We all know the phrase “man is the tool using animal” (my wife claims “man is the tool buying animal”). We all tend to use the tools we know best for lots of purposes, whether they are actually the ideal tool or not.

I generally figure that tool overkill is not a problem. When I don’t have a serial port analyzer handy, I’ve been known to debug RS-232 problems with an oscilloscope.

When I recently had a need to observe the HTML data flow between the browser and server, I started out with `tcpdump`. I’ve used `tcpdump` in the past to debug networking glitches, and I’m fairly familiar with it. Then I started trying to decipher the output.

Of course, the right tool for any job is Tcl, so that’s what I used to extract the information I wanted from the `tcpdump` output.

The `tcpdump` program is fairly easy to install on SunOS and Solaris boxes and comes standard with most Linux distributions these days. On a SunOS system, you may need to play some driver games and run from root to use `tcpdump`, but it can be done.

There are a number of options you can use to customize `tcpdump` behavior. My favorites are:

- `-i interface` The name of the interface to watch.
- `-s size` The number of bytes of data to display as hex.
- `-l` Use line oriented buffering, rather than normal buffered I/O. Just because I’m not patient.
- `-n` Don’t convert addresses to names.
- `-v` Be verbose. Include time-to-live and type-of-service information.
- `-x` Print each packet in hex.

The `-x` option makes a lot of applications possible. Being able to examine an entire data packet is a powerful tool.

The `tcpdump` output with this set of options resembles:

```
ack 329 win 8432 (DF) (ttl 128, id 31790)
  453c 2f7 4 683e 0d0a 3c7 4 6820 616c 6967
  ...
  2045 450d 0a3c 2f74 683e
```

The first two lines in this display are actually a single line in the `tcpdump` output, broken into two lines to fit better on the printed page.

The first line tells us that the packet was transferred at 11:20 p.m. from 192.168.9.63 (port 80) to 192.168.9.2 (port 1823). The packet had the “PSH” flag set in the TCP header, and includes 226 bytes of data.

The rest of the lines are the hex data for the TCP/IP packet including the IP header, TCP header and data.

The first easy step is to separate the packet information lines from the packet data lines.

The data format has a number of features we could use to distinguish information lines from data lines. For example, the first character for an info line is always a number, while the first character of a data line is always a whitespace character.

---

**by Clif Flynt**

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of `Tcl/Tk for Real Programmers` and the `TclTutor` instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

[clif@cflynt.com](mailto:clif@cflynt.com)
However, the first character of an info line can be any number, and I'm not sure which whitespace character (space or tab) is used for data, so testing on the first character would mean checking for one of 10 possible digits, or one of two possible whitespace characters.

The third character in an information line is always a ":". Checking for the ":" is a simple test. We can use the string first command to find the first colon in a line. If there is no colon, then string first will return a -1.

The Tcl code to read the data from an input channel and check for data or info lines looks like this:

```tcl
while {set len [gets $input line] = 0} {
    # A colon in position 2 means a header line xx:yy:zz.abc
    if {string first ":" $line} == 2} {
        # Found info line
    } else {
        # Found data line
        append hexData [string trim $line]
    }
}
```

The `gets` command will return the number of characters it reads from a channel. If there is a failure (like hitting the end of the file), it will return -1.

The `string trim` command will trim whitespace away from the left and right ends of a text string. This gets rid of the leading spaces. The whitespace was useful while parsing the input, but we don't need it anymore.

Whenever the script recognizes an information line, it knows that whatever is in the `hexData` variable is hex data for a complete packet, and this can be processed.

So, the next trick is to decipher that hex data and pull out the HTML page as easy-to-read ASCII text.

I like thinking of data as bytes, rather than 16-bit shorts, or 32-bit words. So, the first step is to convert the data from a string of shorts to a string of bytes.

Rather than think of the data as a string of numbers and spaces, it makes sense to think of it as a list of numbers separated by whitespace. This leads to thinking of list commands to reformat the data, instead of regular expression or string-based solutions.

