The philosophy that drove the design of most of the Unix system data files is that it's best to sacrifice a little speed in favor of ease of maintenance. Thus, most data files (`/etc/password, /etc/hosts,` etc)contain printable ascii data. If you need to examine them you can do it with `more`, and if you need to repair them, you can do it with `vi`.

There are some files that contain binary data: the `utmp` file for example. The `utmp` file contains data about all the users on a system (assuming everything is working correctly), including their login id, the time they logged in, the IP address they logged in from, etc.

It might be nice to watch this file, and report when it changes. For example, if there is suddenly someone running as root on our firewall, it might be something we are interested in knowing about.

Early versions of Tcl supported only ASCII strings, and could not handle binary data. With version 8.0, Tcl moved to a more versatile internal data representation and added the `binary` command. The binary command allows easy conversion from binary representations to printable ASCII representations of data. Tcl is still oriented around printable ASCII strings but the `binary` command makes it possible to handle binary data as well.

The binary command has two subcommands, `binary scan`, and `binary format`

The syntax for these is:

Syntax: `binary format` *`formatString arg1 ?arg2? ... ?argn?`*

`binary format`	Returns a binary string created by convert one or more printable ascii strings to a binary format.
`formatString`	A string that describes the format of the ascii data.
`arg*`	The printable ascii to convert

Syntax: `binary scan` *`binaryString formatString arg1 ?varName1? ... ?varNamen?`*

`binary scan`	Converts a binary string to one or more printable ascii strings
`binaryString`	The binary data.
`formatString`	A string that describes the format of the ascii data.
`varName*`	Names of variables to accept the printable representation of the binary data

The *`formatString`* for the binary commands allows binary data to be collected from or distributed to a number of variables in a variety of formats. It slightly resembles a regular expression string, but has a slightly different flavor.

Like the regular expression string, a `binary` command format string is composed of sets of two fields - a descriptor for the type of data, followed by an optional count modifier.

There are several identifiers defined for this command that support converting strings, decimal data, hex data, or floating point values by 8-bit, 16-bit or 32-bit data widths. Here are a few of the commonly used descriptors:

h Converts from binary to/from hexadecimal digits in little-endian order.
`binary format h2 34` - returns "C" (0x43).
`binary scan "4" h2 x` - stores 0x43 in the variable `x`

H Converts from binary to/from Hexadecimal digits in big-endian order.
`binary format H2 34` - returns "4" (0x34).
`binary scan "4" H2 x` - stores 0x34 in the variable `x`

c Converts an 8 bit value to/from ASCII.
`binary format c 0x34` - returns "4" (0x34).
`binary scan "4" c x` - stores 0x34 in the variable `x`

s Converts a 16 bit value to/from ASCII in little-endian order.
`binary format s 0x3435` - returns "54" (0x350x34).
`binary scan "45" s x` - stores 13620 (0x3534) in the variable `x`

S Converts a 16 bit value to/from ASCII in big-endian order.
`binary format S 0x3435` - returns "45" (0x350x34).
`binary scan "45" S x` - stores 13365 (0x3435) in the variable `x`

i Converts a 32 bit value to/from ASCII in little-endian order.
`binary format i 0x34353637` - returns "7654" (0x350x34).
`binary scan "45" s x` - stores 13620 (0x3534) in the variable `x`

I Converts a 32 bit value to/from ASCII in big-endian order.
`binary format I 0x34353637` - returns "4567" (0x350x34).
`binary scan "45" S x` - stores 13365 (0x3435) in the variable `x`

f Converts 32 bit floating point values to/from ASCII.
`binary format f 1.0` - returns the binary string "0x0000803f" .
`binary scan "0x0000083f" f x` - stores 1.0 in the variable `x`

The optional count can be an integer, to list the exact number of conversions to perform, or a *, to use all remaining data.

The format string can be arbitrarily complex, with multiple descriptor/count pairs separated by spaces.

