

Introduction

What are Berkeley Packet Filters? BPF's are a raw (protocol independent) socket interface to the data link layer that allows filtering of packets in a very granular fashion¹.

Support for BPF is compiled into the kernel in UNIX-like hosts, or if not, libpcap/Wireshark allows this to be done at user mode level. If done via user mode, all packets are copied up from the interface and not just the ones the filter specifies.

BPF were first introduced in 1990 by Steven McCanne of Lawrence Berkeley Laboratory, according the FreeBSD man page on bpf².

Working with BPF

If you use tcpdump for very long, you encounter what are called “primitives”, filter expressions to tune your results to only see certain traffic. Examples of primitives are “net”, “port” “addr” and qualifiers to those such as “src” or “dst”.

With these we can limit our results using filters such as ‘src host 10.10.1.1’ or ‘net 10.10’. There are many of these (see the man page of tcpdump for the full list)

You can also specify protocols, such as “ip”, “tcp”, or “icmp”. Some even make comparisons, such as “less” and “greater” for packet length.

These primitives are short cuts for BPF's. Each one references some field or fields in one of the network protocol headers. For example, the embedded protocol field in the IP header is the 9th byte offset from 0. If the value contained there is a 6, the packet is TCP. So the primitive “tcp” really means show me all the packets in the IP header whose 9th byte offset from 0 contains a 6. If we wrote this as a BPF, it would look like this: ‘ip[9] = 6’ or using hex, ‘ip[9] = 0x06’ .

BPF's can go far beyond the built-in primitives, allowing us to get as granular as needed, down the single bit level. If a field does not span the entire byte, we'll need to write a BPF to look at the bits in question to determine the value there.

Let's look at the first line of the IP header³ to see an example.

Byte	0	Byte 1	Byte 2	Byte 3
IP Version	IP Header length	Type of Service	Total Length	

We see byte 0 (we start counting from 0, which is what we mean by offset from 0) that there are two fields in the byte, the IP Version field and the IP Header Length Field.

If we wanted to see what the IP version of the packet is, how we would do this? We only want the value in the high order nibble (high order = left most as we count bits from right to left, and a nibble is 4 bits, or half a byte). To see that value we have to extract it from the byte of data somehow and look at it singularly. To do this, we employ a method known as bitmasking. Bitmasking is simply filtering out the bits we don't wish to look at and retaining the ones we do.

To accomplish this, we'll perform a bitwise AND operation on all of the bits in the byte. If we AND the bits, only the ones with a value of 1 will be retained. Let's look at this.

Here's a binary representation of a typical first byte in the IP header:

0 1 0 0 0 1 0 1

We've separated the two nibbles here for clarity. We see the low order nibble (right-most) has 0101. This is our IP header length. We want to check the high order nibble, which has the value 0100. To do this we will add 1 to each bit. In a bitwise AND, any values except two 1's equal 0. Two 1's equal one.

So to manipulate the bits to see the first nibble only, we want to add 1's to the high order nibble and 0's to the lower order. Since all 1's will equal F in hex, we will write an expression adding hex F to the first nibble and 0 to the second.

Here's what the BPF will look like:

`ip[0] & 0xF0 = 0x04` (our search value).

Broken down, we are telling tcpdump to look at the IP header (`ip`), first byte offset from 0 (`[0]`), retain all the bits in the first nibble and discard all the bits in the low order nibble (`& 0xF0`) and show us all the packets with a value of 4 in that nibble (`= 4`).

Here's our bit wise operation...

```
0 1 0 0   0 1 0 1
```

```
1 1 1 1   0 0 0 0
```

```
0 1 0 0 0 0 0 0
```

We now see the low order nibble has been filtered (all 0's) and we have the high order nibble left. Binary 0100 = decimal 4, so this shows us the packet has value of 4 in the high order nibble of the first byte; the IP header is set to IPv4.

Sample Filters

Now that we see how BPF's work, here are some samples of filters we can search on:

<code>'ip[9] = 0x11'</code>	udp
<code>'ip[9] = 0x01'</code>	icmp
<code>'tcp[2:2]'</code>	2nd byte, spanning two bytes
<code>'icmp[0] = 0x08'</code>	echo request packet
<code>'tcp[2:2] < 0x14'</code>	tcp dest port < 20

Let's create a filter for one of the more common and more complex uses: TCP Flags

The flags field in TCP is found at the 13th byte offset from 0. The flags themselves inhabit all of the lower order nibble, and the two lower order bits of the high order nibble.

The two high order bits of the high order nibble are used for ECN (Explicit Congestion Notification). Here's our layout...

TCP Byte 13

Flags	CWR	ECE	Urg	Ack	Push	Reset	Syn	Fin
Binary Values	128	64	32	16	8	4	2	1

Let's assume we wish to see all packets with the SYN and FIN flags set. This is anomalous behavior and usually indicative of a port scanning method.

We would need to look at the whole byte and retain all bits except the two highest orders in the high order nibble. To do this we need a mask retaining all of the lower order nibble and the lower order bits of the high order nibble. Here's our bitwise operation mask:

	High order					Low order			
CWR	ECE	Urg	Ack	--	Push	Reset	Syn	Fin	
0	0	1	1	--	1	1	1	1	

So, we would have a hex F in the low order nibble (all 1's), and a 3 in the high order nibble (a 1 in the 0 column, which is 1, and a one in the 2's column, which is 2, equaling 3).

So our mask would be 0x3F. That would show us only the bits that contain TCP flags.

If we use that filter and look for a value of 3, meaning the two lowest order bits are set, the Fin and Syn bit, we would end up with this:

```
'ip[13] & 0x3f = 0x03'
```

This filter tells the system to filter on the 13th byte offset from 0, discarding the two highest order bits, and showing packets that have a total value of 3 in the six remaining bits, which would mean the Fin and the Syn flags were both flipped on.

Now that we know how to look at only the bits we need, we can apply this to any field, in any network header. You can, of course, string multiple filters together to get as specific as needed. Here's a tcpdump query to show us all packets with the Syn flag set, and a datagram (packet) size greater than 134 bytes (probable data on the Syn packet), and an IP version that is NOT 4:

```
tcpdump -nn -i eth0 'tcp[13] & 0x02 = 2 and ip[2:2] > 0x86 and ip[0] & 0xF0 != 4'
```

Summary

Berkeley Packet Filters are a powerful tool for the intrusion detection analysis. Using them will allow the analyst to quickly drill down to the specific packets he/she needs to see and reduce large packet captures down to the essentials. Even a basic knowledge of how to use them will save hours of time during the investigation of packets, or give insight into malicious traffic that wasn't detected using other methods.

References

1. http://en.wikipedia.org/wiki/Berkeley_Packet_Filter
2. <http://www.gsp.com/cgi-bin/man.cgi?section=4&topic=bpf>
3. <http://en.wikipedia.org/wiki/IPv4>