Cryptkit – A Cryptographic Extension for Tcl

Steve Landers
Digital Smarties
steve@DigitalSmarties.com

Abstract

This paper describes the design and implementation of Cryptkit – a Tcl extension providing a powerful toolkit that makes adding encryption and authentication services to Tcl applications straightforward, even for inexperienced crypto programmers. Cryptkit contains an abstracted object-oriented binding to the Cryptlib security toolkit – a powerful, portable and comprehensive security library written in C. The paper will also discuss possible future developments, including a Tcl stacked channel interface and an implementation of the recently endorsed X9.95 Trusted Time Stamp Management and Security standard.

Introduction

The genesis of Cryptkit came about during the 11th Tcl/Tk Conference[1] in New Orleans when Mike Doyle of Eolas Technologies commented that it should be possible to produce a Tcl cryptographic extension based on Cryptlib[2]. His rationale was quite simple – it is trivially easy to build client/server and web applications in Tcl so it should be just as simple to add appropriate security. And although there are a number of good cryptographic extensions for Tcl (in particular TLS[3]) there is no single cryptographic toolkit that supports a wide range of cryptographic and authentication features.

Mike asked this author if he would be interested in producing a Tcl interface to Cryptlib. In one of his more optimistic pronouncements of recent times, this author responded "sure – I can probably throw together an interface quite quickly". Whist that was (strictly speaking) true, producing an appropriate interface turned out to be significantly more work, for a number of reasons:

- the Cryptlib API is large – with over 65 calls and 350 pages of documentation
- a lot of design effort was required to produce a Tcl binding that is both "Tclish" (i.e. comfortable for Tcl developers) and also close enough to the extensive Cryptlib documentation to avoid the need for Tcl specific documentation
- during the implementation it became apparent that there existed an opportunity to provide higher level "wrappers" or utility procedures that encapsulated good practice for Tcl applications
- there was also the opportunity to provide an interface to Tcl channels, using the Tcl transformation and stacked channel facilities
- and finally, there is the opportunity to include non-Cryptlib features within the utility procedures – e.g. access to Tcllib[4] cryptographic functions.

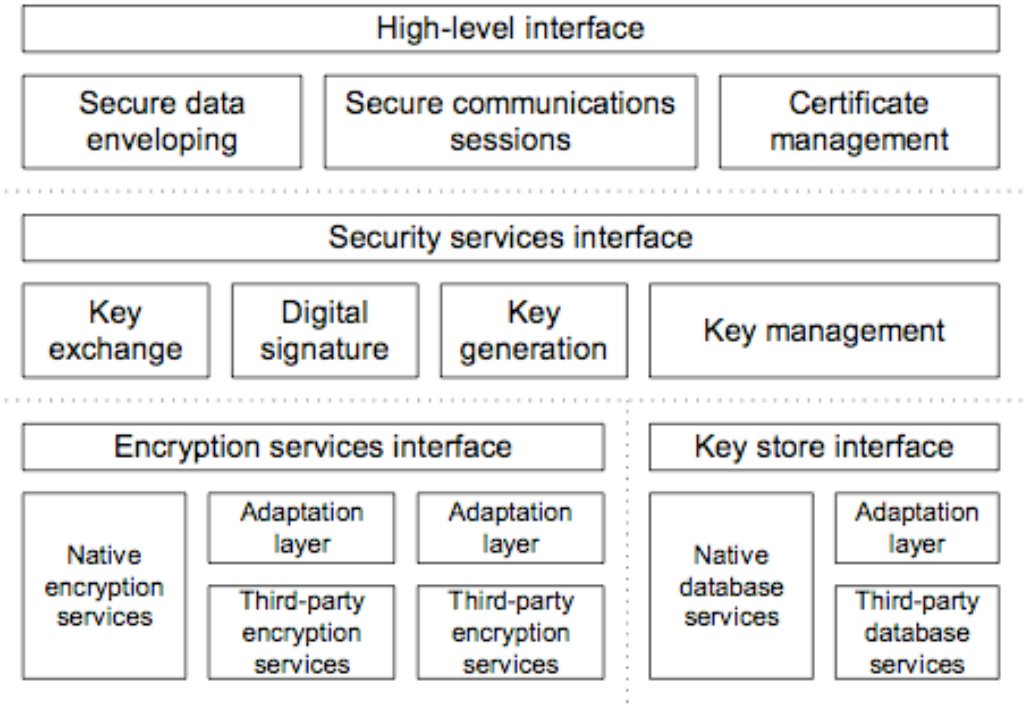
And so the Cryptkit project started – with the goal of producing a comprehensive cryptographic extension for Tcl based on (but not restricted to) the Cryptlib toolkit.

Cryptlib overview

Cryptlib is a general-purpose encryption toolkit written by Peter Gutmann, a self-confessed “Professional Paranoid” in the Department of Computer Science at the University of Auckland. Cryptlib is described as “a layered set of security services and associated programming interfaces that provides an integrated set of information and communications security capabilities”[5]. In summary, it is a powerful, portable and comprehensive security library written in C, but with an “object-like” interface.

