The command `< switchport port-security aging type {absolute|inactivity}&rt;` sets the MAC address timeout method. The command `<switchport port-security aging time {1-1440}&rt;` sets the timeout timer in minutes.

One additional thing about access ports -- it is also possible to assign a port membership to a non-existent VLAN ID. The switch will create the VLAN for you:

```
outlan-rs01(config)#interface FastEthernet2/0/10
outlan-rs01(config-if)#switchport access vlan 67
% Access VLAN does not exist. Creating vlan 67
```

Building VLAN Interfaces in Linux and IOS

Michael J. Martin

```
outlan-rs01(config-if)#
```

Be warned that this method can backfire on you. If you make a mistake and type in a non-existent VLAN, the switch could then assign the port to the wrong VLAN.

Configuring a switchport to support tagging is accomplished using the <switchport trunk {value}&rt; interface subcommand:

```
outlan-rs01(config-if)#switchport trunk encapsulation dot1q
outlan-rs01(config-if)#switchport mode trunk
outlan-rs01(config-if)#switchport nonegotiate
```

Although Cisco IOS supports dynamic trunk negotiation via DTP, it is generally a good idea to statically configure trunk interfaces. The first step in the configuration of a tagged port is defining the encapsulation protocol with the <switchport trunk encapsulation {dot1q | isl | negotiate}&rt; interface subcommand. The encapsulation method defines the tagging protocol that will be used between the switches to tag the frames. Next, the port needs to be hard-set to operate as a tagged (trunked) interface with the <switchport mode {access | trunk}&rt; command. As with untagged ports, DTP should be disabled on tagged interfaces using the <switchport nonegotiate}&rt; command. Now that the interface is set up as a tagged interface, we need to define the VLAN traffic that can be forwarded across the interface:

```
outlan-rs01(config-if)#switchport trunk allowed vlan 40,41,80
outlan-rs01(config-if)#switchport trunk native vlan 71
```

Cisco follows the "forward everything" approach when it comes to the ability of VLAN traffic to traverse tagged interfaces, because it works from the philosophy that tagged interfaces are also STP-included interfaces. You, however, may not want to have every VLAN see a given tagged interface as a potential forwarding path, especially, if you utilize multiple trunk ports to distribute VLAN loads using per-VLAN spanning tree. The sub-command <switchport trunk allowed vlan {vid,vid..}&rt; is used to define the VLANs that can utilize a given tagged interface. If this command not used on a trunk port, all VLANs will be permitted.

The other command used for controlling VLAN access on tagged interfaces is the <switchport trunk native vlan {vid}&rt; command. This command specifies the VLAN to which any non-tagged traffic should be forwarded. At this point the switchport interface is configured to support tagged frames. However, depending on your environment, it may also be desirable to enable port-security on both tagged and untagged interfaces.

```
outlan-rs01(config-if)#switchport port-security maximum 48 vlan 80
outlan-rs01(config-if)#switchport port-security maximum 24 vlan 41
outlan-rs01(config-if)#switchport port-security maximum 24 vlan 40
outlan-rs01(config-if)#switchport port-security aging type inactivity
outlan-rs01(config-if)#switchport port-security aging time 60
```

The port security configuration for tagged ports follows a slightly different model than the one utilized for untagged ports. Untagged ports only support one VLAN, requiring only a single MAC address limit. Port security on tagged interfaces requires a MAC limit definition be set for each VLAN the tagged port is a member of. To accommodate this, the command provides a VLAN assignment option following the MAC definition field, <switchport port-security maximum {1-6272}&rt; vlan {vid}&rt;. Otherwise, the MAC aging and timer definitions are the same as those used when applying port security to an untagged interface.

