# Tcpdump advanced filters

October 1, 2007

## Introduction

In this article, I will explain how to use tcpdump to:

- know if IP options are set
- find DF packets (packets which we don't want to be fragmented)
- find fragmented packets
- find datagrams with low TTL
- find particular TCP flag combinations
- find datagrams with particular data (here, packets with command MAIL from the SMTP protocol and GET command from HTTP)

## Notes

I usually type `tcpdump -n -i eth1 -s 1600` before my filter but I won't do that throughout the article.

- `-n` prevents DNS lookups.
- `-i` specifies the interface.
- `-s` specifies the size of the packets (default is 65536 bytes).

Be careful if you use `-s 0` because depending on the version of tcpdump, you might be capturing 64K or full-lenght packets.

All commands are typed as root.

Feel free to contact me for comments, suggestions or reporting mistakes. Let me know if something is not clear!

I'll try to keep this document updated with new useful rules.

## Let's learn the syntax

Before I begin with advanced filters, let's review the basic syntax of tcpdump.

**Basic syntax**

*Filtering hosts*

Match any traffic involving 192.168.1.1 as destination or source:

```
tcpdump host 192.168.1.1
```

As source only:

```
tcpdump src host 192.168.1.1
```

As destination only:

```
tcpdump dst host 192.168.1.1
```

*Ports filtering*

Match any traffic involving port 25 as source or destination:

```
tcpdump port 25
```

As source only:

```
tcpdump src port 25
```

As destination only:

```
tcpdump dst port 25
```

*Network filtering*

```
tcpdump net 192.168
tcpdump src net 192.168
tcpdump dst net 192.168
```

*Protocol filtering*

```
tcpdump arp
tcpdump ip
tcpdump tcp
tcpdump udp
tcpdump icmp
```

**Let’s combine expressions**

Remember this:

| Type | Expression | Also possible |
|---|---|---|
| Negation | `!` | `not` |
| Concatenate | `&&` | `and` |
| Alternate | `||` | `or` |

For example the following rule will match any TCP traffic on port 80 (web) with 192.168.1.254 or 192.168.1.200 as destination host:

```
tcpdump '((tcp) and (port 80) and ((dst host 192.168.1.254) or (dst host 192.168.1.200)))'
```

This one will match any ICMP traffic involving the destination with physical/MAC address 00:01:02:03:04:05:

```
tcpdump '((icmp) and ((ether dst host 00:01:02:03:04:05)))'
```

This will match any traffic for the destination network 192.168 except destination host 192.168.1.200:

```
tcpdump '((tcp) and ((dst net 192.168) and (not dst host 192.168.1.200)))'
```

**Advanced header filtering**

Before we continue, we need to know how to filter out info from headers:

| Expression | Explanation |
|---|---|
| `proto[x:y]` | will start filtering from byte x for y bytes. `ip[2:2]` would filter bytes 3 and 4 (first byte begins by 0) |
| `proto[x:y] & z = 0` | will match bits set to 0 when applying mask `z` to `proto[x:y]` |
| `proto[x:y] & z !=0` | some bits are set when applying mask `z` to `proto[x:y]` |
| `proto[x:y] & z = z` | every bits are set to z when applying mask `z` to `proto[x:y]` |
| `proto[x:y] = z` | `p[x:y]` has exactly the bits set to `z` |

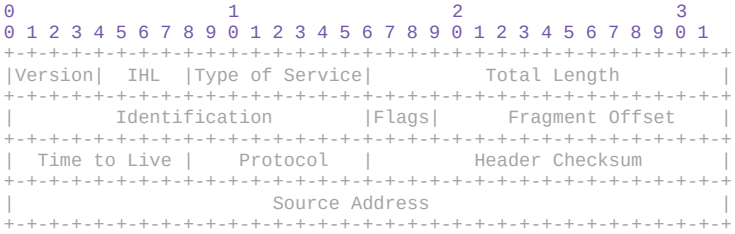
Operators:

| Operator | Meaning |
|---|---|
| > | greater |
| < | lower |
| >= | greater or equal |
| <= | lower or equal |
| = | equal |
| != | different |

This may not be clear in the first place but you’ll find examples below involving these expressions.