Cryptlib provides three levels of interface:

- a high-level interface providing secure data enveloping, secure communications sessions and certificate management
- a medium-level security services interface providing key exchange, digital signature, key generation and key management
- a low-level encryption services interface providing access to native and third-party encryption services (including hardware crypto devices), and a key store interface to native and third party database services



Cryptlib supports a wide range of ciphers (inc. Blowfish, DES/3DES, IDEA, RC2/4/5), message digests (inc. MD2/4/5, SHA-2/256/384/512), message authentication codes (inc. HMAC-MD5, HMAC-SHA1) and public key algorithms (inc. Diffie-Hellman, DSA, RSA, ElGamal). It is also full multi-threaded and extensible with new algorithms.

It is released under a dual license – it is free for non-commercial use and can be licensed for commercial use.

Being a portable and well-documented C library, Cryptlib was an ideal candidate to be used as the basis for a Tcl extension.

First steps – Cryptkit V1

The first version of Cryptkit stayed quite close (in style) to the Cryptlib API. In particular it relied on Cryptlib's data enveloping rather than a more "Tcl friendly" or Object Oriented approach.

There were a number of reasons for taking this approach

- to leverage the existing documentation of the Cryptlib API
- to keep the Tcl bindings relatively thin
- to defer designing a more abstracted interface until there was practical experience at using Cryptkit and Cryptlib

For example, consider the following example that creates a general purpose data envelope "e" using the higher-level Cryptlib API

```
package require cryptkit
namespace import cryptkit::*
cryptInit
cryptCreateEnvelope e CRYPT_UNUSED CRYPT_FORMAT_CRYPTLIB
```

Note that in version 1 all Cryptkit procedures live in the "cryptkit" namespace, and that procedure names correspond directly to the Cryptlib C API function.

The cryptInit procedure must be called once to initialise Cryptlib, otherwise any Cryptkit procedures will return an error. All Cryptkit V1 calls return either CRYPT_OK or CRYPT_ERROR, as appropriate.

Now consider the cryptCreateEnvelope procedure invocation - this calls the equivalent C API function with three arguments

- the first is the name of a Tcl variable to store ID of the data envelope
- the second is the user who owns the envelope – Cryptkit supports one user per process and so passes the Cryptlib symbol CRYPT_UNUSED indicating the default user
- the format for the enveloped data – in this case the default CRYPT_FORMAT_CRYPTLIB format

Assuming the invocation of cryptCreateEnvelope works it will return CRYPT_OK and the Tcl variable "e" will be set to an integer value (e.g. 6) indicating the object number within Cryptlib. This value can be used in subsequent calls to manipulate the Cryptlib encapsulation object.

For example, consider the following example in C - digitally signing an envelope

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;
cryptCreateEnvelope(&cryptEnvelope, CRYPT_UNUSED,
                                    CRYPT_FORMAT_CRYPTLIB);

/* Add the signing key */
cryptSetAttribute(cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
                                 SigKeyContext)
```

```
/* Add the data size information and data, wrap up the processing,
   and pop out the processed data */
cryptSetAttribute(cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
                                 messageLength);
cryptPushData(cryptEnvelope, message, messageLength, &bytesCopied);
cryptFlushData(cryptEnvelope);
cryptPopData(cryptEnvelope, envelopedData, envelopedDataBufferSize,
                            &bytesCopied);
cryptDestroyEnvelope(cryptEnvelope);
```

The Tcl equivalent using Cryptkit V1 is

```
cryptCreateEnvelope e CRYPT_UNUSED CRYPT_FORMAT_CRYPTLIB
cryptSetAttribute $e CRYPT_ENVINFO_SIGNATURE $sigKeyContext
cryptSetAttribute $e CRYPT_ENVINFO_DATASIZE \
                              [string length $message]
cryptPushData $e $message bytesCopied
cryptFlushData $e
cryptPopData $e buffer $bufsize bytesCopied
cryptDestroyEnvelope $e
```

A further example is a script to:

- create a key pair and certificate
- use the key pair to sign the certificate
- add the key pair and certificate to a key set,
- write the certificate chain to a file

```
cryptInit
cryptAddRandom NULL CRYPT_RANDOM_SLOWPOLL

# Create key pair
cryptCreateContext keyPair CRYPT_UNUSED CRYPT_ALGO_RSA
cryptSetAttributeString $keyPair CRYPT_CTXINFO_LABEL "ca"
cryptSetAttribute $keyPair CRYPT_CTXINFO_KEYSIZE 128
cryptGenerateKey $keyPair

# Create cert
cryptCreateCert cert CRYPT_UNUSED CRYPT_CERTTYPE_CERTIFICATE
cryptSetAttribute $cert CRYPT_CERTINFO_SELFSIGNED 1
cryptSetAttribute $cert CRYPT_CERTINFO_CA 1
cryptSetAttribute $cert CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO $keyPair
cryptSetAttributeString $cert CRYPT_CERTINFO_COUNTRYNAME "au"
cryptSetAttributeString $cert CRYPT_CERTINFO_COMMONNAME $commonName