Building VLAN Interfaces in Linux and IOS

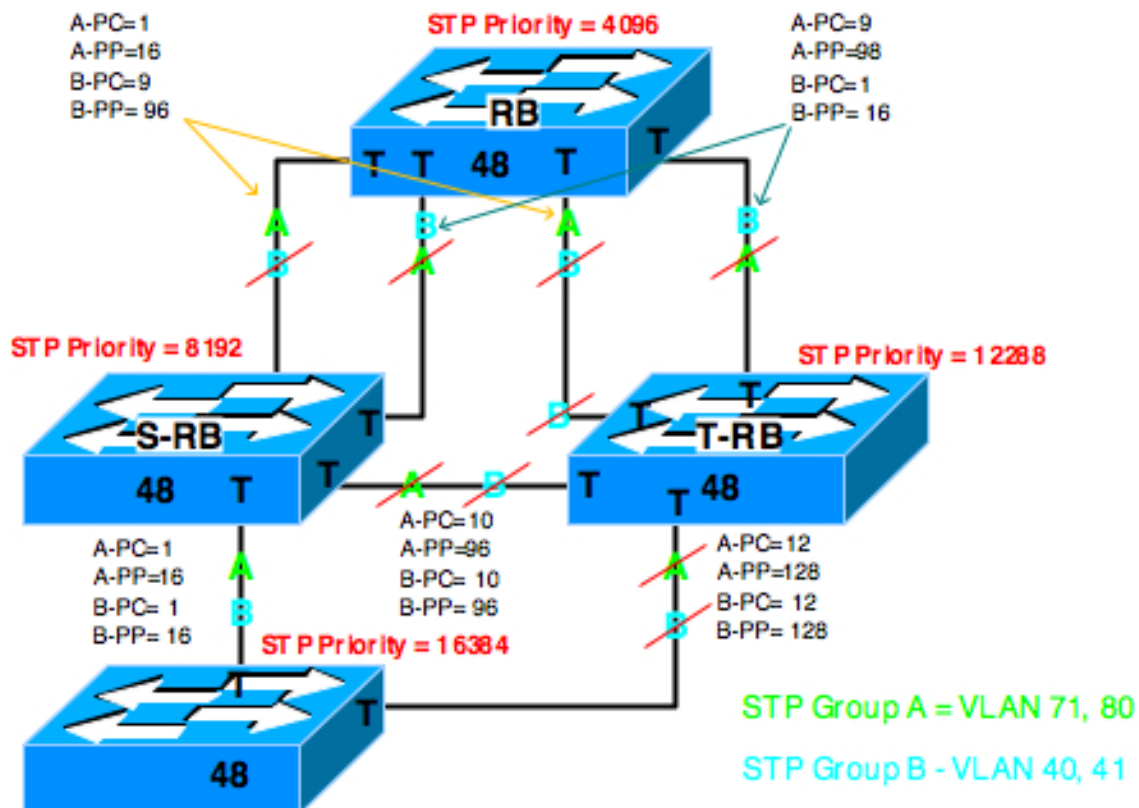
Michael J. Martin

When implementing port security on tagged interfaces, it's important to get the MAC address limits correct or you can potentially "black hole" legitimate hosts from having their traffic forwarded. There are a couple of different ways to calculate the allocations:

First, to determine your per-port MAC allocation, multiply the value by 2, then multiply the sum by the total number of switch ports that make up the LAN: $PPMA * 2 * TSPD = \text{MAC Address Limit}$. Apply the result to each VLAN on each trunk port. This approach will provide enough MAC table headroom for each switch port * PPA in the LAN to be assigned to the same VLAN and still be forwarded. This is clumsy and relies on the untagged ports' limitations to be correct for proper enforcement, but has the benefit of not needing to be readjusted each time a port is assigned to a VLAN. Remember, you have to plan for the possibility that each trunk port could have to forward all of the traffic generated by the active ports in the LAN.

The second option is more secure, but also has higher administrative overhead. Determine the per-port MAC allocation. Then calculate the number member ports in each VLAN. Multiply the PPA value by 2, then multiply the sum by the number of port members in a VLAN: $PPMA * 2 * TSPV = \text{MAC Address Limit for VLAN X}$. Perform this calculation for each VLAN, then apply the MAC allocation limitation on a per-VLAN basis on each trunk port. This approach provides a check against the port-level allocations, but has the downside of requiring the allocation to be recalculated each time a VLAN port membership changes.

The last thing you need to consider when configuring tagged switch interfaces is the per-VLAN port-level STP calculation values. The STP topology of the network should always be predictable. The way to ensure this is to set the STP values -- on the switches and per trunk -- on a per-VLAN basis, as these ports are typically the bridged interfaces that tie the LAN switching topology together. Here is an example STP topology:



Building VLAN Interfaces in Linux and IOS

Michael J. Martin

The STP topology depicted above splits the LAN's VLANs into the two STP groups A and B. The factors that need to be considered are:

- The election of the root bridge and its successors in the event of a hardware failure
- The selection of the primary path to the root bridge and the succeeding paths in the event of a link failure.