Of course, it is important to know what the protocol headers look like before diving into more advanced filters.

IP header:



```
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                   |    Padding    | <-- optional
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             DATA ...                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

I'll consider we are only working with the IPv4 protocol suite for these examples.

In an ideal world, every field would fit inside one byte. This is not the case, of course.

# Exercises

## Exercise: Are IP options set?

Let's say we want to know if the IP header has options set. We can't just try to filter out the 21st byte because if no options are set, data start at the 21st byte. We know a "normal" header is usually 20 bytes (160 bits) long. With options set, the header is longer than that. The IP header has the header length field which we will filter here to know if the header is longer than 20 bytes.

```
+-+-+-+-+-+-+-+-+
|Version|  IHL  |
+-+-+-+-+-+-+-+-+
```

Usually the first byte has a value of 01000101 in binary.

Anyhow, we need to divide the first byte in half…

```
0100 0101

0100 = 4 in decimal. This is the IP version.
0101 = 5 in decimal. This is the number of blocks of 32 bits in the headers. 5 x 32 bits = 160 bits or 20 bytes.
```

The second half of the first byte would be bigger than 5 if the header had IP options set.

We have two ways of dealing with that kind of filters.

- Either try to match a value bigger than 01000101. This would trigger matches for IPv4 traffic with IP options set, but ALSO any IPv6 traffic !

In decimal `01000101` equals `69`.

Let's recap how to calculate in decimal.

```
0 : 0       \
1 : 2^6 = 64  \ First field (IP version)
0 : 0         /
0 : 0       /
-
0 : 0       \
1 : 2^2 = 4   \ Second field (Header length)
0 : 0         /
1 : 2^0 = 1  /
```

64 + 4 + 1 = 69

The first field in the IP header would usually have a decimal value of 69. If we had IP options set, we would probably have 01000110 (IPv4 = 4 + header = 6), which in decimal equals 70.

This rule should do the job:

```
tcpdump 'ip[0] > 69'
```

Somehow, the proper way is to mask the first half/field of the first byte, because as mentioned earlier, this filter would match any IPv6 traffic.

- The proper/right way : "masking" the first half of the byte

`0100 0101` will become `0000 0101` thanks to a mask of `0000 1111`

```
0100 0101 : 1st byte originally
0000 1111 : mask (0xf in hex or 15 in decimal). 0 will mask the values while 1 will keep the values intact.
---------
0000 0101 : final result
```

You should see the mask as a "power switch". 1 means on/enabled, 0 means off/disabled.

The correct filter:

In decimal:

```
tcpdump 'ip[0] & 15 > 5'
```

or

In hexadecimal:

```
tcpdump 'ip[0] & 0xf > 5'
```

I use hex masks.

Let’s recap.. That’s rather simple, if you want to:

- keep the last 4 bits intact, use `0xf` (binary 00001111)
- keep the first 4 bits intact, use `0xf0` (binary 11110000)

## Exercise: Is DF bit (don’t fragment) set?

Let’s now try to know if we fragmentation is occuring. Fragmentation is not desirable. Fragmentation occurs when a the MTU of the sender is bigger than the path MTU on the path to destination.

Fragmentation info can be found in the 7th and 8th byte of the IP header.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Bit 0:     reserved, must be zero
Bit 1:     (DF) 0 = May Fragment, 1 = Don't Fragment.
Bit 2:     (MF) 0 = Last Fragment, 1 = More Fragments.
```

The fragment offset field is only used when fragmentation occurs.

If we want to match the DF bit (don’t fragment bit, to avoid IP fragmentation):

The 7th byte would have a value of `01000000` or `64` in decimal

```
tcpdump 'ip[6] = 64'
```

Exercise: Matching fragmentation

Matching MF (more fragment set)? This would match the fragmented datagrams but wouldn’t match the last fragment (which has the 2nd bit set to 0):

```
tcpdump 'ip[6] = 32'
```

The last fragment have the first 3 bits set to 0… but has data in the fragment offset field.