Tcl has a couple of commands for converting data from strings to lists.

**Syntax: join list ?separatorCharacter?**

- `join` Converts a list into a character-delimited string.
- `list` The list of elements to join into a string.
- `?separatorCharacter` A character to place between each element.

The `join` command is very useful for converting a Tcl list into a character-delimited string to export to some other program. You can use the `join` command to create a comma-delimited string to export to an Excel or SC spreadsheet program, for instance.

If we declare the separator to be no character (a ""'), Tcl will strip all the spaces from a list.
This code will convert the data string of space-delimited words to a long string of hex digits.

```tcl
set hexData [join $hexData ""]
```

The next step is to convert the block of hex digits into a list of bytes.

The Tcl `split` command will split string data into a list.

**Syntax:** `split string ?splitChars?`

- `split` Split a string into a list. Elements are delimited by a marker character.
- `string` The string to split.
- `?splitChars?` A string of characters to use to mark elements. By default the markers are whitespace characters (tab, new line, space, carriage return).

The `split` command is often used to load data that was exported as a comma-delimited list from some other program like `sc` or `Excel`.

We can use the `split` command to split a string at each character by setting the `splitChar` to an empty string.

This code will take the mass of hex digits and convert it into a list of hex bytes.

```tcl
foreach {h l} [split $hex ""] {
    lappend bytes $h$l
}
```

The packet consists of an IP header, a TCP header, and then the data. Our script should skip the header information and just process the data packet.

A TCP header is always 20 bytes long, but an IP header can vary in size.

The IP Header starts with two nibbles:

- **Identifier** The version number of the IP packet. Most commonly this is “4” for IP version 4. We’ll be seeing “6” in this field more often as systems move to IPv6.
- **length** The number of 32-bit words in the IP header.

We can extract the IP header length with the string range and `lindex` commands.

**Syntax:** `lindex list position`

- `lindex` Return the list element at `position`.
- `list` A list of to extract an element from.
- `position` The position of the element to extract.

The first element of the hex bytes is the first byte of the IP header. It can be extracted as: `[lindex $bytes 0]`, since Tcl lists and strings are zero based.

We could use the same `split` command to split the hex byte into nibbles, but it’s a bit easier to extract one character with the `string range` command.

**Syntax:** `string range string startPosition endPosition`

- `string range` Return a subset of characters from a string.
- `string` The string to extract a subset of characters from.
- `startPosition` The position of the first character to extract.
- `endPosition` The position of the last character to extract.
The string range is zero based and uses inclusive selection, so the command to get the second character from the byte is:

```tcl
set headerLen [string range [lindex $bytes 0] 1 1]
```

This header length is in 32-bit words, not bytes. The code to find the position of the first data value is:

```tcl
set pos [expr ($headerLen * 4) + 20]
```

Now to convert the hex values to ASCII characters. As usual, there are several ways to solve the problem.

For instance, we could use an associative array as a lookup table with code like:

```tcl
array set hex2ascii {
   ... 41 A
   42 B
   43 C
   ...
}
```

```tcl
foreach byte $bytes {
   append str $hex2ascii($byte)
}
```

This would work, but gets a bit large.

The Tcl format command works much like the C sprintf command, and makes a shorter and simpler solution.

**Syntax:** `format formatString value1 ?value2?...`

```tcl
format Return a new string formatted as defined by the formatString.
```

**formatString** A string that defines how to format the following data units. This string uses the same conventions as printf.

- `%i` The value is an integer to be formatted as a decimal integer.
- `%f` The value is a floating point number to be formatted as xx.yy.
- `%c` The value is character data to be formatted as a string.

