;login:

THE MAGAZINE OF USENIX & SAGE

October 2002 volume 27 • number 5

inside:

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

generating ethernet packets

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

This article is the first of a series on building network and firewall testing and validation tools using Tcl, open source packages, and some special-purpose hardware. This time I will describe building and testing a Tcl extension for generating Ethernet packets. Subsequent articles will expand on techniques for using this and other extensions.

When I'm building a firewall system I always worry about what I might have missed. Did I install the new security patches in the right places, define the rules correctly, leave no holes?

There are online services like *http://scan.sygatetech.com/* that will scan my system for common flaws, but that requires that I put the system on the Net to test it.

SATAN or SAINT, for example, will check the system for a lot of holes, but they don't test all the firewall rules.

So, I decided to write my own firewall test framework and add new tests as I find and need them.

The first thing I needed for this toolkit was some way to send arbitrary IP packets, to confirm that things like packets on the outside interface with inside addresses are blocked, malformed packets are discarded, and so on.

A little Net searching found a few tools that would almost do what I want. These tools include the Tcl extensions psh from Sun and pkt from USC, and the programs mgen from the Naval Research Laboratory and sendip from Project Purple.

After a little bit of looking, I decided to work with the libdnet library, written by Dug Song (*http://libdnet.sourceforge.net/*). The advantage of this package is that it has the low-level support I need, is currently supported, and has adequate documentation.

The disadvantage is that it's a C library, not a Tcl extension, but that's easily changed.

Tcl was designed to be easily extended. With just a few hours' labor you can pick up a random library and generate the interface code to use it as a Tcl extension. However, this does require some knowledge of how Tcl extensions are constructed. Doing this the first time can take closer to eight hours.

If you don't feel like spending that much time and learning Tcl internals, you can use the SWIG (SoftWare Interface Generator) program to create the interface code for you (*http://www.swig.org/*).

SWIG was developed by David Beazley (*beazley@cs.uchicago.edu*) to make his life easier while he was developing software at Los Alamos. Even in its early forms the program was very useful.

I downloaded the version 1.3.13 for this work. SWIG's built-in support for structures and complex data types is constantly improving. Some details described in this article may be different on the version of SWIG you are using.

SWIG works by examining a definition file that describes the functions and data structures in a library and generating some C code to allow those functions to be loaded into a Perl, Python, Tcl/Tk, Ruby, Guile, or MzScheme interpreter.

Generating a definition file is fairly simple. The basic format is just a list of function declarations.

For example, if you have a file named fibon.c that contains this Fibonacci function:

```
int fib (int i) {
      if (i <= 1) {return 1;}
return fib(i-2) + fib(i-1);
}
```

it could be turned into a Tcl extension with this one-line definition file:

```
$> cat fibon.i
int fib(int i);
```

and this SWIG command line:

```
swig -tcl -module fib -prefix fib -namespace  -v fibon.i
```

The -module fib argument defines the name for this module. The module name can be defined on the command line (as done here) or in the definition file, with the line %module fib.

The -prefix fib argument sets a value that will be used to prevent command name collisions. When used with the -namespace argument, SWIG will generate code to create the new commands in the fib namespace. Placing the extension commands

in a namespace has become the preferred style for Tcl extensions.

Running this SWIG command will create a wrapper file named fibon_wrap.c, which can be compiled into a shared library with a command line resembling this:

```
gcc -shared -L/usr/lib fibon_wrap.c fibon.o -o libfib.so
```

Once this is done, you can load the libfib.so library and use the Fibonacci code in your Tcl scripts just as you would any other Tcl extension.

```
load ./libfib.so
    for {set i 1} {$i &lt; 5} {incr i} {
        puts "The Fibonacci series at level $i is [fib::fib $i]"
    }
```

Unfortunately, most projects are a bit more complex than this.

One problem you run into is that C is a lower level language than Tcl. The C compiler supports data structures that reflect the organization of the data in physical memory, while the Tcl interpreter insulates the programmer from the hardware.

The SWIG solution for this is to generate new Tcl commands for creating and accessing C data structures. The new Tcl commands to create a C data structure will allocate memory for the data structure and return an identifier that Tcl scripts can use to reference the structure. The Tcl script can then pass that identifier to the interface for C functions that need to access the data. As an added benefit, SWIG uses some magic naming conventions to do runtime data checking, so you can't accidentally pass a structure of type a to a function expecting a structure of type b.

You can create an extension with support for creating and using C arrays by adding a little bit of code to the definition file.

The easiest way to do this is to use SWIG's %inline directive. This directive defines functions which should be both included in the final wrapper and exposed to the SWIG parser and code generator.