Here's an example of some C code to write a structure to a disk file, and the Tcl code to read and translate the data:

C Code to generate a structure

```
#include
#include
main () {
    struct a {
       int i;
       float f[2];
       char s[20];
       } aa;

       FILE *of;

       aa.i = 100;
```

Tcl Code to read the structure

```
# Open the input file, and read data
set if [open tstStruct r]
set d [read $if]
close $if

# scan the binary data into variables.

binary scan $d "i f2 a*" i f s

# The string data includes any binary garbage
#   after the NULL byte.
# Strip off that junk.
```

```
        aa.f[0] = 2.5;
        aa.f[1] = 3.8;
        strcpy(aa.s, "This is a test");

        of = fopen("tstStruct", "w");
        fwrite(&aa, sizeof(aa), 1, of);
        fclose(of);
}
```

```
set 0pos [string first [binary format c 0x00]
incr 0pos -1
set s [string range $s 0 $0pos]

# Display the results
puts $i
puts $f
puts $s
```

The output from the Tcl code is:

```
100
2.5 3.79999995232
This is a test
```

The flip side to this is to write a structure in Tcl, and read it with a C program. This pair of programs will perform that operation.

Tcl Code to generate a structure **C**

```
set str [binary format "i f2 a20" 100 {23.4 56.78} "the other test"]

set if [open tstStruct2 w]
puts -nonewline $if $str
close $if
```

```
#include
#include
main () {
    struct a {
       int i;
       float f[
       char s[2
       } aa;

       FILE *of

       of = fop
       fread(&a
       fclose(c

       printf("
           aa.i
}
```

The C program generates this output:

```
I: 100
f[0]: 23.400000 f[1]: 56.779999
str: the other test
```

The `binary` command makes it (relatively) easy to parse the `utmp` file. All we need to do is look up the `utmp.h` include file on our system, examine the structure definition, and create a format string that the `binary` command can use to parse each structure in the `utmp` file.

On a Linux system, the utmp structure looks like this:

```
/* The structure describing an entry in the user accounting database.  */
struct utmp
{
  short int ut_type;            /* Type of login.  */
  pid_t ut_pid;                 /* Process ID of login process.  */
  char ut_line[UT_LINESIZE];    /* Devicename.  */
  char ut_id[4];                /* Inittab ID.  */
  char ut_user[UT_NAMESIZE];    /* Username.  */
  char ut_host[UT_HOSTSIZE];    /* Hostname for remote login.  */
  struct exit_status ut_exit;   /* Exit status of a process marked
                                   as DEAD_PROCESS.  */
  long int ut_session;          /* Session ID, used for windowing.  */
  struct timeval ut_tv;         /* Time entry was made.  */
  int32_t ut_addr_v6[4];        /* Internet address of remote host.  */
  char __unused[20];            /* Reserved for future use.  */
};
```

This would lead you to believe that a format string like this should do the trick for extracting the members of the structure:

```
#    type pid line id user host exit session time addr
set f "s  i   a32  a4 a32  a256 s2   i       i2   i4"
```

In a perfect world, this would work fine.

In this world, certain processors require that an integer start on an integer boundaary, and if a structure declares a single short followed by a long integer, then there will be two bytes of padding added so that the integer can start on a long-word boundary.

The `binary` command includes a type descriptor that is not a data type: the `@` character. Where the other type descriptors accept a count modifier, the `@` descriptor will accept an absolute location in the data as a modifier.

This can be used to set the imaginary cursor in the binary data to a longword boundary, skipping the padding. The `@` can also be used to set the cursor location to the start of each structure in the data set.

The next trick is that while the `utmp` structure is allowing for an IPV6 address, only the first 4 bytes are actually being used today. So, rather than using `i4` for the IP address, we can use `c4` (to read 4 bytes of address).

This format string works for Linux:

```
# set st to the start of the structure
      start type padding      pid line id user host exit session time addr
set f "@$st  s   @[expr $st+4] i  a32  a4 a32  a256 s2   i       i2   c4"
```

This would let us generate a report with all the numbers converted to a printable format. This is better than nothing, (but not a lot better).

The first field is the type of process described in this record. The `utmp.h` file describes the meanings of these types. It’s a relatively easy task to cut and paste from that file, edit a little, and convert the

`#define` lines to a Tcl associative array that we can use as a lookup table to convert the numeric types to a more human friendly value:

```
foreach {name num} {
     EMPTY           0     RUN_LVL         1
     BOOT_TIME       2     NEW_TIME        3
     OLD_TIME        4     INIT_PROCESS    5
     LOGIN_PROCESS   6     USER_PROCESS    7
     DEAD_PROCESS    8     ACCOUNTING      9} {
  set types($num) $name
}
```

The time stamp can be converted to a more human friendly form with the Tcl `clock` command.

The Tcl `clock` command has several subcommands that will obtain the current time (in seconds since the epoch), convert a time in seconds to human readable format, or convert a human style time/date into seconds.

