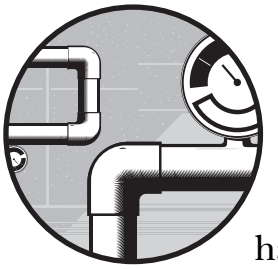
4

FILTERING FLOWS



The good news is, you now have actual data about your network. The bad news is, you have far too much data about your network. An Internet T1 might generate millions of flow records in a single day, while a busy Ethernet core might generate billions or more. How can you possibly manage or evaluate that heap of data? You must filter your data to display only interesting flows. The `flow-nfilter` program lets you include or exclude flows as needed.

You can filter traffic in almost any way you can imagine. For example, if a particular server is behaving oddly, you can filter on its IP address. If you're interested in HTTP traffic, you can filter on TCP port 80. You can reduce your data to include only interesting traffic, which will help you evaluate and diagnose issues. For example, if you have a large internal corporate network, you might want to view only the traffic exchanged with a particular branch office, filtering on all of its network addresses.

In Chapter 3, you viewed flow information by running `flow-cat` and feeding the resulting data stream to `flow-print`. Filtering takes place between these two processes: `flow-nfilter` accepts the data stream from `flow-cat` and examines each flow. Flows that match the filter pass on to `flow-print` (or other flow-processing programs); flows that do not match the filter drop from the data stream.

Filter Fundamentals

In this chapter, you'll start by building a few simple filters. Once you understand the basics of filter construction, you'll examine the various filter types and functions in depth.

NOTE *Define your filters in the file filter.cfg, which is probably in* /usr/local/flow-tools/etc/cfg/filter.cfg *or* /usr/local/etc/flow-tools/filter.cfg*, depending on your operating system and how you installed flow-tools.*

Common Primitives

You'll build your filters out of *primitives*. A primitive is a simple traffic characteristic, such as "port 80," "TCP," or "IP address 192.0.2.1." For example, those three primitives could be combined to create one filter that passes all TCP traffic to the host 192.0.2.1 on port 80.

`flow-nfilter` supports more than a dozen different primitives and can compare them with flows in more than two dozen different ways. A primitive looks much like this:

```
filter-primitive name
❶    type primitive-type
❷    permit value
```

The first line defines a filtering primitive and assigns the primitive a name.

The type at ❶ defines the characteristic you want to match on, such as an IP address, a port, or a time. (I'll cover the most commonly useful filter types.)

The permit statement at ❷ defines the values you're looking for. By default, a primitive denies everything, so you must explicitly state what your filter permits. Alternatively, you could use a `deny` statement to create a primitive that matches everything except what you're looking for and explicitly put a `default permit` statement at the end.

For example, a complete primitive that matches the IP address 192.168.1.1 looks like this:

```
filter-primitive ❶ 192.0.2.1
❷    type ip-address
❸    permit 192.0.2.1
```

At ❶ I've named my primitive after the address it matches. You can use any one-word name that makes sense to you, such as "mailserver" or "firewall," if you prefer. The `ip-address` primitive at ❷ matches network addresses. Finally, at ❸ this primitive matches any IP address equal to 192.0.2.1. If you include this primitive in a filter, it will pass traffic to or from this IP address only.

Similarly, the following primitive defines port 25:

```
filter-primitive ❶ port25
    type ❷ ip-port
    permit 25
```

Although I could have called this primitive `25`, at ❶ I used the name `port25` to make it absolutely clear that this primitive matches a port because the number 25 by itself could be a number of seconds, a count of octets or packets per second, an autonomous system, a floor number, and so on. (An IP address is unmistakable, so using the address as a name probably won't confuse you.)

The `ip-port` primitive at ❷ is another commonly used filter component. Including this primitive in a filter means that the filter will pass traffic only on port 25.

The default *filter.cfg* includes a primitive for TCP traffic, as shown here:

```
filter-primitive ❶ TCP
    type ❷ ip-protocol
    permit ❸ tcp
```

You're unlikely to mistake the name TCP at ❶ for anything other than the protocol, but the `ip-protocol` primitive at ❷ lets you create a primitive for any TCP/IP protocol. Of course, if you have obscure network protocols, you'll probably need to create additional protocol primitives, and your permit statements at ❸ can use either the protocol number or the protocol name from */etc/protocols*.

Each primitive can include only one type of match. For example, the following is invalid:

```
filter-primitive bogus-primitive
❶    type ip-port
     permit 25
❷    type ip-address
     permit 192.0.2.1
```

This primitive tries to match on both a port number (❶) and an IP address (❷). A primitive cannot do this. To filter out connections to the IP address 192.0.2.1 on port 25, you must assemble a filter from multiple primitives.

Now that you have a few primitives, you can create your first filter.

Creating a Simple Filter with Conditions and Primitives

Combine primitives into filters with the `filter-definition` keyword, like so:

```
❶ filter-definition name
❷     match condition primitive1
      match condition primitive1
      ...
```

Every filter begins with `filter-definition` (❶) and a name. Filters can share a name with a primitive but not with other filter definitions.

The filter contains a series of `match` keywords (❷), followed by conditions and primitives. The `match` keyword specifies the part of the flow this entry checks and the primitive to compare it to.

