

Chapter 17: Filters to Detect, Filters to Protect

You learned about the concepts of discovering events of interest by configuring filters in Chapter 8, "Introduction to Filters and Signatures." That chapter showed you various different products with unique filtering languages to assist you in discovering malicious types of traffic. What if you are a do-it-yourself type, inclined to watch Bob Vila or "Men in Tool Belts?" The time-honored TCPdump program comes with an extensive filter language that you can use to look at any field, combination of fields, or bits found in an IP datagram. If you like puzzles and don't mind a bit of tinkering, you can use TCPdump filters to extract different anomalous traffic. Mind you, this is not the tool for the faint of heart or for those of you who shy away from getting your brain a bit frazzled. Those who prefer a packaged solution might be better off using the commercial products and their GUIs or filters.

This chapter introduces the concept of using TCPdump and TCPdump filters to detect events of interest. TCPdump and TCPdump filters are the backbone of Shadow, and so the recommended suggestion is to download the current version of Shadow at <http://www.nswc.navy.mil/ISSEC/CID/> to examine and enhance its native filters. This takes care of automating the collection and processing of traffic, freeing you to concentrate on improving the TCPdump filters for better detects.

Specifically, this chapter discusses the mechanics of creating TCPdump filters. You learn different techniques for excavating bytes and bits within bytes of the IP datagram using these filters. Different TCPdump filters are developed to show you how to extract anomalous content found in the IP header, UDP header, and TCP header. This chapter tries to build on these foundations and lead up to developing more complex and advanced filters.

The Mechanics of Writing TCPdump Filters

Often, you will want to examine certain fields in the IP datagram for signs of malicious activity directed at your network. TCPdump filters need to specify an item of interest, such as a field in the IP datagram, for record selection. Such items might be part of the IP header (the IP header length, for example), the TCP header (TCP flags, for example), the UDP header (the destination port, for example), or the ICMP message (message type, for example).

TCPdump provides some macros for commonly used fields, such as "port" to indicate a source or destination port or "host" to indicate an IP number or name of a source or destination host. We won't use these in the examples—not for the sake of proud academics, but because the fields we are interested in do not have macros, and so we must use the format of referencing a field by the protocol and displacement in terms of bytes into that protocol.

TCPdump assigns a designated name for each type of header associated with a protocol. Much as you would expect, "ip" is used to denote a field in the IP header or data portion of the IP datagram, "tcp" for a field in the TCP header or data of the TCP segment, "udp" for a field in the UDP header or data of the UDP datagram, and "icmp" for a field in the ICMP header or data of the ICMP message.

Now, we have to reference a field in a given protocol by its displacement in bytes from the beginning of the protocol header. For instance, ip[0] indicates the first byte of the IP datagram, which happens to be part of the IP header (remember counting starts at 0). tcp[13] is byte 14 into the TCP segment, which is also part of the TCP header, and icmp[0] is the first byte of the ICMP message, which is the

ICMP message type.

For this discussion, we use the following format to create a TCPdump filter:

```
<protocol header>[offset:length] <relation> <value>
```

All of the initial filters that this chapter covers reference [Figure 17.1](#), which is the standard layout of the IP header. Notice that each of the rows has 32 bits, ranging in value from 0 through 31. Essentially, each row is composed of 4 bytes; and don't forget that counting starts with 0. That is one of the hardest things to commit to memory.

Figure 17.1

The IP header.

Suppose that you want to use TCPdump to select any datagram that has an embedded protocol of ICMP. Refer to [Figure 17.1](#) and notice this particular protocol field is located 9 bytes (last reminder: start counting at 0) into the IP header. Therefore, we denote this field as `ip[9]`. Notice also that the TCPdump filter format called for an `offset:length`; the implied length is 1 byte, and the length is used if you want to span more than a single byte. Now that you have located the 1-byte field that stores the embedded protocol, you need to know that a value of 1 in this field represents ICMP. To compose the entire filter to find ICMP records, use the filter `ip[9] = 1`. If this were used to collect records off the network, you would run TCPdump as follows:

```
tcpdump 'ip[9] = 1'
```

This reads from the default network interface and collects only ICMP records. You embed the filter in single quotation marks to keep the UNIX shell from trying to interpret the filter. Another TCPdump option used for more complicated filters is the `-F` option of TCPdump, which points TCPdump to a file where the filter is located. You could have put the filter `ip[9] = 1` in a file `/tmp/filter` and executed the following command:

```
tcpdump -F /tmp/filter
```

This would have yielded the same results as the TCPdump command that included the filter in the command line itself. This option is usually used for long filters or automated TCPdump processes to avoid command-line entry of the filter.

