the tclsh spot

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

Previous Tclsh Spot articles have described some of the details in building a firewall validation system. This application uses an agent style of client-server relationship, with one master program controlling several agents that run on different physical platforms to generate packets and analyze how they are processed.

Tcl's clean-socket mechanism, support for importing code at runtime, and support for safe child interpreters make it a powerful tool for agent applications.

However, the base distribution of Tcl supports only unencrypted TCP sockets. When transferring code snippets to be executed on a client machine, we want to ensure that the data has not been modified and that the sender is a system we trust.

The SSL protocol (initially developed by Netscape to provide secure Web transactions) is an application-independent protocol that supports server and client authentication. With its support for authenticating servers and negotiating encryption keys, SSL provides the strong encryption and validation tools that this type of application requires.

Since it's so easy to extend Tcl with new commands, it wasn't long before SSL support was added with an extension.

The TLS (Transport Layer Security) extension is an interface to the OpenSSL libraries that provides for validated, encrypted conversations over a TCP socket. This is a mature package used by the Tcl http daemon to support secure Web transactions as well as numerous other applications. The TLS extension was originally written by Matt Newman and has been maintained by several folks, including Jeff Hobbs and Dan Razell.

This article describes the TLS extension, and how to set up a secure link between a master and the remote agents, and introduces use of a safe child interpreter to evaluate suspect code from a remote site. Many thanks go to Dan Razell for his help in understanding how TLS works and his tips on how to construct a secure client and server.

The first step is to download the TLS code from *http://sourceforge.net/projects/tls/*. The latest released version (1.4.1) has some minor bugs that are fixed in the CVS archives and will be released soon as version 1.5.

Once downloaded, the usual ./configure; make; make install will install the TLS extension on your system.

Once this is done, you can prove that it worked by starting your Tcl shell and trying to package require the TLS extension:

```
$> tclsh
% package require tls
    1.50
%
```

The program flow for opening a secure client socket is similar to opening a standard Tcl client socket. Your script will open an I/O channel and then evaluate puts, gets, read, and flush commands as necessary. The difference between using a normal socket and a secure socket is that instead of opening the channel with the socket command, the channel is opened with the secure socket (tls::socket) command.

Syntax: tls::socket ?-switch value? host port

-switch value — A key-value pair to define how this socket should behave. There are several options including:

-requestbool
: Request a certificate from peer during SSL handshake. Default: true

-serverscript
: Handshake as a server. The script will be evaluated when a client attempts to connect.

-require bool
: Require a valid certificate during SSL handshake. Default: false

-password script
: A script to invoke when OpenSSL requires a password. The script should return a plaintext password to be used, perhaps by querying a user.

-keyfile fileName
: A private key file to use.

-cafile fileName
: Defines a CA file to use.

-certfile fileName
Defines the certificate to use.

host The name or IP address of the host.

port The name or port number of the port to connect to.

Once OpenSSL and the TLS extension are installed, you can write a Tcl script that will interact with a secure Web site. This code snippet will send an HTTP GET request to a secure Web server and retrieve a few lines of the reply.

```
package require tls
set s [tls::socket -request 0 127.0.0.1 8443]
puts $s "GET / HTTP/1.0\n";
flush $s
set page [read $s]
puts $page
```

This script generates this output:

```
HTTP/1.0 200 Data follows
Date: Sun, 07 Dec 2003 02:42:09 GMT
Server: Tcl-Webserver/3.4.2 September 3, 2002
...
```

SSL supports several authentication modes when establishing a socket connection. Either or both endpoints can authenticate each other, or you can suppress authentication entirely. This example demonstrates the simplest case, establishing a connection *without* authentication of either endpoint.

The -request 0 option allows the two ends of the socket to negotiate an encryption key without requiring authentication. The connection will be encrypted, and thus be *private*, but the identity of the two endpoints is not confirmed.

For simple encrypted conversations without authenticating the identity of the participants, the SSL handshake only needs to exchange public keys and confirm that messages can be encrypted and decrypted. To authenticate a sender's identity, the participants need access to a trusted certificate that provides a digital fingerprint to identify the sender.

Since the agents will be evaluating code that could be malicious, it's important to use both the encryption and validation features of SSL. (The code could still be malicious, but at least the agent will be certain of the source.)