Conditions include things such as IP addresses, ports, protocols, types of service, and so on. All of the conditions listed must match for the filter to match a flow. For example, the following filter combines the `TCP` primitive and the `port25` primitive:

```
filter-definition TCPport25
❶     match ip-protocol TCP
❷     match ip-source-port port25
```

This filter passes all flows coming from TCP port 25. Any flow that does not come from TCP port 25 will not pass through the filter.

Although primitives and conditions look similar, their names can differ. For example, both filter conditions and filter primitives use the `ip-protocol` keyword (❶). When matching ports, however, primitives use the `ip-port` keyword (❷), but filter definitions use the `ip-source-port` and `ip-destination-port` keywords instead.

NOTE *The most common cause of filtering errors is using incorrect keywords. Use filter keywords only in filters, and use primitive keywords only in primitives.*

NAMING CONVENTIONS FOR FILTERS AND PRIMITIVES

Assign names to your filters and primitives carefully. If you initially choose ambiguous or confusing names, you'll trip over them when you have dozens or hundreds of filters! Make your names easy to recognize and unmistakable in purpose.

Primitives can share a name with a filter. For example, you can name a primitive TCP and a filter TCP, but you cannot name two primitives TCP or two filters UDP. Also, filter and primitive names are case insensitive. You cannot name one primitive `tcp` and another primitive TCP.

Using Your Filter

Use `flow-nfilter`'s `-F` option and the filter name to pass only the traffic that matches your filters. For example, here I'm printing only the flows that match the `TCPport25` report:

```
# flow-cat * | flow-nfilter -F TCPport25 | flow-print | less
srcIP            dstIP            prot  srcPort  dstPort  octets      packets
192.0.2.37       216.82.253.163   6     25       62627    1294        12
192.0.2.36       81.30.219.92     6     25       63946    1064        15
203.16.60.9      192.0.2.36       6     25       1054     1628        31
...
```

In this example, you can see only the flows where the protocol is 6 (TCP) and the source port is 25. This filter would be useful if you were investigating mail issues, for example. The filter shows that the mail server sent traffic from port 25, and hence the network level of the mail system is functioning.

Useful Primitives

Now that you understand how primitives and filters work together, I'll discuss primitives in depth. `flow-nfilter` supports many different primitives, but I'll cover only the most commonly useful ones here. The `flow-nfilter` man page includes the complete primitive list, but this book contains every one that I have used during several years of flow analysis.

Protocol, Port, and Control Bit Primitives

Filtering on network protocol and port information is one of the most common ways to strip a list of flow records down to only interesting traffic.

IP Protocol Primitives

You saw a basic IP protocol primitive earlier, but you can check for protocols other than TCP. For example, if you use IPSec, OSPF, or other network protocols that run over IP but that are not over TCP or UDP, you'll eventually need to view them separately. Filtering by protocol is the only way to differentiate between network applications that share port numbers, such as syslog (UDP/514) and rsh (TCP/514).

When defining a protocol filter, you can use either the protocol number or name from */etc/protocols*. I prefer to use the number so that */etc/protocols* changes won't interfere with traffic analysis. For example, OSPF runs over protocol 89, so here's a filter to match it:

```
filter-primitive OSPF
    type ip-protocol
    permit 89
```

Similarly, IPSec uses two different protocols: ESP (protocol 50) and AH (protocol 51). The following primitive matches all IPSec traffic. (Separate multiple entries with commas.)

```
filter-primitive IPSec
    type ip-protocol
    permit 50,51
```

Although the IPSec protocols don't have port numbers, `flow-nfilter` can show you how much bandwidth an IPSec VPN between any two points uses and where your VPN clients connect from.

NOTE *The default filter.cfg includes primitives for TCP, UDP, and ICMP.*

Port Number Primitives

Most network applications run on one or more ports. By filtering your output to include the port only for the network service you're interested in, you ease troubleshooting. To do so, use the `ip-port` primitive you saw earlier.

```
filter-primitive port80
    type ip-port
    permit 80
```

A single primitive can include multiple ports, separated with commas like so:

```
filter-primitive webPorts
    type ip-port
    permit 80,443
```

If you have a long list of ports, you can give each its own line and add comments. This example includes services that run over TCP (telnet and POP3) as well as UDP (SMB).

```
filter-primitive unwantedPorts
    type ip-port
    permit 23   #telnet
    permit 110  #unencrypted POP3
    permit 138  #Windows SMB
...
```

You can also create primitives for ranges of ports.

```
filter-primitive msSqlRpc
    type ip-port
    permit 1024-5000
```

IP port primitives can use names from */etc/services*, but I recommend using numbers to insulate you from changes or errors in that file. `flow-print` and `flow-report` can perform number-to-name translations if necessary.

TCP Control Bit Primitives

Filtering by TCP control bits identifies abnormal network flows. Use the `ip-tcp-flags` primitive to filter by control bits. (See "TCP Control Bits and Flow Records" on page 50.)

```
filter-primitive syn-only
    type ip-tcp-flags
    permit 0x2
```

This primitive matches flows with only a SYN control bit, also known as a *SYN-only flow*. Either the server never responded to the request, a firewall blocked the connection request, or no server exists at the destination address.

These flows are fairly common on the naked Internet, where viruses and automated port scanners constantly probe every Internet address, but they should be comparatively uncommon on your internal network. Numerous SYN-only flows on an internal network usually indicate misconfigured software, a virus infection, or actual intruder probes.