Bit Masking

We need to introduce a couple more pointers while we're at it. The TCPdump filter language is not a robust language compared to the constructs and operations available in other languages, such as C or Perl, for instance. Often, we have to go back to the ancient roots of assembler language-like manipulations to extract fields that don't fall on byte boundaries.

TCPdump is fairly straightforward and coherent when you are dealing with a field that falls on a byte boundary and you are looking at all 8 bits. Although you have discovered how to span bytes by specifying the length after the offset, what happens if you want to look at only certain bits or a range of bits in a byte. In other words, you don't want to look at the entire byte. This is where things get a little hairy, and this discussion assumes that you have mastered the rudiments of binary and

hexadecimal.

Preserving and Discarding Individual Bits

Take a look at the structure of the IP header again. Now look at the first byte in the IP header and notice that it is actually two 4-bit fields. Each of these 4-bit fields is known as a *nibble*. What if you wanted to examine the 4-bit header length only, and didn't care about the value in the 4-bit version field. You really just want to look at the low-order nibble. How do you discard the high-order nibble so that you can concentrate on the value of the 4-bit IP header length alone? In essence, you want to turn the high-order 4 bits into 0s. Doing so enables you to reference the first byte and look at the low-order nibble alone. If the question "how the heck do I do that?" is rolling around the tip of your tongue, you are following this discussion in hot pursuit.

Remember back to Boolean arithmetic? A well-deserved groan or two is merited or even expected. Personally, I don't remember anyone who enjoyed a good truth table, but unfortunately, you have to delve back into the far recesses of your brain to resurrect the Boolean AND operator. Does Table 17.1 bring back any nightmares?

Table 17.1 AND Truth Table

BIT A	AND	BIT B	RESULT
0		0	0
1		0	0
0		1	0
1		1	1

This table shows all the possible binary bit values and the results of AND'ing the bits. The only time that 2 bits have a resulting value of 1 is when both AND'ed bits are 1. What does this mean to this discussion of TCPdump filters? You might have forgotten the original challenge: You need to zero-out the high-order nibble of the first byte in the IP header so that you can focus on the low-order nibble. Well, what if you can AND the value found in the first byte of the IP header with all 0s in the high-order nibble, which has the effect of discarding them. Then, you can preserve the original value in the low-order nibble by AND'ing all those bits with 1s.

Consider how this is done. Take a look at [Figure 17.2](#). In the rectangles, you see the first byte of an actual IP header divided into two 4-bit chunks. Examine the value in the datagram; the high-order nibble has a value of 0100 with a 1 in the 2^2 position, which yields 4. This is the version of IP – IP version 4, in this case. Now look at the low-order nibble. It has a value of a 1 in the 2^2 position and a 1 in the 2^0 position, so we have a $4 + 1$ (or 5). This is the IP header length. Very unfortunately, the metric for this is not bytes as you might expect. It would be a lot easier that way, but to save on space required to store this value, this represents not a byte, but a word. A word is 32 bits, or 4 bytes. To convert a value that you find in this length field to bytes, you must multiply by 4. What this means is that this is a 20-byte header length, which is typical for a header that has no options.

Creating the Mask

Let's get on with the task of discarding the 4 high-order bits. Look at [Figure 17.2](#) again, but this time at the line under the actual value found in the first byte of the IP datagram. This is what we have designated the "mask," or the byte that will be AND'ed with the original value, bit by bit to discard the high-order bits and preserve the low-order bits. If you were to start the process at the high-order (leftmost) bit, you would find a 0 in the value bit and a 0 in the mask bit. On the line below it, you see the resulting bit is 0. Logically, we have a 0 in the value bit that we AND with a 0 in the mask bit and the result is a 0. Remember if we AND any value bit with a 0, the result is a 0. Using this line of thought, our other mask bits for the high-order nibble are also 0s. As you see, the resulting value for the high-order nibble is 0000, which is exactly what we wanted—to zero-out this field to focus on the lower-order nibble.

Because we are dealing with an entire byte, we also need to mask the low-order nibble as well; we cannot ignore that. Starting with the leftmost bit of the low-order nibble, we find a 0 in the value bit and a 1 in the mask bit. These two values AND'ed yield a 0, thereby preserving the original value bit. Next we see that a 1 in a value bit AND'ed with a 1 in the mask bit also preserves the value bit. You can see the pattern; all 1s in the mask for the low-order nibble preserves the low-order nibble. And, looking at the resulting value, we see that we have accomplished what we set out to do—to look exclusively at the value of the IP header length. Yes, we have to go through all of this because we cannot look at just part of a byte! *Whew!* We need to cover just one more step about the mechanics of writing filters and then we can turn to the actual filters themselves. How do we tell TCPdump to perform the AND operation, and with what value?