# Sign cert with keyPair
cryptSignCert $cert $KeyPair

# open key store
cryptKeysetOpen keySet CRYPT_UNUSED CRYPT_KEYSET_FILE "keyset.p15" \
                       CRYPT_KEYOPT_CREATE

# Add keyPair and cert chain to pkcs15 keyset (.p15)
cryptAddPrivateKey $keySet $keyPair "password"
cryptAddPublicKey $keySet $cert
cryptKeysetClose $keySet
```

```
# Write out cert chain to pkcs7 file (.p7b)
cryptExportCert buffer bufferLength CRYPT_CERTFORMAT_CERTCHAIN $cert
set fos [open $label.p7b w]
puts $fos $buffer
close $fos

# Cleanup
cryptDestroyContext $cert
cryptDestroyContext $keyPair
cryptEnd
```

Problems and opportunities

As can be seen in the examples given – the Cryptkit V1 Tcl API is very similar in style to the C API. Whilst this does have the advantages listed previously, it does result in an interface that is less than ideal for a Tcl developer - for a number of reasons:

- the Tcl programmer has to be concerned about integer versus string attributes even though Tcl is a type-less language (e.g. cryptSetAttribute versus cryptSetAttributeString)
- arguments that are never used, or are always set to a fixed value still appear in the Tcl procedure call (e.g. the user arguments such as that in cryptCreateEnvelope are always set to CRYPT_UNUSED)
- the interface is somewhat verbose – e.g. prefixing all symbols with CRYPT_ is unnecessary when we know that (for example) the third argument to cryptCreateEnvelope will always be the format
- likewise, given Cryptkit procedures are in their own namespace there is no need to prefix each procedure with “crypt”
- the cryptkit namespace could be shorted to crypt – giving more readable procedure names like crypt::CreateEnvelope

But perhaps the main problem is that the interface isn’t “Tclish” – that is, it doesn’t follow the practices and conventions that have become typical in the Tcl world.

The second iteration – a more Tclish API

The most significant change in the V2 API is that procedures now return a “handle” which can then be manipulated in an object oriented fashion

Consider again the example of digitally signing an envelope. Using the V2 API this looks like the following:

```
set e [crypt::CreateEnvelope FORMAT_CRYPTLIB]
$e set ENVINFO_SIGNATURE $sigKeyContext
$e set ENVINFO_DATASIZE [string length $message]
$e PushData $message
$e FlushData
set buffer [$e PopData $bufsize]
$e destroy
```

Some points to note are:

- CreateEnvelope returns an object oriented “handle”, rather than just an internal ID returned from Cryptlib
- the “handle” is a Tcl command which can be invoked like any other Tcl command
- procedures that operate on an envelope handle are available as methods (or sub-commands) to the handle
- the envelope handle has set and get methods which replace the SetAttribute/SetAttributeString and GetAttribute/GetAttributeString commands
- Cryptkit now lives in the crypt namespace and the “crypt” prefix has been removed from the Cryptkit procedure names
- the CRYPT_ prefix is now optional on Cryptlib symbols

A more complex example follows:

```
# Hash some data
set sha [crypt::CreateContext ALGO_SHA]
$sha Encrypt "hello I am data to hash"
$sha Encrypt ""      ;# end of data

# Generate DSA key
set dsa [crypt::CreateContext ALGO_DSA]
$dsa set CTXINFO_LABEL "testkey"
$dsa GenerateKey

# Produce a signature
set buffer [$dsa CreateSignature $sha]

# Validate the signature
if {[$dsa CheckSignature $buffer $sha]} { puts OK! }

# Generate a symmetric key
set aes [crypt::CreateContext ALGO_AES]
$aes GenerateKey

# Generate a symmetric key to wrap the first one
set des [crypt::CreateContext ALGO_3DES]
$des set CTXINFO_KEYING_SALT "this is salt"
$des set CTXINFO_KEYING_VALUE "secret password"

# Wrap the first key in the second
set buffer [$des ExportKey $aes]

# Query the signature blob
QueryObject $buffer i
parray i
```

The parray command outputs the following

```
i(cryptAlgo)  = CRYPT_MODE_CBC
i(cryptMode)  = CRYPT_MODE_CBC
i(hashAlgo)   = CRYPT_MODE_NONE
i(objectType) = CRYPT_MODE_ECB
i(salt)       = this is the salt
i(saltSize)   = 16
```

Higher-level utility procedures

Cryptkit V2 goes further and introduces a number of utility procedures that make it even easier to perform common cryptographic functions from Tcl. These include:

- cert
- key
- encrypt
- decrypt
- hash
- option
- session

Utility procedures accept an argument that specifies the procedure type (e.g. a certificate type), and a series of argument pairs that correspond to Cryptlib attributes and values.

The **cert** utility procedure is used to create a certificate - the certificate type defaults to CERTTYPE_CERTIFICATE if not specified. For example, the command

```
crypt::cert -selfsigned 1 -ca 1 -publickey 1 –simple 1 \
            -name "My name"
```

will run the following V2 commands