Similarly, you can filter on flows that contain only an RST. An RST-only flow indicates that a connection request was received and immediately rejected, generally because a host is requesting service on a TCP port that isn't open. For example, if you ask a host for a web page when that host doesn't run a web server, you'll probably get a TCP RST.

```
filter-primitive rst-only
    type ip-tcp-flags
    permit 0x4
```

Although a certain level of this activity is normal, identifying the peak senders of SYN-only and RST-only flows can narrow down performance problems and unnecessary network congestion.

To identify flows with multiple control bits set, add the control bits together. For example, flows that contain only the SYN and RST control bits indicate system problems. To identify these flows, write a filter that matches SYN+RST packets.

```
filter-primitive syn-rst
    type ip-tcp-flags
    permit 0x6  # 0x2 (SYN) plus 0x4 (RST)
```

Once you start examining TCP control bits on even a small network, you'll find all sorts of problems and quickly ruin your blissful ignorance.

ICMP Type and Code Primitives

Different ICMP type and code messages can illuminate network activity. Although you can filter flows based on ICMP type and code, it's not exactly easy to do so.

Flows encode the ICMP type and code as the destination port. A primitive that matches a particular type and code uses the `ip-port` primitive. ICMP type and code are usually expressed as hexadecimal, but `ip-port` takes decimal values. (Use Table 3-4 on page 53 to identify the appropriate decimal values.)

For example, suppose you're looking for hosts that send ICMP redirects. Redirects are ICMP type 5 and come in two codes, 0 (redirect subnet) and 1 (redirect host). In hexadecimal, these would be 500 and 501. Table 3-4 shows their decimal values as 1280 and 1281, so write a primitive like this:

```
filter-primitive redirects
    type ip-port
    permit 1280-1281
    default deny
```

Used in a filter by itself, this primitive would pass ICMP, TCP, and UDP flows. When you create the actual filter, use both this primitive and the ICMP primitive to see only ICMP redirects.

IP Address and Subnet Primitives

Filtering flows by addresses and subnets lets you narrow down data to hosts and networks of interest.

IP Addresses

Primitives for IP addresses use the `ip-address` type. It's reasonable to name primitives after the IP address they match, because IP addresses are difficult to confuse with other types of filter primitives.

```
filter-primitive 192.0.2.1
    type ip-address
    permit 192.0.2.1
```

One primitive can include any number of addresses.

```
filter-primitive MailServers
    type ip-address
    permit 192.0.2.10
    permit 192.0.2.11
```

A primitive such as this `MailServers` example lets you match multiple hosts that serve a particular function, such as "all web servers," "all file servers," and so on.

Subnet Primitives

Primitives can also match subnets using the `ip-address-mask` and `ip-address-prefix` primitives. Flow-tools provides two different formats for subnets, `ip-address-mask` and `ip-address-prefix`, to match the two common notations for expressing subnets.

The `ip-address-mask` primitive expects a full IP network address with the netmask in decimal form, as follows:

```
filter-primitive our-network
    type ip-address-mask
    permit 192.0.2.0 255.255.255.0
```

This primitive matches all hosts with an IP between 192.0.2.0 and 192.0.2.255.

The `ip-address-prefix` primitive uses prefix (slash) notation.

```
filter-primitive our-network
    type ip-address-prefix
    permit 192.168.0/24
    permit 192.168.1/24
```

You can include multiple subnets, each on its own line, in the subnet primitive, and the subnet masks or prefixes do not have to be equal in all the entries. For example, the following is a perfectly valid primitive:

```
filter-primitive mixed-netmasks
    type ip-address-prefix
    permit 192.168.0/23
    permit 192.168.2/24
```

This primitive matches any IP address between 192.168.0.0 and 192.168.2.255.

Time, Counter, and Double Primitives

You can filter flows by times during the day or by arbitrary counter values.

Comparison Operators in Primitives

Time and counter primitives use logical comparison operators, as shown in Table 4-1.

Table 4-1: Time and Counter Comparison Operators

Operator	Comparison	Time
`gt`	Greater than	Later than
`ge`	Greater than or equal to	This time or later
`lt`	Less than	Earlier than
`le`	Less than or equal to	Earlier than or equal to
`eq`	Equal	Exactly this time

Use these comparison operators *only* in time and counter primitives, not in filter definitions.

Time Primitives

To filter according to when flows began or stopped, use a `time` primitive. For example, here, you're looking for flows that stop or start some time during the minute of 8:03 AM.

```
filter-primitive 0803
    type time
    permit eq 08:03
```

NOTE *Remember, flow records use a 24-hour clock, so 8:03 PM is filtered as 20:03.*

You can narrow down a time period even further. For example, if you know that the traffic you're interested in started and stopped during the second of 8:03:30 AM, you can write a primitive for that.

```
filter-primitive 0803
    type time
    permit eq 08:03:30
```

You cannot filter on millisecond time intervals. Sensors and collectors are rarely accurate to milliseconds, however.

To define a time interval, use other comparison operators. For example, suppose you know that something happened on your network between 7:58 AM and 8:03 AM. To filter traffic during this time period, define a time window from 7:58 to 8:03, inclusive, with the `ge` and `lt` operators, like so:

```
filter-primitive crashTime
    type time
    permit ge 07:58
    permit le 08:03
```

Although you can control the data you report on by selecting which flow files to analyze, using times helps narrow your searches even further. This is invaluable when examining large files, and it demonstrates the need for accurate time on your network.

