tomcrypt(3) 0.8.0 | libtomcrypt Tcl wrapper

Cyan Ogilvie

TOMCRYPT

libtomcrypt Tcl wrapper - use cryptographic primitives in Tcl scripts

SYNOPSIS

package require tomcrypt ?0.8.0?

tomcrypt::hash *algorithm bytes*
tomcrypt::hmac *algorithm key message*
tomcrypt::hkdf *algorithm salt info in length*
tomcrypt::base64url encode|strict_encode *bytes*
tomcrypt::base64url decode|strict_decode *string*
tomcrypt::encrypt *spec key iv bytes*
tomcrypt::decrypt *spec key iv bytes*
tomcrypt::ecc_generate_key *curve ?prng?*
tomcrypt::ecc_extract_pubkey *privkey*
tomcrypt::ecc_ansi_x963_import *bytes ?curve?*
tomcrypt::ecc_ansi_x963_export *pubkey*
tomcrypt::ecc_verify *sig message pubkey*
tomcrypt::ecc_sign *privkey message ?prng?*
tomcrypt::ecc_shared_secret *privkey pubkey*
tomcrypt::rsa_make_key ?**-keysize** *bits*? ?**-exponent** *e*? ?**-prng** *prng*?
tomcrypt::rsa_extract_pubkey *privkey*
tomcrypt::rsa_sign_hash -key *privkey* **-hash** *hash* ?**-padding** *type*? ?**-hashalg** *algorithm*? ?**-saltlen** *bytes*? ?**-prng** *prng*?
tomcrypt::rsa_verify_hash -key *pubkey* **-sig** *signature* **-hash** *hash* ?**-padding** *type*? ?**-hashalg** *algorithm*? ?**-saltlen** *bytes*?
tomcrypt::rsa_encrypt_key -key *pubkey* **-msg** *message* ?**-padding** *type*? ?**-hashalg** *algorithm*? ?**-lparam** *label*? ?**-prng** *prng*?
tomcrypt::rsa_decrypt_key -key *privkey* **-ciphertext** *ciphertext* ?**-padding** *type*? ?**-hashalg** *algorithm*? ?**-lparam** *label*?
tomcrypt::rng_bytes *count*
tomcrypt::prng create *prngInstance type ?entropy?*
tomcrypt::prng new *type ?entropy?*

PRNG instance methods:

prngInstance **bytes** *count*
prngInstance **add_entropy** *entropy*
prngInstance **integer** *lower upper*
prngInstance **double**
prngInstance **export**
prngInstance **destroy**

DESCRIPTION

This package provides a thin wrapper around a subset of libtomcrypt's functionality.

COMMANDS

tomcrypt::hash *algorithm bytes* Return the hash of *bytes*, using the *algorithm*. The values available for *algorithm* are those that are known by libtomcrypt. The returned value is the raw bytearray.

tomcrypt::hmac *algorithm key message* Compute the HMAC (Hash-based Message Authentication Code) of *message* using the hash *algorithm* and *key*. The *algorithm* must be one of the hash algorithms known to libtomcrypt (like sha256). Both *key* and *message* must be byte arrays. Returns the HMAC result as a raw byte array.

tomcrypt::hkdf *algorithm salt info in length* Perform HKDF (HMAC-based Extract-and-Expand Key Derivation Function) using the specified *algorithm* (a hash algorithm known to libtomcrypt, like sha256), with the given *salt* (a bytearray, can be empty), *info* (a bytearray, can be empty), and input keying material *in* (a bytearray). The derived key will be *length* bytes long. Returns the derived key as a raw bytearray.

tomcrypt::base64url encode|strict_encode *bytes* Return the base64url encoding of *bytes*, which is the same as the regular base64 encoding except for two substitutions: '+' -> '-' and '/' -> '_', so that the result can be represented in a URL part without needing to be escaped. Also useful when using the result as a filename. If **strict_encode** is used, then the result will have '=' padding characters appended to ensure that its length is a multiple of 4. **encode** does not pad its output.

tomcrypt::base64url decode|strict_decode *string* Inverts the encoding applied by **encode** or **strict_encode**. Both **decode** and **strict_decode** accept both padded and unpadded input, but strict does not allow pad characters or characters outside of the valid base64url alphabet within the encoded value.

