NETWORK MANIPULATION IN A HEX FASHION: An introduction to HexInject

```
A1 EA CC 1B F5 36 32 BC 40 D0 81 1E C6 87 DC CD D1 62 6F 4D 29 FD 44 44
22 35 27 D5 28 54 81 A7 4F 92 65 23 79 E7 22 3F BC AD BA BB AD E9 6E 74
8C 78 8A             2D          18 FE 9F 0A 5C FF 3F 72 31 60 CD 1D 74
73 DA A8 AE       33 FE 8E    81 00 0A 04 14 47 B8 94 26 87 8B 83 4E 2F
37 3A EB E9       52 DF A5    8E A2 8B 99 50 C8 D6 67 43 3E 3B BF FF D9
FF EE 70 BB       52 B5 46    19          24    9B 97 40    04 4D 06 EE
0B D8 30 1C       33 D2 D7    3C    B7 2F 66 CB    0C    0F 8D FF FB 78
64 D0 00 04       63 69 E9    CE       EC 2D 08 03    91 B1 D7 78 67 A0
73 F8 CF 92                   3A    0E 22 E2 4B    3A    8F D3 D2 92 65
CF A3 89 C4       24 ED 5F    A2          6C    3D 50 8B    C0 D9 E2 93
2F 0C 93 69       51 1F FF    32 38 70 B9 21 38 90 D3 47 D5 A3 D4 8B 17
0F 32 88 21       CB F5 FD    D6 E2 B5 90 8A 40 5B D7 DC E5 78 02 49 97
95 FE 9A 6D       0F 18 FF    2E 0F          CC 8D 22 16 18          FB
C2 2C 0C 28       82 8E C2    52 20 E4    EE A5 3B 7E CA F4 44    AC 02
E4 7F ED FE       FC EE EE    66 1D 37    11 A4    3D 56    74    C4 A4
04 A3 BB             2C          8A BA    87 24    AA C5    AF    6A AF
BB 37 87 ED 71 33 D0 B5 4A F1 D5 76 EB    EA 3C       F1    F4    30 8D
53 75 73 EC C7 3B 91 79 16 8C B9 E0 F1    CC E3    5A       3E    A1 D1
C3 9C 7D A6 0A 46 A8 54 D2 23 83 69 5A    21 8C    10 85    BE    9C 45
EA AD A0 23 21 9C D0 C5 36 83 37 C9 D9    66 7F    2A 08    DC    98 17
85 3F FA 78 60 AB 00 81 05 19 4C 22          7F F3 B8 88 09 14    02 36
00 00 3B BD 8E BD 0C AE 8E 1E 8B 7E 20 30 10 94 7A 8E 88 FD 5F    FF FF
FF FE DA 2E F1 AC 17 BB A7 12 93 DF 64 A9 3B 3B 0B 3D          37 56 00
82 CA 4A B4 52 6A 5E 92 16 A7 95 7D B6 5D C9 A1 C1 D9 EA FB 05 8F FC 16
C5 90 45 8D A7 49 AD 81 B3 B2 0A A6 63 8D 77 92 EA 98 D1 D4 25 FF FB 78
```

2010 - Emanuele Acri

<crossbower@backtrack-linux.org>

Special thank to:

BackTrack Linux Staff

<http://www.backtrack-linux.org/>

Index

Introduction

HexInject (http://hexinject.sourceforge.net/) is a very versatile packet injector and sniffer, that provide a command-line framework for raw network access.

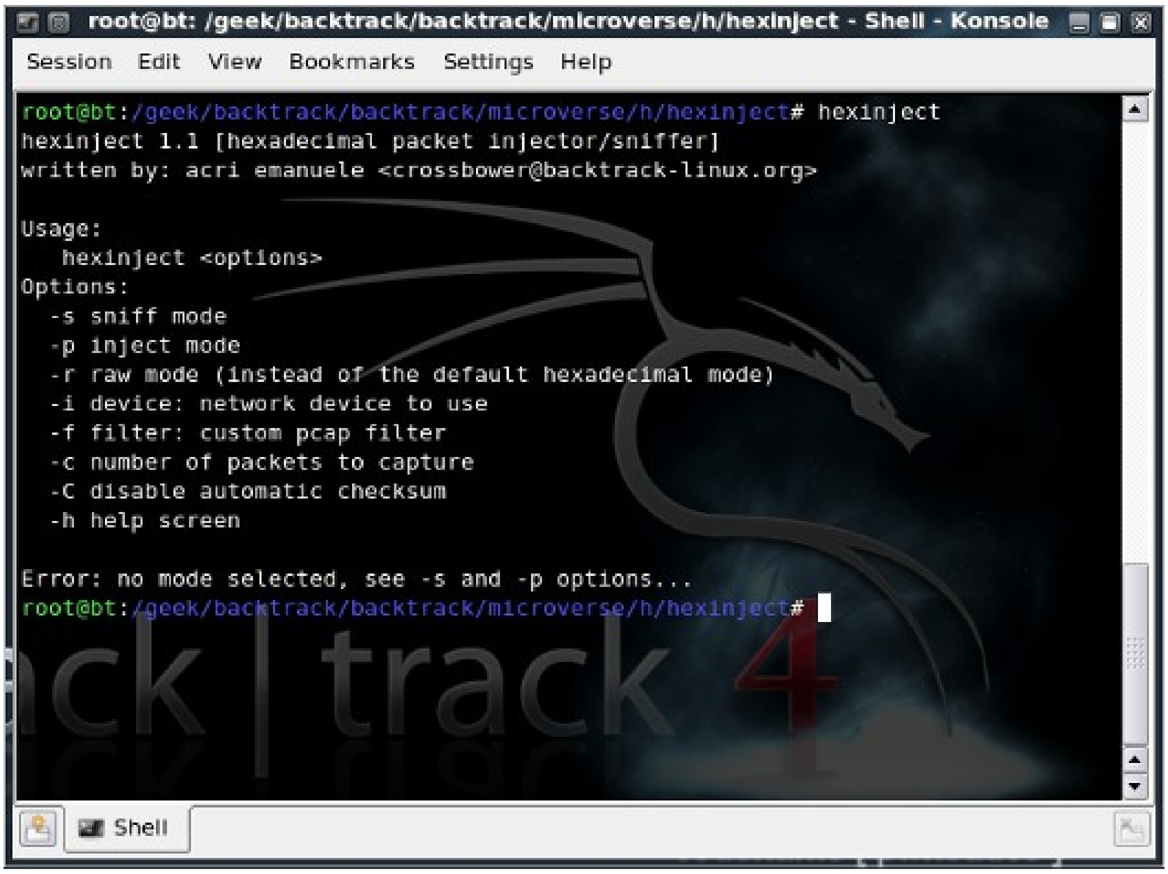
It's designed to work together with others command-line utilities, the utilities you usually use for processing text output from executables or scripts (replace, sed, awk).

This “compatibility” facilitates the creation of powerful shell scripts, in a very little time, capable of reading, intercepting and modifying network traffic.

It's like using libpcap (http://www.tcpdump.org/), from the command line, without messing with the API. From a (lazy) programmer perspective that's fantastic!