Matching the fragments and the last fragments:

```
tcpdump '((ip[6:2] > 0) and (not ip[6] = 64))'
```

A bit of explanation:

`ip[6:2] > 0` would return anything with a value of at least 1.

We don’t want datagrams with the DF bit set though.. the reason of the `not ip[6] = 64`

If you want to test fragmentation use something like:

```
ping -M want -s 3000 192.168.1.1
```

## Exercise: Matching datagrams with low TTL

The TTL field is located in the 9th byte and fits perfectly inside 1 byte.

The maximum decimal value of the TTL field is thus `255` (`11111111` in binary).

This can be verified, we’re going to try to specify a TTL of 256:

```
ping -M want -s 3000 -t 256 192.168.1.200
```

Indeed ping tells use 256 is out of range:

```
ping: ttl 256 out of range
```

The TTL field:

```
+-+-+-+-+-+-+-+-+
|  Time to Live |
+-+-+-+-+-+-+-+-+
```

We can try to find if someone on our network is using trace route by using something like this on the gateway:

```
tcpdump 'ip[8] < 5'
```

### Exercise: Matching packets longer than X bytes

Where X is 600 bytes:

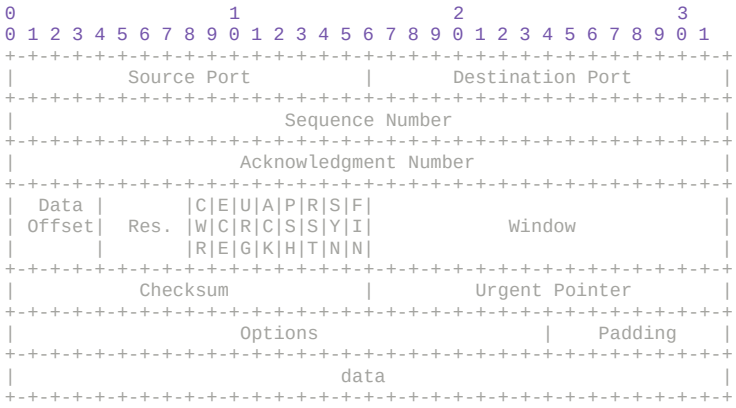
```
tcpdump 'ip[2:2] > 600'
```

## More IP filtering

We could imagine filtering source and destination addresses directly in decimal addressing. We could also match the protocol by filtering the 10th byte.

It would be pointless anyhow, because tcpdump makes it already easy to filter out that kind of info.

TCP header:




### Matching any TCP traffic with a source port > 1024:

```
tcpdump 'tcp[0:2] > 1024'
```

or

```
tcpdump 'tcp src portrange 1025-65535'
```

### Matching TCP traffic with particular flag combinations:

The flags are defined in the 14th byte of the TCP header.


In the TCP 3-way handshakes, the exchange between hosts goes like this:

1. Source sends SYN
2. Destination answers with SYN, ACK
3. Source sends ACK

If we want to match packets with only the SYN flag set, the 14th byte would have a binary value of `00000010` which equals 2 in decimal:

```
tcpdump 'tcp[13] = 2'
```

### Matching SYN, ACK

`00010010` or `18` in decimal:

```
tcpdump 'tcp[13] = 18'
```

### Matching either SYN only or SYN-ACK datagrams

```
tcpdump 'tcp[13] & 2 = 2'
```

We used a mask here. It will returns anything with the ACK bit set (thus the SYN-ACK combination as well)

Let’s assume the following examples (SYN-ACK)

```
00010010 : SYN-ACK packet
00000010 : mask (2 in decimal)
--------
00000010 : result (2 in decimal)
```

Every bits of the mask match!