tomcrypt::encrypt *spec key iv data* Encrypt the plaintext bytes in *data* using the key *key* using the cipher and mode specified in *spec*. See **CIPHER**

SPEC for details.

tomcrypt::decrypt *spec key iv data* Decrypt the ciphertext bytes in *data* using the key *key* using the cipher and mode specified in *spec*. See **CIPHER SPEC** for details.

tomcrypt::aead encrypt *mode cipher key iv aad plaintext* Encrypt *plaintext* using authenticated encryption with associated data (AEAD). The *mode* can be one of: **gcm**, **eax**, **ocb**, **ocb3**, **ccm**, or **chacha20poly1305**. Most modes require a *cipher* (like "aes"), except chacha20poly1305 which uses its own cipher (pass "" for cipher in that case). The *key*, initialization vector *iv*, and additional authenticated data *aad* are all byte arrays. *aad* can be empty if no metadata needs to be authenticated. Returns a 2-element list: {ciphertext tag} where tag is the authentication tag (typically 16 bytes).

tomcrypt::aead decrypt *mode cipher key iv aad ciphertext tag* Decrypt *ciphertext* using AEAD, verifying the authentication *tag*. Parameters must match those used during encryption. Returns the plaintext if successful, or throws an error if the tag verification fails (indicating tampering or corruption).

tomcrypt::ecc_generate_key *curve ?prng?* Generate a new ECC key using the PRNG instance *prng* (defaulting to the system PRNG if not specified), using the specified *curve*. See **CURVE SPEC** for details of how to specify the curve. Returns the private key in PEM formatted OpenSSL compatible DER format.

tomcrypt::ecc_extract_pubkey *privkey* Extract the public key from an ECC private key *privkey* (in OpenSSL's DER format, possibly PEM encoded, as returned by **ecc_generate_key**). Returns the public key in OpenSSL DER format, PEM encoded.

tomcrypt::ecc_verify *sig message pubkey* Verify the signature *sig* over the message *message* with public key *pubkey*. *sig* is in ANSI X9.62 format, *pubkey* is in ANSI X9.63 section 4.3.6 format or the native libtomcrypt format, and message is the raw bytearray (typically a hash result) that was signed. Returns true if the signature is valid, false if not, and throws an error if it couldn't parse *sig* or *pubkey*.

tomcrypt::ecc_sign *privkey message ?prng?* Sign *message* using the private key *privkey* (in openssl's DER format (possibly PEM encoded), as returned by **ecc_generate_key**). If *prng* is provided, use that PRNG instance for the signing operation, otherwise use the system's secure random number generator. Returns the signature in ANSI X9.62 format, suitable for verification with **ecc_verify**.

tomcrypt::ecc_shared_secret *privkey pubkey* Compute an ECDH shared secret between the local private key *privkey* and the remote public key *pubkey*. Both parties compute the same shared secret value, which can

be used to derive encryption keys. Returns the raw x-coordinate of the shared elliptic curve point in binary format (conforms to EC-DH from ANSI X9.63). The shared secret should be passed through a key derivation function like **hkdf** before being used as a key.

tomcrypt::rsa_make_key ?-keysize *bits*? ?-exponent *e*? ?-prng *prng*? Generate a new RSA keypair. The **-keysize** option specifies the key size in bits (must be a multiple of 8 between 1024 and 4096, defaults to 2048). The **-exponent** option sets the public exponent (defaults to 0x10001). The **-prng** option specifies a PRNG instance to use; if omitted, the system's secure random number generator is used. Returns the private key in PKCS#1 PEM format. The private key is suitable for use with **rsa_sign_hash** and **rsa_decrypt_key**. Use **rsa_extract_pubkey** to derive the corresponding public key for use with **rsa_verify_hash** and **rsa_encrypt_key**.

tomcrypt::rsa_extract_pubkey *privkey* Extract the public key from an RSA private key *privkey* (in PKCS#1 DER/PEM format). Returns the public key in PKCS#1 PEM format, suitable for use with **rsa_verify_hash** and **rsa_encrypt_key**.