Usage

This is a screenshot of the current usage of the tool, the options should be self-explanating:


root@bt: /geek/backtrack/backtrack/microverse/h/hexinject - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
root@bt:/geek/backtrack/backtrack/microverse/h/hexinject# hexinject
hexinject 1.1 [hexadecimal packet injector/sniffer]
written by: acri emanuele <crossbower@backtrack-linux.org>

Usage:
   hexinject <options>
Options:
  -s sniff mode
  -p inject mode
  -r raw mode (instead of the default hexadecimal mode)
  -i device: network device to use
  -f filter: custom pcap filter
  -c number of packets to capture
  -C disable automatic checksum
  -h help screen

Error: no mode selected, see -s and -p options...
root@bt:/geek/backtrack/backtrack/microverse/h/hexinject# 
```

Basically it has two execution modalities: **sniff** and **inject**, and two data format: **hexadecimal** and **raw** (the first data format is the default, the second is unparsed network traffic).

You can provide a custom pcap filters to select traffic to capture, very useful for advanced uses.

And finally, because HexInject is capable to set the correct checksum for the packets injected, there's a flag to disable this feature.

Why hexadecimal?

The tools is written to read and inject data in hexadecimal, why?

We'll not dwell on the advantages of hexadecimal to represent the binary format.
The reasons are just two: because a fixed two character string can represent all the possible values of a byte (but you already know this...) and because the hexadecimal allow to follow the principles of *Data-Driven Programming*.

“When doing data-driven programming, one clearly distinguishes code from the data structures on which it acts, and designs both so that one can make changes to the logic of the program by editing not the code but the data structure.” (http://www.faqs.org/docs/artu/ch09s01.html).

This is very important to make clear and maintainable programs or scripts. However, not all the libraries that provide raw network access follow this principle.
For example, libnet uses functions to hide data, as we can see from this snippet of code (from netdiscover, http://www.nixgeneration.com/~jaime/netdiscover/):

```
/* Forge Arp Packet, using libnet */
void forge_arp(char *source_ip, char *dest_ip, char *disp)
{
      static libnet_ptag_t arp=0, eth=0;
      static u_char dmac[ETH_ALEN] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
      static u_char sip[IP_ALEN];
      static u_char dip[IP_ALEN];
      u_int32_t otherip, myip;

      /* get src & dst ip address */
      otherip = libnet_name2addr4(libnet, dest_ip, LIBNET_RESOLVE);
   memcpy(dip, (char*)&otherip, IP_ALEN);
      myip = libnet_name2addr4(libnet, source_ip, LIBNET_RESOLVE);
   memcpy(sip, (char*)&myip, IP_ALEN);

      /* forge arp data */
      arp = libnet_build_arp(
      ARPHRD_ETHER,
      ETHERTYPE_IP,
      ETH_ALEN, IP_ALEN,
      ARPOP_REQUEST,
      smac, sip,
      dmac, dip,
      NULL, 0,
      libnet,
      arp);

      /* forge ethernet header */
   eth = libnet_build_ethernet(
      dmac, smac,
      ETHERTYPE_ARP,
      NULL, 0,
      libnet,
      eth);

      /* Inject the packet */
   libnet_write(libnet);
}
```

Libnet_build_arp() and libnet_build_ethernet() are complex functions, that requires a lot of variables and pointers (many of them libnet-specific). Certainly their use is not intuitive.

The result is, in my opinion, confusing and ugly. But, of course, the function can be rewritten in a different style:

```
/* Forge Arp Packet, using libpcap */
void forge_arp(char *source_ip, char *dest_ip, char *disp)
{
      in_addr_t sip, dip;

      char raw_arp[] =
            "\xff\xff\xff\xff\xff\xff" // mac destination
            "\x00\x00\x00\x00\x00\x00" // mac source
            "\x08\x06"                 // type
            "\x00\x01"                 // hw type
            "\x08\x00"                 // protocol type
            "\x06"                     // hw size
            "\x04"                     // protocol size
            "\x00\x01"                 // opcode
            "\x00\x00\x00\x00\x00\x00" // sender mac
            "\x00\x00\x00\x00"         // sender ip
            "\xff\xff\xff\xff\xff\xff" // target mac
            "\x00\x00\x00\x00";        // target ip

      /* get src & dst ip address */
      dip = inet_addr(dest_ip);
      sip = inet_addr(source_ip);

      memcpy(raw_arp + 28, (char*) &sip, IP_ALEN);
      memcpy(raw_arp + 38, (char*) &dip, IP_ALEN);

      /* set mac addr */
      memcpy(raw_arp + 6,  smac, ETH_ALEN);
      memcpy(raw_arp + 22, smac, ETH_ALEN);

      /* Inject the packet */
      pcap_sendpacket(inject, (unsigned char *) raw_arp, sizeof(raw_arp)-1);
}
```

The second version of forge_arp() uses a *data-driven* approach: less variables, a clear representation of the ARP packet, standard functions and standard data-types. The packet can be modified in every aspect without altering the code.

This approach is somewhat similar to (well-written) exploits, where the assembly shellcode (hexadecimal, of course) has every opcode commented, and it's easy to adapt to the target.

HexInject is similar to this second version of forge_arp(), only much simpler.

Note: you can download a patch to eliminate the libnet dependency from the last release of netdiscover, from my site: http://backtrack.it/~crossbower/netdiscover0.3-beta7-no-libnet.patch. Useful for recent systems that do not support old versions of libnet...

Basic usage

"I believe without exception that theory follows practice.
Whenever there is a conflict between theory and practice, theory is wrong."
David Baker

This practical section of the document show various uses of HexInject, using a lot of examples.

The operating environment is **BackTrack 4 R1** (downloadable from here: http://www.backtrack-linux.org/downloads/), virtualized with VirtualBox.

It's assumed that the system has two network interfaces (**eth0**, **eth1**), if these differ from your, you must adapt the examples to your system (not difficult).

HexInject as Sniffer

As seen before, HexInject can be used as sniffer when the options "-s" is provided. It can print network traffic in both hexadecimal and raw format.

A first test of the functionality can be:

```
root@backtrack-base# hexinject -s -i eth0
1C AF F7 6B 0E 4D AA 00 04 00 0A 04 08 00 45 00 00 3C 9A 88 40 00 40 06 51 04 C0
A8 01 09 5B 05 32 79 C9 45 01 BB 61 5E 85 79 00 00 00 00 A0 02 16 D0 0D 2F 00 00
02 04 05 B4 04 02 08 0A 00 0D 22 EC 00 00 00 00 01 03 03 07 FF FF FF FF FF FF AA
00 04 00 0A 04 08 06 00 01 08 00 06 04 00 01 AA 00 04 00 0A 04 C0 A8 01 09 00 00
00 00 00 00 C0 A8 01 04
AB 00 00 03 00 00 AA 00 04 00 0A 04 60 03 22 00 0D 02 00 00 AA 00 04 00 0A 04 03
DA 05 00 00 00 00 00 00 00 00 00 AA 00 04 00 00 00 0A 00 00 02 AA AA FF FF FF FF
FF FF AA 00 04 00 0A 04 08 06 00 01 08 00 06 04 00 01 AA 00 04 00 0A 04 C0 A8 01
09 00 00 00 00 00 00 C0 A8 01 04
```

The default data format is hexadecimal: bytes are separated by a single space, and packets are separated by the "newline" character. This format is very easy to parse by standard unix cmd-line utilities (like sed, awk, tr...) and by scripting language interpreters (perl, python, tcl... or even bash).

Instead, if raw data format is specified, the output will be similar to this:

```
root@backtrack-base# hexinject -s -i eth0 -r
�#
##��kEk��@3#5E[y#H��#   Ė.Y#�r��#C�##$O�##
(��9  1D50%�z#9#�.
H���.�
�Ëm�; (��ߊ�Cd�Y�#��kM�#
E4�j@@#X���#      [y#H�.Ĵ�#CY#���##���##
      7{(��9^C
```

Network packets are are mainly composed of non printable characters. For this reason, if we want to extract useful information from network streams we need the help of some utilities.

Very useful is the tool **strings**, that extracts and prints printable character sequences delimited by unprintable characters. Mainly developed for determining the contents of non-text files, it works very well for our purpose.

Let's see:

```
root@backtrack-base# hexinject -s -i eth0 -r | strings
 w220 Ftp firmware update utility
USER test
      331 Password please.
PASS test
Z421 Login incorrect.
[421 Login incorrect.
QlQp
^C
```

Interesting... we just intercepted an FTP connection to an embedded device (in this case a DLink Router).

FTP commands are plain text so it's easy to extract the username and the password from the stream:

```
USER test
PASS test
```

Textual protocols

We can of course do something a little more advanced... For example we can extract and print some HTTP headers.

HTTP is a textual protocol used to retrieve web pages from webservers (but not only). An HTTP request or response is composed by the message headers and the message body.

The headers are easy to parse because they are separated by the character sequence “\r\n” (0x0D 0x0A), so, from a “raw format” perspective, one header per line.

Let's try to extract the *host* header to see what websites are being visited on our LAN:

```
root@backtrack-base# hexinject -s -i eth0 -r | strings | grep 'Host:'
Host: youtube.com
Host: www.youtube.com
Host: s.ytimg.com
...
```

Even in this example we used only common utilities (**strings** and **grep**), creating a “specialized” sniffer without writing a line of code.

“Ok”, You can think, “this is easy if the protocol is textual, but what about binary protocols?”

Don't worry, with a bit of black sorcery, we can do that and more...

Mixed protocols

We just introduced HTTP, but there's another protocol without which the user experience of the web would not be the same: Domain Name System.

The DNS protocol, translates domain names, meaningful to humans, into binary identifiers (IP adresses). So, if a user types “www.backtrack-linux.org” into his browser location bar, he will be properly addressed to 67.23.70.62, the IP address of the webserver.

DNS is a binary protocol, but contains sequences of printable characters and it's widely used, for this reason it's a good example of “mixed protocol” (binary data + printable sequences). We will now write a request sniffer for it.

The strings we want to extract (the domain names requested), are not transmitted entirely in a printable format: the domain levels are separated by one byte containing the number of characters of the next string.

Just to visualize the sequence, a resolution request for "www.google.com" will appear as:

HEADER	**...**	**3**	**www**	**6**	**google**	**3**	**com**	**...**

We need to extract and decode something like this:

“\x03www\x06google\x03com\x00”

Since we want to use only common shell tools, a good choice to convert binary characters in printable ones is **tr**, a tool used to translate set of character. Sets can be strings of printable characters (represent themselves), or interpreted sequences.

From the manual of **tr**:

```
Interpreted sequences are:
    \NNN   character with octal value NNN (1 to 3 octal digits)
```

So, with a little trick, we can convert the domain name of DNS requests in a printable string:

```
    tr '\001-\015' '.'
```

This convert binary values between 1 (\001) and 13 (\015) into dots, joining the domain levels of DNS requests.

But it's not enough... We must do a better selection of the data. This can be done with the option **-f**, providing a custom pcap filter (http://www.manpagez.com/man/7/pcap-filter/).

The following is the filter to capture only DNS requests, since DNS uses UDP as network protocol and the standard server port is 53:

```
    udp dstport=53
```

The last thing to do is to ignore the (few) extracted fields that are not domain name:

```
    grep -o -E '[a-zA-Z0-9_-]+[a-zA-Z0-9\._-]+'
```

Putting it all together:

```
root@backtrack-base# hexinject -s -i eth0 -f 'udp dstport=53' -r -c 10 | tr
'\001-\015' '.' | strings --bytes=8 | grep -o -E '[a-zA-Z0-9_-]+[a-zA-Z0-9\._-]
+'
www.xkcd.org
www.xkcd.org
www.google.co.uk
www.google.com
www.google.co.uk
www.google.com
www.xkcd.org
```

Et voila! A one-line DNS sniffer! Compared to writing a sniffer in C or even Python or Perl, how much time was saved?

We'll explore more example of binary protocol analysis in the advanced section.

HexInject as Injector

HexInject can be used as injector when the option "**-p**" is provided. This functionality is complementary to the sniffing mode, and, when combined together, they can lead to rather interesting results.

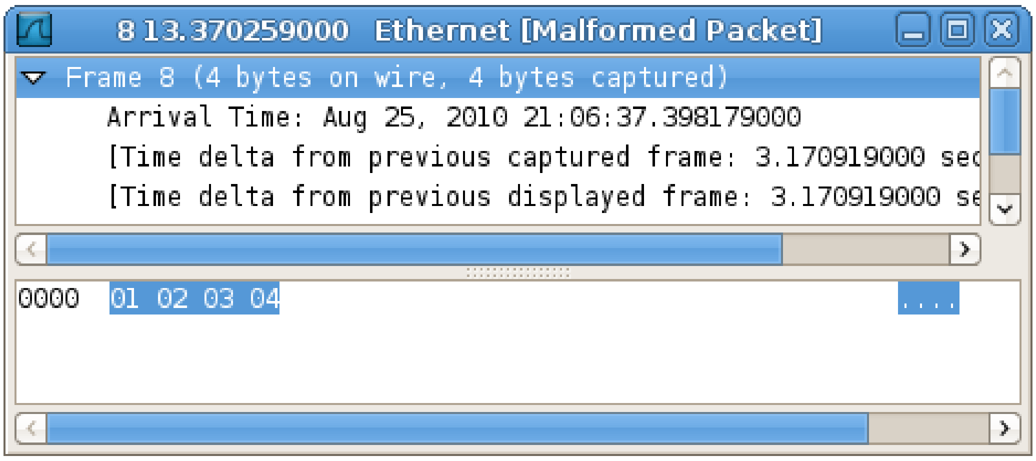
For now we'll briefly explore how to use HexInject as an injection tool.

Injecting

Simply, we can choose to inject data in raw or hexadecimal format, as seen before with the sniffing mode. HexInject reads data from the standard input (stdint), so, to provide him custom strings, we must use the pipe operator:

```
root@backtrack-base# echo "01 02 03 04" | hexinject -p -i eth0
```

Some hex bytes have just been injected:



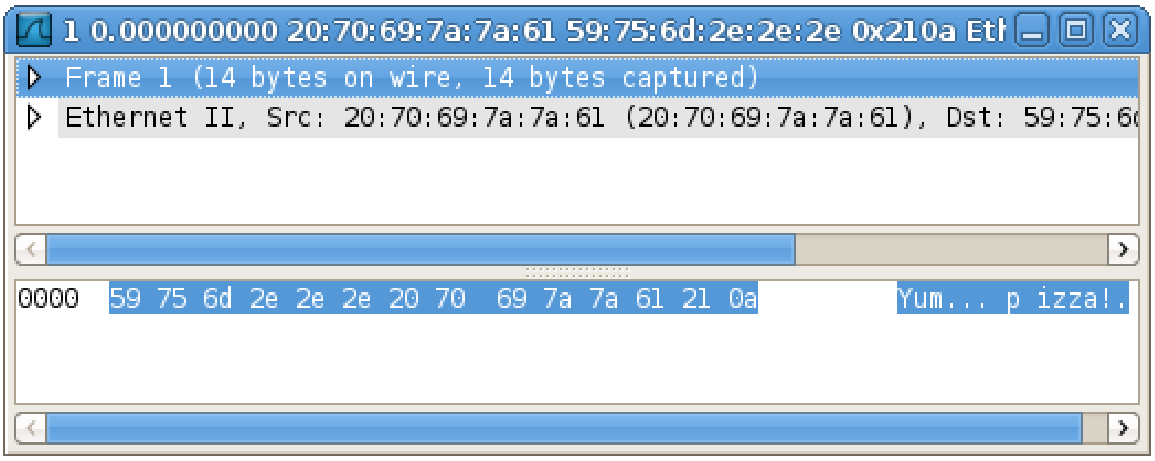
```
8 13.370259000   Ethernet [Malformed Packet]
▽ Frame 8 (4 bytes on wire, 4 bytes captured)
      Arrival Time: Aug 25, 2010 21:06:37.398179000
      [Time delta from previous captured frame: 3.170919000 sec
      [Time delta from previous displayed frame: 3.170919000 se

0000  01 02 03 04                                          ....
```

The same thing can be done in raw mode:

```
root@backtrack-base# echo 'Yum... pizza!' | hexinject -p -i eth0 -r
```

The result, as you can imagine, is:



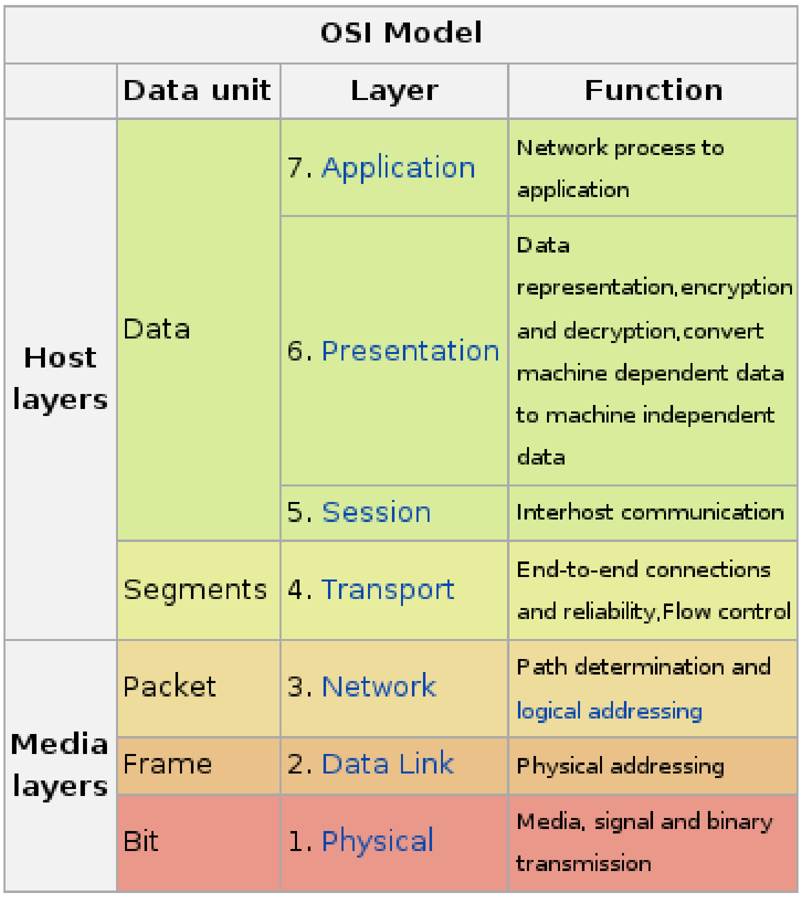
```
1 0.000000000 20:70:69:7a:7a:61 59:75:6d:2e:2e:2e 0x210a Eth
▷ Frame 1 (14 bytes on wire, 14 bytes captured)
▷ Ethernet II, Src: 20:70:69:7a:7a:61 (20:70:69:7a:7a:61), Dst: 59:75:6

0000  59 75 6d 2e 2e 2e 20 70  69 7a 7a 61 21 0a         Yum... p izza!.
```

The **important** thing to note, is that the tool injects packets *"as they are"* in the network, without performing any kind of parsing (the only exception is the checksum calculation, but the feature can be disabled).

So, to be correctly interpreted by other hosts on the network, the packets must have a correct structure, and must be properly encapsulated.

HexInject operates at the Data Link layer of the OSI model (image from http://en.wikipedia.org/wiki/OSI_model):



OSI Model

	Data unit	Layer	Function
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication
	Segments	4. Transport	End-to-end connections and reliability, Flow control
Media layers	Packet	3. Network	Path determination and logical addressing
	Frame	2. Data Link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

Build your own packages taking this into account.

The "basic usage" part of this document is over. The next sections will show more advanced uses of the tool.

Advanced usage

“You don't have to cook fancy or complicated masterpieces,
just good food from fresh ingredients.”
Julia Child

This section will show more advanced uses of HexInject, but the material will always be presented as simply as possible and with extensive use of examples.

The operating environment is **BackTrack 4 R1** (downloadable from here: http://www.backtrack-linux.org/downloads/), virtualized with *VirtualBox*.

It's assumed that the system has two network interfaces (**eth0**, **eth1**), if these differ from your, you must adapt the examples to your system (not difficult).

Binary protocols

We have seen how to extract information from textual and “mixed” protocols. Now we'll see how to extract information from binary protocols and how to read binary header fields.

A common LAN protocol we can easily spot “in-the-wild” is Address Resolution Protocol (ARP). This protocol is used to determine a network host's hardware address (MAC) when only it's network layer address (IP address) is known.

The structure of the protocol is simple: it includes the addresses of the sender and recipient and a field indicating whether the packet is a request or a response.

Let's try to capture a packet to experiment on:

```
root@backtrack-base# hexinject -s -i eth0 -f 'arp' -c 1
FF FF FF FF FF FF AA 00 04 00 0A 04 08 06 00 01 08 00 06 04 00 01 AA 00 04 00 0A
04 C0 A8 01 09 00 00 00 00 00 00 C0 A8 01 04
```

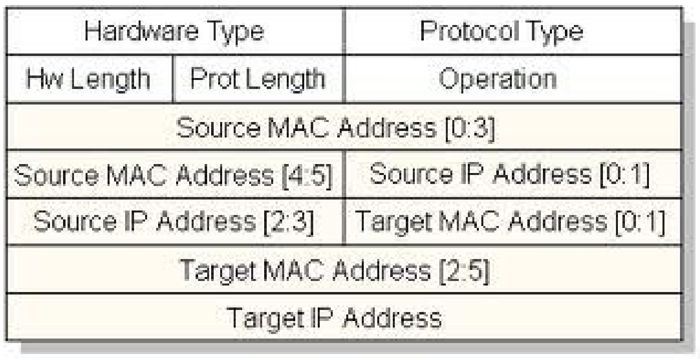
We can save the captured packet in the file 'arp.example'.

Having a little knowledge of the ARP structure it's possible to divide the protocol header from the ethernet frame:

```
FF FF FF FF FF FF AA 00 04 00 0A 04 08 06 00 01 08 00 06 04 00 01 AA 00 04 00 0A
04 C0 A8 01 09 00 00 00 00 00 00 C0 A8 01 04
```

The red part is the ethernet frame. It contains the destination MAC address (in this case ff:ff:ff:ff:ff:ff, broadcast), the sender MAC address (aa:00:04:00:0a:04) and the next header type (in this case ARP, 0x0806).

The red part is the ARP header:



Hardware Type		Protocol Type	
Hw Length	Prot Length	Operation	
Source MAC Address [0:3]			
Source MAC Address [4:5]		Source IP Address [0:1]	
Source IP Address [2:3]		Target MAC Address [0:1]	
Target MAC Address [2:5]			
Target IP Address			

Let's see what can we extract from this bunch of bytes.

Extract binary fields

Our goal is to create a complete ARP sniffer to print hexadecimal data in a comprehensible manner. We'll write it in form of shell script using only bash and awk.

The first two fields to extract are Hardware Type and Protocol Type, usually set to "0x0001" (Ethernet) and "0x0800" (Ipv4). Since these fields are of a fixed length of 2 bytes we can easily print them using **awk**.