```
set c [crypt::CreateCert CERTTYPE_CERTIFICATE]
$c set selfsigned 1
$c set CERTINFO_SUBJECTPUBLICKEYINFO 1
$c set commonname "My name"
$c set CERTINFO_XYZZY 1
$c SignCert ""
```

The **key** utility procedure accepts an argument that specifies the key type (e.g. IDEA), and a flag to specify the key text. For example, the command

```
set k [crypt::key idea -key $key]
```

will run the following V2 commands

```
set k [crypt::CreateContext ALGO_IDEA]
$k set CTXINFO_KEY $key
$k GenerateKey
```

The **encrypt** utility procedure accepts a session key and a list of data values. Similarly the **decrypt** utility procedure does the reverse and returns clear text. So, to use the above key to encrypt and decrypt a clear text message we could do something like

```
set c [crypt::encrypt -sessionkey $k $message]
set u [crypt::decrypt –sessionkey $k $c]
if {$u eq $message} {
    puts worked
} else {
    puts failed
}
```

The **hash** utility procedure is used to hash text using one of the algorithms supported by Cryptlib. For example

```
set msg "this is the message"
foreach algo {md2 md4 md5 ripemd160 sha} {
   puts "[format %-10s $algo] = [crypt::hash $algo $msg]"
}
```

will output the following

```
md2        = fa11a5d972d18efb8acd4e70242890f0
md4        = ba370f9c73940645ff32dab913764697
md5        = e30d2a38a4a5d155aa18f09ae1a155ab
ripemd160  = c721e9ed4bfa87039049c1baa8f73e8ea8d37367
sha        = c37c7b5326ce400d23d9807b13f1ea396861a0b5
```

The hash procedure does quite a bit of work behind the scenes. For example, the following command:

```
crypt::hash md5 $msg
```

This will run the following V2 commands, which return the hash value as a text formatted string (which is much more useful to a Tcl program than the binary data)

```
set k [crypt:CreateContext ALGO_MD5]
$k Encrypt $msg
$k Encrypt ""
set bin [$k get CTXINFO_HASHVALUE]
set len [expr {2*[string length $bin]}]
binary scan $bin H$len hash
$k destroy
return $hash
```

The **session** utility procedure can be used to create client or server sessions using one of the Cryptlib secure session facilities. Supported session types are:

- SSH
- SSL
- TLS
- CMP
- SCEP
- RTCS
- OCSP
- TSP

For example, to create an SSL server session listening on port 1080 use the following:

```
set s [crypt::session ssl -type server -port 1080]
```

Similarly, to create an ssh client session for the current user, connected to host www.server.com on the default ssh port use:

```
set c [crypt::session ssh -privatekey $key -host www.server.com]
```

where $key is a key object created by the **key** utility procedure.

You can then write data to the ssh server using the following:

```
$c puts "ls -l"
```

The session "puts" sub-command is analogous to the Tcl puts command, and will call crypt::PushData and crypt::FlushData.

You can read data back from the ssh server using

```
while {[$c gets line] >=0 } {
    puts $line
}
```

The session "gets" sub-command is analogous to the Tcl gets command, and will call crypt::PopData and split the returned value into lines.

And finally there is an **option** utility procedure which can be used to set and get Cryptlib global configuration options. To get a Cryptlib setting use the following form:

```
crypt::option optionclass attribute
```

The two arguments are concatenated to produce the name of the Cryptlib option – so to retrieve the default hash algorithm (CRYPT_OPTION_ENCR_HASH) you would use

```
crypt::option encr hash
```

and to produce a hash using the default algorithm we'd use something like

```
set def [crypt::option encr hash]
puts "default hash ($def) = [crypt::hash $message]"
```

which would produce

```
default hash (ALGO_SHA) = c37c7b5326ce400d23d9807b13f1ea396861a0b5
```

To set global options specify a value as the third parameter. For example, to set the default hash algorithm to MD5 you would use

```
crypt::option encr hash ALGO_MD5
```

Setting an option usually only remains in effect during the current Cryptkit session (i.e. until the Tcl interpreter using Cryptkit exits or the crypt::End command is called).

Options can be saved to permanent storage using the option "save" sub-command. This will make them available to be used in subsequent Cryptkit sessions:

```
crypt::option save
```

The option “dump” sub-command can be used to display all Cryptlib global options, grouped by option class (particularly useful when developing an application using Cryptkit)

```
% crypt::option dump

Cert
     COMPLIANCELEVEL = COMPLIANCELEVEL_STANDARD
     UPDATEINTERVAL = 90
     REQUIREPOLICY = COMPLIANCELEVEL_REDUCED
     VALIDITY = 365
     SIGNUNRECOGNISEDATTRIBUTES = COMPLIANCELEVEL_OBLIVIOUS

Cms
     DEFAULTATTRIBUTES = 1

Device
     PKCS11_HARDWAREONLY = 0
     PKCS11_DVR01 =
     PKCS11_DVR02 =
     PKCS11_DVR03 =
     PKCS11_DVR04 =
     PKCS11_DVR05 =