The SWIG documentation includes this example to show how an array of doubles can be created and accessed:

```
// SWIG helper functions for double arrays
%inline %{
// Create a new array of doubles of a given length
double *new_double(int size) {
    return (double *) malloc(size*sizeof(double));
}
// Delete an array of double
void delete_double(double *a) {
    free (a);
}
// Retrieve the value of an element of the array
double get_double(double *a, int index) {
    return a[index];
}
// Set the value of an element in the array
void set_double(double *a, int index, double val) {
    a[index] = val;
}
%}
```

The new commands can be used like this.

```
# Create an array of doubles
set squares [new_double 5]
# Fill the array with the square of the index
for {set i 0} {$i < 5} {incr i} {
    set_double $squares $i [expr $i * $i]
}
# Invoke a library procedure that requires a
# pointer to an array of doubles as an argument
foo $squares
```

Structures can be a bit more tricky to use but are very simple to describe in the definition file. All you need to do is include the C struct in the body or in an %inline section of the definition file and SWIG will generate a set of interface functions and include them in the Tcl extension.

For example, a structure like this:

```
struct arp {
    unsigned char mac_address[6];
    unsigned char ip_address[4];
}
```

can be accessed with a Tcl script by making a definition file that looks like this:

```
%inline %{
// Define an ARP structure for Machine and IP address
struct arp {
    unsigned char mac_address[6];
    unsigned char ip_address[4];
}
//  A helper utility to set values in an array of
//  unsigned chars -
//  copied from the array example

int unsigned_char_set (unsigned char *ar, \
        int index, unsigned char val) {
    ar[index] = val;
}
//  A test function to display the contents of an
//  arp structure

int showArp (struct arp *p) {
    int i;
    for (i=0; i<6; i++) {
```

```
        printf("0x%x ", p->mac_address[i]);
    }
    printf("\n");
    for (i=0; i<4; i++) {
        printf("0x%x ", p->ip_address[i]);
    }
    printf("\n");
}
%}
```

When SWIG processes this code, it creates these new Tcl commands:

::fib::new_arp *::fib::arp*	Allocates memory for a new arp structure and returns the name to the Tcl script that invokes it.
::fib::delete_arp	Frees the memory associated with an arp structure
::fib::arp_ip_address_get	Returns a handle to access the ip_address C array element of the arp structure.
::fib::arp_mac_address_get	Returns a handle to access the mac_address C array element of the arp structure.
::fib::showArp	An interface into the showArp C function.
::fib::unsigned_char_set	An interface into the unsigned_char_set C function to assign values to elements in a C array.

The Tcl code to test this resembles the following:

```
set arp [arp::new_arp]
set mac [arp::arp_mac_address_get $arp]
for {set i 0} {$i < 6} {incr i} {
    arp::unsigned_char_set $mac $i $i
    }
set ip [arp::arp_ip_address_get $arp]
for {set i 0} {$i < 4} {incr i} {
    arp::unsigned_char_set $ip $i $i
    }

arp::showArp $arp
```

The body of a definition file can usually be extracted from an include file. If you are lucky, you can just use the package's primary include file as a definition file.

The dnet.h file has too much information that's not relevant to creating a wrapper (and is confusing to the SWIG parser), so the simple solution of using dnet.h as a definition file didn't work.

However, all the critical pieces of information (the functions, declarations, and structures used as arguments) are described in the man page, so a set of cut-and-paste operations will create a minimal definition file.

To ensure portability across different word-size machines, the libdnet package uses several data types that aren't part of the basic C language. The SWIG parser doesn't recognize these new datatypes. The SWIG solution for unrecognized data types is to consider them to be pointers.

However, the SWIG parser will recognize a #define or typedef directive to define these datatypes. Adding these lines to the definition file satisfies the SWIG parser:

```
typedef unsigned short uint16_t;
typedef unsigned char  uint_8_t;
typedef unsigned int   uint_32_t;
typedef unsigned int   ip_addr_t;
typedef unsigned int   size_t;
```

To finish the dnet.i definition file, I added versions of the C array access code described above to handle arrays of uint_32_t, uint_16_t, and uint_8_t data.

Once the definition file is complete, SWIG can create a Tcl extension in seconds. The next step is to test the new extension and see if it works.

One of the features of the libdnet library is the ability to send raw packets over the Ethernet. This is as low-level as you can get, and will let me generate whatever type of malformed IP packet I need.

The two critical commands are eth_open, to open a connection to an Ethernet device, and eth_send, to transmit a buffer of binary data (an Ethernet frame).

Syntax: eth_t *eth_open(const char *device);

	Open a connection to an Ethernet device and return a handle for future use.
char *device	The name of the Ethernet device to be connected to, such as eth0, pn0, etc.

Syntax: ssize_t eth_send(eth_t *e, const void *buf, size_t len);

	Transmit a buffer of data over the Ethernet. The buffer should be a valid Ethernet frame. Returns the number of bytes sent. The checksum will be appended automatically.
eth_t *e	The handle returned by eth_open
void *buf	The data to send over the link
size_t len	The number of 8-bit characters to transmit

One problem is that eth_send requires that the buf buffer be a pointer to an area of memory. A Tcl string won't be accepted by the SWIG wrapper. Fortunately, the SWIG wrapper's data validity checking will accept any pointer as a void pointer, so we can use the uint_8_t array commands to create and fill an array of unsigned chars.