Figure 17.2 Bit masking.

First, we want to represent the mask bytes as two hexadecimal characters; 0000 1111 can be translated to 0x0f. The 0x informs TCPdump that this value is in hexadecimal; its default base is decimal. Here is how to construct the partial filter:

```
ip[0] & 0x0f
```

This says to take the value found in the first byte of the IP header and AND it with a hexadecimal value of 0f.

Putting It All Together

We are dealing with the first byte of the IP header; we AND that byte with a hexadecimal 0x0f and we have just managed to focus on the IP header length. Why might you want to isolate this field? One very good reason is to test for the presence of IP options. The normal IP header is 20 bytes, or 5 32-bit words. That means that an IP header that might contain a dangerous IP option, such as source routing, would have a length of greater than 5 found in this field. IP options are almost never used any more for anything other than evil intent, so we want to know whether IP options exist. Recall from the TCPdump filter syntax that you need a relation and a value. The entire filter to find a signature of an IP datagram that has IP options is as follows:

```
ip[0] & 0x0f > 5
```

That is it; end of a very long story. I know this seems like a lot of work and a lot of theory, but it truly

does get easier as you get more practice. I warned you about the tinkering part; if you followed this and think you understand, however, you're well on your way to examining any field including bits of the IP datagram. Not many IDS offer this capability. With TCPdump, you lose no fidelity in your ability to capture and analyze data. Again, not many other IDS can make this claim. That is why it might be worth your while to become familiar with TCPdump and TCPdump filters.

TCPdump IP Filters

Some of the telltale indications in the IP header that you might be a target of reconnaissance include traffic sent to your broadcast address, fragmentation, and the presence of IP options. You should never see legitimate traffic sent to your broadcast address from outside your network, and you should block this traffic as previously mentioned to prevent the likes of mapping and Smurf attacks. As you learned, fragmentation is a natural enough byproduct of a datagram traveling to your network that originated on a network with a larger MTU than one of the routes it took to get to your network. But, you also saw how fragmentation can be used for denial-of-service attacks or to try to bypass notice by an IDS or routers that cannot keep track of state.

Detecting Traffic to the Broadcast Addresses

Let's define the broadcast address as one with a final octet of 255 or 0. This includes most broadcast addresses subdivided on classic byte boundaries. Take a look again at [Figure 17.1](#). The destination address is found in bytes 16 through 19 (32 bits) of the IP header. We are only concerned with the final octet, or byte 19. We can describe the broadcast addresses as follows:

```
ip[19] = 0xff
ip[19] = 0x00
```

or as a combined filter as follows:

```
ip[19] = 0xff or ip[19] = 0x00
```

We tend to express ourselves in hexadecimal and not decimal, but you could have as easily written this filter:

```
ip[19] = 255 or ip[19] = 0
```

Depending on where the sensor host is that runs the TCPdump filter, you might pick up broadcast traffic inside your network. Assume, for example, that your inside network is 192.168.x.x. To further qualify this filter to examine only traffic directed toward your network from a foreign source, you tweak the filter as follows:

```
not src net 192.168 and (ip[19] = 0xff or ip[19] = 0x00)
```

This just introduced a new operator, the `not`, to negate; and a couple of new macros: `src`, to indicate the traffic originated from this source; and `net`, to indicate a subnet. This filter says you want to look at any traffic that originates from a source network other than your own that is destined for the broadcast addresses. If you start TCPdump with this filter or collect TCPdump data and later read it back with this filter, it picks up attempted efforts to send broadcast traffic to your network.

Detecting Fragmentation

In this section, you exercise some of your new knowledge of the mechanics of writing TCPdump filters to look for fragmentation. All fragments in the fragment train except the last one have the more fragments (MF) bit set. If you can discover how to locate this field and see whether it is set, you can find most of the fragmented traffic directed your way. Look again at [Figure 17.1](#). You see that the more fragments bit is in the second row of the IP header. Can you figure out what byte it is in?