```
root@backtrack-base# cat arp.example | awk '{ print "0x"$15$16 }'
0x0001
root@backtrack-base# cat arp.example | awk '{ print "0x"$17$18 }'
0x0800
```

In the script we'll add a function to convert the value in the protocol name.

The next step is printing the length of the protocol addresses. Since a decimal value it's more useful we have to change a little the awk command:

```
root@backtrack-base# cat arp.example | awk --non-decimal-data
'{ printf("%d","0x"$19) }'
6
root@backtrack-base# cat arp.example | awk --non-decimal-data
'{ printf("%d","0x"$20) }'
4
```

The result is correct: 6 bytes for MAC addresses and 4 bytes for IP numbers. Note: the option '--*non-decimal-data*' has been introduced more recently in GNU awk, and is optional because it's not compatible with old scripts, but it is very useful to interpret hexadecimal numbers as inputs.

Now we can analyze the opcode:

```
root@backtrack-base# cat arp.example | awk '{ print "0x"$21$22 }'
0x0001
```

In this case it's an ARP request (opcode 0x0001), but you can encounter also responses (opcade 0x0002).

The last things left to be extracted are the MAC and IP address of the source and the target.

Source addresses:

```
root@backtrack-base# cat arp.example | awk '{ print
$23":"$24":"$25":"$26":"$27":"$28 }'
AA:00:04:00:0A:04

root@backtrack-base# cat arp.example | awk --non-decimal-data '{ printf("%d.%d.
%d.%d", "0x"$29, "0x"$30, "0x"$31, "0x"$32); }'
192.168.1.9
```

Target addresses:

```
root@backtrack-base# cat arp.example | awk '{ print
$33":"$34":"$35":"$36":"$37":"$38 }'
00:00:00:00:00:00
root@backtrack-base# cat arp.example | awk --non-decimal-data '{ printf("%d.%d.
%d.%d", "0x"$39, "0x"$40, "0x"$41, "0x"$42); }'
192.168.1.4
```

Easy, isn't it?

We've converted IP address to dotted decimal style using awk's printf() (as before we need the option –non-decimal-data), and “decoded” MAC addresses just joining the 6 bytes with “:” characters.

Now that we know how to get all the information we need, let's see if we can put these commands together to create a script. It will display the packet in a pretty manner:

```
#!/bin/bash

awk --non-decimal-data '
{

  print "+--- hw type ---+--- pr type ---+";
  print "|     0x" $15$16 "      |     0x" $17$18 "      |";

  print "+--- hw size ---+--- pr size ---+";
  print "|      0x" $19 "       |      0x" $20 "       |";

  print "+-------- opcode (type) --------+";
  print "|         0x" $21$22 "   " ($22 == 1 ? "request" : "response") "         |";

  print "+---------- source hw ----------+";
  print "|        " $23":"$24":"$25":"$26":"$27":"$28 "        |";


  print "+---------- source pr ----------+";
  ip1 = sprintf("%d.%d.%d.%d", "0x"$29, "0x"$30, "0x"$31, "0x"$32);
  len1 = length(ip1);
  printf("|        %s%*c\n", ip1, 24-len1, "|");

  print "+---------- target hw ----------+";
  print "|        " $33":"$34":"$35":"$36":"$37":"$38 "        |";

  print "+---------- target pr ----------+";
  ip2 = sprintf("%d.%d.%d.%d", "0x"$39, "0x"$40, "0x"$41, "0x"$42);
  len2 = length(ip2);
  printf("|        %s%*c\n", ip2, 24-len2, "|");

  print "+-------------------------------+";
  print "";

}
'
```

The script is very simple, but it is difficult to mentally visualize its output without running it (obviously you can also pipe the script to a running HexInject process to format packets in real time):

```
root@backtrack-base# cat arp.example | ./arp_decode.sh
+--- hw type ---+--- pr type ---+
|    0x0001     |    0x0800     |
+--- hw size ---+--- pr size ---+
|     0x06      |     0x04      |
+-------- opcode (type) --------+
|        0x0001  request        |
+---------- source hw ----------+
|       AA:00:04:00:0A:04       |
+---------- source pr ----------+
|        192.168.1.9            |
+---------- target hw ----------+
|       00:00:00:00:00:00       |
+---------- target pr ----------+
|        192.168.1.4            |
+-------------------------------+
```

Although it is only a script, through the awk language it's possible to convert, edit and format any kind of textual output, making this the perfect language for our purposes.

As you've seen, sometimes, simple scripts and pipelines are as powerful as complex programs. The difference is in development time and versatility.

In the appendix you can find various cheatsheet to extract field from the most common protocols. Be sure to give a look...

Advanced piping

So far we only used pipes comprising one instance of HexInject, either reading or injecting data.

Is, of course, possible to combine two different HexInject processes, which, running in different modalities, allow the modification of network packets "on the fly".

An intuitive pattern, generally applicable when we need to alter a flow of data is the following:

hexinject -s -i 'src int' -f 'filter' | … | … | hexinject -p -i 'dst int'

A first instance of HexInject read the data from the *source interface* usually selecting the data through a *pcap filter* (it's rare the necessity to analyze all the traffic, so the use of filters is strongly encouraged).

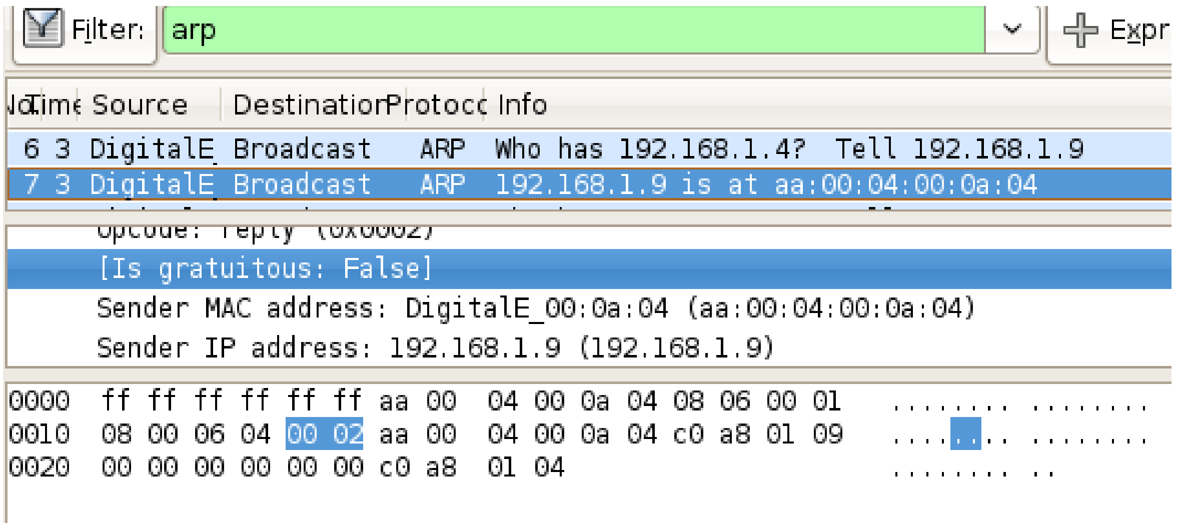
Then the traffic is analysed by a serie of filters (cmd-line tools) and modified.

Finally the traffic is re-injected in the network by a second instance of HexInject, running on the *destination interface* (which may be or not the same as the source interface).

A simple example of this is a "conversion" of an ARP request in an ARP response just changing one bit of the packet:

```
root@backtrack-base# hexinject -s -i eth0 -c 1 -f 'arp' | replace '06 04 00 01'
'06 04 00 02' | hexinject -p -i eth0
```

Wireshark dump:



```
Filter: arp
No. Time Source     Destination Protocol Info
6  3  DigitalE_  Broadcast   ARP  Who has 192.168.1.4?  Tell 192.168.1.9
7  3  DigitalE_  Broadcast   ARP  192.168.1.9 is at aa:00:04:00:0a:04

        Opcode: reply (0x0002)
        [Is gratuitous: False]
        Sender MAC address: DigitalE_00:0a:04 (aa:00:04:00:0a:04)
        Sender IP address: 192.168.1.9 (192.168.1.9)