Simple code like this will generate garbage packets on the local Ethernet. The data is illegal Ethernet frames, which aren't accepted by other nodes on the network, but running the script will cause the activity lights on an interface card to blink, demonstrating that frames are being sent.

```
# Load the new extension
load ./libdnet.so
# Open a connection to the Ethernet device
set e [dnet::eth_open eth1]
# Create a buffer
set buf [dnet::new_uint_8Array 60]
# Stuff the buffer with incrementing values
for {set i 0} {$i < 60} {incr i} {
    dnet::set_uint_8Array $buf $i $i
}
# And shove it onto the wire 10 times
for {set i 0} {$i < 10} {incr i} {
    dnet::eth_send $e $buf 60

    # Pause for 100 milliseconds
    after 100
}
```

The next step is to send a legal packet and see if it's recognized.

An Ethernet frame consists of five fields of data:

Field size (bytes)	Description
6	The destination MAC address.
6	The source MAC address.
2	A type definition. This is 0x0800 for IP datagrams.
46–1500	The datagram.
4	A Cyclic Redundancy Checksum.

The arp -a command gave me a list of IP addresses and corresponding MAC addresses to fill in the source MAC address and destination MAC address fields; the type field for an IP datagram is 0x0800, and the CRC will be appended by the transmission code.

To generate a valid IP datagram, I used tcpdump with the -x option to get a hex dump of an IP packet. I decided to ping the target node from the node running the Tcl script and grab one of those packets. Using an Echo Request packet provides two sets of validation. Using tcpdump, I can watch the packet arrive on the target node, and I can also see if the target machine responds to the fabricated ping request.

Tcl has full support for operating with lists of data. It makes sense to treat a packet as a Tcl list of hex values until it needs to be converted to an array of unsigned chars for the eth_send command.

The code below creates an Ethernet frame from the various pieces of data. It uses the split command, to convert a colon-delimited MAC address into a list of hex bytes, and the eval command, to combine two lists into a single list.

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

Syntax: split *string ?splitChars?*

split	Splits a string into a list. Elements are delimited by a marker character.
string	The string to split.
?splitChars?	A string of characters to mark elements. By default the markers are whitespace characters (tab, newline, space, carriage return). In this example, the character to split on is the colon separating the bytes in a MAC address.

The eval command concatenates the arguments into a string before starting the evaluation. This causes a set of data to lose one level of data grouping. Without eval, a command like lappend list $list2 would be evaluated as lappend list {a b c}, which will append the list element {a b c} to a list. The command eval lappend list $list2 would be evaluated as lappend list a b c, which will append three list elements, a, b, and c to a list.

This script will generate an Ethernet frame and transmit it to the local network:

```
# The MAC address, obtained with arp -s
set destEther 00:E0:4C:00:14:4D
set srcEther 00:A0:CC:D1:B6:00
# A valid echo request packet,
#  obtained with tcpdump
set echo_Request [list 45 00 00 54 00 00 40 00 40 01 \
    05 16 c0 a8 5a 40 c0 a8 5a 02 08 00 98 d9 \
    df 22 00 00 63 cf 4d 3d d5 f3 0e 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]
# Fill a list with hex values
set packet [split $destEther :]
eval lappend packet [split $srcEther :]
```

```
lappend packet 08 00
eval lappend packet $echo_Request
# How many bytes are we using?
set len [llength $packet]
# Create a C array and fill it.
set buf [dnet::new_uint_8Array $len]
for {set i 0} {$i < $len} {incr i} {
   dnet::set_uint_8Array $buf $i 0x[lindex $packet $i]
}
# And shove it onto the wire 10 times
for {set i 0} {$i < 10} {incr i} {
     dnet::eth_send $e $buf $len

     # Wait 100 milliseconds between frames
     after 100
}
```

This extension provides a platform for generating IP packets. The next article will start describing techniques for validating the packet generator before using the generator to validate another system.

USENIX and SAGE Need You

People often ask how they can contribute to our organizations. Here is a list of tasks for which we hope to find volunteers (some contributions not only reap the rewards of fame and the good feeling of having helped the community, but authors also receive a small honorarium). Each issue we hope to have a list of openings and opportunities.

The SAGEwire and SAGEweb staff are seeking:

- Interview candidates
- Short article contributors (see *http://sagewire.sage.org*)
- White paper contributors for topics like these:

Back-ups	Emerging technology	Privacy
Career development	User education/training	Product round-ups
Certification	Ethics	SAGEwire
Consulting	Great new products	Scaling
Culture	Group tools	Scripting
Databases	Networking	Security implementation
Displays	New challenges	Standards
E-mail	Performance analysis	Storage
Education	Politics and the sysadim	Tools, system

- Local user groups: If you have a local user group affiliated with USENIX or SAGE, please mail the particulars to *kolstad@sage.org* so they can be posted on the Web site.

;login: is seeking attendees of non-USENIX conferences who can write lucid conference summaries. Contact Tina Darmohray, *tmd@usenix.org*, for eligibility and remuneration info. Conferences of interest include (but are not limited to): Interop, SOSP, O'Reilly Open Source Conference, Blackhat (multiple venues), SANS, and IEEE networking conferences. Contact *login@usenix.org*.

;login: always needs conference summarizers for USENIX conferences too! Contact Alain Hénon, *ah@usenix.org*, if you'd like to help.