Specifically, if you count into the IP header, you will find it in the sixth byte. It is the third bit from the left of the high order-bit. Look at [Figure 17.3](#) to see how you might mask all surrounding bits except this one. Your mask needs to be 0010 0000, which is a hexadecimal 0x20. Your filter becomes `ip[6] & 0x20 != 0`. You use a generic relation and value of `!= 0`. This means that the more fragments bit is set. Why not just say `ip[6] & 0x02 = 1`? After all, aren't you testing that the exact bit is set? Not really; the problem with this is that you are not testing the bit value, but the value of the result of masking the original byte and the mask byte. Therefore, you need to examine the resulting value in context of where it falls in the whole byte. If the more fragments bit is set, it falls in the byte in the 2^5 position of the byte, which is 32. A generic `!= 0` is a little easier to express the result. Alternatively, you can write the filter as `ip[6] & 0x02 = 32`. Keep in mind that because fragmentation is not always malicious, you are likely to generate false positives with this filter.

Figure 17.3

Identifying the more fragments bit.

You have now seen how to express three TCPdump filters for potentially anomalous settings in the IP header. Now, turn your attention to some of the other protocols and how you can use TCPdump filters to discover other sorts of events of interest.

TCPdump UDP Filters

Many backdoors and Trojans use UDP ports, most notably port 31337 or Back Orifice. To detect UDP connections, you must decide on which UDP ports you want to examine directed activity. Take a look at <http://www.sans.org/y2k/ports.htm> for an idea of some of the types of ports you might want to watch. Configure your filters to watch for activity to these ports. If you want to look for traffic to Back Orifice, for example, your filter is as follows:

```
udp and dst port 31337
```

The labor is not in figuring how to express this as a TCPdump filter; as you see, it is trivial. The labor is involved in deciding which ports you want to include, adding them to the filter, and keeping the filter current with the real world of ever-expanding UDP exploits.

Consider a popular UDP application, traceroute. The UNIX traceroute works by attempting to send UDP datagrams to high-numbered ports of the destination host. If a host on your network is that destination host, you want to be alerted of the attempted or successful traceroute. If you begin by looking at UDP activity to ports in the 33000–33999 range, you will find most of the traceroute activity. Be warned that Windows traceroutes use ICMP echo requests and replies, so this signature does not detect that activity. And, be forewarned that some versions of traceroute enable the user to

provide command-line options, one of which is a destination port. Therefore, this filter might not capture all traceroute activity, but it will find most of the conventional activity.

Figure 17.4 shows the layout of the UDP header. Notice that the UDP destination port number is found in bytes 2 and 3 of the UDP header. A very insightful question to ask is "why don't we use the port macro rather than byte displacements?" For instance, why can't we use this filter:

```
dst port >= 33000 and dst port < 34000
```

Figure 17.4

The UDP header.

The problem is that when TCPdump uses a range such as this and not one exact value, you have to express that field in terms of the primitive protocol and displacement and forgo the use of macros. The correct syntax to discover traceroutes then becomes this:

```
udp[2:2] >= 33000 and udp[2:2] < 34000
```

Notice the first use of the length option [2:2] to span bytes. You need to examine 2 consecutive bytes starting at byte 2. You can further limit the amount of traffic that this filter extracts by examining the TTL value along with the destination port. traceroute operates by manipulating the TTL value found in the IP header. traceroute records the routers that it traverses and does so using an incrementing TTL value. More often than not, you will see a TTL of 1 on the sensor host running TCPdump before it crosses a router that will expire it. This is a signature of traceroute. Therefore, let's embellish the traceroute filter to include the TTL value to eliminate some of the noise associated with discovering traceroutes. The TTL field is found in the IP header; it has no macro to reference it, and if you look once again at Figure 17.1, you find it in the eighth byte. Here is what the new filter would look like:

```
udp[2:2] >= 33000 and udp[2:2] < 34000 and ip[8] = 1
```

This gives you an idea of some of the UDP filters. TCPdump filters can also be used for ICMP traffic. Specifically, some of the good candidates for detection of anomalous ICMP traffic are address mask requests, someone trying to discover the MTU of your network sending datagrams with the don't fragment bit set and receiving back messages from your router with the MTU, and loki. All these filters are so simple to write; we will leave these for you to try. Here are the signatures of TCPdump filters for you to write on your own. (You can check your filters against the ones at the end of this chapter.)

- The address mask request has a value of 17 in the first byte of the ICMP message.
- The fragmentation required, but DF flag set message has a 3 in the first byte of the ICMP message and a 4 in the second byte of the message.
- A signature for Loki was an echo request (an 8 in the first byte of the ICMP message) or an echo reply (a 0 in the first byte of the ICMP message) and in the sixth and seventh bytes of the ICMP message, you would have a hexadecimal value of 0xf001 or 0x01f0.