0000  ff ff ff ff ff ff aa 00  04 00 0a 04 08 06 00 01   ........ ........
0010  08 00 06 04 00 02 aa 00  04 00 0a 04 c0 a8 01 09   ........ ........
0020  00 00 00 00 00 00 c0 a8  01 04                     ........ ..
```

Note that the strings passed to *replace* are quite long, though they differs in only one bit. This is because the pipe is "stateless", so there's the risk of altering wrong parts of the packets.

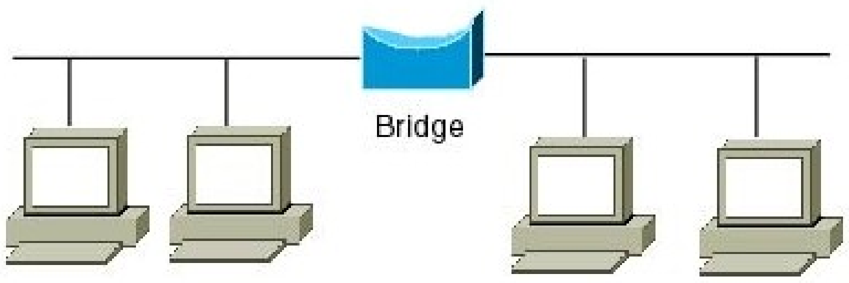
If you plan to do the same thing with a “smarter” pipe you can adapt the shell script seen previously for the parsing and dumping of ARP packets.

We said that the source interface may not coincide with the destination interface. This opens up several new possibilities.

We could put up a pseudo transparent bridge built using only two lines of bash:

```
root@backtrack-base# hexinject -s -i eth0 -c 1 -f 'src host 192.168.1.9' |
hexinject -p -i eth1
root@backtrack-base# hexinject -s -i eth1 -c 1 -f 'dst host 192.168.1.9' |
hexinject -p -i eth0
```

Actually this example can surely be improved, it just demonstrate the versatility of the tools.



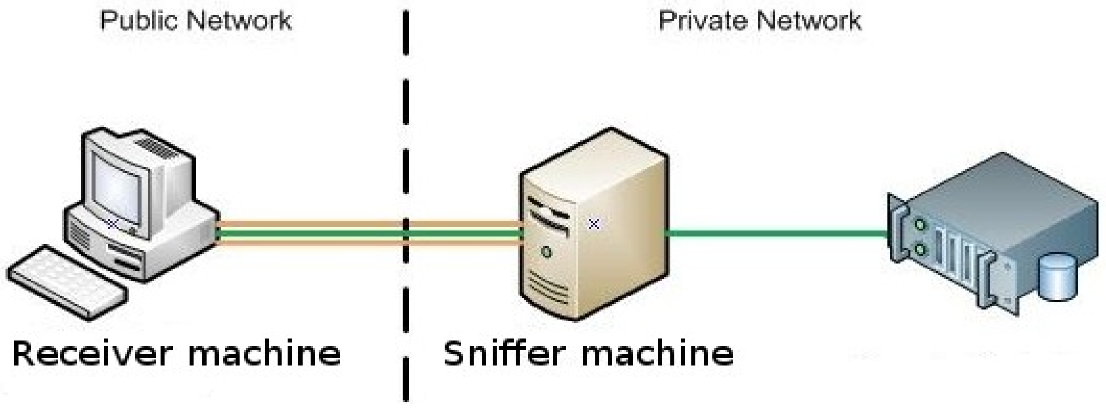
It's even possible to emulate NAT opportunely replacing the IP:

```
root@backtrack-base# hexinject -s -i eth0 -c 1 -f 'src host 192.168.1.9' |
replace 'C0 A8 01 09' 'C0 A8 01 04' | hexinject -p -i eth1

root@backtrack-base# hexinject -s -i eth1 -c 1 -f 'dst host 192.168.1.9' |
replace 'C0 A8 01 04' 'C0 A8 01 09' | hexinject -p -i eth0
```

Note that these two examples lack the management of MAC addresses, that can be implemented as a script placed in the middle of the pipe. Nevertheless the examples give an idea of what is possible to do.

A final, but very interesting, example of advanced pipes is the combined use of netcat and hexinject to create a **remote sniffer**.



The sniffer is located on the machine "**192.168.56.101**", a backtrack box running **inside virtualbox**, that has not direct access to the internet:

```
host@192.168.56.101# hexinject -s -i eth0 -f 'not dst host 192.168.56.1' | nc -u 192.168.56.1 5555
```

Note the use of pcap filters to avoid an infinite loops of sniffed and transmitted packets.

The receiver is located on "**192.168.1.9**" or "**192.168.56.1**" inside the virtualbox network. To receive traffic we need just a listening netcat:

```
host@192.168.1.9# nc -l -p 5555 -u
0A 00 27 00 00 00 08 00 27 49 48 03 08 00 45 00 00 54 00 00 40 00 40 01 7F EC C0
A8 38 65 C0 A8 01 07 08 00 17 93 59 1A 00 02 DB 42 A7 4C 18 BE 01 00 08 09 0A 0B
0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
0A 00 27 00 00 00 08 00 27 49 48 03 08 00 45 00 00 54 00 00 40 00 40 01 7F EC C0
A8 38 65 C0 A8 01 07 08 00 AC 92 59 1A 00 03 DC 42 A7 4C 82 BD 01 00 08 09 0A 0B
0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
...
```

As you can see, some ICMP packets (generated by the ping utility) has been captured and remotely transmitted to the receiver machine. No tunneling protocols, no GRE, just netcat and hexinject, two simple standalone tools.

To conclude this last section we can only say that the possibilities and combinations of tools are virtually endless, and the only limit is our imagination.

I hope this document has not bored you, please contact me via email if you find new interesting uses of the tool, so that i can add new examples to this document.

Appendix

"If you can't explain it simply, you don't understand it well enough"

Albert Einstein

"The ability to simplify means to eliminate the unnecessary so that the necessary may speak"

Hans Hofmann

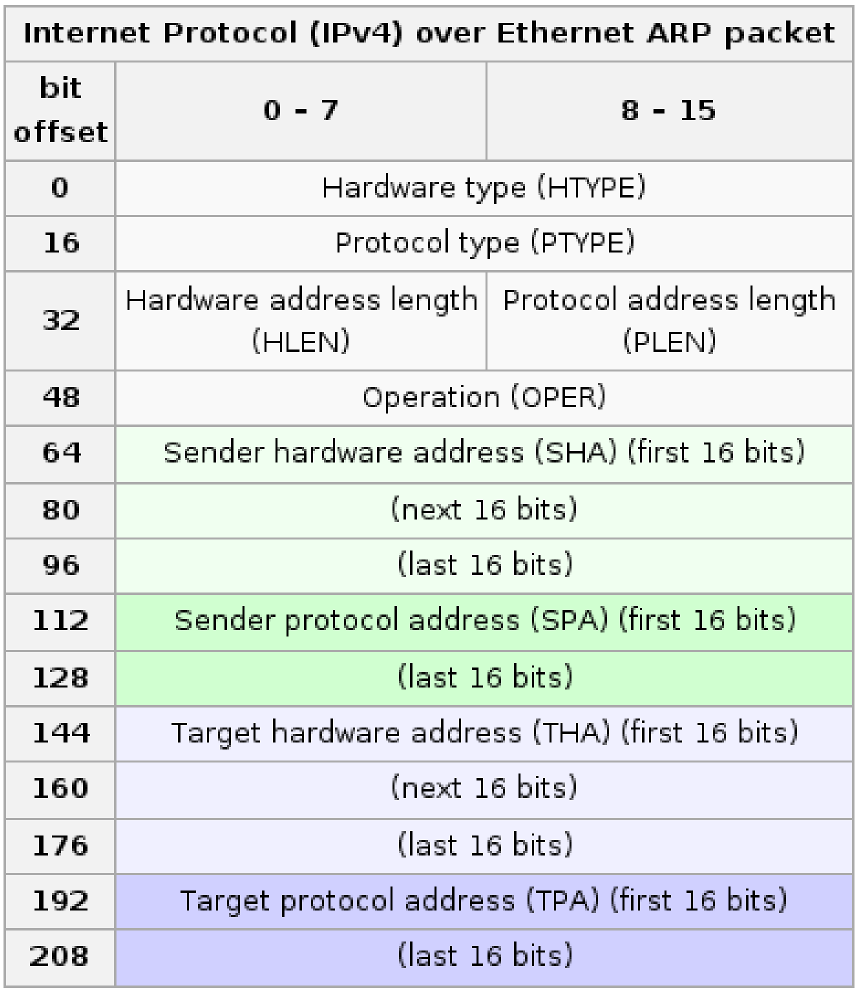
This appendix contains various cheatsheets useful to "decrypt" hexadecimal packet dump. They describe the structure of the most common protocols and how to extract their fields using only simple command-line tools.

Surely a lazy programmer/penstester aid ;)

The appendix includes also some visual representation of protocol headers from Wikipedia (http://www.wikipedia.org).

ARP cheatsheet

Visual representation:



Internet Protocol (IPv4) over Ethernet ARP packet		
bit offset	**0 - 7**	**8 - 15**
0	Hardware type (HTYPE)	
16	Protocol type (PTYPE)	
32	Hardware address length (HLEN)	Protocol address length (PLEN)
48	Operation (OPER)	
64	Sender hardware address (SHA) (first 16 bits)	
80	(next 16 bits)	
96	(last 16 bits)	
112	Sender protocol address (SPA) (first 16 bits)	
128	(last 16 bits)	
144	Target hardware address (THA) (first 16 bits)	
160	(next 16 bits)	
176	(last 16 bits)	
192	Target protocol address (TPA) (first 16 bits)	
208	(last 16 bits)	

Capture example:

```
FF FF FF FF FF FF AA 00 04 00 0A 04 08 06 00 01 08 00 06 04 00 01 AA 00 04 00 0A
04 C0 A8 01 09 00 00 00 00 00 00 C0 A8 01 04
```

Capture explanation:

`FF FF FF FF FF FF`	Destination hardware address
`AA 00 04 00 0A 04`	Source hardware address
`08 06`	Type
`00 01`	Hardware type
`08 00`	Protocol type
`06`	Hardware size
`04`	Protocol size
`00 01`	Opcode
`AA 00 04 00 0A 04`	Sender hardware address
`C0 A8 01 09`	Sender protocol address
`00 00 00 00 00 00`	Target hardware address
`C0 A8 01 04`	Target protocol address

Field extraction cheatsheet:

Field	**Command(s)**	**Result**
Destination hw address	awk '{ print $1":"$2":"$3":"$4":"$5":"$6 }'	FF:FF:FF:FF:FF:FF
Source hw address	awk '{ print $7":"$8":"$9":"$10":"$11":"$12 }'	AA:00:04:00:0A:04
Type	awk '{ print "0x"$13$14 }'	0x0806
Hardware type	awk '{ print "0x"$15$16 }'	0x0001
Protocol type	awk '{ print "0x"$17$18 }'	0x0800
Hardware size	awk '{ print "0x"$19 }'	0x06
Protocol size	awk '{ print "0x"$20 }'	0x04
Opcode	awk '{ print "0x"$21$22 }'	0x0001
Sender hw address	awk '{ print $23":"$24":"$25":"$26":"$27":"$28 }'	AA:00:04:00:0A:04
Sender proto address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$29, "0x"$30, "0x"$31, "0x"$32); }'	192.168.1.9
Target hw address	awk '{ print $33":"$34":"$35":"$36":"$37":"$38 }'	00:00:00:00:00:00
Target proto address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$39, "0x"$40, "0x"$41, "0x"$42); }'	192.168.1.4

ICMP cheatsheet

Visual representation (IP):

bit offset	0-3	4-7	8-13	14-15	16-18	19-31
0	Version	Header Length	Differentiated Services Code Point	Explicit Congestion Notification	Total Length	
32	Identification				Flags	Fragment Offset
64	Time to Live		Protocol		Header Checksum	
96	Source IP Address					
128	Destination IP Address					
160	Options (if Header Length > 5)					
160 or 192+	Data					

Visual representation (ICMP):

Bits	0-7	8-15	16-23	24-31
0	Type	Code	Checksum	
32	ID		Sequence	

Capture example:

```
1C AF F7 6B 0E 4D AA 00 04 00 0A 04 08 00 45 00 00 54 00 00 40 00 40 01 54 4E C0
A8 01 09 C0 A8 64 01 08 00 34 98 D7 10 00 01 5B 68 98 4C 00 00 00 00 2D CE 0C 00
00 00 00 00 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
```

Capture explanation:

1C AF F7 6B 0E 4D	Destination hardware address
AA 00 04 00 0A 04	Source hardware address
08 00	Type
45	Version / Header length
00	ToS/DSF
00 54	Total length
00 00	ID
40 00	Flags/Fragment offset

`40`	TTL
`01`	Protocol
`54 4E`	Checksum
`C0 A8 01 09`	Source address
`C0 A8 64 01`	Destination address
`08`	Type
`00`	Code
`34 98`	Checksum
`D7 10`	ID
`00 01`	Sequence number
`5B 68 98 4C 00 00 00 00 2D CE 0C 00 00 00 00 00 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37`	Data

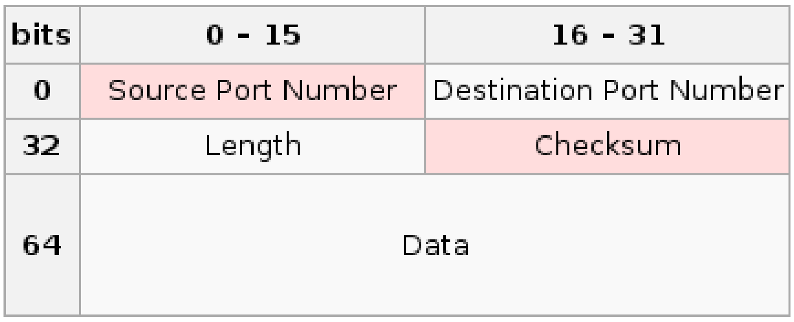
Field extraction cheatsheet:

Field	**Command(s)**	**Result**
Destination hw address	awk '{ print $1":"$2":"$3":"$4":"$5":"$6 }'	`1C:AF:F7:6B:0E:4D`
Source hw address	awk '{ print $7":"$8":"$9":"$10":"$11":"$12 }'	AA:00:04:00:0A:04
Type	awk '{ print "0x"$13$14 }'	0x0800
Version	awk --non-decimal-data '{ print rshift("0x"$15, 4) }'	4
Header length	awk --non-decimal-data '{ print and("0x"$15, 0xf)*4 }'	20
ToS/DSF	awk '{ print "0x"$16 }'	0x00
Total length	awk --non-decimal-data '{ printf("%d","0x"$17$18) }'	84
ID	awk '{ print "0x"$19$20 }'	0x0000
Flags	awk --non-decimal-data '{ print rshift("0x"$21$22,13) }'	2
	awk --non-decimal-data '{ $a=rshift("0x"$21$22,13); if(and($a,4)) { print "Reserved" } if(and($a,2)) { print "Do not fragment" } if(and($a,1)) { print "More fragments" } }'	Do not fragment
Fragment offset	awk --non-decimal-data '{ print and("0x"$21$22,13) }'	0
TTL	awk '{ print "0x"$23 }'	0x40
Protocol	awk '{ print "0x"$24 }'	0x01

Checksum	awk '{ print "0x"$25$26 }'	0x544E
Source address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$27, "0x"$28, "0x"$29, "0x"$30); }'	192.168.1.9
Destination address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$31, "0x"$32, "0x"$33, "0x"$34); }'	192.168.100.1
Type	awk '{ print "0x"$35 }'	0x08
Code	awk '{ print "0x"$36 }'	0x00
Checksum	awk '{ print "0x"$37$38 }'	0x3498
ID	awk '{ print "0x"$39$40 }'	0xD710
Sequence number	awk '{ print "0x"$41$42 }'	0x0001
Data	sed 's/.\{125\}//' | replace ' ' '\\x' | xargs printf	[h�L-� ############## ## !"#$ %&'()*+,-./01234567

UDP cheatsheet

Visual representation:



bits	0 - 15	16 - 31
0	Source Port Number	Destination Port Number
32	Length	Checksum
64	Data	

Capture example:

```
1C AF F7 6B 0E 4D AA 00 04 00 0A 04 08 00 45 00 00 3C 9B 23 00 00 40 11 70 BC C0
A8 01 09 D0 43 DC DC 91 02 00 35 00 28 6F 0B AE 9C 01 00 00 01 00 00 00 00 00 00
03 77 77 77 06 67 6F 6F 67 6C 65 03 63 6F 6D 00 00 01 00 01
```

Capture explanation:

`1C AF F7 6B 0E 4D`	Destination hardware address
`AA 00 04 00 0A 04`	Source hardware address
`08 00`	Type
`45`	Version / Header length
`00`	ToS/DSF
`00 3C`	Total length
`9B 23`	ID
`00 00`	Flags/Fragment offset
`40`	TTL
`11`	Protocol
`70 BC`	Checksum
`C0 A8 01 09`	Source address
`D0 43 DC DC`	Destination address
`91 02`	Sorce port
`00 35`	Destination port
`00 28`	Length
`6F 0B`	Checksum
`AE 9C 01 00 00 01 00 00 00 00 00 00 03 77 77 77 06 67 6F 6F 67 6C 65 03 63 6F`	Data

```
6D 00 00 01 00 01
```

Field extraction cheatsheet:

Field	**Command(s)**	**Result**
Destination hw address	awk '{ print $1":"$2":"$3":"$4":"$5":"$6 }'	`1C:AF:F7:6B:0E:4D`
Source hw address	awk '{ print $7":"$8":"$9":"$10":"$11":"$12 }'	AA:00:04:00:0A:04
Type	awk '{ print "0x"$13$14 }'	0x0800
Version	awk --non-decimal-data '{ print rshift("0x"$15, 4) }'	4
Header length	awk --non-decimal-data '{ print and("0x"$15, 0xf)*4 }'	20
ToS/DSF	awk '{ print "0x"$16 }'	0x00
Total length	awk --non-decimal-data '{ printf("%d","0x"$17$18) }'	60
ID	awk '{ print "0x"$19$20 }'	0x9B23
Flags	awk --non-decimal-data '{ print rshift("0x"$21$22,13) }'	0
	awk --non-decimal-data '{ $a=rshift("0x"$21$22,13); if(and($a,4)) { print "Reserved" } if(and($a,2)) { print "Do not fragment" } if(and($a,1)) { print "More fragments" } }'	
Fragment offset	awk --non-decimal-data '{ print and("0x"$21$22,13) }'	0
TTL	awk '{ print "0x"$23 }'	0x40
Protocol	awk '{ print "0x"$24 }'	0x11
Checksum	awk '{ print "0x"$25$26 }'	0x70BC
Source address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$27, "0x"$28, "0x"$29, "0x"$30); }'	192.168.1.9
Destination address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$31, "0x"$32, "0x"$33, "0x"$34); }'	208.67.220.220
Source port	awk --non-decimal-data '{ printf("%d", "0x"$35$36) }'	37122
Destination port	awk --non-decimal-data '{ printf("%d", "0x"$37$38) }'	53
Length	awk --non-decimal-data '{ printf("%d", "0x"$39$40) }'	`40`
Checksum	awk '{ print "0x"$41$42 }'	0x6F0B
Data	sed 's/.\{125\}//' | replace ' ' '\\x' | xargs printf	��###www#google#com##

TCP cheatsheet

Visual representation:

Bit offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Source port																Destination port															
32	Sequence number																															
64	Acknowledgment number																															
96	Data offset				Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
128	Checksum																Urgent pointer															
160 ...	Options (if Data Offset > 5) ...																															

Capture example:

```
1C AF F7 6B 0E 4D AA 00 04 00 0A 04 08 00 45 00 00 34 5A AE 40 00 40 06 5E 67 C0
A8 01 09 58 BF 67 3E 9B 44 00 50 8E B5 C6 AC 15 93 47 9E 80 10 00 58 A5 A0 00 00
01 01 08 0A 00 09 C3 B2 42 5B FA D6
```

Capture explanation:

1C AF F7 6B 0E 4D	Destination hardware address
AA 00 04 00 0A 04	Source hardware address
08 00	Type
45	Version / Header length
00	ToS/DSF
00 34	Total length
5A AE	ID
40 00	Flags/Fragment offset
40	TTL
06	Protocol
5E 67	Checksum
C0 A8 01 09	Source address
58 BF 67 3E	Destination address
9B 44	Sorce port
00 50	Destination port
8E B5 C6 AC	Sequence number
15 93 47 9E	Ack number

`80`	Header length
`10`	Flags
`00 58`	Window
`A5 A0`	Checksum
`00 00`	Padding
`01 01 08 0A 00 09 C3 B2 42 5B FA D6`	Options

Field extraction cheatsheet:

Field	**Command(s)**	**Result**
Destination hw address	awk '{ print $1":"$2":"$3":"$4":"$5":"$6 }'	`1C:AF:F7:6B:0E:4D`
Source hw address	awk '{ print $7":"$8":"$9":"$10":"$11":"$12 }'	AA:00:04:00:0A:04
Type	awk '{ print "0x"$13$14 }'	0x0800
Version	awk --non-decimal-data '{ print rshift("0x"$15, 4) }'	4
Header length	awk --non-decimal-data '{ print and("0x"$15, 0xf)*4 }'	20
ToS/DSF	awk '{ print "0x"$16 }'	0x00
Total length	awk --non-decimal-data '{ printf("%d","0x"$17$18) }'	52
ID	awk '{ print "0x"$19$20 }'	0x5AAE
Flags	awk --non-decimal-data '{ print rshift("0x"$21$22,13) }'	2
	awk --non-decimal-data '{ $a=rshift("0x"$21$22,13); if(and($a,4)) { print "Reserved" } if(and($a,2)) { print "Do not fragment" } if(and($a,1)) { print "More fragments" } }'	Do not fragment
Fragment offset	awk --non-decimal-data '{ print and("0x"$21$22,13) }'	0
TTL	awk '{ print "0x"$23 }'	0x40
Protocol	awk '{ print "0x"$24 }'	0x06
Checksum	awk '{ print "0x"$25$26 }'	0x70BC
Source address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$27, "0x"$28, "0x"$29, "0x"$30); }'	192.168.1.9
Destination address	awk --non-decimal-data '{ printf("%d.%d.%d.%d", "0x"$31, "0x"$32, "0x"$33, "0x"$34); }'	88.191.103.62
Source port	awk --non-decimal-data '{ printf("%d", "0x"$35$36) }'	39748
Destination port	awk --non-decimal-data '{ printf("%d", "0x"$37$38) }'	80
Sequence number	awk --non-decimal-data '{ printf("%d",	2394277548

	"0x"$39$40$41$42) }'	
Ack number	awk --non-decimal-data '{ printf("%d", "0x"$43$44$45$46) }'	361973662
Header length	awk --non-decimal-data '{ printf("%d", rshift("0x"$47,4)*4) }'	32
Flags	awk --non-decimal-data '{ $a="0x"$48; if(and($a,128)) { print "CWR" } if(and($a,64)) { print "ECN-Echo" } if(and($a,32)) { print "Urg" } if(and($a,16)) { print "Ack" } if(and($a,8)) { print "Push" } if(and($a,4)) { print "Rst" } if(and($a,2)) { print "Syn" } if(and($a,1)) { print "Fin" } }'	Ack
Window	awk --non-decimal-data '{ printf("%d", "0x"$49$50) }'	88
Checksum	awk '{ print "0x"$51$52 }'	0
Padding	...	...
Options	...	...
Data	sed 's/.\{197\}//' | replace ' ' '\\x' | xargs printf (The offset may vary, options dependent)	...