### Matching PSH-ACK packets

```
tcpdump 'tcp[13] = 24'
```

### Matching any combination containing FIN

FIN usually always comes with an ACK so we either need to use a mask or match the combination ACK-FIN.

```
tcpdump 'tcp[13] & 1 = 1'
```

### Matching RST flag

```
tcpdump 'tcp[13] & 4 = 4'
```

### Matching with tcpflags

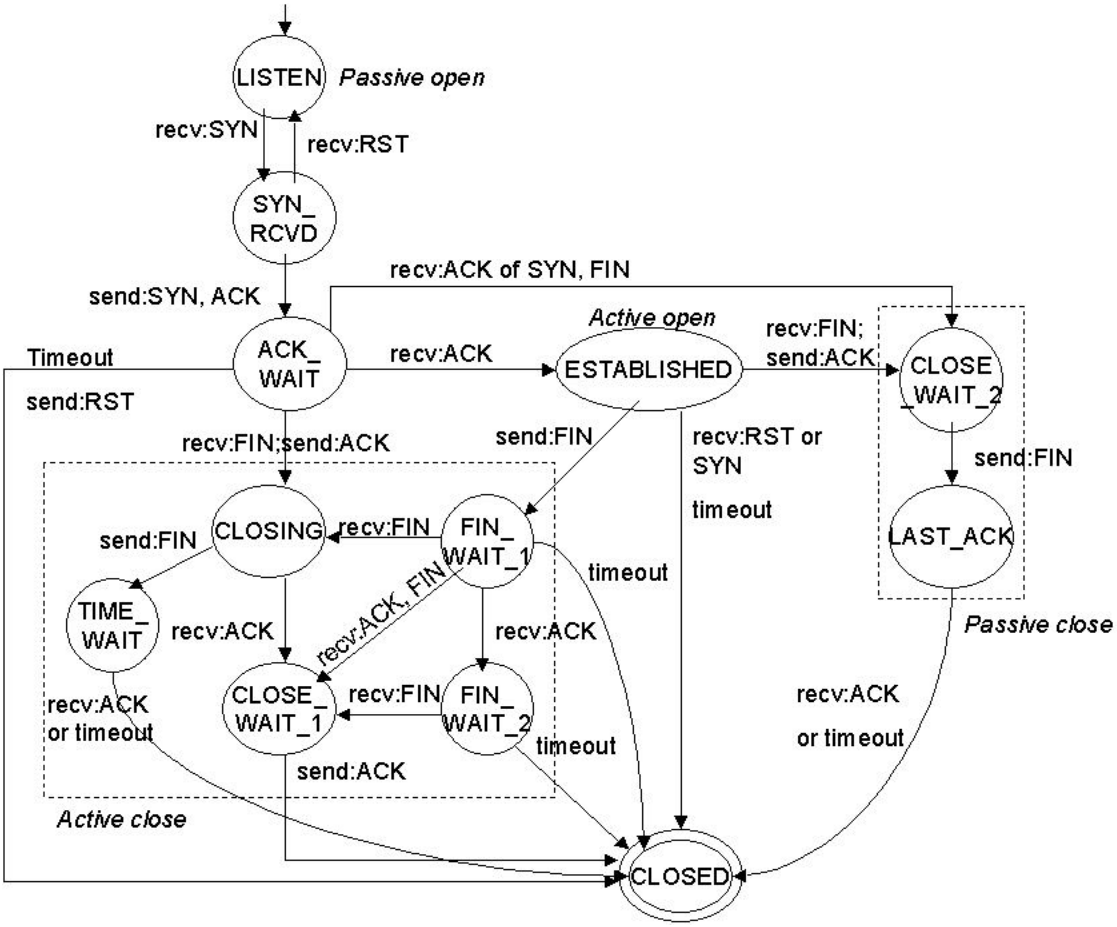
Actually, there's an easier way to filter flags (`man pcap-filter` and look for tcpflags):

```
tcpdump 'tcp[tcpflags] == tcp-ack'
```

Matching all packages with TCP-SYN or TCP-FIN set:

```
tcpdump 'tcp[tcpflags] & (tcp-syn|tcp-fin) != 0'
```

By looking at the TCP state machine we can find the different flag combinations we may want to analyze.




Ideally, a socket in ACK_WAIT mode should not have to send a RST. It means the 3 way handshake has not completed. We may want to analyse that kind of traffic.