The switch with the highest bridge priority wins the STP root bridge election (highest priority being the lowest priority numeric value). Because each VLAN runs its own STP instance, bridge priority is set on a per-VLAN basis. The default value is 32768. The bridge priority is set using the global configuration command `<spanning-tree vlan {vid} priority {4096 – 61440}&rt;`. The priority values are set in 4096 increments. In the example above, the RB switch has a priority value for each VLAN of 4096, S-RB uses a bridge priority of 8192, and so on. Once the root bridge has been established, the next step is to configure the path section values, which are port-priority and port cost. In path selection, the port with the lowest priority will be selected to forward frames for the VLAN. In the event that all ports have the same priority, the port with the lowest cost will forward frames for the VLAN.

```
outlan-rs01(config-if)#spanning-tree vlan 40,41 port-priority 96
outlan-rs01(config-if)#spanning-tree vlan 71,80 port-priority 16
outlan-rs01(config-if)#spanning-tree vlan 40,41 cost 9
outlan-rs01(config-if)#spanning-tree vlan 71,80 cost 1
```

The VLANs port priority is set using the interface command `<spanning-tree vlan {vid,vid} port-priority {0-240}&rt;`. The port priority is set in increments of 16 up to 240. The lower the number, the higher the port's priority. The VLAN's port cost is set with the interface command `<spanning-tree vlan {vid,vid} cost {1- 200000000}&rt;`. Once again, as with port priority and bridge priority, the lower the number value, the more desirable the path. The switch assigns a default cost of 4 for Gigabit interfaces, 19 for 100 Mbps interfaces, and 100 for 10 Mbps interfaces. Keep this in mind when using custom values. Technically, either of the values can be used to manipulate the preferred forwarding path to the root bridge. Both are adjusted here for example purposes.

Now that our Cisco switch tagged port is configured, the last configuration aspect we need to review is setting up 802.1Q VLAN interfaces on Linux. The current 802.1Q Linux implementation supports the following:

- 4094 VLANs per interface (per IEEE standard)
- MAC-based VLANs (non-standard)
- Per-interface multicast
- Support for native VLAN

802.1Q support is available as a loadable kernel module or can be built right into the kernel (my preferred method). The kernel we built for our WLAN proxy server included support for 802.1Q in the kernel. Of course, like all 802.1Q tagged port implementations, 802.1Q Ethernet frames generated from (and sent to) VLAN interfaces on Linux have the additional 4 bytes added to the frame header to support the TPCD/TCI fields.

Although the Linux kernel supports VLANs, support at the Ethernet driver level is a whole other issue. The Ethernet drivers that come with the standard Linux kernel distribution do not support the 4 byte increase in the Ethernet frame that's required for VLAN support. Instead, the 4 byte increase, which enlarges the maximum Ethernet frame size from 1514 to 1518, is interpreted by the driver as a

Building VLAN Interfaces in Linux and IOS

Michael J. Martin

framing error known as a "giant" and the perceived oversized frames are then discarded. There are two workarounds for this problem. The first is to use drivers that support the increased frame size. There are patches available for the Intel EtherPro 10/100/1000 cards, HP 10/100/1000 Proliant cards, D-Link, 3Com, and others. The best bet is to check to see if modified drivers exist for your network card.

The other solution is to reduce the maximum transmission unit (MTU) size by 4 bytes. This is easily accomplished on the command line using the MTU option of the `/sbin/ifconfig` or `/sbin/ip` commands, alone or as part of the interface configuration string:

```
[root@vulcan root]# ifconfig eth1.41 mtu 1496 or
```

```
[root@vulcan root]# ip link set eth1.41 mtu 1496
```

The MTU value can also be set in interface configuration scripts located in `/etc/sysconfig/network-scripts/ifcfg-*`. It is a good idea to also adjust the TCP maximum segment size (MSS) when adjusting the MTU. On Linux, the TCP-MSS adjustments can be made per route basis using the `/sbin/route` or `/sbin/ip` command.

```
/sbin/route {route...} mss {1-1460}
/sbin/ip route {route...} advmss {1-1460}
```