NOTE *`flow-nfilter` also supports the `time-date` primitive for a specific date and time, such as January 20, 2011, at 8:03 AM. If you're interested in a specific date, however, you're better off analyzing the flow files for that date. Flow files are named for the year, month, day, and time of their creation for a reason.*

Counter Primitives

The `counter` primitive lets you create filters like "more than 100 octets" or "between 500 and 700 packets." When creating filters of this sort, use one or more comparison operators with integers to define counters, as follows:

```
filter-primitive clipping
    type counter
    permit gt 10000
```

This particular filter would pass anything that has more than 10,000 of what you're trying to measure. As another example, suppose you want to look at flows that last only 1,000 milliseconds (1 second) or longer. Here's how you could do that:

```
filter-primitive 1second
    type counter
    permit ge 1000
```

Or, perhaps you want only flows of 1KB or larger.

```
filter-primitive 1kB
    type counter
    permit ge 1024
```

You can use multiple comparisons in a counter. For example, here, I'm permitting everything greater than 1,000 and less than 2,000:

```
filter-primitive average
    type counter
    permit gt 1000
    permit lt 2000
```

NOTE *When using the `counter` primitive, keep in mind that counters work only when filtering based on octets, packets, and/or duration. Counters will not match TCP ports or IP addresses.*

Double Primitives

No, a `double` primitive isn't twice as primitive as the rest of flow-tools. A `double` primitive is a `counter` with a decimal point. It matches either packets per second or bits per second.

For example, suppose you want to ignore all connections that send 100 or more packets per second. You need a primitive to define the 100 part of that.

```
filter-primitive lessThan100
    type double
    permit lt 100.0
```

You'll see how to tie this to the number of packets per second in a filter definition, but this primitive defines the "less than 100" part of the filter.

Like the `counter` primitive, the `double` cannot match arbitrary data. It can match only octets, packets, and duration.

Interface and BGP Primitives

Flow records exported from a router include routing information, but most of this information is useful only if you're using dynamic routing such as Border Gateway Protocol (BGP). If you are not using BGP or other dynamic routing protocols, you can skip this section.

Identifying Interface Numbers Using SNMP

Most router configuration interfaces (such as Cisco's command line) give each router interface a human-friendly name such as FastEthernet0 or Serial1/0. Internally, the router knows each interface by a number. The router uses the interface number in flow records, rather than the human-friendly name.

The simplest way to get the list of interface names and their corresponding numbers is through Simple Network Management Protocol (SNMP). If you're using multiple Internet providers, you almost certainly have some sort of SNMP capability. Most Unix-like systems include the net-snmp software suite, so I'll use that as an example. Other SNMP browsers should present similar results.

Remember, SNMP presents information as a hierarchical tree. To get a list of network interfaces, check the `RFC1213-MIB::ifDescr` branch of the SNMP tree. To see interface names and numbers, use `snmpwalk` to query the router's `RFC1213-MIB::ifDescr` values. If your MIB browser doesn't support human-friendly names, `RFC1213-MIB::ifDescr` is equivalent to `.1.3.6.1.2.1.2.2.1.2`.

```
# snmpwalk -v ❶ 2 -c ❷ community ❸ router RFC1213-MIB::ifDescr
RFC1213-MIB::ifDescr.❹1 = STRING: ❺ "FastEthernet0/0"
RFC1213-MIB::ifDescr.2 = STRING: "FastEthernet0/1"
RFC1213-MIB::ifDescr.4 = STRING: "Null0"
RFC1213-MIB::ifDescr.5 = STRING: "T1 0/0/0"
RFC1213-MIB::ifDescr.6 = STRING: "T1 0/0/1"
RFC1213-MIB::ifDescr.7 = STRING: "Serial0/0/0:0"
RFC1213-MIB::ifDescr.8 = STRING: "Serial0/0/1:1"
RFC1213-MIB::ifDescr.9 = STRING: "Tunnel1"
```

In the previous example, at ❶ you query a router with SNMP version 2, using its community name (❷) and the router's hostname or IP address (❸). In response, you get a list of router interface names.

The SNMP index is the router's internal number for the interface. For example, at ❹ interface 1 is named *FastEthernet0/0* (❺). Interface 7 is named *Serial0/0/0:0*, and so on.

Network engineers should notice that of the eight interfaces listed, interface 4 (null0) is a logical interface and should never see any traffic. Similarly, interfaces 5 and 6 are not real interfaces; they are interface cards supporting interfaces 7 and 8. Only five of the eight interfaces should ever pass traffic.

By default, Cisco routers can change their interface numbering on a reboot, which prevents gaps in interface numbering when interfaces are added or removed. Interface numbers that change arbitrarily really confuse long-term reporting, however. I recommend making your router maintain consistent interface numbering across reboots. It's true that this leaves gaps in the interface list; note the absence of interface 3 on the example router. On the other hand, interface 7 is always Serial 0/0/0:0, even years later. Tell a Cisco device to leave interface numbering unchanged with the configuration option `snmp-server ifindex persist`.

Also, note that if you have multiple routers exporting data to a single collector, you must separate the data to get meaningful interface information. For example, interface 8 on router A might be a local Ethernet interface, while interface 8 on router B might be an upstream T1 interface. You can filter data by exporter IP address, but this creates the need for an extra layer of filtering.