**valueX** Values to be used in the return string. There must be one value for each “%” field in the format string.

The format command is most often used to generate tabular data with something like:

```tcl
foreach {name address phone} $addressBook {
   puts [format "%20s %30s %10s", $name $address $phone]
}
```

We can use the `%c` option to convert the hex bytes to ASCII characters.

```tcl
foreach b [lrange $bytes $pos end] {
   append str [format %c 0x$b]
}
```

Note the `0x$b`. In Tcl (like C) a decimal number starts with a digit between 1 and 9. If a number starts with “0”, it is considered to be an octal value, and if it starts with “0x”, the number is treated like a hexadecimal value.
Code like this:

```tcl
set hex 0x41
puts $hex
```

would print `0x41`, since `puts` is expecting a text string, and there's no need to consider `0x41` to be anything but a text string.

However, code like:

```tcl
set hex 0x41
puts [expr $hex + 2]
```

will print `67`. The `expr` command expects a number, and will interpret `0x41` as an integer (decimal `65`), add two, and return the result as a decimal value.

The `format` command expects a numeric argument for the `%c` format specifier. When Tcl interpreter encounters the `0x$b` it substitutes the current value for `$b`, creating an ASCII string like “0x41”. The `format` command will interpret “0x41” as a hexadecimal value, and then convert the binary value to a printable ASCII character.

This code will convert the `tcpdump` hex dumps to printable ASCII.

```tcl
proc hex2Text {hex} {
    set hex [join $hex ""]
    foreach {h l} [split $hex ""] {
        lappend bytes "$h$l"
    }
    set headerLen [string range [lindex $bytes 0] 1 1]
    # The IP header length is the second nibble.
    # The TCP header is 20 bytes.
    set pos [expr ($headerLen * 4) + 20]
    foreach b [range $bytes $pos end] {
        append str [format %c 0x$b]
    }
    return "$str"
}
```

The default `tcpdump` output includes every packet seen passing by an interface. This includes a lot of packets that I’m not interested in.

The `tcpdump` program has support for filtering the output by fields like port, IP address, type of packet, etc., which is great if you know just which packets you want to examine.

Sometimes I find that I want to grab all the data for a few minutes, save it in a file, and then examine different subsets of the data. This means that I need to filter out the unwanted data in my script.

The values I usually filter on are the source and/or destination IP address, source and/or destination ports, TCP flag and length of packet.

Most of this information can be extracted from the `tcpdump` info line with the Tcl `regexp` command, which was discussed in great and tedious detail in a Tclsh Spot arti-
The basic form of the `regexp` command is:

```
```

Since a regular expression can include symbols that have meaning to the Tcl interpreter, we commonly put the `expression` within curly braces to disable any Tcl special character processing.

A simple `regexp` command might resemble:

```
set string {Regular expressions are useful and powerful}
regexp {R.*r +(e.*s) } $string all e1
```

The regular expression says to look for a pattern that starts with uppercase `R`, has 0 or more undefined characters, a lowercase `r` followed by one or more spaces, followed by lowercase `e`, 0 or more other characters, and finally a lowercase `s` followed by a space.

The parenthesis around the `e.*s` tells the `regexp` command to extract the part of the string that matches this part of the pattern and save that in the second variable.

When this command is run, it will match the expression pattern to the string `Regular expressions`. The entire matching string will be placed in the variable `all`, and the string "expressions" will be placed in the variable `e1`.

We can look for an IP address with a pattern like: `{([0-9]+.[0-9]+.[0-9]+.[0-9]+)}`. This pattern calls for one or more digits followed by a period, followed by one or more digits, etc. The periods need to be escaped with a backslash because the period has meaning to the regular expression parser – the period means any character. If we left out the backslash, then any collection of numbers would match the pattern.

We could match two IP addresses by putting two copies of that string (with appropriate other values to match the rest of the string) in a regular expression pattern.

In this case, the pattern starts to get very long, very quickly.