### Exercise: Matching SMTP data

I will make a filter that will match any packet containing the "MAIL" command from an SMTP exchange.

You can use something like http://www.easycalculation.com/ascii-hex.php to convert values from ASCII to hexadecimal or you can use Python.

```
$ python -c 'print "MAIL".encode("hex")'
4d41494c
```

So “MAIL” in hex is `0x4d41494c`

The rule would be:

```
tcpdump '((port 25) and (tcp[20:4] = 0x4d41494c))'
```

It will check the bytes 21 to 24. “MAIL” is 4 bytes/32 bits long..

This rule would not match packets with IP options set.

This is an example of packet (a spam, of course):

```
Capturing on eth0
Frame 1 (92 bytes on wire, 92 bytes captured)
    Arrival Time: Sep 25, 2007 00:06:10.875424000
    [Time delta from previous packet: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Packet Length: 92 bytes
    Capture Length: 92 bytes
    [Frame is marked: False]
    [Protocols in frame: eth:ip:tcp:smtp]
Ethernet II, Src: Cisco_X (00:11:5c:X), Dst: 3Com_X (00:04:75:X)
    Destination: 3Com_X (00:04:75:X)
        Address: 3Com_X (00:04:75:X)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
    Source: Cisco_X (00:11:5c:X)
        Address: Cisco_X (00:11:5c:X)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
    Type: IP (0x0800)
Internet Protocol, Src: 62.163.X (62.163.X), Dst: 192.168.X (192.168.X)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 78
    Identification: 0x4078 (16504)
    Flags: 0x04 (Don't Fragment)
        0... = Reserved bit: Not set
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 118
    Protocol: TCP (0x06)
    Header checksum: 0x08cb [correct]
        [Good: True]
        [Bad : False]
    Source: 62.163.X (62.163.X)
    Destination: 192.168.X (192.168.XX)
Transmission Control Protocol, Src Port: 4760 (4760), Dst Port: smtp (25), Seq: 0, Ack: 0, Len: 38
    Source port: 4760 (4760)
    Destination port: smtp (25)
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 38    (relative sequence number)]
    Acknowledgement number: 0    (relative ack number)
    Header length: 20 bytes
    Flags: 0x18 (PSH, ACK)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...1 .... = Acknowledgment: Set
        .... 1... = Push: Set
        .... .0.. = Reset: Not set
        .... ..0. = Syn: Not set
        .... ...0 = Fin: Not set
    Window size: 17375
    Checksum: 0x6320 [correct]
        [Good Checksum: True]
        [Bad Checksum: False]
Simple Mail Transfer Protocol
    Command: MAIL FROM:<wguthrie_at_mysickworld--dot--com>\r\n
        Command: MAIL
        Request parameter: FROM:<wguthrie_at_mysickworld--dot--com>
```

## Matching HTTP data

Let’s make a filter that will find any packets containing GET requests.

The HTTP request will begin with:

```
GET / HTTP/1.1\r\n
```

For a total of 16 bytes counting the carriage return but not the backslashes!

If no IP options are set.. the GET command will use bytes 20, 21 and 22. Usually, options takes 12 bytes (12th byte indicates the header length, which should report 32 bytes). So we should match bytes 32, 33 and 34 (1st byte = byte 0).

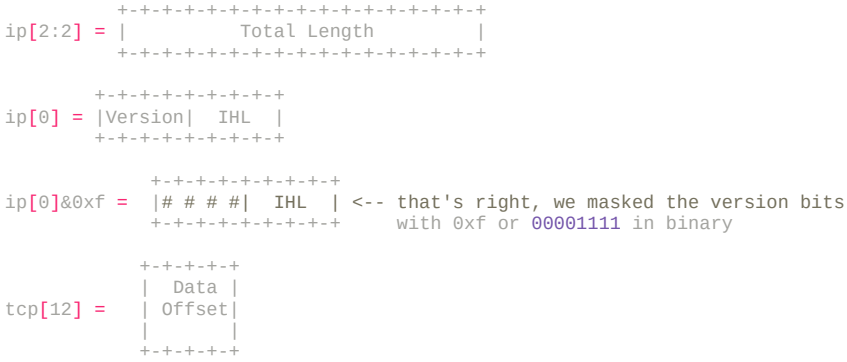
Tcpdump can only match data size of either 1, 2 or 4 bytes, we will take the following ASCII character following the GET command (a space).