I'll use the previous interface list in the upcoming examples. Interfaces 1 and 2 are local Ethernet ports, interfaces 7 and 8 are T1 circuits to two different Internet service providers, and interface 9 is a VPN tunnel. The other interfaces should never see traffic.

Interface Number Primitive

Filtering by interface passes only the traffic that traversed that interface. Use the `ifindex` primitive for this purpose.

```
filter-primitive vpnInterface
    type ifindex
    permit 9
```

Interface 9 is the VPN interface. Filtering on it shows you only traffic that goes over the VPN.

(You can list multiple interfaces on one line.)

```
filter-primitive localEthernet
    type ifindex
    permit 1,2
```

Filtering by interface lets you focus on how traffic flows between particular network segments.

Autonomous System Primitives

The Autonomous System (AS) is the core of BGP routing, and routers with BGP peers include AS number information in their flow exports. You can pull out traffic from particular AS numbers with the `as` primitive as follows:

```
filter-primitive uunet
    type as
    permit 701
```

You can list multiple AS numbers separated by commas on a single line, or you can even list a range of AS numbers. Of course, you can also add multiple AS numbers on separate lines. (ARIN, RIPE, and other AS registrars frequently issue AS numbers to large organizations in blocks, so you might need to create such a filter.)

```
filter-primitive uunet
    type as
    permit 701-705
```

You can also write filters for route announcement prefix length using the `ip-address-prefix-len` primitive. I haven't found a use for a filter that says "Show me all the routes we're getting that are /25 or longer," but carriers and transit providers might find it useful to identify clients that are trying to announce tiny networks.[1]

Filter Match Statements

To assemble primitives into filters, use `match` statements. `flow-nfilter` compares each flow against every `match` statement in a filter, and if a flow fits every `match` statement, the flow passes through. If the flow does not fit every `match` statement, the flow is removed from the data stream.

Many match types have names that are similar to their associated primitives. For example, the `ip-protocol` primitive has a corresponding `ip-protocol` match. Other primitives have no single matching condition. For example, the `ip-port` primitive can match either the `ip-source-port` primitive or the `ip-destination-port` primitive. If you use an incorrect `match` statement in your configuration, `flow-nfilter` exits with an error.

Filter definitions support many different types of match condition. The `flow-nfilter` manual page has the complete list, but the ones I find useful are described here.

Protocols, Ports, and Control Bits

Matching protocols and ports is very common. Control bits and ICMP type and code are less common but powerful in a different way.

Network Protocol Filters

Use the `ip-protocol` match type to check each flow against an `ip-protocol` primitive.

I previously defined a primitive for OSPF. Here I'm using that primitive to pass only OSPF traffic:

```
filter-definition OSPF
  match ip-protocol OSPF
```

Listing multiple protocol primitives in a filter will cause no packets to match. After all, very few single flows are both TCP and UDP.

Source or Destination Port Filters

`flow-nfilter` has separate matches for source ports (`ip-source-port`) and destination ports (`ip-destination-port`). These match against the `ip-port`

1. If you're not a transit provider but are trying to announce tiny networks, the lesson you should learn here is this: Tiny route announcements won't work, and if they do, they c*an* find you.

primitive. Here I'm using the `port80` primitive defined earlier to filter traffic to a web server:

```
filter-definition port80
    match ip-destination-port port80
```

To match multiple ports for one service, define a primitive that includes all the ports for that service. For example, earlier I defined a `webTraffic` primitive for ports 80 and 443.

```
filter-definition webTraffic
    match ip-destination-port webTraffic
```

Use the `ip-source-port` similarly. For example, to capture traffic leaving your web server, filter the flows leaving ports 80 and 443. (You'll see how to write reports that match both arriving and departing traffic in "Logical Operators in Filter Definitions" on page 76.)

```
filter-definition webTraffic
    match ip-source-port webTraffic
```

TCP Control Bit Filters

Use the `ip-tcp-flags` keyword to match TCP control bit primitives. For example, I previously defined a `rst-only` primitive that matched flows that contained TCP resets only.

```
filter-definition resets
    match ip-tcp-flags rst-only
```

This filter displays only the flows that match the `rst-only` primitive. You don't need to specify a protocol, because flow records contain control bits only for TCP flows. You could use very similar filters for the other TCP control bit primitives.

ICMP Type and Code Filters

Remember that flows record the ICMP type and code in the destination port field of ICMP flows. However, unlike TCP control bits, which appear only in the records of TCP flows, destination ports appear in TCP, UDP, and ICMP flows. To specifically match ICMP type and code, your filter must include the destination port and the protocol as follows:

```
filter-definition redirects
❶    match ip-destination-port redirects
❷    match ip-protocol ICMP
```

I previously defined a `redirects` primitive at ❶ that matched both codes within the ICMP redirect type. Here, I'm adding a match (❷) for the `ICMP` protocol primitive as well. This filter passes only the flows that contain ICMP redirects.

Addresses and Subnets

`flow-nfilter` supports two match types for IP addresses: source (`ip-source-address`) or destination address (`ip-destination-address`). These match types can work on any of the three IP address primitives: `ip-address`, `ip-address-mask`, or `ip-address-prefix`.

