suse.com

# Making Sense of Hexdump - SUSE Communities

31-39 minutes

I often work with binary data that has a format I could interpret if only I could see it in a human readable form. Most text editors aren't much use. One way (of many) to solve that is to use the hexdump utility. hexdump is very versatile and allows you to look at the structure inside binary files as you see fit and once you learn how to use it you can apply it quickly to many problems. I think that easily qualifies as cool.

If you've used the hexdump man page then you may have found the formatting syntax quite intimidating. Part of the format syntax is pretty much the same as a commonly used function in the C programming language. I found hexdump syntax difficult and I am a C programmer. In truth it isn't really difficult to understand but without an example to use while reading the documentation it may not be obvious how the formatting options are applied.

**Contents:**

**Basic usage**

The place to start is with a guess and just give hexdump the name of a data file and get whatever default format hexdump uses:

```
# hexdump mydata
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000010 0002 0003 0001 0000 8430 0804 0034 0000
0000020 22ec 0000 0000 0000 0034 0020 0008 0028
*
0001030 0027 0024 0006 0000 0034 0000 8034 0804
```

What you are seeing there is the binary contents of the mydata file shown as lines of 8 individual 16-bit values in hexadecimal. The first number on each line is the starting offset in the file for the first of the 8 following values on that line. The * indicates that all of the lines from 0000030 to 0001020 inclusive would be the same value as the 0000020 line and that keeps the output condensed for large files with long and aligned repeating sequences. Be aware that, with no command line options, hexdump will dump the entire file to screen so use it with care on large files.

The first 16 bits of the example file are the hexadecimal value 457f. It’s important to note that the value shown is as interpreted on an

Intel x86 CPU. On some other types of CPU that value could be output as 7f45 due to differences in something called byte ordering. If you are not familiar with the term it's probably worth reading about it, for example on Wikipedia.

If 16-bit values isn't really what you were looking for then a quick review of the hexdump man page indicates there are several switches that select some other pre-defined output formats. There's only one other option, -C, that produces hexadecimal output:

```
# hexdump -C mydata
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00
00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  30 84 04 08 34
00 00 00  |........0...4...|
00000020  ec 22 00 00 00 00 00 00  34 00 20 00 08
00 28 00  |."......4. ...(.|
*
00001030  27 00 24 00 06 00 00 00  34 00 00 00 34
80 04 08  |'.$.....4...4...|
```

Now you can see individual byte values and even see the text for those parts of the file that are normal text characters. So, now I would guess from the "ELF" signature that this file is probably a program executable.

**Viewing only part of a file**

If you have a really big file where you are only interested in seeing some of the data you can use the -n option to specify how many bytes to dump. The -n 32 in the following example causes hexdump to show only 32 bytes of the file.

```
# hexdump -C
```

**-n 32**

```
 mydata
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00
00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  30 84 04 08 34
00 00 00  |........0...4...|
00000020
```

If you are only interested in the data some distance into the file you can use the -s option to specify how far into the file to start dumping from. The -s 16 in the following example causes hexdump to show the content of the file beginning 16 bytes in from the start and continuing to the end of the file. :

```
# hexdump -C
```

**-s 16**

```
 mydata
00000010  02 00 03 00 01 00 00 00  30 84 04 08 34
00 00 00  |........0...4...|
00000020  ec 22 00 00 00 00 00 00  34 00 20 00 08
00 28 00  |."......4. ...(.|
*
00001030  27 00 24 00 06 00 00 00  34 00 00 00 34
80 04 08  |'.$.....4...4...|
```

You can also combine these two switches to see a subset of the data some offset into the file. In this example, using both -s 16 and -n 32 causes hexdump to show 32 bytes of the file beginning 16 bytes in from the start.:

```
# hexdump -C
```

**-s 16 -n 32**

```
 mydata
00000010  02 00 03 00 01 00 00 00  30 84 04 08 34
00 00 00  |........0...4...|
00000020  ec 22 00 00 00 00 00 00  34 00 20 00 08
00 28 00  |."......4. ...(.|
00000030
```

You can stop hexdump from replacing duplicate lines with a * by using the -v option. In the following example -v causes the previously collapsed duplicate rows at offsets 30 hex and 40 hex to be displayed:

```
# hexdump -C -s 16 -n 64
```

**-v**

```
 mydata
00000010  02 00 03 00 01 00 00 00  30 84 04 08 34
00 00 00  |........0...4...|
00000020  ec 22 00 00 00 00 00 00  34 00 20 00 08
00 28 00  |."......4. ...(.|
00000030  ec 22 00 00 00 00 00 00  34 00 20 00 08
00 28 00  |."......4. ...(.|
00000040  ec 22 00 00 00 00 00 00  34 00 20 00 08
00 28 00  |."......4. ...(.|
00000050
```

That covers the basic usage of hexdump and it has been sufficient for all my needs until recently.