In order for the two ends of a conversation to authenticate each other, they need some way to refer to a trusted third party that will authenticate their identities. Rather than actively involve this third party in every transaction, the handshake supports using signed certificates. This allows the authentication to be recorded ahead of time. During the SSL handshake, an endpoint can request that a peer transmit its certificate. The signature on the certificate can then be verified against the public signature of the third party.

The third party is called a certificate authority (CA). Each endpoint maintains a set of these signatures for use whenever it requests a certificate from some other endpoint. The signatures are themselves represented as certificates. Web browsers usually come supplied with CA certificates from parties such as RSA Security and Verisign, which the browser will implicitly trust.

Thus, in order to establish a secure connection (one which is both authenticated and private), SSL needs a certificate and key to compare with the values sent from the other process. The SSL protocol requires a CA certificate at the receiving endpoint to verify the certificate from the sender. The sender obviously needs to transmit a certificate containing its own public key, but it also needs its corresponding private key in order to sign the message itself, so that the receiver can ensure that the message indeed came from the sender.

For a Web site running SSL, you'll want a certificate signed by a well-known and trusted authority, such as Verisign, RSA, or Microsoft. For this application, where the participants must install custom software and configuration files, the certificates can be signed by a local CA. The applications trust the certificates because they are part of the installation.

You can create the keys and certificates you'll need from the openssl command line, or using a GUI like the SimpleCA Tcl script developed by Joris Ballet (*http://users.skynet.be/ballet/joris/SimpleCA*). SimpleCA is a thin front end over openssl that lets you fill in a form instead of following a challenge-response script. The interaction with openssl is recorded in a log file for later examination.

The latest (Rev 28) version of SimpleCA will create a root certificate for you (if none exists) when you start the application. It will start by requesting the basic information with forms like this:

When the forms are complete openssl will be used to generate a rootca.pem file containing the root CA.

The file can be viewed with the OpenSSL command openssl x509 -in rootca.pem, or with cat. It will resemble this:

```
———-BEGIN CERTIFICATE———-
MIIClDCCAf2gAwIBAgIBADANBgkqhkiG9w0BAQQFADB2MQswCQYDVQQGEwJVUzEc
MBoGA1UEChMTTm91bWVuYSBDb3Jwb3JhdGlvbjEQMA4GA1UECxMHVGVzdGluZzEW
...
———-END CERTIFICATE———-
```

Once this is complete, you should create client and server certificates using the Certificates/New Certificate Request menu choice. As with the root certificate, the forms will prompt you for the basic information required to create the certificate and what file to save it in. On the first screen, select Personal for a client certificate, and SSL Server for the server-side certificate.

The forms for a server certificate will request information about the company requesting the certificate (such as physical address), while the personal/client certificate only needs a common name and email address.

When this is complete, SimpleCA will have created files named certificates/SITENAME.csr and certificates/SITENAME.key for the server certificate and certificates/EMAIL.csr and certificates/EMAIL.key for the client certificate (assuming you accept default names and paths).

The Certificate Request files (*.csr) will resemble this:

```
———-BEGIN CERTIFICATE REQUEST———-
MIICEDCCAXkCAQAwgZ0xCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhNaWNoaWdhbjEP
MA0GA1UEBxMGRGV4dGVyMRwwGgYDVQQKExNOb3VtZW5hIENvcnBvcmF0aW9uMREw
...
———-END CERTIFICATE REQUEST———-
```

and the key files will resemble this:

```
———-BEGIN RSA PRIVATE KEY———-
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,5FBB5CBE38B2C08A

3Jv8/+t74zvyY7qlkUxhb1aKdqpOEebRhg/LbdFrSaPWKQkQ2nITyAmQR6d3QW6p
tffYqBm3d0YiOTFLcgNCjhEGhLRYgpw1MxRKq6VRXTjknB6fJn2Zjai0jVXERU44
...
———-END RSA PRIVATE KEY———-
```

The next step is signing the certificates, which is handled under the CA/Sign PKCS#10 Certificate Request menu. This will allow you to select the certificate to sign, display the information about that certificate, request the root authority password, and, finally, summarize what is going to happen with a screen like this:

This sequence of steps will generate files named certificates/SITENAME.crt and certificates/EMAIL.crt. These are binary datafiles by default, though you can force them to be generated as ASCII export files by specifying a .pem suffix when you select the file name.