This Tcl command will convert the system format time in the first field of the `timeval` structure into a date and time in the format MM/DD/YY HH:MM:SS:

```
clock format [lindex $time 0] -format "%D %r"
```

Finally, we don't want to report on the contents of the `utmp` file every 10 seconds, or even every minute. We just want to know what's happened if it changes.

The Tcl `file` command also has many subcommands. A useful one for this application is the `file mtime` that reports the last modification time for a file.

We can't set a trigger to go off when a file is modified, but we can loop on the `file mtime` value, and only report the contents of the `utmp` file when the modification time changes.

```
set mtime 0
...
  while {[file mtime /var/run/utmp] == $mtime} {
      after 10000
  }
  set mtime [file mtime /var/run/utmp]
```

The code shown below will generate output resembling this when someone logs in:

```
Type               Pid         Line                 ID User           Host  Exit
DEAD_PROCESS         7                              si                      0 0
BOOT_TIME            0            ~                 ~~ reboot               0 0
RUN_LVL          20019            ~                 ~~ runlevel             0 0
DEAD_PROCESS       157                              l3                      0 0
DEAD_PROCESS       701                              ud                      0 0
USER_PROCESS       702         tty1                  1 lclif                 0 0
LOGIN_PROCESS      703         tty2                  2 LOGIN                 0 0
LOGIN_PROCESS      704         tty3                  3 LOGIN                 0 0
LOGIN_PROCESS      705         tty4                  4 LOGIN                 0 0
```

```
LOGIN_PROCESS       706         tty5                    5 LOGIN                0 0
LOGIN_PROCESS       707         tty6                    6 LOGIN                0 0
USER_PROCESS        746        pts/2                   /2 lclif                0 0
USER_PROCESS        743        pts/1                   /1 lclif                0 0
USER_PROCESS        744        pts/0                   /0 lclif                0 0
USER_PROCESS        745        pts/3                   /3 lclif                0 0
DEAD_PROCESS        999        pts/5                   /5                      0 0
USER_PROCESS       1163        pts/4                   /4 clif           vlad   0 0
```

In the code below, I've assigned the `output` channel to be `stdout` for my testing. Since the first thing a hacker is likely to do after they've broken into your system is rewrite utmp to hide their presence, this monitor might be a good one to combine with the client/server based monitors discussed in the previous articles to push the infomation off the possibly compromised system and onto a (hopefully) more secure (or at least less obvious) machine inside your network.

As usual, this code is available online at `http://noucorp.com`.

```
# Grab the type definitions from /usr/include/bits/utmp.h

foreach {name num} {
         EMPTY           0     RUN_LVL         1
         BOOT_TIME       2     NEW_TIME        3
         OLD_TIME        4     INIT_PROCESS    5
         LOGIN_PROCESS   6     USER_PROCESS    7
         DEAD_PROCESS    8     ACCOUNTING      9 } {
    set types($num) $name
}


# We'll use $zero to trim trailing zeros from the data.
set zero [binary format c 0x00]

# $reportFmt defines the widths of the columns in the report
set reportFmt {%-16s %6s %12s %20s %-12s %6s %5s %7s %21s %16s}

set output stdout

set mtime 0
while {1} {
    while {[file mtime /var/run/utmp] == $mtime} {
        after 10000
    }
    set mtime [file mtime /var/run/utmp]

    # Display a header
    puts [format $reportFmt Type Pid Line ID User Host Exit Session Time Addr]

    # Open read and close the utmp file
    set if [open /var/run/utmp r]
    set d [read $if]
    close $if

    # Save the length of the data buffer for future use.
    set utmpLen [string length $d]

    # This is the start of the utmp structure being examined
    set start 0
```

```
    # As long as there is data to parse, parse it and step to the
    #   next structure.

    while {$start < $utmpLen} {
        # start of struct  type padding          pid line id user host exit\
          session time addr
        set fmt "@$start    s  @[expr $start +4] i   a32  a4 a32  a256 s2  \
          i       i2   c4"
        binary scan $d $fmt type pid line id user host exit session time addr

        # Trim trailing zeros
        foreach v {line id user host} {
            set $v [string trim [set $v] $zero]
        }

        puts $output [format $reportFmt \
          $types($type) $pid $line $id $user $host $exit $session \
          [clock format [lindex $time 0] -format "%D %r"] [join $addr "."]]

        incr start 384
    }
}
```