**Example dumping a partition table using hexdump**

I wanted to dump the partition table on a disk so that I could mount it via a loopback device. It's easy to get the necessary information from fdisk by switching it to show sector sized units but I thought I'd play with hexdump to force myself to use the format syntax. This is where we are headed:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3d" 1/1 " %02x" 3/1 " %3d" 2/4 " %9d" "\n"'
diskdump.ddimg
00   1   1   0 83 254 255 255        63  16771797
00   0   0   0 00   0   0   0         0         0
00   0   0   0 00   0   0   0         0         0
00   0   0   0 00   0   0   0         0         0
```

The partition table consists of four 16-byte entries at offset 446 bytes into the disk (i.e. in the first sector of a typical disk). Here's an example from an image of a disk that had a single partition:

```
# hexdump -C -s 446 -n 64 diskdump.ddimg
```

```
000001be  00 01 01 00 83 fe ff ff  3f 00 00 00 d5
ea ff 00  |........?.......|
000001ce  00 00 00 00 00 00 00 00  00 00 00 00 00
00 00 00  |................|
*
000001fe
```

By coincidence the partition table entry length and the default line length of hexdump are 16 bytes so each line is a complete partition table entry.

It's probably better to have hexdump show the whole of the block rather than condense the duplicate lines:

```
# hexdump -v -s 446 -n 64 diskdump.ddimg
000001be  00 01 01 00 83 fe ff ff  3f 00 00 00 d5
ea ff 00  |........?.......|
000001ce  00 00 00 00 00 00 00 00  00 00 00 00 00
00 00 00  |................|
000001de  00 00 00 00 00 00 00 00  00 00 00 00 00
00 00 00  |................|
000001ee  00 00 00 00 00 00 00 00  00 00 00 00 00
00 00 00  |................|
000001fe
```

It's not especially difficult to decode that by hand based on published information about the format of partition table entries but you can make hexdump do the work on its own and learn enough about hexdump to be able to apply it to other things in the future. For that it is necessary to dig into the formatting syntax.

### Introducing the format string

A format string is specified with the -e command line option. The hexdump man page describes or references all the syntax elements and gives some examples but it takes some work to determine how to do something not in the examples. Here's something really simple that just outputs the whole table as space separated single byte values in hexadecimal:

```
# hexdump -s 446 -n 64 -v
```

**-e '1/1 " %02X"'**

```
 diskdump.ddimg
 00 01 01 00 83 FE FF FF 3F 00 00 00 D5 EA FF 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00#
```

You'll recognize the use of -n, -s and -v so that it dumps 64 bytes at offset 446 with no collapsing of duplicates. The rest is the format expression and the file to dump. Note that the format expression is enclosed in single quotes: '. The 1/1 means process one byte in groups of one. The n/m syntax is specific to hexdump and there are a few other hexdump specifics but the element beginning with a % comes from the C printf function family, it is referred to in the hexdump man page and you can look at the syntax in more detail with man 3 printf.

In the example, the " %02X" means output a space followed by the value of a single byte as two hexadecimal digits with a leading zero. The % means insert values from the file at this point in the output and what follows the % describes how to do that. The X means output in hexadecimal and the 2 means output as a two digit number and the zero means output leading zeros when the number is less than two digits long.

The use of an uppercase X means that the A through F digits in hexadecimal numbers will be in uppercase. Using a lowercase x causes those digits to be output as a through f in lowercase.

**Getting output in lines**

The # at the end of the output in the example above is the next shell prompt, I didn't include any end of line formatting in the example so I'll fix that first. The \n character sequence is used to specify a new line and also comes from the C printf family. But I'll get 64 lines of output with one byte value on each line if I just add a \n:

```
# hexdump -s 446 -n 64 -v -e '1/1 " %02X"
```

**“\n”**

```
' diskdump.ddimg
 00
 01
 01
 00
 83
 FE
 FF
 FF
 3F
 00
etc
```

That’s because the whole quoted expression is parsed in full repeatedly until the whole of the value for the -n switch is satisfied. The example above the expression processes one byte one time and adds a line feed. That needs to be done 64 times to meet the -n 64 requirement.

**Dumping blocks of data (m/n)**

If we go back to that 1/1 we can tell hexdump to output bytes in groups of 16 (the size of one partition table entry) by specifying 16/1 instead of 1/1:

```
# hexdump -s 446 -n 64 -v -e '
```

**16/1**

```
 " %02X" "\n"' diskdump.ddimg
 00 01 01 00 83 FE FF FF 3F 00 00 00 D5 EA FF 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The 16/1 isn’t reversible, the first number is the number of elements to process in a group and the second number is the element length.

Sixteen single byte values is what I am looking for. The other way around, 1/16, would be a single sixteen byte value. That results in a “bad byte count for conversion character X” error. The value after the / is the number of bytes to process for each %02X and the value before the / is the number of times to repeat that operation. %02X will not decode a 128 bit (16 byte) value, hence the error. Therefore, 16/1 is the appropriate syntax.

**Splitting out individual elements of the structured data**

Now I need to fill in some details. The first byte of a partition table is a status value that indicates if the partition is bootable or not. I’ll separate it out as its own decode option by handling the first byte in isolation and the remaining 15 bytes per partition as a group:

```
# hexdump -s 446 -n 64 -v -e '
```

**1/1 “%02x” 15/1**

```
 " %02X" "\n"' diskdump.ddimg
00 01 01 00 83 FE FF FF 3F 00 00 00 D5 EA FF 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

That doesn’t look any different of course but I have barely started. Next in a partition table entry is a relic of the first PCs with hard disks. There are three individual byte values that specify the starting head, sector and cylinder (least significant 8 bits only) location for the partition. That’s a group of three single byte values:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x"
```

**3/1 ” %02x” 12/1**

```
 " %02X" "\n"' diskdump.ddimg
00 01 01 00 83 FE FF FF 3F 00 00 00 D5 EA FF 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

**Decimal output**

That’s still no different, the head, sector and cylinder have numerical significance so I’ll show them in decimal for human readability:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
```

**%3d**

```
" 12/1 " %02x" "\n"' diskdump.ddimg
00   1   1   0 83 fe ff ff 3f 00 00 00 d5 ea ff 00
00   0   0   0 00 00 00 00 00 00 00 00 00 00 00 00
00   0   0   0 00 00 00 00 00 00 00 00 00 00 00 00
00   0   0   0 00 00 00 00 00 00 00 00 00 00 00 00
```

The new format element is “ %3d”, that means output a space followed by a three character wide field in decimal. Byte values can be between zero and 255 in decimal so can’t be longer than three digits. Since the numeric value is relevant I dropped the zero prefix without dropping the length prefix and get a space padded, right justified field. You can see that partition 1 has a zero status code and begins on cylinder zero, head 1, sector 1. That is the normal location for the first partition on a disk.

Next is the partition type, another single byte:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3d"
```

**1/1 ” %02x” 11/1**

```
 " %02x" "\n"' diskdump.ddimg
00   1   1   0 83 fe ff ff 3f 00 00 00 d5 ea ff 00
00   0   0   0 00 00 00 00 00 00 00 00 00 00 00 00
00   0   0   0 00 00 00 00 00 00 00 00 00 00 00 00
00   0   0   0 00 00 00 00 00 00 00 00 00 00 00 00
```

Next is the head, sector and cylinder (least significant 8 bits only) location of the end of the partition. I’ll do those in decimal again:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3d" 1/1 " %02x"
```

**3/1 " %3d" 8/1**

```
 " %02x" "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255 3f 00 00 00 d5 ea ff
00
00   0   0   0 00   0   0   0 00 00 00 00 00 00 00
00
00   0   0   0 00   0   0   0 00 00 00 00 00 00 00
00
00   0   0   0 00   0   0   0 00 00 00 00 00 00 00
00
```

**32 bit data**

Now the structure is beginning to show. After that I get to the values that will be useful in mounting the file systems and for which single byte output is not useful. The start sector number (logical block address, or LBA) and length (in sectors) for the partition are given as 32 bit values:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3d" 1/1 " %02x" 3/1 " %3d"
```

**2/4 " %08x"**