The main GUI shows a summary of the certificates that have been created:

If you've used the defaults (and created binary datafiles), the final step is to transform the client and server certificates into the flat ASCII export format, using the Tools/Export Certificate menu option. This will generate files named for their serial number. In this example, the default names will be 1000.pem and 1001.pem.

The important files for creating validated TLS sockets are the rootca.pem, *.key, and SERIAL_NUM.pem files.

Creating a client socket with validation is very similar to the previous example. For a validated connection, however, the application needs paths for the certificates and keys, needs to request a certificate from the peer, and must be able to provide a password.

This example provides all the necessary information, using a trivial hardcoded procedure to provide the password:

```
package require tls

proc getPassword {} {
    return "testing"
}

set port        2000
set host        localhost
set certDir     /usr/SimpleCA28/certificates

set s [tls::socket -password getPassword \
    -keyfile $certDir/clif@noucorp.com.key \
    -certfile $certDir/../1000.pem \
    -cafile $certDir/../rootca.pem \
    -request true -require true $host $port]

puts $s "this is a message from the client"
```

The server side of this socket is a bit more complex. Like a basic TCP server socket, rather than opening a channel to a remote site and immediately transferring data, the secure server waits until a client requests a connection to a particular port. When a client requests a connection, a new port is assigned for the conversation and a callback script defined in the tls::socket -server command is evaluated.

A simple TCP server to report the current time and close the connection looks like this:

```
#!/usr/local/bin/tclsh
socket -server openConnection 12345

proc openConnection {channel ip port} {
    puts $channel [clock format [clock seconds]]
    close $channel
}
```

A secure server needs to wait until the handshake is complete (and successful) before processing data messages.

The tls::handshake command forces a handshake and returns the status of a handshake that's in process. The tls::handshake command will read a message if one is available, and returns 0 if the handshake is still in progress (non-blocking) or 1 if the handshake was successful. If the handshake failed, this routine will throw an error.

Syntax: tls::handshake channel

channel	The channel being opened.

This would lead to a trivial solution, such as the one used for the openConnection procedure:

```
proc openConnection {channel clientaddr clientport } {

    # Wait until the handshake is complete

    set fail [catch {tls::handshake $channel} complete]
    while {!$fail && !$complete} {
        after 100
        set fail [catch {tls::handshake $channel} complete]
    }
    puts $channel [clock format [clock seconds]]
    close $channel
}
```

This solution has a big problem. It will hang in the openConnection procedure until the handshake is complete and successful. At best, this blocks the server from accepting other connections until the handshake is complete, and at worst, if the handshake cannot be successful (e.g., the client closes the connection), it will hang forever.

A better solution makes use of the Tcl fileevent (described in the previous Tclsh Spot article) to watch for messages and force them to be absorbed by tls::handshake until tls::handshake returns a successful handshake (or until the channel is closed). This technique supports having several clients connecting at once.

```
proc openConnection {channel clientaddr clientport } {
    global tlssignal
    global msgsignal

    # Wait until the handshake is complete.

    fileevent $channel readable [list handshakeHandler
                                 $channel $clientaddr]
    vwait tlssignal($channel)

    puts $channel [clock format [clock seconds]]
    close $channel
}

proc handshakeHandler {channel clientaddr} {
    global tlssignal

    # Optionally, reject connection based on IP
    # address–based access control list.

    # Check for death of client channel.

    if {[eof $channel]} {
        close $channel
        return
    }

    # Absorb message and report status.

    set fail [catch {tls::handshake $channel} complete]

    if {!$fail && $complete} {
        set tlssignal($channel) " "
    }
}
```

The final step for making a useful agent is to support real data messages, as well as handshakes. This uses the fileevent command again — in this case, to grab messages and process them:

```
proc openConnection {channel clientaddr clientport } {
    global tlssignal
    global msgsignal

    # Wait until the handshake is complete.

    fileevent $channel readable [list handshakeHandler
                    $channel $clientaddr]
    vwait tlssignal($channel)

    # Eval processMessage when data is available.

    fileevent $channel readable [list processMessage
                    $channel]
}
```

For a simple agent test, the processing might be to return the number of characters sent:

```
proc processMessage {channel} {
    if {[eof $channel]} {
        close $channel
        return
    }
    set len [gets $channel message]
    puts $channel "Message length is: $len"
    flush $channel
}
```

Real agents, however, execute code rather than evaluating predefined procedures.