You can match the source address on one line and the destination address on another line. For example, suppose you have an `ip-address-prefix` primitive for your client's network and another for your web servers. The following definition passes traffic from your client to your web server:

```
filter-definition clientsToWeb
    match ip-destination-address webServers
    match ip-source-address clientNetwork
```

You cannot list multiple matches of the same type in a single filter because a single flow cannot have multiple source or destination addresses! To pass traffic from several source or destination addresses, use a primitive that contains all the desired addresses.

The next filter captures data coming into the server from web clients. You need a corresponding report to catch traffic from your web servers to the client network (or a slightly more complicated filter to capture traffic moving in both directions, as you'll see in "Logical Operators in Filter Definitions" on page 76). Because you want to see only web traffic, you also filter with primitives for web traffic and TCP.

```
filter-definition clientsToWebHttpTraffic
    match ip-port webTraffic
    match ip-protocol TCP
    match ip-destination-address webServers
    match ip-source-address clientNetwork
```

You'll see other ways to achieve this same effect in "Using Multiple Filters" on page 75.

Filtering by Sensor or Exporter

Multiple flow sensors can export to a single collector, but at times you'll want to see only the flows that came from a particular sensor. You can use the `ip-exporter-address` match with any IP address primitive to create a filter that passes flows from a particular sensor, as follows:

```
filter-primitive router1
    type ip-address
    permit 192.0.2.1

filter-definition router1-exports
    match ip-exporter-address router1
```

This particular filter passes only the flows exported from the router at 192.0.2.1.

Time Filters

The `start-time` and `end-time` match types let you filter on when flows begin and end, using the `time` primitive. For example, the following sample captures all flows that take place entirely within a particular minute, using the `0803` time primitive defined earlier:

```
filter-definition 0803
    match start-time 0803
    match end-time 0803
```

You can define a filter to match flows starting or ending at any time that you can express with a primitive.

In most cases, you won't have accurate time information about problems. Human beings have a notoriously fuzzy time sense: "A few minutes ago" might be anything from 30 seconds to an hour, and after a few days even that is unreliable. Remember that each flow file covers a five-minute period. Most of the time you're better off searching entire flow files for issues rather than trying to filter on times. I find that filtering on times is useful only on very large flow files and then only when you have precise timing information from the flow files themselves. A human saying that the website broke at 8:15 AM is not reliable. If your flow records say that you had unusual traffic at 8:15 AM, however, you might want to see what else happened during that minute. Filtering on times can be useful in that instance.

Clipping Levels

A *clipping level* is the point at which you start ignoring data. For example, you might not care about flows that contain tiny amounts of data, or perhaps you want to see only tiny flows. To clip data, you can set clipping levels on the amount of traffic transmitted, the connection speed, and the duration of connections.

Octets, Packets, and Duration Filters

Use `counter` primitives to filter based on the number of octets per flow, the packets per flow, or the duration of flows. For example, earlier I defined a primitive for 1KB or larger. Let's use that primitive now to remove the tiny connections from the flow data.

```
filter-definition 1kBplus
    match octets 1kB
```

Similarly, you created a primitive for anything that totaled 1,000 or more, called `1second`. You can write a filter that uses this primitive to allow only flows of 1,000 milliseconds (1 second) or longer.

```
filter-definition over1second
    match duration 1second
```

Counters are arbitrary numbers and can apply to octets, packets, or duration. For example, if you want a filter that includes only flows with 1,024 or more packets, you could easily reuse the `1kB` primitive for that.

```
filter-definition 1024plusPackets
    match packets 1kB
```

Even though you can, I try not to reuse primitives in this way. You never hear of a kilobyte of packets! Such filters confuse me. Being confused while trying to identify network problems is not good.[2]

Packets or Bits per Second Filters

Perhaps you're interested in how quickly connections move or you're interested only in the really fast or really slow connections. If so, you can use `double` primitives to filter based on packets per second or bits per second.

For example, earlier you defined a `double` primitive for less than 100. You can use this for either packets per second or bits per second.

```
filter-definition lessThan100pps
    match pps lessThan100

filter-definition lessThan100bps
    match bps lessThan100
```

In this particular case, I don't mind reusing the `lessThan100` primitive, because the name isn't so closely tied to a particular data type.

BGP and Routing Filters

You can filter flows based on the routing information included in the flow records. (If you are not using BGP, you can skip this section.)

Autonomous System Number Filters

The `source-as` and `destination-as` match types let you match based on AS numbers. For example, this filter lets you see what traffic you're receiving (from what was the UUnet network) using the `uunet` AS primitive defined earlier:

```
filter-definition uunet
    match source-as uunet
```

You could also turn this around to create a filter to permit the traffic you're sending to UUnet systems.

2. I don't need to waste my time calling myself an idiot because I gave a filter an ambiguous name. Many other people are delighted to call me an idiot for all sorts of reasons.

Next-Hop Address Filters

The *next hop* is the IP address where a router sends a flow. This is usually the IP address on the remote end of an ISP circuit (for outgoing flows) or the external address of your firewall (for inbound flows). Routers include the next hop in flow records. However, software flow sensors like `softflowd` know nothing of interfaces on remote hosts or how packets are routed, so flows exported from software flow sensors do not contain next-hop addresses.

Now suppose that the next-hop IP address for one of your Internet providers is 61.118.12.45. To filter all traffic leaving your network via that ISP, you could use a primitive and a definition like this:

```
filter-primitive ispA
    type ip-address
    permit 61.118.12.45

filter-definition ispA
    match ip-nexthop-address ispA
```

The `ip-nexthop-address` match type works with the primitives `ip-address`, `ip-address-mask`, and `ip-address-prefix`.