```
 "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255 0000003f 00ffead5
00   0   0   0 00   0   0   0 00000000 00000000
00   0   0   0 00   0   0   0 00000000 00000000
00   0   0   0 00   0   0   0 00000000 00000000
```

Note that the latest change adds a 2/4 count specifier, that means 2 instances of a 4 byte (32 bit) value. The %08x format means output the value as an 8 digit hexadecimal number with leading zeros. Since the LBA values have numeric significance it will be more useful in decimal and without the zero paddingl

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3d" 1/1 " %02x" 3/1 " %3d" 2/4 "
```

**%9d"**

```
 "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255        63  16771797
00   0   0   0 00   0   0   0         0         0
00   0   0   0 00   0   0   0         0         0
00   0   0   0 00   0   0   0         0         0
```

The longest 32 bit decimal number has nine digits, versus eight digits for the same number in hexadecimal so I changed the " %08x" to " %9d".

**Signed/unsigned**

That's all I needed but I was a little careless, though it didn't affect the output this time. The d format character says the value is signed (can be positive or negative). Partitions can't have negative sector, head, cylinder or LBA values. On most platforms an integer value can't be identified as signed or unsigned based only on the values of the bytes. External knowledge is required that states whether to interpret a value as signed or unsigned. Since I know the values are all unsigned then everywhere I had a d in a % format I should replace it with a u to indicate the value is unsigned (positive values only):

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1
```

**" %3u"**

```
 1/1 " %02x" 3/1
```

**" %3u"**

```
 2/4
```

**" %9u"**

```
 "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255        63  16771797
00   0   0   0 00   0   0   0         0         0
00   0   0   0 00   0   0   0         0         0
00   0   0   0 00   0   0   0         0         0
```

**Adding extra text to the output**

You can have literal text of your choosing in the output from hexdump. We've actually already had lots of it in the spaces and \n values that we've used. You can add other things and make the output more meaningful:

```
# hexdump -s 446 -n 64 -v -e '1/1
```

**"Partition:| %02x"**

```
 3/1
```

" | **%3u"**

```
 1/1 "
```

| **%02x"**

```
 3/1 "
```

| **%3u"**

```
 2/4 "
```

| **%9u"**

```
 "\n"' /data/vms/vmware/vmdk/2raw/rawfromvmdk.img
Partition:| 00 |   1 |   1 |   0 | 83 | 254 | 255
| 255 |        63 |  16771797
Partition:| 00 |   0 |   0 |   0 | 00 |   0 |   0
|   0 |         0 |         0
Partition:| 00 |   0 |   0 |   0 | 00 |   0 |   0
|   0 |         0 |         0
Partition:| 00 |   0 |   0 |   0 | 00 |   0 |   0
|   0 |         0 |         0
```

Those changes gave us column borders and an introductory word on each line.

**Put it in a shell script with some layout**

So, now I have a hexdump command line that will dump a partition table. Here it is in a shell script with some cell borders added:

```
#!/bin/bash
# Dump what would be the partition table of the
```

```
command line specified
# file if that file was partitioned media
echo
echo 'PARTITION TABLE IN' $1
echo
echo '   -------Start-------
--------End--------'
echo 'St |  Hd | Cyl | Sec | Tp |  Hd | Cyl | Sec
| LBA Start |   LBA End'
echo
'---|-----|-----|-----|----|-----|-----|---------------------
'
hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 " |
%3u" 1/1 " | %02x" 3/1 " | %3u" 2/4 " | %9u" "\n"'
$1
echo
'-------------------------------------------------------------
'
echo
echo
```

Save it as partdump somewhere in the path, change the mode to executable and you have a cool new tool:

```
# partdump diskdump.ddimg

PARTITION TABLE IN diskdump.ddimg

   -------Start-------    --------End--------
St |  Hd | Cyl | Sec | Tp |  Hd | Cyl | Sec | LBA
Start |   LBA End
---|-----|-----|-----|----|-----|-----|---------------------
00 |   1 |   1 |   0 | 83 | 254 | 255 | 255 |
63 |  16771797
00 |   0 |   0 |   0 | 00 |   0 |   0 |   0 |
0 |         0
00 |   0 |   0 |   0 | 00 |   0 |   0 |   0 |
0 |          0
00 |    0 |    0 |    0 | 00 |    0 |    0 |    0 |
0 |          0
--------------------------------------------------------------
```

**It’s a 64 bit world!**

As nice as that is, it barely scratches the surface of what you can do with hexdump. All we’ve done is handle 8 bit and 32 bit values in hexadecimal and signed/unsigned decimal with various field widths and value layouts such as with leading zeros or spaces. These are integer operations. These days you have to be able to deal with values longer than 32 bits. The printf syntax for 64 bit values isn’t necessary in hexdump. You can just specify that you are working with a 64 bit (8 byte) value and what output format you want. For example, if the LBA values happened to be a single 64 bit you could do this to see them in hexadecimal – note that this is no longer a valid interpretation of a partition table:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3u" 1/1 " %02x" 3/1 " %3u"
```

**1/8** “

```
 %x" "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255 ffead50000003f
00   0   0   0 00   0   0   0 0
00   0   0   0 00   0   0   0 0
00   0   0   0 00   0   0   0 0
```

Note the value ffead50000003f at the end was formatted as a single 8 byte value (1/8) and output in hexadecimal %x. You could use %16x to preserve the numeric column alignment:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3u" 1/1 " %02x" 3/1 " %3u" 1/8 "
```

**%16x**”

```
 "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255   ffead50000003f
```

```
00   0   0   0 00   0   0   0                0
00   0   0   0 00   0   0   0                0
00   0   0   0 00   0   0   0                0
```

And you can include leading zeros:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3u" 1/1 " %02x" 3/1 " %3u" 1/8 " %016x" "\n"'
diskdump.ddimg
00   1   1   0 83 254 255 255 00ffead50000003f
00   0   0   0 00   0   0   0 0000000000000000
00   0   0   0 00   0   0   0 0000000000000000
00   0   0   0 00   0   0   0 0000000000000000
```

And you can use signed decimal instead of hexadecimal:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3u" 1/1 " %02x" 3/1 " %3u" 1/8 "
```

**%20d”**

```
 "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255    72034319610150975
00   0   0   0 00   0   0   0                    0
00   0   0   0 00   0   0   0                    0
00   0   0   0 00   0   0   0                    0
```

And you can use unsigned decimal instead of hexadecimal:

```
# hexdump -s 446 -n 64 -v -e '1/1 "%02x" 3/1 "
%3u" 1/1 " %02x" 3/1 " %3u" 1/8 "
```

**%20u”**

```
 "\n"' diskdump.ddimg
00   1   1   0 83 254 255 255    72034319610150975
00   0   0   0 00   0   0   0                    0
00   0   0   0 00   0   0   0                    0
00   0   0   0 00   0   0   0                    0
```

**Text in a file**

Not all values in a binary file are numeric, the file may also contain text. hexdump provides a way to show that too. Unlike the C printf family of functions, you have to know in advance how long the text is and there are many ways to achieve the same thing. Going back to the ELF executable file that I started with, you could just dump the signature as-is:

```
# hexdump
```

**-s 1 -n 3**

```
 -v -e
```

**‘1/3 “%s” “\n”‘**

```
 mydata
ELF
```

You get the same effect with every one of these:

```
# hexdump -s 1 -n 3 -v -e '3/1
```

**“%c”**

```
 "\n"'  mydata
ELF
# hexdump -s 1 -n 3 -v -e '3/1
```

**“%_p”**

```
 "\n"'  mydata
ELF
# hexdump -s 1 -n 3 -v -e '3/1
```

**“%_c”**

```
 "\n"'  mydata
ELF
# hexdump -s 1 -n 3 -v -e '3/1
```

**“%_u”**

```
 "\n"'  mydata
ELF
```

There are many other examples that will output the same three characters but with different arrangements of line feeds. The _p, _c and _u examples are specific to hexdump and not part of the printf syntax. The _p causes characters that have no standard glyph to be output as a . – we can see that by going a couple of bytes beyond the F in the ELF signature:

```
# hexdump -s 1 -n 5 -v -e '5/1 "%c" "\n"'  mydata
ELF
# hexdump -s 1 -n 5 -v -e '5/1 "%_p" "\n"'  mydata
ELF..
```

Note that the first one, using the printf standard %c displayed nothing for the two characters following the ELF but the second one displayed a dot for each character after the ELF. That helps with alignment and it indicates when the data changes from text (printing characters) to non-text (non-printing characters).

The _c example outputs the text in the host's default character set and displays non-printing characters by value in octal (base 8):