Encr
     MAC = ALGO_HMAC_SHA
     ALGO = ALGO_3DES
     HASH = ALGO_SHA

Info
     MINORVERSION = 1
     STEPPING = 1
     MAJORVERSION = 3
     COPYRIGHT = Copyright Peter Gutmann, Eric Young, OpenSSL,
                                                          1994-2004
     DESCRIPTION = cryptlib security toolkit

Keying
     ITERATIONS = 500
     ALGO = ALGO_SHA

Keys
     LDAP_OBJECTCLASS = inetOrgPerson
     LDAP_EMAILNAME = mail
     LDAP_CRLNAME = certificateRevocationList;binary
     LDAP_OBJECTTYPE = 0
     LDAP_CACERTNAME = cACertificate;binary
…
```

In summary, the V2 utility procedures provide a more convenient and simplified way to access the underlying Cryptlib functionality for common tasks, without precluding the use of the underlying API for more complex tasks.

And since they are scripted in Tcl, it is likely that their number and scope will improve over time as more experience is gained with using Cryptlib from Tcl.

Implementation

Interfacing Cryptlib with Tcl

There are three main ways of interfacing a C library such as Cryptlib with Tcl

- manually create the interface code in C using the Tcl C API
- generate an interface using SWIG (the Simplified Wrapper and Interface Generator) [6]
- use Critcl ("C Runtime in Tcl") [7] to generate an interface

With 65+ functions to wrap the first option wasn't strongly considered.

Both SWIG and Critcl are used to generate interfaces from Tcl to C libraries. SWIG uses a small interface file that specifies which functions are to be wrapped, and generates C code that calls the Tcl C API to perform the binding. When compiled, the resulting shared library can be loaded into a Tcl interpreter and the C functions called from Tcl.

Critcl uses a different approach – C code is embedded in a Tcl script and transparently compiled into a shared library that is installed in a Tcl package. At runtime the library for the current platform is loaded when the calling application invokes "package require".

Critcl was chosen for a number of reasons

- the author's involvement with the Critcl project and resulting level of familiarity
- the Tcl specific nature of Critcl
- the opportunity to further improve Critcl as a facility for wrapping and binding to C libraries from Tcl

But rather than build an interface procedure for each of the Cryptlib functions, the decision was taken to try a higher level approach – develop a "little language" that describes the various Cryptlib functions and their arguments. This description is used to generate Critcl code, which in turn builds Cryptkit.

Cryptlib is particularly suited to this approach because it uses integer values for symbolic constants and object names, and all Cryptlib functions return an integer that indicates success (i.e. CRYPT_OK) or failure (i.e. CRYPT_ERROR). Any return values are set via pointer (i.e. reference) arguments, with additional arguments specifying the maximum length and actual length for void* return values.

So function arguments fall into just a few categories:

- integer values corresponding to Cryptlib symbols (e.g. CRYPT_CONTEXT)
- integer values representing Cryptlib encapsulated object identifiers
- char* arguments (e.g. a null-terminated character string holding a password)
- void* arguments (e.g. a character array holding random data) with an associated integer argument specifying the length of the data
- integer and char* return arguments
- void* return arguments, with an associated length and maximum length integer argument

Armed with this information, we can begin to describe the Cryptlib API in our little language.

Describing Cryptlib functions

Cryptlib functions are defined by a “generate” procedure within the Cryptkit source code. This accepts a list of function names and arguments.

For example

```
generate {
    AddPrivateKey    { keyset cryptKey password }
    AddPublicKey     { keyset certificate }
    AddRandom        { randomData randomDataLength }
    …
}
```

The first field is the name of each function, which is followed by a list of arguments. Note that the actual Cryptlib function names are prefixed by “crypt”, and the Tcl procedures that access them are defined in the ::crypt namespace, for example:

```
::crypt::AddPrivateKey calls cryptAddPrivateKey
```

Note also that the above description doesn’t include the type of each argument. This is specified by optionally adding a single character “suffix modifier” to each argument definition. The common suffix modifiers are

*	const char* argument
=	const void* argument, next argument is the length
&	return integer
%	return void*, next argument is the length being returned
@	const int argument that is absent from Tcl procedures calls and always set to CRYPT_UNUSED when calling C functions

For example