tomcrypt::ecc_ansi_x963_import *bytes* ?*curve*? Import an ECC public key from ANSI X9.63 section 4.3.6 format (a bytearray starting with 0x04 followed by the x and y coordinates). If *curve* is provided, it specifies the curve to use (see **CURVE SPEC**), otherwise the curve is inferred from the length of the x and y coordinates (only works with uncompressed format). Returns the public key suitable for use with **ecc_verify** and **ecc_shared_secret**.

tomcrypt::ecc_ansi_x963_export *pubkey* Export an ECC public key *pubkey* to ANSI X9.63 section 4.3.6 format (a bytearray starting with 0x04 followed by the x and y coordinates).

tomcrypt::rsa_sign_hash -key *privkey* -hash *hash* ?-padding *type*? ?-hashalg *algorithm*? ?-saltl
Sign a message hash using the RSA private key *privkey* (in PKCS#1 DER/PEM format, as returned by **rsa_make_key** or from other sources). The *hash* should be the raw bytes of a message digest. The **-padding** option specifies the padding scheme: **v1.5** for PKCS#1 v1.5, **pss** for PSS (default), or **v1.5_na1** for v1.5 without ASN.1 encoding (for SSL 3.0 compatibility). The **-hashalg** option only applies to **pss** and is the name of the hash function to use for that padding (e.g., "sha1", "sha256", defaults to "sha256"). For PSS padding, **-saltlen** specifies the salt length in bytes (defaults to 0). If **-prng** is provided, use that PRNG instance, otherwise use the system's secure RNG (only applicable to **pss**). Returns the signature as raw bytes.

tomcrypt::rsa_verify_hash -key *pubkey* -sig *signature* -hash *hash* ?-padding *type*? ?-hashalg *alg*
Verify an RSA signature *signature* over the message hash *hash* using the public key *pubkey* (in PKCS#1 DER/PEM format). The **-padding**

(defaults to "pss"), **-hashalg** (defaults to "sha256"), and **-saltlen** (defaults to 0) options must match those used during signing. Returns true if the signature is valid, false otherwise.

tomcrypt::rsa__encrypt__key -key *pubkey* -msg *message* ?-padding *type*? ?-hashalg *algorithm*? ?
Encrypt a short message using the RSA public key *pubkey* (in PKCS#1 DER/PEM format). The **-padding** option can be **v1.5** for PKCS#1 v1.5 or **oaep** for OAEP padding (default). For OAEP, the **-hashalg** option (only applicable for **oaep** specifies the hash to use (defaults to "sha256") he **-lparam** option is an optional label for OAEP (can be empty bytes). If **-prng** is provided, use that PRNG instance, otherwise use the system's secure RNG. Returns the encrypted bytes. Message size limits depend on key size and padding: for a 2048-bit key with OAEP/SHA-256, the maximum message size is about 190 bytes.

tomcrypt::rsa__decrypt__key -key *privkey* -ciphertext *ciphertext* ?-padding *type*? ?-hashalg *algo*
Decrypt RSA-encrypted data using the private key *privkey* (in PKCS#1 DER/PEM format). The **-padding** (defaults to "oaep"), **-hashalg** (defaults to "sha256"), and **-lparam** options must match those used during encryption. Returns the decrypted message bytes, or throws an error if decryption fails or padding is invalid.

tomcrypt::prng create *prngInstance type ?entropy?* Create a PRNG (pseudorandom number generator) instance accessed by the command name *prngInstance*, using the implementation *type*, such as **fortuna** or **chacha20** (as known to libtomcrypt), or "" (an empty string) to select the recommended default which may change between releases, and bootstrapped with *entropy* which must be a bytearray of high entropy bytes. If *entropy* is omitted the PRNG will be bootstrapped with at least 256 bits of entropy from the platform's default cryptographic RNG. Returns the *prngInstance* command name.

tomcrypt::prng new *type ?entropy?* As above, but the *prngInstance* command name is picked automatically.

PRNG INSTANCE METHODS

prngInstance* bytes *count Retrieve *count* random bytes from the PRNG. Returned as a raw bytearray.

prngInstance* add__entropy *entropy Add entropy to the PRNG, given as a bytearray *entropy*, which should come from a high quality source of random bytes such as the platform's secure RNG or a previously exported state by *prngInstance* **export**.

prngInstance* integer *lower upper Generate a random integer between *lower* and *upper*, inclusive, with uniform distribution. Either *lower* or *upper*, or both, may be bignums, and negative, but *lower* must be <= *upper*.