```
# hexdump -s 1 -n 5 -v -e '5/1 "%_c" "\n"' mydata
ELF002001
```

So, the dots after the ELF are the byte values 2 and 1 (the octal is output here as three digits).

The _u example outputs the text in US ASCII with control characters (those with decimal byte values less than 32 and the value 255) displayed by their three-character control name:

```
# hexdump -s 1 -n 5 -v -e '5/1 "%_u" "\n"' mydata
ELFstxsoh
```

That's a Start of TeXt control character followed by a Start Of Heading control character. In this file the byte values don't represent those ASCII control characters but you can see how you could use %s, %_p, %_c and %_u as appropriate for different types of input files.

**Files with text and numeric data**

Just as we combined 8, 32 and 64 bit %x output with each other and with 8, 32 and 64 bit decimal values earlier, we can combine numeric and text values on the same command line:

```
# hexdump -n 6 -v -e
```

**‘1/1 “%02x ” 3/1 “%_u” 2/1 ” %02x”**

```
 "\n"' mydata
7f ELF 02 01
```

I removed the -s 1 so that you could see the text in between some other types of data.

**Numbers with a fractional part**

In the programming world they are known as floating point numbers. hexdump can display them but there is no guarantee that a floating point value in a file is in the native floating point format of the host. Nevertheless. If you knew there was a floating point value 5213 bytes into that ELF file (which there isn’t in reality):

```
# hexdump -s 5213
```

**-n 8 -v -e ‘1/8 “%f “**

```
 "\n"' mydata
311143424.000000
```

Once again you can combine it with other format options to look at sequential elements of the file that have different formats. The default length of the %f conversion is 8 bytes so you can actually drop the 1/8 from the example because there are no other format values and the dump length is 8. If the number has too many digits then you can show it in exponential format:

```
hexdump -s 5295 -n 8 -v -e '1/8
```

**“%g”**

```
 "\n"' mydata
1.07497e-255
```

If you prefer the e to be uppercase you can use %G:

```
hexdump -s 5295 -n 8 -v -e '1/8
```

**“%G”**

```
 "\n"' mydata
```

```
1.07497E-255
```

All three of these floating point formats can also operate on 4 byte floating point types:

```
hexdump -s 5255 -n 8 -v -e '1/4 "%G" "\n"' mydata
1.67511E-10
1.07861E-38
```

**I no longer know where I am in the file**

When you decode a large number of elements in a file you can end up with so much output that you can't tell where in the file a given value is. In the first examples you saw hexdump output the offset into the file in the first column:

```
# hexdump mydata
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000010 0002 0003 0001 0000 8430 0804 0034 0000
```

You can add the same to your own decoding adding a _a prefix to any of the integer numeric format types: x, d, o. If the sample file had 8 floating point values 4091 bytes into the file and I wanted to know the location of each one:

```
# hexdump -s 4091 -n 64 -v -e '
```

**"%_ax** | "

```
 1/8 "%G" "\n"' mydata
ffb | 2.56934
1003 | 2.57715
100b | 2.58496
1013 | 2.59277
101b | 0
1023 | 0
102b | 1.29478E+16
1033 | 9.34901E+25
```

I included the bar so that you could see the problem when the value overflows to add a new digit. We can use the same prefixes to %_ax and %_ad as we could to the normal %x and %d:

```
# hexdump -s 4091 -n 64 -v -e '
```

**“%08_ax** | “

```
 1/8 "%G" "\n"' mydata
00000ffb | 2.56934
00001003 | 2.57715
0000100b | 2.58496
00001013 | 2.59277
0000101b | 0
00001023 | 0
0000102b | 1.29478E+16
00001033 | 9.34901E+25
```

Decimal is probably more useful for human readability:

```
# hexdump -s 4091 -n 64 -v -e '
```

**“%9_ad** | “

```
 1/8 "%G" "\n"' mydata
     4091 | 2.56934
     4099 | 2.57715
     4107 | 2.58496
     4115 | 2.59277
     4123 | 0
     4131 | 0
     4139 | 1.29478E+16
     4147 | 9.34901E+25
```

There’s a special version of the file-position option that will show the position of the next byte in the file after the last value output, _A:

```
hexdump -s 4091 -n 64 -v -e '"%9_ad | " 1/8 "%G"
"\n"
```

**“%_Ad \n**”