```
generate {
    AddPrivateKey    { keyset cryptKey password*             }
    AddPublicKey     { keyset certificate                    }
    AddRandom        { randomData= randomDataLength          }
    …
    CreateCert       { certificate+ cryptUser@ certType      }
    …
    PopData          { envelope buffer% length bytesCopied&  }
}
```

So, the “little language” is just an invocation of the “generate” procedure, enumerating each function and its arguments, with suffix modifiers to specify the argument types.

Variable length argument data

There are a number of Cryptlib functions that return pieces of data of varying lengths.

For example, cyptCreateSignature is used to digitally sign a piece of data and it returns a signature in a buffer specified in the function call.

Within Cryptlib such functions can be called with a NULL argument and Cryptlib will return the length of the buffer required, without actually generating the data. Also, a number of function arguments are integer variables that will be set to the length of the data being returned by Cryptlib, and so don't need to be in the Tcl API

Similarly, there are a number of functions that accept void * arguments with a companion integer argument to specify the length of the data. Since the length of all Tcl data can be obtained at run time, it makes sense for the interface to insert the length automatically.

To support these features we use introduce four additional suffix modifiers

:	Instructs Cryptkit to call the function first with a NULL argument, allocate a return buffer of appropriate size and then call the function again
!	Tells Cryptkit that the argument should not appear in the Tcl procedure interface, and that instead Cryptkit should pass a predefined maximum length value (arbitrarily set to 32000)
?	Tells Cryptkit that the variable will be set by Cryptlib, and doesn't need to be in the Tcl API
#	Tells Cryptkit to use the length of the previous void* argument (i.e. the length of the Tcl ByteArray corresponding to the void* argument)

For example, the description of cryptCreateSignature and cryptCheckSignature is

```
generate {

   CheckSignature  { signature= length# sigCheckKey hashContext }
   CreateSignature { signature: maxlength! signatureLength?
                                signContext hashContext }

}
```

When invoking these from Tcl

- from crypt::CheckSignature omit the *length* argument
- from crypt::CreateSignature omit the *maxlength* and *signatureLength* arguments

Cryptlib objects

Several functions accept integer arguments that are object (i.e. envelope) identifiers within Cryptlib and, discussed previously, Cryptlib V2 encapsulates these within an object command.

These need special handling in Cryptkit V2 via the following suffix modifiers

+	creates a Cryptkit object and returns a handle comprising the argument name concatenated with the Cryptlib envelope ID (e.g. cert123)
-	if the argument is an integer then it is assumed to be a Cryptlib envelope ID and passed straight to Cryptlib, otherwise Cryptkit tries to find an object of this name and passes the object's envelope ID to Cryptlib (e.g. cert123 will pass 123)
_	same as "-" except it destroys the corresponding Tcl object before returning from the procedure

This allows us to add descriptions such as

```
generate {
    CreateCert       { certificate+ cryptUser@ certType      }
    AddPublicKey     { keyset- certificate-                  }
    AddPrivateKey    { keyset- cryptKey- password*           }
    DestroyCert      { cryptCert_                            }
}
```

Taking the CreateCert command as an example, this is invoked as follows

```
% set cert [crypt::CreateCert CERTTYPE_CERTIFICATE]
% $cert set CERTINFO_SELFSIGNED 1
% $cert set CERTINFO_CA 1
% puts $cert
certificate395
```

So Cryptkit has created an object command “certificate395” which corresponds to the Cryptlib certificate object whose ID is 395.

Cryptlib symbols

There are a number of Cryptlib symbols that need to be made available to the Cryptkit program (e.g. CRYPT_CERTTYPE_CERTIFICATE). These symbols are defined by a #define or enum in the cryptlib.h header file.

These were made available to the Tcl programmer by extending Critcl with a new critcl::cdefines command to allow mapping between C defines and Tcl variables. For example:

```
critcl::cdefines CRYPT_* ::crypt
```

This command tells Critcl to make Cryptlib symbols beginning with CRYPT_ available as variables within the Tcl ::crypt namespace – both with and without the leading CRYPT_.

Critcl does this by pre-processing any C code in cryptlib.tcl script using the appropriate C compiler preprocessor flag (e.g. gcc –E). The preprocessed C code is scanned for #defines and enums, and the Tcl C API function calls are generated to set Tcl variables to the corresponding integer values. When the resulting shared library is loaded by the Cryptkit package it sets these variables within the crypt namespace.

This will allow a program to reference a Cryptlib symbol like any other Tcl variable, for example:

```
crypt::CreateCert $CERTTYPE_CERTIFICATE
```

But it is more convenient to just to use the symbol name (i.e. without de-referencing the Tcl variable) when calling a Cryptkit command, so we introduce another suffix modifier

^ the integer argument is the name of a Cryptlib #define/enum

which allows us to declare CreateCert as

```
generate {
    CreateCert       { certificate+ cryptUser@ certType^     }
```

If the argument is an integer it is passed throught to the Cryptlib function, otherwise it is assumed to be a symbol name and the value of the corresponding Tcl variable is passed through.

This allows us to invoke CreateCert via

```
crypt::CreateCert CERTTYPE_CERTIFICATE
```

Generated C code

The description of the Cryptlib API is used to generate interface Critcl ccommand procedures (i.e. C embedded in Tcl, with access to arguments via the Tcl C Object API).

For example, the description of the crypt::CreateCert function:

```
CreateCert       { certificate+ cryptUser@ certType^    }
```

results in the following Critcl commands being generated

