tomcrypt(3) 0.6.0 | libtomcrypt Tcl wrapper

Cyan Ogilvie

TOMCRYPT

libtomcrypt Tcl wrapper - use cryptographic primitives in Tcl scripts

SYNOPSIS

package require tomcrypt ?0.6.0?

tomcrypt::hash *algorithm bytes*
tomcrypt::hmac *algorithm key message*
tomcrypt::base64url encode|strict_encode *bytes*
tomcrypt::base64url decode|strict_decode *string*
tomcrypt::ecc_make_key *prng keysize*
tomcrypt::ecc_verify *sig message pbkey*
tomcrypt::ecc_sign *privkey message ?prng?*
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::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 outsite of the valid base64url alphabet within the encoded value.

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::ecc_make_key *prng keysize* Generate a new ECC keypair using the PRNG instance *prng* with the given *keysize* in bytes. Returns a two-element list containing the private key in libtomcrypt's internal format and the public key in ANSI X9.63 format. The private key is suitable for use with **ecc_sign** and the public key with **ecc_verify**.

tomcrypt::ecc_verify *sig message pbkey* Verify the signature *sig* over the message *message* with public key *pbkey*. *sig* is in ANSI X9.62 format, *pbkey* 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 *pbkey*.

tomcrypt::ecc_sign *privkey message ?prng?* Sign *message* using the private key *privkey* (in libtomcrypt's internal format, as returned by **ecc_make_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::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.

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
lassign [tomcrypt::ecc_make_key rand 32] privkey pubkey
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]]
```

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/v0.6.0/tomcrypt0.6.0.tar.
tar xf tomcrypt0.6.0.tar.gz
cd tomcrypt0.6.0
./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/v0.6.0/tomcrypt0.6.0.
    ./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 very early work in progress. Currently all that is implemented and tested are the **hash** and **ecc_verify** commands. More to come soon.

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

an author's ability to disclaim copyright this package can be used under the terms of the CC0, BSD, or MIT licenses. No attribution, permission or fees are required to use this for whatever you like, commercial or otherwise, though I would urge its users to do good and not evil to the world.