```
' mydata
     4091 | 2.56934
     4099 | 2.57715
```

```
4107 | 2.58496
4115 | 2.59277
4123 | 0
4131 | 0
4139 | 1.29478E+16
4147 | 9.34901E+25
 4155
```

The 4155 is the effect of the %_Ad, the extra indent was deliberate. The _A option can be useful when you are building output from multiple hexdump command lines and want to know what -s value to use on the next command without having to calculate it.

**Custom field widths**

When specifying the field widths after a % you can use another syntax to describe the overall width to output the value in and how many digits of output to include. For example, the following are equivalent:

```
# hexdump -s 5091 -n 32 -v -e '1/4 "%08x" "\n"'
mydata
2e697472
752f0053
732f7273
702f6372
616b6361
2f736567
4c495542
6c672f44
# hexdump -s 5091 -n 32 -v -e '1/4
```

**“%8.8x”**

```
 "\n"' mydata
2e697472
752f0053
732f7273
702f6372
```

```
616b6361
2f736567
4c495542
6c672f44
```

In the second one I replaced %08x with %8.8x. In the second syntax, the number before the dot is the overall width for the field and the number after the dot is the number of digits of output to generate. The value is output right justified if the first number is larger than the second. So, we could have the 8 digit hex value in a 10 character field:

```
# hexdump -s 5299 -n 32 -v -e '1/4
```

**"%10.8x"**

```
 "\n"' mydata
  0b002402
  030b3e0b
  0300000e
  0b0b0024
  08030b3e
  24040000
  3e0b0b00
  0500000b
```

Whatever the value of the first or second number, the whole length implied by the input length and the format character is output, so you can't fool hexdump with %0.0x:

```
# hexdump -s 5299 -n 32 -v -e '1/4 "%0.0x" "\n"'
mydata
b002402
30b3e0b
300000e
b0b0024
8030b3e
24040000
3e0b0b00
500000b
```

The values were output in full and left justified.

**Putting it all together**

The following is a really big example that uses lots of hexdump commands with shell variables, arithmetic and logic operations to dump a packet from a tcpdump file down to the tcp header – inserting the correct offset for some other tcpdump file is left as an exercise for the reader:

```
#!/bin/bash
# Decode a UDP packet raw binary dataDone

let offs=40
declare -i ws
declare -i end
declare -i rmndr
declare -i byteval
declare -i bitws

NetworkOrder16Bit() {
        ws=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
        ((offs+=1))
        ((ws*=256))
        byteval=`hexdump -s $offs -n 1 -e '1/1
"%u"' test2.pkts`
        ((offs+=1))
        ((ws+=byteval))
}

NetworkOrder32Bit() {
        ws=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
        ((offs+=1))
        ((ws*=256))
        byteval=`hexdump -s $offs -n 1 -e '1/1
"%u"' test2.pkts`
        ((offs+=1))
        ((ws+=byteval))
        ((ws*=256))
        byteval=`hexdump -s $offs -n 1 -e '1/1
"%u"' test2.pkts`
        ((offs+=1))
        ((ws+=byteval))
        ((ws*=256))
        byteval=`hexdump -s $offs -n 1 -e '1/1
"%u"' test2.pkts`
        ((offs+=1))
        ((ws+=byteval))
}

echo
echo

end=$offs+74

#Start with the destination MAC address
hexdump -s $offs -n 6 -e '"Ether Dest:        "
5/1 "%02x:" 1/1 "%02x\n"' test2.pkts
((offs+=6))

# Then the source
hexdump -s $offs -n 6 -e '"Ether Src:         "
5/1 "%02x:" 1/1 "%02x\n"' test2.pkts
((offs+=6))

# Next is the packet type, but its in network byte
order so we can't
# use '1/2 "%04x"' - we may be run on a system
with another byte order and you'll
# get different output
hexdump -s $offs -n 2 -e '"Ether Type:        "
```

```
2/1 "%02x" "\n"' test2.pkts
((offs+=2))

# IP header

# Get the version/len value into a variable we can
do arithmetic with
byteval=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
((offs+=1))

# The most significant 4 bits are the IP version

((bitws=byteval&0xF0))
((bitws/=16))
echo "IP Version:       $bitws"