The `/sbin/route` command does not display the MSS setting of the kernel routes, but the `/sbin/ip` command does. If the `/sbin/ip` command is not part of your Linux distribution, the `/bin/netstat` command can be used with `-nr` flags to print the kernel routing table with the MSS values:

```
[root@vulcan root]# netstat -nr
Kernel IP routing table
Destination      Gateway          Genmask         Flags   MSS Window  irtt  Iface
0.0.0.0          172.30.71.1    0.0.0.0        UG      1360 0        0     eth0
```

Configuring VLAN interfaces is two-step process. Step 1 creates the virtual physical interface. This is done with the `/sbin/vconfig` command. The prerequisite for using this command is that the physical interface to which the VLAN interface is bound is activated during the boot process or on the command line. For example, if we were adding VLAN interfaces to `eth1`, which was not activated at boot, we would execute the commands:

```
[root@vulcan root]# ifconfig eth1 up
[root@vulcan root]# vconfig add eth1 44
Added VLAN with VID == 44 to IF -:eth1:-
[root@vulcan root]# vconfig add eth1 45
Added VLAN with VID == 45 to IF -:eth1:-
[root@vulcan root]#
```

Once the VLAN interfaces have been created, the next step is to configure the IP address assignments on the VLAN interfaces using `/sbin/ifconfig` or `/sbin/ip` :

```
[root@vulcan root]# ifconfig eth1.44 192.168.20.4 netmask 255.255.255.0 mtu 1496
```

or

```
[root@vulcan root]# ip addr add 10.10.1.4/24 dev eth1.45
```

Building VLAN Interfaces in Linux and IOS

Michael J. Martin

```
[root@vulcan root]# ip link set eth1.45 mtu 1496
```

Once the IP interfaces are set we can verify the configuration with `/sbin/ifconfig` or `/sbin/ip addr`. The kernel stores VLAN status information in the `/proc` file system. To see this VLAN specific information, I.E., interfaces set, physical interface binding, tx/rx packets, etc. We can parse the `/proc net/vlan/*` file system data using the `cat` or `more` command:

```
[root@vulcan root]# cat /proc/net/vlan/*
VLAN Dev name      | VLAN ID
Name-Type: VLAN_NAME_TYPE_RAW_PLUS_VID_NO_PAD
eth1.44           | 44 | eth1
eth1.45           | 45 | eth1
eth1.44  VID: 44          REORDER_HDR: 1 dev->priv_flags: 1
      total frames received:          0
      total bytes received:           0
Broadcast/Multicast Rcvd:            0

      total frames transmitted:       0
      total bytes transmitted:        0
      total headroom inc:              0
      total encap on xmit:             0
Device: eth1
INGRESS priority mappings: 0:0  1:0  2:0  3:0  4:0  5:0  6:0  7:0
EGRESSSS priority Mappings:
eth1.45  VID: 45          REORDER_HDR: 1 dev->priv_flags: 1
      total frames received:          0
      total bytes received:           0
Broadcast/Multicast Rcvd:            0

      total frames transmitted:       0
      total bytes transmitted:        0
      total headroom inc:              0
      total encap on xmit:             0
Device: eth1
INGRESS priority mappings: 0:0  1:0  2:0  3:0  4:0  5:0  6:0  7:0
EGRESSSS priority Mappings:
[root@vulcan root]#
```

At this point, all of the VLAN interfaces are configured and online. Now there is only one small problem. In the event the server is rebooted, the VLAN configuration will be lost. To avoid having to build all of these VLAN interfaces after each reboot, the entire VLAN interface set-up process can be automated using a simple script that can be run from `/etc/rc.d/rc.local`:

```
#!/bin/sh
# Shutdown the Interface
/sbin/ifconfig eth1 down

# Bring up the Interface with the correct MTU
/sbin/ifconfig eth1 mtu 1496

# Create the VLAN interfaces:
/sbin/vconfig add eth1 44
```

Building VLAN Interfaces in Linux and IOS

Michael J. Martin

```
/sbin/vconfig add eth1 45

# Configure IP information and MTU values for the VLAN interfaces:
# The /sbin/ifconfig method
/sbin/ifconfig eth1.44 192.168.20.4 netmask 255.255.255.0 mtu 1496

# The /sbin/ip method
/sbin/ip addr add 10.10.1.4/24 dev eth1.45
/sbin/ip link set eth1.45 mtu 1496

# Print out the VLAN status info:
cat /proc/net/vlan/*

#end
```