Running code from an outside source on my system (even if I've proven that the source is known and trusted) gives me the heebie-jeebies. Even in code that's not supposed to be malicious, there's the potential for a bug.

The Tcl solution for this problem is the interp command, and the support for creating *safe* child interpreters.

Syntax: interp create ?-safe? ?interpName?

interp create	Create a new slave interpreter
-safe	Make this a *safe* interpreter with no ability to do damage to your system.
interpName	An optional name for this interpreter.

The interp command creates a new child interpreter within a running Tcl process. In actual terms, this means a new state structure and hashtables for variables and procedures.

Using a hashtable technique to map the names of commands to the compiled code that implements them makes it easy to create a *safe* interpreter. The interpreter simply leaves commands like open, exec, and socket out of the hashtable. This makes it impossible for a script running in a *safe* interpreter to invoke those commands.

The next example shows both how a full-featured interpreter can be created and how to create one without access to the file system:

```
# Create an interpreter with full access.

set int1 [interp create fullservice]

# Load the Tk extension and build a GUI.

$int1 eval "load /usr/local/lib/libtk8.2.so"
$int1 eval "label .l1 -text "OK"; grid .l1"

# Create an interpreter that cannot access the file system.

set int2 [interp create -safe limited]

# This throws an error.

$int2 eval "load /usr/local/lib/libtk8.2.so"
```

When a new interpreter is created, Tcl creates a new command with the same name as the new interpreter. The command interp create newInterp creates a new interpreter named newInterp and a new command newInterp. As with other Tcl objects, a script will interact with the interpreter by using the new command.

As shown in the previous example, you can evaluate a script within the child interpreter with the interpName eval command.

This example shows how we can create a new *safe* interpreter to evaluate commands received from a remote system. In this example, Tcl commands could be evaluated.

```
# Create a new safe interpreter.

interp create -safe safeInterp

# Receive

proc processMessage {channel} {
    if {[eof $channel]} {
        close $channel
        return
    }

    set rply [safeInterp eval [gets $channel]]
    puts $channel "$rply"
    flush $channel
}
```

If we need to add new procedures to the interpreter, they can be added within an interpName eval command like this:

```
safeInterp eval {
    proc checksum {string} {
        set total 0
        foreach c [split $string " "] {
            scan $c %c x
            incr total $x
        }
    return $total
  }
}
```

However, to be useful, even a *safe* interpreter needs to be able to interact with a file system, or perform some other *unsafe* interaction. The interpreter alias command can be invoked within a parent interpreter to allow a slave to run certain scripts within the parent environment (which may be a full-service interpreter). This leads to a tightrope act in which we open small holes in the safe interpreter to perform tightly defined unsafe actions"

Syntax: interpName alias targetName sourceName

targetName	The name by which a procedure will be referenced in the child interpreter.
sourceName	The name by which a procedure is referenced in the parent interpreter.

For example, if we needed to add a logging facility to the checksum procedure shown above, it couldn't be done – that procedure runs in a *safe* interpreter and can't open any channels.

However, using the alias command, we can link a logging procedure in the main interpreter to a procedure name in the *safe* child interpreter:

```
# Create a safe interpreter.

interp create -safe safeInterp

# Link the 'writeLog' procedure in this environment
# to the 'log' procedure in the safe child interpreter.

safeInterp alias log writeLog

# Define writeLog.

proc writeLog {data} {
    set of [open /tmp/agent.log "a"]
    puts $of $data
    close $of
}

# Define a procedure to use the logging facility.

safeInterp eval {
    proc checksum {string} {
        set total 0
        foreach c [split $string " "] {
            scan $c %c x
            incr total $x
        }
    log "$total $string"
    return $total
}
```

And that provides all the tools for creating a cryptographically secure agent system in Tcl. The complete code for the client and server is just 131 lines and is available at *http://www.noucorp.com*.

The next Tclsh Spot will show you how to start using these tools to evaluate a firewall.