Interface Filters

Another way to filter by provider or network segment is to filter by the router interface. The match types `input-interface` and `output-interface` let you filter by traffic arriving or leaving your router.

You defined a primitive for router interface 9 earlier. Here I'm using it in a filter:

```
filter-definition vpn
    match input-interface vpnInterface
```

This shows traffic entering the router on this interface.

Using Multiple Filters

Suppose you want to identify all traffic between two machines. You could define primitives for those two hosts and then write a filter that specifically defines those hosts. However, this common situation will keep you very busy writing new filters. Instead, I find it much easier to define smaller filters and tie them together on the command line.

You can invoke `flow-nfilter` repeatedly in a single command. Find the flow files for the times you're interested in, filter them for the first host, and then filter them a second time for the second host.

```
# flow-cat ft-* | ❶ flow-nfilter -F host1 | ❷ flow-nfilter -F host2 | flow-print | less
```

The first `flow-nfilter` invocation at ❶ passes only flows that include traffic from `host1`. The second at ❷ passes only flows that include traffic from `host2`.

Similarly, you can write separate filters for certain protocols, like all web traffic. You previously created a filter for all HTTP and HTTPS traffic, called `webTraffic`.

```
# flow-cat ft-* | ❶ flow-nfilter -F host1 | ❷ flow-nfilter -F webTraffic | flow-print | less
```

The first filter at ❶ passes only traffic for the interesting host, and the second (❷) passes only HTTP and HTTPS traffic.

You can create simple filters for important hosts and subnets on your network. For example, if you have a customer who reports problems reaching your website, you could write one flow filter for your site and one for the customer's addresses and use them both to see what traffic passed between your networks. You could then look for SYN-only or RST-only flows that would indicate problems. Or you might find that traffic from the customer's network never reaches you at all. In any case, these two filters will tell you exactly what traffic appeared on your network and how it behaved.

By combining filters on the command line, you will write fewer filters and get more use out of the filters you create.

Logical Operators in Filter Definitions

When you put multiple match conditions in a filter definition, `flow-nfilter` places a logical "and" between them. For example, the following filter shows all traffic that runs over TCP and has a source port of 25. This passes an email server's responses to a connection.

```
filter-definition TCPport25
    match ip-protocol TCP
    match ip-source-port port25
```

You can use other logical operators to build very complicated filters.

Logical "or"

When I try to analyze a connection problem, I usually want to see both sides of the conversation. I want a filter that will show connections to port 25 as well as from port 25. For this, use the `or` operator as follows:

```
filter-definition email
    match ip-protocol TCP
    match ip-source-port port25
❶   or
❷   match ip-protocol TCP
❸   match ip-destination-port port25
```

After the `or` statement at ❶, a whole new filter definition begins. Although I listed `TCP` in the first filter, if you're interested in TCP in the second filter, you must repeat the match on TCP at ❷, after which you can add the new `match` statement at ❸ to catch flows that end on port 25. Now, if you apply this filter to your flow data, you'll see something like this:

```
# flow-cat ft-v05.2011-12-20.12* | flow-nfilter -F email | flow-print | less
srcIP            dstIP            prot  srcPort  dstPort  octets      packets
❶ 217.199.0.33     192.0.2.37       6     5673     25       192726      298
❷ 192.0.2.37       217.199.0.33     6     25       5673     8558        181
  206.165.246.249  192.0.2.37       6     38904    25       13283       22
  192.0.2.37       206.165.246.249  6     25       38904    1484        16
  ...
```

The first flow at ❶ is from a remote IP to the address of the local email server, with a destination port of 25. This is an incoming mail transmission. The second flow at ❷ is from the mail server to the same remote IP address; it's coming from port 25. This is the response to the first flow.

I could use more sophisticated `flow-print` formats to view this in more detail, run `flow-report` on this data to check for errors, or add another filter to specifically point out TCP errors in the email stream. This simple check shows me that the mail server is exchanging substantial amounts of traffic on TCP port 25, however. I would tell my mail administrator to check the logs for errors or provide more information.

Filter Inversion

Sometimes it's easier to write a filter for the traffic you're *not* interested in. For example, suppose you want to see all the traffic to or from your email servers that isn't email. Although you could write primitives that included all port numbers except those for email, that's annoying and tedious.

Instead, use the `invert` keyword to reverse the meaning of a filter, like so:

```
filter-definition not-email
❶    invert
     match ip-protocol TCP
     match ip-source-port port25
     or
     match ip-protocol TCP
     match ip-destination-port port25
```

By adding `invert` to the report at ❶, you pass everything that doesn't match the defined filters. In this example, I'm passing every network transaction that doesn't involve TCP port 25.

But there's a problem with this filter: It will match all nonemail traffic on all the hosts for which you're capturing data. You, however, need to view only traffic for your email hosts.

To solve this problem, you could add your email servers into the `not-email` filter, but the email servers both send and receive email. You would need a definition section for remote servers connecting to your mail servers, a section for your servers' response to those remote servers, a third section for your mail servers connecting to remote mail servers, and a fourth for the remote servers' responses to your servers' requests. That's pretty ugly.