```
critcl::ccommand CreateCert {data ip objc objv} {
  int ret;
  int arg1;
  int arg2;
  int arg3;
  Tcl_Obj *obj3;
  if (objc != 2) {
    Tcl_WrongNumArgs(ip, 1, objv, "certType");
    return TCL_ERROR;
  }
  arg2 = CRYPT_UNUSED;
  if (GetIntOrConstArg(ip, objv[1], &arg3) == TCL_ERROR) {
    Tcl_AddErrorInfo(ip, "\" (argument 1 to CreateCert)");
    return TCL_ERROR;
  }
  ret = cryptCreateCert(&arg1, arg2, arg3);
  if (ret < CRYPT_OK)
    return cryptkitError(ip, ret, 0);
  cryptkitNewObj(ip, Tcl_NewStringObj("certificate",-1),
                     Tcl_NewIntObj(arg1));
  return TCL_OK;
}
```

Points to note include

- the Cryptlib cryptCreateCert function call (highlighted)
- the helper functions GetInOrConstArg, cryptkitError and cryptkitNewObj

- arguments and return values are handled via Tcl Objects
- the first argument (arg1) is returned as a Tcl object (cryptkitNewObj sets the return value) as per the + suffix modifier
- arg2 (is set to CRYPT_UNUSED, as per its description with the @ suffix modifier
- the third argument is either an integer or symbol as per the ^ suffix modifier

Building Cryptkit

Building Cryptkit is a two-stage process

- first you build a static Cryptlib library (the details of which vary by platform and are beyond the scope of this paper)
- then you build Cryptkit using Critcl,

The result is a cross-platform Tcl package that can be included into your application to make it “Cryptkit enabled”.

The Cryptkit source is a single file – cryptkit.tcl. It assumes a copy (or link to) the cryptlib.h header is in the same directory/folder. Likewise you need a copy of the static Cryptlib library for each platform you want to build on. And finally you need a copy of the Critcl program, downloadable from the Critcl home page [7].

Once all these are available, on each platform you build Cryptkit using:

```
$ critcl –pkg cryptkit
```

If your source directory is cross-mounted to multiple machines then the lib/cryptkit directory will have the Cryptkit shared library for each platform added to it. This directory adheres to the Tcl “package” convention and so is ready for use in any Tcl application.

Cryptkit has been deployed on Linux (both x86 and ARM), MacOS X and Windows. It should be portable to any platform supported by Cryptlib (which seems to be most platforms) and Critcl (anywhere a GNU toolchain is available, and then some).

The Future

Future developments include providing access to non-Cryptlib functionality. In particular, this will include interfaces to the Tcllib cryptographic modules from within the Cryptkit utility procedures, including the pure Tcl implementations of AES, DES, MD4, MD5, MD5crypt and SHA1.

Another development will be support for Tcl stacked channels. Stacked channels allow transformations or filters to be applied to I/O channels (both files and socket). This allows the filters to intercept all read/write activity on the channel and modify the data. This will allow Cryptkit functionality to be applied to existing Tcl channels, combining the full power of Cryptkit with Tcl’s powerful event driven I/O model.

And finally, there is a planned implementation of the recently endorsed X9.95 Trusted Time Stamp Management and Security standard.

Licensing

Cryptkit is released under a Tcl-friendly BSD style Open Source license.

As mentioned previously, the underlying Cryptlib security toolkit is distributed under a dual license – it is free for non-commercial use and commercial licenses are available. Developers wishing to use Cryptkit in commercial products should ensure that they review the Cryptlib license available at [2].

Conclusion

Cryptkit has proven useful at two levels

- it has provided a cryptographic extension for Tcl, that is both comprehensive and "Tcl friendly"
- it has been led to the development of new ways of wrapping C libraries for use as Tcl extensions

And along the way, the project has led to further improvements to the already useful Critcl utility.

It's certainly not perfect, and in many ways the rough edges are yet to be "knocked off" as it begins to be used in "real" projects.

But by any measure, it's been worthwhile. And fun[1].

Acknowledgements

Thanks are due to Mike Doyle of Eolas Technologies Inc, for the initial Cryptkit concept, the name and the ongoing sponsorship of the development.

Thanks also to Pat Thoyts for his help in building a Windows version of Cryptlib, and for his general suggestions, support and encouragement. And for Jean-Claude Wippler's help in building the Windows port.

Likewise Jeff Hobbs who has provided ongoing insight into several deep technical issues – in particular on the quality of the Tcl C API code generated.

Steve Redler IV assisted with the Linux/ARM port, as well as providing feedback and suggestions. As did Donal Fellows, Andreas Kupries and several other people in the Tcl community.

Mark Roseman, Donal Fellows, Jean-Claude Wippler and Mike Doyle reviewed the draft of this paper.

And finally Peter Gutmann, for producing Cryptlib and for his assistance during the development of Cryptkit.

[1] For some definition of "fun"

References

[1] *The 11th Annual Tcl/Tk Conference* - http://www.tcl.tk/community/tcl2004/

[2] *Cryptlib* - http://www.cs.auckland.ac.nz/~pgut001/cryptlib/

[3] *TLS* - http://tls.sourceforge.net/ and http://wiki.tcl.tk/tls

[4] *Tcllib* – http://tcllib.sourceforge.net

[5] *The Cryptlib Security Toolkit Version 3.2*, auth. Peter Gutmann, pg 2 ftp://ftp.franken.de/pub/crypt/cryptlib/manual.pdf

[6] *SWIG – the Simplified Wrapper and Interface Generator* - http://swig.sourceforge.net/

[7] *Critcl – C Runtime in Tcl* - http://www.equi4.com/critcl.html

Appendix 1 – Cryptlib C API Description