***prngInstance* double** Generate a random double precision floating point value in the range [0, 1) (inclusive of the lower bound but not the upper). The result is picked from a set of 2**53 discrete values, with uniform distribution and equal resolution (uniformly spaced) across the range. The gap between each discrete value is 2**-53. This subset - 2/1023 of the possible doubles in [0, 1) - is the largest subset that satisfies the uniform resolution requirement. See [1] for a discussion of the nuances of random floating point values.

***prngInstance* export** Export entropy, returning the random bytearray. Intended to preserve entropy across PRNG instances and reduce the demands on scarce platform entropy. To do that, supply the result of this command to the *entropy* argument when creating a new PRNG instance.

***prngInstance* destroy** Destroy the instance. After returning, the *prngInstance* command no longer exists and all resources are released. Renaming the instance command to {} is equivalent.

CIPHER SPEC

The choice of cipher and mode for encrypting and decrypting is given by a list of 3 or 4 elements: *cipher*, *keysize*, *mode*, and *mode_opt* (if the mode takes options).

cipher is a name of a symmetric cipher supported by libtomcrypt, such as "blowfish", "aes", etc. *keysize* is the size of the key (in bits). *mode* is the streaming mode, such as "cbc", "ctr", etc. Choose "ctr" if you don't have a good reason not to.

CURVE SPEC

The curve for ECC operations can be specified by any of the names libtomcrypt understands: by name like **secp256r1**, an alias like **P-256**, or the OID like **1.2.840.10045.3.1.7**. Custom curves can also be specified by a dictionary of parameters with the following keys: - **prime** - **A** - **B** - **order** - **Gx** - **Gy** - **cofactor** (optional, defaults to 1) - **OID** (optional)

EXAMPLES

Print out the hex-encoded md5 of "hello, tomcrypt" (normally, when hashing strings, they should be converted to an encoding like utf-8 first, but this example leaves that out for simplicity's sake):

```
puts [binary encode hex [tomcrypt::hash md5 "hello, tomcrypt"]]
```

Verify an ECC signature:

[1] Goualard F. Generating Random Floating-Point Numbers by Dividing Integers: A Case Study. Computational Science – ICCS 2020. 2020 Jun 15;12138:15–28. doi: 10.1007/978-3-030-50417-5_2. PMCID: PMC7302591.

```
set verified    [tomcrypt::ecc_verify \
    [binary decode base64 MEUCIQDr/iC/fbEVKDydJ6/Jw95f53b6SGOXo7dMQtVGR48lMQIgeSKKZOph5MMqq
    [binary decode hex 41091b1b32c6cd42f06b36f72801e01915bd99115f120c119ef7b781f7140dda] \
    [binary decode hex 046ddc90ba0fd79c53bd70060192211631d11ec581302e91c3559df4b20cdf747dbd
]
if {$verified} {
    puts "signature is valid"
} else {
    puts "signature is not valid"
}
```

Create a Fortuna PRNG with automatic entropy bootstrapping and use it to generate 10 random bytearrays:

```
tomcrypt::prng create csprng fortuna
for {set i 0} {$i < 10} {incr i} {
    puts "random bytes $i: [binary encode hex [csprng bytes 8]]"
}
csprng destroy
```

Preserve scarce platform entropy between runs, and leave the choice of the PRNG implementation up to the library, and mix in 8 bytes of entropy from the platform RNG every 10 minutes:

```
proc readbin filename {
    set h [open $filename rb]
    try {read $h} finally {close $h}
}

proc writebin {filename bytes} {
    set h [open $filename wb]
    try {puts -nonewline $h $bytes} finally {close $h}
}

# Bootstrap using saved entropy if we have it
set saved_entropy_filename  somefile.bin
if {[file exists $saved_entropy_filename]} {
    tomcrypt::prng create csprng {} [readbin $saved_entropy_filename]
} else {
    tomcrypt::prng create csprng {}
}

# Save entropy for next time
writebin $saved_entropy_filename [csprng export]

# Mix in entropy periodically
coroutine background_add_entropy eval {
    trace add command csprng delete [list [info coroutine] done]
```

```
    while 1 {
        after [expr {10 * 60 * 1000}] [list [info coroutine] stir]
        switch -- [lindex [yield] 0] {
            stir    { csprng add_entropy [tomcrypt::rng_bytes 8] }
            done    { break }
            default { error "expecting stir or done" }
        }
    }
}

# Generate a random 256 bit integer
set key [csprng integer 0 [expr {2**256-1}]]

# Enter the event loop
if {![info exists exit]} {
    vwait exit
}
exit $exit
```