This approach is fine if the interfaces can be activated once the boot process has been completed. However, if you need network services such as DHCP to support these interfaces and DHCP loads during the boot process, the alternative is to use the network configuration scripts. This is relatively easy to do; first just create needed interface configuration scripts in `/etc/sysconfig/network-scripts`. Here is an example:

```
# VLAN interface VID 40
DEVICE=eth1.40
BOOTPROTO=static
IPADDR=172.30.40.4
NETMASK=255.255.255.0
BROADCAST=172.30.40.225
NETWORK=172.30.40.0
TYPE=Ethernet
MTU=1496
ONBOOT=yes
```

If you chose to use DHCP to assign IP address information to VLAN interfaces and run into problems getting an address, enable the Ethernet header reordering option for the interface using the `<set_flag {vlan-int} 1` vconfig command option:

```
[root@vulcan root]# vconfig set_flag eth1.41 1
```

Set flag on device `-:eth1.41:-` Should be visible in `/proc/net/vlan/eth1.41`

```
[root@vulcan root]#
```

This option reorders the header to look like a physical Ethernet interface. This will reduce performance slightly, so the developer warns to use it only when needed.

Once the new network configuration scripts have been created, we need to finish the network scripts. The startup configuration file for the physical interface to which we are binding the VLAN interfaces needs to be configured to load at boot, without an IP address. Here is the boot script for interface eth1:

```
# Intel Corp. | 82540EM Gigabit Ethernet Controller
DEVICE=eth1
BOOTPROTO=none
```

Building VLAN Interfaces in Linux and IOS

Michael J. Martin

```
TYPE=Ethernet
ONBOOT=yes
MTU=1496
```

The last step is to create an rc script to create the VLAN interfaces during the boot cycle. Although Red Hat supports VLAN interfaces, there is no support for the VLAN interface set up in the S10Network rc script (other distributions do have this support). In this case, one needs to be created. The simplest way is to copy the S99local rc script and name the copy S11Vconfig. Then edit the file and add the needed /sbin/vconfig commands. Here is an example:

```
#!/bin/sh
vconfig add eth1 44
vconfig add eth1 45
```

By setting the Vconfig rc script name number one higher than the S10network rc script, the S11Vconfig script is executed immediately after the physical interface has been enabled. The best part is that the kernel accepts the VLAN interface network configuration information even though the logical interfaces have not been created. Once the S11Vconfig executes, the VLAN interfaces become active.

It is possible to configure VLAN interfaces on an active physical interface already configured with IP information. When covering the tagged port configuration on a Cisco IOS based switch, there was mention of the "native VLAN" option. The Linux VLAN implementation can take advantage of the native VLAN. Here is the /sbin/ifconfig output of an interface configuration utilizing the native VLAN option:

```
[root@vulcan root]# ifconfig
eth1      Link encap:Ethernet  HWaddr 00:30:48:43:86:99
          inet addr:172.30.40.4  Bcast:172.30.255.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1401 errors:0 dropped:0 overruns:0 frame:0
          TX packets:470 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:309904 (302.6 Kb)  TX bytes:244560 (238.8 Kb)
          Base address:0xa400 Memory:ec020000-ec040000

eth1.41   Link encap:Ethernet  HWaddr 00:30:48:43:86:99
          inet addr:172.30.41.4  Bcast:172.30.41.225  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1496  Metric:1
          RX packets:683 errors:0 dropped:0 overruns:0 frame:0
          TX packets:257 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:140276 (136.9 Kb)  TX bytes:120930 (118.0 Kb)
```

Notice in the output above that both interfaces have different MTUs. Support for this configuration is not "officially" mentioned in the developer's documentation. However, I have tested the configuration and found no issues so far. What is nice about this option is that if you are truly limited to a single physical interface, it provides the ability to run DHCP services on an interface that is also supporting VLAN interfaces without any compatibility worries.

That concludes the Linux networking portion of our WLAN proxy server. Next, we'll move onto configuring DHCP and DNS services. As always, questions and comments are welcome.