TCPdump TCP Filters

TCPdump filters for TCP traffic are mostly concerned with initial SYN connections and other types of anomalous flag combinations that might indicate some kind of reconnaissance or mapping efforts. We want to look for initial SYN connections because they inform us of attempted connections to a TCP port. This doesn't necessarily mean that they were successful. If your TCPdump sensor is located outside a packet-filtering device that blocks access to the TCP destination port, it will never reach the host. And, if the traffic is allowed through the packet-filtering device, it is possible that the host doesn't offer the attempted service. You can glean a lot of intelligence by detecting this activity, the least of which is discovering rogue TCP ports that hosts on your network might be offering.

Filters for Examining TCP Flags

Figure 17.5 relates to most of the remaining filters into this chapter.

The TCP flag bits are located 13 bytes into the TCP header. Because you are looking for individual bits in the bytes, you need to perform some bit-masking to select the flag or flags you want to examine. Begin by writing a filter to extract records with the SYN flag alone set:

```
tcp[13] & 0xff = 2
```

Why this filter? We see that our mask consists of all 1s. Why didn't we use a mask of 0s in all fields except the SYN flag (`tcp[13] & 0x02 = 2`)? By masking a bit with a 0, the resulting value is necessarily 0. The value bit could be 1, however, and the 0 mask would discard it. If this is confusing, try an example.

Suppose that you want to look at TCP segments with the SYN flag alone set. Okay, now suppose that you have a TCP flag byte with both the SYN and ACK flags set. The binary value that you would see for the TCP flag byte would be 0001 0010. If that were masked with 0000 0010, you would end up with a result of 0000 0010, which is 2. Therefore, masking with 0s in fields other than the SYN flag selects TCP segments with other flags set along with the SYN flag. To prevent this from occurring, you use the original filter and look for an exact value. This filter does not select records with other flags set along with the SYN flag.

Figure 17.5

The TCP flag byte.

Take a look at some other TCP flag combinations you might want to know about:

- **tcp[13] = 0** This shows null scans with no flags set. This condition should never occur.
- **tcp[13] = 3** This shows activity where both the SYN and FIN flags are set simultaneously; this is definitely an anomalous condition. You might want to alter the filter to `tcp[13] & 0x03 = 3`, because this gets any activity with both the SYN and FIN flags set, as well as any other flags set. In this case, you don't necessarily want to limit this to SYN and FIN alone.
- **tcp[13] = 0x10 and tcp[8:4] = 0** This shows activity with the ACK flag set, but with an acknowledgement value of 0. Any TCP segment with the ACK flag on should have a minimum acknowledgement value of 1 that occurs during the three-way handshake. At least 1 sequence number has to be consumed to elicit a valid acknowledgement; otherwise no acknowledgement

would be returned. This filter captures nmap operating system fingerprinting scans that send TCP traffic to various destination ports with the ACK flag alone set, but a 0 value in the acknowledgement field.

- **tcp[13] >= 64** Figure 17.5 shows two high-order bits in the TCP flag byte that are labeled reserved bits. These 2 bits should be 0s; if they are not, something is amiss. The first reserved bit is found in the 2^6 (64) position, and the second is found in the 2^7 (128) position. If either or both bits are set, the value for the TCP flag byte is greater than or equal to 64. Our old friend nmap sometimes sets one or both of these bits to perform operating system fingerprinting. Most hosts reset these values to 0s, but some leave the set value. This is used by nmap to help classify the operating system behavior.

These are just some of the different combinations of TCP flags that you can examine. This is not an exhaustive list, and I encourage you to play with these filters and develop different combinations. Be warned, however, that there are a lot of anomalous flag combinations and you might run out of TCPdump registers if your filter becomes too expansive.

Detecting Data on SYN Connections

Before letting you loose to develop some TCPdump filters of your own, let's take a look at one advanced filter that will summon up all the various bits and pieces you have learned in the chapter about developing filters and then some. In Chapter 2, "Introduction to TCPdump and Transmission Control Protocol (TCP)," you learned that data should not be sent before the three-way handshake has been completed. You saw this activity with the 3DNS product, which is a nuisance but ostensibly not malicious. You also read about the example of a scan that a site received in which there was data included on the SYN. It was feared that this type of activity might be an attempt to elude an IDS that started stream or data assembly for data received after the three-way handshake only.