Generate a new ECC keypair and use it to sign a message:

```
tomcrypt::prng create rand fortuna
set msg      [binary decode hex 41091b1b32c6cd42f06b36f72801e01915bd99115f120c119ef7b781f714
set privkey  [tomcrypt::ecc_generate_key secp256r1 rand]
set pubkey   [tomcrypt::ecc_extract_pubkey $privkey]
set sig      [tomcrypt::ecc_sign $privkey $msg]
set verified [tomcrypt::ecc_verify $sig $msg $pubkey]
if {$verified} {
    puts "signature verified successfully"
} else {
    puts "signature verification failed"
}
rand destroy
```

Compute HMAC-SHA256:

```
set key     [binary decode hex 0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b]
set message "Hi There"
puts [binary encode hex [tomcrypt::hmac sha256 $key $message]]
```

Encrypt and decrypt with AES-GCM authenticated encryption:

```
# Generate a random key and IV
set key [tomcrypt::rng_bytes 32]  ;# 256-bit AES key
set iv  [tomcrypt::rng_bytes 12]  ;# 96-bit IV (recommended for GCM)
set aad "user-id:12345,timestamp:2024-01-15"  ;# Metadata to authenticate
set plaintext "Secret message that needs both confidentiality and authenticity"

# Encrypt: returns {ciphertext tag}
```

```
lassign [tomcrypt::aead encrypt gcm aes $key $iv $aad $plaintext] ciphertext tag

# Store or transmit: $ciphertext, $tag, and $aad
# The IV can be transmitted in the clear (but must not be reused with the same key)

# Decrypt and verify
set decrypted [tomcrypt::aead decrypt gcm aes $key $iv $aad $ciphertext $tag]
puts "Decrypted: $decrypted"

# Tag verification failure (tampering detected)
set bad_tag [string repeat "\x00" 16]
if {[catch {tomcrypt::aead decrypt gcm aes $key $iv $aad $ciphertext $bad_tag} err]} {
    puts "Authentication failed: $err"
}
```

Generate RSA keypair and create CloudFront-style signature (PKCS#1 v1.5 + SHA-1):

```
set privkey [tomcrypt::rsa_make_key]
set pubkey [tomcrypt::rsa_extract_pubkey $privkey]

# CloudFront policy string
set policy {{"Statement":[{"Resource":"http://example.com/*","Condition":{"DateLessThan":{"A
set hash [tomcrypt::hash sha1 [encoding convertto utf-8 $policy]]

# Sign with PKCS#1 v1.5 and SHA-1
set signature [tomcrypt::rsa_sign_hash -key $privkey -hash $hash -padding v1.5 -hashalg sha1

# Verify signature
set valid [tomcrypt::rsa_verify_hash -key $pubkey -sig $signature -hash $hash -padding v1.5
if {$valid} {
    puts "CloudFront signature verified successfully"
} else {
    puts "CloudFront signature verification failed"
}
```

RSA encryption and decryption with OAEP padding:

```
set privkey [tomcrypt::rsa_make_key]
set pubkey [tomcrypt::rsa_extract_pubkey $privkey]

set message "Secret message for RSA encryption"
set msgbytes [encoding convertto utf-8 $message]
set lparam "MyApplication"
set lparambytes [encoding convertto utf-8 $lparam]

# Encrypt with OAEP padding using SHA-256
set ciphertext [tomcrypt::rsa_encrypt_key -key $pubkey -msg $msgbytes -padding oaep -hashalg
```

```
# Decrypt
set decrypted [tomcrypt::rsa_decrypt_key -key $privkey -ciphertext $ciphertext -padding oaep
set decrypted_message [encoding convertfrom utf-8 $decrypted]

puts "Original: $message"
puts "Decrypted: $decrypted_message"
```