It's much simpler to define a separate filter that strips the flow data down to just the email servers and then to concatenate the two, as follows:

```
❶ filter-primitive emailServers
  type ip-address
  permit 192.0.2.37
  permit 192.0.2.36

❷ filter-definition emailServers
  match ip-source-address emailServers
  or
  match ip-destination-address emailServers
```

The `emailServers` primitive at ❶ includes the IP addresses of all the mail servers. Next, at ❷ I create a filter definition to match all traffic leaving or going to those servers. Then, to see all nonemail traffic to or from my email servers, I do this:

```
# flow-cat * | ❶ flow-nfilter -F emailServers | ❷ flow-nfilter -F not-email | flow-print | less
```

The `emailServers` filter at ❶ passes only the flows that involve my email servers. The `not-email` filter at ❷ passes only flows that are not SMTP. By combining these two filters, I see only interesting traffic. I'll probably need to adjust the filter further to remove other uninteresting traffic, such as DNS queries to the DNS server, but I'm almost there.

Of course, after reviewing the filtered traffic, I can go ask my email administrator why he's running his own DNS server on the mail server instead of using the corporate name servers and why he browses the Web from those machines instead of using the proxy server and its adult content filters.[3]

Filters and Variables

Flow-tools also includes filters that can be configured on the command line, which can be useful for very simple filters, such as identifying traffic from a particular IP address. The default filters that use these are fairly limited, but they'll suffice for simple traffic analysis. It's also easy to write your own variable-driven reports.

3. Yes, I could take this straight to human resources, but HR won't wash and wax my car.

Using Variable-Driven Filters

The filters that are configurable on the command line use three variables: `ADDR` (address), `PORT` (port), and `PROT` (protocol). These support five reports, letting you filter by protocol as well as by source and destination address and port: `ip-src-addr`, `ip-dst-addr`, `ip-src-port`, `ip-dest-port`, and `ip-prot`.

Suppose your boss calls. She's connecting from a random open wireless hotspot in some inconvenient city and can't get into the corporate VPN concentrator. You get her IP address, either by asking her for it or by accessing system logs to see where she's coming from. To see all the traffic coming to your network from her IP, without writing a custom filter, you could use a command-line variable on the flow files for that time window. For example, if she's at the IP address 192.0.2.8, you'd use a command like this:

```
# flow-cat * | flow-nfilter -F ip-src-addr ❶ -v ADDR=192.0.2.8 | flow-print
```

The `-v` argument at ❶ tells `flow-nfilter` that you're assigning a value to a variable. In this example, I've assigned the value 192.0.2.8 to the variable `ADDR`. You'll see all traffic originating from that IP address.

WHEN TO USE VARIABLE-DRIVEN FILTERS?

For simple filters on individual hosts and ports, use variable-driven filters. If you must filter on multiple hosts or ranges of ports, define primitives and filters in *filter.cfg*.

Defining Your Own Variable-Driven Filters

Variable-driven filters take advantage of the primitives `VAR_ADDR` (address), `VAR_PORT` (port), and `VAR_PROT` (protocol), as defined in *filter.cfg*. For example, the following is a default variable-driven filter that uses the `ADDR` variable. This looks exactly like a standard report, except that it uses the variable name instead of a primitive.

```
filter-definition ip-src-addr
    match ip-source-address VAR_ADDR
```

Use these variables to define your own variable-driven filters. For example, I like to see all traffic to *and* from a host of interest. Writing a command-line version of this report is easy.

```
filter-definition ip-addr
    match ip-destination-address VAR_ADDR
    or
    match ip-source-address VAR_ADDR
```

Similarly, I prefer to see all traffic to *and* from a port simultaneously.

```
filter-definition ip-port
    match ip-destination-address VAR_PORT
    or
    match ip-source-address VAR_PORT
```

With these reports, I can dynamically filter for any individual host or port on the fly.

Creating Your Own Variables

`VAR_ADDR`, `VAR_PORT`, and `VAR_PROT` are not magic variables hard-coded into `flow-nfilter`; they're defined in *filter.cfg*. Here's the definition of `VAR_PORT`:

```
filter-primitive VAR_PORT
    type ip-port
    permit ❶ @{PORT:-0}
```

Most of this primitive looks like any other primitive for a port number, but the `permit` statement (❶) is very different. This example takes the variable `PORT` as defined on the command line and turns it into a number. The specifics of how this works aren't important, but you can use this sample as a model for your own primitives.

Now here's another example. I frequently work with BGP, so I need an AS number primitive.

```
❶ filter-primitive VAR_AS
❷ type as
❸ permit @{AS:-0}
```

I've assigned this primitive the name `VAR_AS` at ❶ to correspond with the existing variable names, and I've assigned it the `as` type (❷). The `permit` statement at ❸ is copied from the `VAR_PORT` primitive, substituting the variable name `AS` for the port. Now I can create a filter using this variable.

```
filter-definition AS
❶     match source-as VAR_AS
      or
❷     match destination-as VAR_AS
```

This closely resembles the earlier custom variable–based filters in that you pass traffic going to ❶ and from the specified AS (❷). Now you can use this filter to get the traffic to a particular autonomous system.

```
# flow-cat * | flow-nfilter -F as-traffic -v AS=701 | flow-print -f 4 | less
```

When you apply this filter, you'll see only the flows involving AS number 701.

At this point, you should be able to filter traffic in any way you like. Now let's run analysis on that data.