```
# function arguments
#   default is "const int"
# suffix modifiers to alter this
#   ^   arg is name of #define/enum in cryptkit namespace
#   *   char *
#   =   void *
#  object handling
#   +   creates Cryptkit object and returns handle comprising argument
#       name+cryptlib number (e.g. cert123)
#   -   if integer then pass to Cryptlib, otherwise tries to find
#       Cryptkit object and passes integer value to cryptlib
#   _   same as -, excepts destroys the corresponding Tcl object
# the following indicate values to be returned from the Tcl proc
#   &   return int
#   %   return void *, next arg is length
#   :   return void *, call with NULL to get length of string
# the following arguments are not in the Tcl API, only the C API
#   #   use length of previous void *
#   !   max length - always follows :
#   ?   actual length returned by C API
#   @   set to CRYPT_UNUSED in C API

generate {
   AddCertExtension    { certificate- oid* criticalFlag extension=
                             extensionLength# }
   AddPrivateKey       { keyset- cryptKey- password* }
   AddPublicKey        { keyset- certificate- }
   AddRandom           { randomData= randomDataLength^ }
   AsyncCancel         { cryptObject- }
   AsyncQuery          { cryptObject- }
   CAAddItem           { keyset- certificate- }
   CACertManagement    { certificate+ action keyset- caKey-
                         certRequest- }
   CAGetItem           { keyset- certificate+ certType keyIDtype keyID=
                       }
   CheckCert           { certificate- sigCheckKey- }
   CheckSignature      { signature= length# sigCheckKey- hashContext- }
   CheckSignatureEx    { signature= length# sigCheckKey- hashContext-
                             extraData& }
   CreateCert          { certificate+ cryptUser@ certType^ }
```

```
    CreateContext       { context+ cryptUser@ cryptAlgo^ }
    CreateEnvelope      { envelope+ cryptUser@ formatType^ }
    CreateSession       { session+ cryptUser@ sessionType^ }
    CreateSignature     { signature: maxlength! signatureLength?
                              signContext- hashContext- }
    CreateSignatureEx   { signature: maxlength! signatureLength?
                          formatType signContext- hashContext-
                          extraData- }
    Decrypt             { cryptContext- buffer= length# }
    DeleteAttribute     { cryptObject- attributeType }
    DeleteCertExtension { certificate- oid* }
    DeleteKey           { cryptObject- keyIDtype keyID= }
    DestroyCert         { cryptCert_ }
    DestroyContext      { cryptContext_ }
    DestroyEnvelope     { cryptEnvelope_ }
    DestroyObject       { cryptObject_ }
    DestroySession      { cryptSession_ }
    DeviceClose         { device }
    DeviceCreateContext { cryptDevice context+ cryptAlgo }
    DeviceOpen          { device& cryptUser@ deviceType name* }
    DeviceQueryCapability { # handled explicitly by C function }
    Encrypt             { cryptContext% buffer= length# }
    End                 { }
    ExportCert          { certObject: maxlength! certObjectLength?
                              certFormatType^ certificate- }
    ExportKey           { encryptedKey: maxlength! encryptedKeyLength?
                              exportKey- sessionKeyContext- }
    ExportKeyEx         { encryptedKey: maxlength! encryptedKeyLength?
                              formatType exportKey- sessionKeyContext- }
    FlushData           { cryptHandle- }
    GenerateKey         { cryptContext- }
    GenerateKeyAsync    { cryptContext- }
    GetAttribute        { # handled explicitly by C function }
    GetAttributeString  { cryptObject- attributeType^ value:
                          valueLength? }
    GetCertExtension    { certificate- oid* criticalFlag& extension:
                              maxlength! extensionLength? }
    GetPrivateKey       { cryptHandle- context+ keyIDtype^ keyID*
                              password* }
    GetPublicKey        { cryptObject- publicKey+ keyIDtype keyID= }
    ImportCert          { certObject= certObjectLength cryptUser@
                              certificate+ }
    ImportKey           { encryptedKey= maxlength# importContext-
                              sessionKeyContext- }
    Init                { # handled explicitly by Tcl procedure }
    KeysetClose         { keyset_ }
    KeysetOpen          { keyset+ cryptUser@ keysetType^ name* options^
                        }
    PopData             { envelope- buffer% length bytesCopied? }
    PushData            { envelope- buffer= length# bytesCopied& }
    QueryCapability     { # handled explicitly by C function }
    QueryObject         { # handled explicitly by C function }
    SetAttribute        { # handled explicitly by C function }
    SetAttributeString  { cryptObject^ attributeType^ value=
                          valueLength# }
    SignCert            { certificate- signContext- }
}
```