Another trick is to put the pieces of the regular expression into a set of variables, and then build the expression from those:

```
set port {([^ ]+)}
set sep { <[ ]>}
set flag {([^ ])}
```

This code will look for the IP address, port, and flag information in a `tcpdump` info line and parse the values into five variables. The string of data from `tcpdump` is in the variable `oldLine`.

```
set m [regexp "$addr$port$sep$addr$port $flag" $oldLine all srcIP srcPORT destIP destPORT TcpFlag]
```


Once we’ve extracted that information, we can decide whether or not the data associated with this line is interesting.
This is an obvious place for an if command. The Tcl if command supports either a simple boolean expressions like $i < 10 or complex expressions like ($i < 10) && ($i > 4).

Since the boolean expression goes through the normal Tcl substitution phase, any Tcl command that returns a value can be used as part of the boolean expression.

We could check that a packet starts with the number 4 (the first nibble defines this to be IPv4) with code like:

```tcl
if {\[string range $bytes 0 0\] == 4} {...}
```

We could check that one string matches another with one of several string commands like `string compare` (which returns whether one string is greater or less than another, like the C library `strcmp` function) or `string first` (that returns the first occurrence of one string within another). The `string match` command gives us the most power for this application.

**Syntax:** `string match pattern string`

- `string match` Returns a TRUE or FALSE depending on whether the pattern matches the string it is being compared to.
- `pattern` A glob style pattern to compare to a string.
- `string` The string to compare the pattern with.

A go/no go decision on each set of hex data is made with this code:

```tcl
if {\[string length $hex\] $packetLen} &&
   ([string match 4 [string range [string trimleft $hex 0 0]]] &&
   ([string match $destIPpattern $destIP ] &&
   ([string match $destPORTpattern $destPORT ] &&
   ([string match $srcIPpattern $srcIP ] &&
   ([string match $srcPORTpattern $srcPORT ] &&
   ([string match $flagPattern $TcpFlag ]){
    set txt [hex2Text $hex]
    puts "\n$txt
———————————-\n";
}
```

This if statement checks that

1. The data packet is at least $packetLen bytes long.
2. The data is an IPv4 packet (or looks a lot like one).
3. The destination IP address matches the requested destination pattern.
4. The destination port matches the requested destination pattern.
5. The source IP address matches the requested source pattern.
6. The source port matches the requested source pattern.
7. The TCP flag matches the requested flag pattern.

The patterns could be hardcoded, but our code will be more configurable if the patterns are saved in a set of variables.

The body of this application looks like this:

```tcl
# Define the patterns
set packetLen 100
set flagPattern "P"
set destIPpattern "192.168.9.63"
set srcIPpattern "192.168.9.2"
set srcPORTpattern "*"
set destPORTpattern "www:"
```
set input stdin

# Initialize variables.
set hex ""
set oldLine ""
set TcpFlag ""
set destPORT ""

# Define the parts of the regular expression
set port {{[^ ]+}}
set sep {<|>|+}
set flag {{[^ ]+}}

# Read lines of data and process as necessary.
while {set len [gets $input line] = 0} {
    if {$len < 2} continue
    if {[string first ":" $line] == 2} {
        set m [regexp "$addr$port$sep$addr$port +$flag" \
               $oldLine all srcIP srcPORT destIP destPORT TcpFlag]
        if {([string length $hex] $packetLen) &&
           ([string match 4 [string trimleft $hex] 0 0]]) &&
           ([string match $destIPpattern $destIP] ) &&
           ([string match $srcPORTpattern $srcPORT] ) &&
           ([string match $flagPattern $TcpFlag] )} {
            set txt [hex2Text $hex]
            puts "$txt
———————————-
"
        }
        set oldLine $line
        set hex ""
    } else {
        append hex [string trim $line]
    }
}

This code takes output from tcpdump and prints output that resembles this:

POST /cgi-bin/login.cgi HTTP/1.0
Accept: */*
Host: 192.168.9.63
UserAgent: Tcl http client package 2.3
Content-Type: application/x-www-form-urlencoded
Content-Length: 49
submitButton=Login&password=PASSWORD&loginName=loginID

As usual, the code described in this article is available at http://www.noucorp.com.