“GET " in hex is `47455420`

```
tcpdump 'tcp[32:4] = 0x47455420'
```

Matching HTTP data (exemple taken from tcpdump man page):

```
tcpdump 'tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)'
```




So what we are doing here is “(IP total length - IP header length - TCP header length) != 0”

We are matching any packet that contains data.

We are taking the IHL (total IP length)

## Exercise: Matching other interesting TCP things

### SSH connection (on any port)

We will be looking for the reply given by the SSH server.

OpenSSH usually replies with something like “SSH-2.0-OpenSSH_3.6.1p2”. The first 4 bytes (SSH-) have an hex value of `0x5353482D`.

```
tcpdump 'tcp[(tcp[12]>>2):4] = 0x5353482D'
```

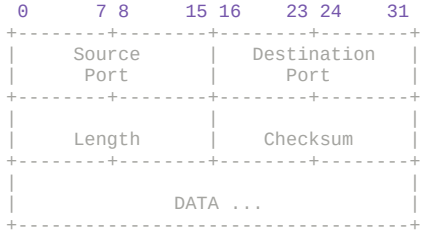
If we want to find any connection made to older version of OpenSSH (version 1, which are insecure and subject to MITM attacks).

The reply from the server would be something like “SSH-1.99..”

```
tcpdump '(tcp[(tcp[12]>>2):4] = 0x5353482D) and (tcp[((tcp[12]>>2)+4):2] = 0x312E)'
```

Explanation of `>>2` can be found below in the references section.

## UDP header



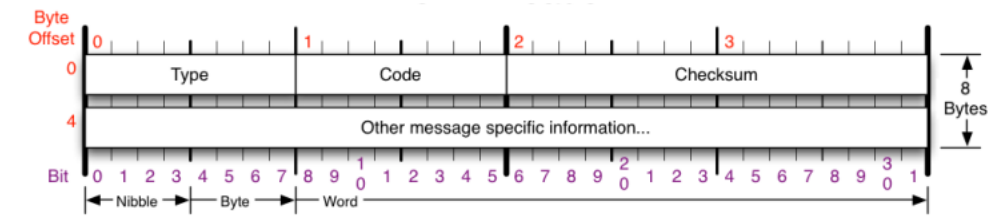

Nothing really interesting here.

If we want to filter ports we would use something like:

```
tcpdump udp dst port 53
```

### ICMP header

See different ICMP messages:




**ICMP Message Types**

| Type | Code/Name |
|---|---|
| 0 | Echo Reply |
| 3 | Destination Unreachable |
| | 0 Net Unreachable |
| | 1 Host Unreachable |
| | 2 Protocol Unreachable |
| | 3 Port Unreachable |
| | 4 Fragmentation required, and DF set |
| | 5 Source Route Failed |
| | 6 Destination Network Unknown |
| | 7 Destination Host Unknown |
| | 8 Source Host Isolated |
| | 9 Network Administratively Prohibited |
| | 10 Host Administratively Prohibited |
| | 11 Network Unreachable for TOS |
| 3 | Destination Unreachable (continued) |
| | 12 Host Unreachable for TOS |
| | 13 Communication Administratively Prohibited |
| 4 | Source Quench |
| 5 | Redirect |
| | 0 Redirect Datagram for the Network |
| | 1 Redirect Datagram for the Host |
| | 2 Redirect Datagram for the TOS & Network |
| | 3 Redirect Datagram for the TOS & Host |
| 8 | Echo |
| 9 | Router Advertisement |
| 10 | Router Selection |
| 11 | Time Exceeded |
| | 0 TTL Exceeded |
| | 1 Fragment Reassembly Time Exceeded |
| 12 | Parameter Problem |
| | 0 Pointer Problem |
| | 1 Missing a Required Operand |
| | 2 Bad Length |
| 13 | Timestamp |
| 14 | Timestamp Reply |
| 15 | Information Request |
| 16 | Information Reply |
| 17 | Address Mask Request |
| 18 | Address Mask Reply |
| 30 | Traceroute |

**Checksum**

Checksum of ICMP header

**RFC 792**

Please refer to RFC 792 for the Internet Control Message protocol (ICMP) specification.