# Least significant 4 bits are the header length
((bitws=byteval&0x0F))
((bitws*=4))
echo "Header Length:    $bitws"

# Type of service
byteval=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
((offs+=1))

# Precedence
((bitws=byteval&0xE0))
((bitws/=32))
echo "Precedence:       $bitws"

# Delay
((bitws=byteval&0x10))
if (( bitws ))
then
```

```
        echo "Minimize delay"
fi

# Throughput
((bitws=byteval&0x08))
if (( bitws ))
then
        echo "High throughput"
fi

# Reliability
((bitws=byteval&0x08))
if (( bitws ))
then
        echo "High reliability"
fi

# Cost
((bitws=byteval&0x08))
if (( bitws ))
then
        echo "Minimize cost"
fi

# Message length - in network byte order
NetworkOrder16Bit
echo "Message Length:    $ws"

# ID
hexdump -s $offs -n 2 -e '"ID:                "
2/1 "%02x" "\n"' test2.pkts
((offs+=2))

# Flags
byteval=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
```

```
((offs+=1))

# DF
((bitws=byteval&0x40))
if (( bitws != 0))
then
        echo "Don't fragment"
else
        # More fragments
        ((bitws=byteval&0x20))
        if (( bitws ))
        then
                echo "More fragments"
        fi

        # Fragment number
        ((ws=bitws&0x1F))
        ((ws*=256))
        byteval=`hexdump -s $offs -n 1 -e '1/1
"%u"' test2.pkts`
        ((ws+=byteval))
        echo "Fragment number:    $ws"
fi
((offs+=1))

hexdump -s $offs -n 1 -e '"TTL:               "
1/1 "%u\n"' test2.pkts
((offs+=1))

hexdump -s $offs -n 1 -e '"Protocol:          "
1/1 "%u\n"' test2.pkts
((offs+=1))

hexdump -s $offs -n 2 -e '"Header checksum:   "
2/1 "%02x" "\n"' test2.pkts
((offs+=2))
```

```
hexdump -s $offs -n 4 -e '"Source IP Addr:    "
3/1 "%02u." 1/1 "%02u\n"' test2.pkts
((offs+=4))

hexdump -s $offs -n 4 -e '"Dest IP Addr:      "
3/1 "%02u." 1/1 "%02u\n"' test2.pkts
((offs+=4))

# TCP header
NetworkOrder16Bit
echo "Source Port:       $ws"

NetworkOrder16Bit
echo "Dest Port:         $ws"

hexdump -s $offs -n 4 -e '"Sequence Number:   "
4/1 "%02x" "\n"' test2.pkts
((offs+=4))

hexdump -s $offs -n 4 -e '"ACK Number:        "
4/1 "%02x" "\n"' test2.pkts
((offs+=4))

ws=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
((offs+=1))
((ws/=16))
((ws*=4))
echo "Header Length:     $ws"

byteval=`hexdump -s $offs -n 1 -e '1/1 "%u"'
test2.pkts`
((offs+=1))
((bitws=byteval&0x80))
echo -n "Flags:             "
```

```
if (( bitws ))
then
        echo -n "CWR "
fi
((bitws=byteval&0x80))
if (( bitws ))
then
        echo -n "CWR "
fi
((bitws=byteval&0x40))
if (( bitws ))
then
        echo -n "ECN "
fi
((bitws=byteval&0x20))
if (( bitws ))
then
        echo -n "URG "
fi
((bitws=byteval&0x10))
if (( bitws ))
then
        echo -n "ACK "
fi
((bitws=byteval&0x08))
if (( bitws ))
then
        echo -n "PSH "
fi
((bitws=byteval&0x04))
if (( bitws ))
then
        echo -n "RES "
fi
((bitws=byteval&0x02))
if (( bitws ))
```

```
then
        echo -n "SYN "
fi

((bitws=byteval&0x01))
if (( bitws ))
then
        echo -n "FIN "
fi
echo

NetworkOrder16Bit
echo "Window Size:       $ws"

hexdump -s $offs -n 2 -e '"Checksum:          "
2/1 "%02x" "\n"' test2.pkts
((offs+=1))

# Some more of the packet
echo
echo
hexdump -s $offs -n 128 -C test2.pkts
echo
echo
```

That shouldn't be considered as a viable packet analyzer but it shows most of the elements necessary to build a custom tool to investigate some arbitrary binary data. Hopefully there's something in that you will find useful when applying hexdump to your own binary file decode problems.

(Visited 1 times, 1 visits today)