It seems prudent then to try to develop a TCPdump filter that would detect this activity. You could later put in exclusions for annoying false alarms from 3DNS activity. The problem is that no field in the TCP segment has the number of bytes in the TCP payload. You do have a bevy of other fields that have length values in them, however. Specifically, you have two length fields in the IP header; one is the length of the entire IP datagram, the other is the length of the IP header itself. In the TCP segment, you have the length of the TCP header. Figure 17.6 shows that the length of the IP datagram minus the length of the IP header minus the length of the TCP should leave the TCP payload length.

"Piece of cake," you say? You will encounter some complications, or challenges (your choice). Notice the different metrics in different fields; the IP datagram length is in bytes, whereas the IP header and TCP header are in 32-bit words. You must standardize to bytes and convert the header lengths to bytes by multiplying them by a factor of four. This is quite manageable. You have already dealt with the IP header length, and so you have pretty much conquered that.

One final bit of nastiness that you need to address is the TCP header length seen in Figure 17.7. Look carefully at where this is located; it is in the high-order nibble of the 12th byte. You already know that you have to zero-out the low-order nibble to deal with the high-order nibble exclusively, but you aren't quite ready to tackle the formula just yet. Because this is in the high-order nibble, it is really multiplied by a factor of 16 more than you want, so it has to be normalized.

Suppose, for example, that you have a TCP header length of 24 bytes that includes a 20-byte header

and some TCP options. Remember that you have to convert to 32-bit words, so you need to divide by 4 to compute which value would be found in the TCP header length field. You would find in this field a value of 6. Assume you have also masked the low-order nibble so that the hexadecimal value remaining in this byte is 60. The binary representation of this is 0110 0000. A 1 is in the 2^6 position (64) and a 1 is in the 2^5 position (32), which really means you have 96. Because this field is in the high-order nibble, it is really 16 times a value found in a low-order nibble. To normalize this back to 6, you need to divide by 16. Summing up all the manipulations to this field, you want to normalize by dividing by 16 and then convert to bytes by multiplying by 4. Now you are ready to tackle this filter.

Figure 17.6

Calculating the TCP payload length.

Figure 17.7

The TCP header.

Let's revisit the conditions and formula we want in pseudo-code before attempting the TCPdump filter.

If the SYN flag alone is set, subtract from the IP datagram total length, the IP header length converted to bytes, and the TCP header length normalized and converted to bytes, and check to see whether the resulting value is non-zero.

SYN flag alone is set:

```
tcp[13] = 2
```

Total length of the IP datagram:

```
ip[2:2]
```

IP header length converted to bytes:

```
((ip[0] & 0x0f)*4)
```

TCP header length normalized and converted to bytes:

```
((tcp[12] & 0xf0)/16*4)
```

which is the same as:

```
((tcp[12] & 0xf0)/4)
```

Now put it all together to see the final filter:

```
tcp[13] = 2
and
( ip[2:2] -
  ((ip[0] & 0x0f)*4) -
  ((tcp[12] & 0xf0)/4)
) != 0
```

This discovers any traffic that attempts to include data on the initial SYN. Pretty awesome!

Summary

This chapter has shown that although TCPdump filters might not win most-likely-to-succeed in a beauty pageant of IDS filters, they can do some amazing things. Yes indeed, you need to get your hands soiled and you need to think pretty darn hard many times when attempting to debug a filter that does not work. But, these filters give you full access to your data. I cannot emphasize enough that when you smell something foul with your data, you want the ability to analyze at the bit level. (TCPdump filters afford you this power.) Literally, you want to leave no bit unturned when you are conducting in-depth analysis.

Most of the filters that you write using TCPdump will probably use macros and probably won't require any bit masking. When you need to examine individual bits or disjoint bits in a byte, however, you must isolate the bits of interest using bit masking. The other gotcha with TCPdump discussed in this chapter is standardizing on metrics with different length fields; make sure you convert to bytes. Finally, remember that the location that bits fall in the byte is significant. It might be necessary to normalize if you are dealing with bits in the high-order nibble. If you are up to the challenge of all of this, I think you will get a true sense of satisfaction after you have mastered the deciphering of data and the creation of potentially revealing filters.

Answers to ICMP filters:

```
- icmp[0] = 17
- ((icmp[0] = 3) and (icmp[1] = 4))
- (((icmp[0] = 0) or (icmp[0] = 8)) and
  ((icmp[6:2] = 0xf001) or (icmp[6:2] = 0x01f0)))
```

© Copyright Pearson Education. All rights reserved.