We will usually filter the type (1 byte) and code (1 byte) of the ICMP messages.

Here are common ICMP types:

```
  0    Echo Reply                [RFC792]
  3    Destination Unreachable   [RFC792]
  4    Source Quench             [RFC792]
  5    Redirect                  [RFC792]
  8    Echo                      [RFC792]
 11    Time Exceeded             [RFC792]
```

We may want to filter ICMP messages type 4, these kind of messages are sent in case of congestion of the network.

```
tcpdump 'icmp[0] = 4'
```

If we want to find the ICMP echo replies only, having an ID of 500. By looking at the image with all the ICMP packet description we see the ICMP echo reply have the ID spread across the 5th and 6th byte. For some reason, we have to filter out with the value in hex.

```
tcpdump -i eth0 '(icmp[0] = 0) and (icmp[4:2] = 0x1f4)'
```

## Acknowledgments


- Contribution and updates: Yousong Zhou (China)
  - tcpflags
  - Python hex conversion
  - portrange
  - typo corrections
- Publication/reference: Keith Makan (South Africa)
  - “Penetration Testing with the Bash shell” published at Packt Publishing, available here.
- Documentation/reference: Tcpdump project
  - https://www.tcpdump.org/
- Documentation/reference: pfSense project
  - https://docs.netgate.com/pfsense/en/latest/diagnostics/packetcapture/references.html
- Reference in blog post: Daniel Miessler (California)
  - https://danielmiessler.com/study/tcpdump/
- Reference in Thesis: Andrés Geovanny Muñoz Ponguillo (Ecuador)
  - http://repositorio.ug.edu.ec/bitstream/redug/16465/1/B_CISC_PTG.1167.Mu%C3%B1oz%20Ponguillo%20Andr%C


## References

- tcpdump man page : http://www.tcpdump.org/tcpdump_man.html
- Conversions: http://easycalculation.com/hex-converter.php
- Filtering HTTP requests: http://www.wireshark.org/tools/string-cf.html
- Filtering data regardless of TCP options: http://www.wireshark.org/lists/wireshark-users/201003/msg00024.html

Just in case the post in the last link ever disappears, here's a copy of the content:

```
From: Sake Blok <sake@xxxxxxxxxx>
Date: Wed, 3 Mar 2010 22:42:29 +0100
Subject: Wireshark-users: Re: [Wireshark-users] Hex Offset Needed
Or if your capturing device is capable of interpreting tcpdump style filters (or more accurately, BPF style filters), you could use

tcp[((((tcp[12:1] & 0xf0) >> 2) + 8):2] = 0x2030

Which in English would be:
- take the upper 4 bits of the 12th octet in the tcp header ( tcp[12:1] & 0xf0 )
- multiply it by four ( (tcp[12:1] & 0xf0)>>2 ) which should give the tcp header length
- add 8 ( ((tcp[12:1] & 0xf0) >> 2) + 8 ) gives the offset into the tcp header of the space before the first octet of the response (
- now take two octets from the tcp stream, starting at that offset ( tcp[((((tcp[12:1] & 0xf0) >> 2) + 8):2]  )
- and verify that they are " 0" ( = 0x2030 )

Of course this can give you false positives, so you might want to add a test for "HTTP" and the start of the tcp payload with:

tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x48545450

resulting in the filter:

tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x48545450 and tcp[((((tcp[12:1] & 0xf0) >> 2) + 8):2] = 0x2030

A bit cryptic, but it works, even when TCP options are present (which would mess up a fixed offset into the tcp data).

Cheers,
Sake
```