RSA signature with PSS padding:

```
set privkey [tomcrypt::rsa_make_key]
set pubkey [tomcrypt::rsa_extract_pubkey $privkey]

set message "Document to be signed with PSS"
set hash [tomcrypt::hash sha256 [encoding convertto utf-8 $message]]

# Sign with PSS padding and salt length 32
set signature [tomcrypt::rsa_sign_hash -key $privkey -hash $hash -padding pss -hashalg sha25

# Verify signature
set valid [tomcrypt::rsa_verify_hash -key $pubkey -sig $signature -hash $hash -padding pss -
if {$valid} {
    puts "PSS signature verified successfully"
} else {
    puts "PSS signature verification failed"
}
```

BUILDING

This package has no external dependencies other than Tcl. The libtom libraries it depends on are included as submodules (or baked into the release tarball) and are built and statically linked as part of the package build process.

Currently Tcl 8.7 is required, but if needed polyfills could be built to support 8.6.

From a Release Tarball

Download and extract the release, then build in the standard TEA way:

```
wget https://github.com/cyanogilvie/tcl-tomcrypt/releases/download/v0.8.0/tomcrypt0.8.0.tar.
tar xf tomcrypt0.8.0.tar.gz
cd tomcrypt0.8.0
./configure
make
sudo make install
```

From the Git Sources

Fetch the code and submodules recursively, then build in the standard autoconf / TEA way:

```
git clone --recurse-submodules https://github.com/cyanogilvie/tcl-tomcrypt
cd tcl-tomcrypt
autoconf
./configure
make
sudo make install
```

In a Docker Build

Build from a specified release version, avoiding layer pollution and only adding the installed package without documentation to the image, and strip debug symbols, minimising image size:

```
WORKDIR /tmp/tcl-tomcrypt
RUN wget https://github.com/cyanogilvie/tcl-tomcrypt/releases/download/v0.8.0/tomcrypt0.8.0.
    ./configure; make test install-binaries install-libraries && \
    strip /usr/local/lib/libtomcrypt*.so && \
    cd .. && rm -rf tcl-tomcrypt
```

For any of the build methods you may need to pass `--with-tcl /path/to/tcl/lib` to `configure` if your Tcl install is somewhere nonstandard.

Testing

Since this package deals with security sensitive code, it's a good idea to run the test suite after building (especially in any automated build or CI/CD pipeline):

```
make test
```

And maybe also the memory checker `valgrind` (requires that Tcl and this package are built with suitable memory debugging flags, like `CFLAGS="-DPURIFY -Og" --enable-symbols`):

```
make valgrind
```

SECURITY

Given the limitations of a scripting language environment, this package's code does not have sufficient control over freed memory contents (or memory paged to disk) to guarantee that key material or other sensitive material (like decrypted messages) can't leak in a way that could be exploited by other code running on the shared memory (or disk) machine. For this reason, careful consideration should be given to the security requirements of the application as a whole when using this package in a shared execution context, or in a virtual machine. That

said, operations that do not rely on secret values (like verifying cryptographic signatures) safe in these shared environments.

FUZZING

TODO

AVAILABLE IN

The most recent release of this package is available by default in the `alpine-tcl` container image: docker.io/cyanogilvie/alpine-tcl and the `cftcl` Tcl runtime snap: https://github.com/cyanogilvie/cftcl.

SEE ALSO

This package is built on the libtomcrypt library, the libtommath library, and tomsfastmath.

PROJECT STATUS

This is a work in progress, but the commands documented here are implemented and tested and the package is in limited production use. The ECC related functions are not yet production ready.

With the nature of this package a lot of care is taken with memory handling and test coverage. There are no known memory leaks or errors, and the package is routinely tested by running its test suite (which aims at full coverage) through valgrind. The `make valgrind`, `make test` and `make coverage` build targets support these goals.

SOURCE CODE

This package's source code is available at https://github.com/cyanogilvie/tcl-tomcrypt. Please create issues there for any bugs discovered.

LICENSE