Programming with Crypticl 0.2

Tåle Skogan*

February 17, 2007

Contents

1 Introduction

This is a short introduction to the Crypticl cryptography library written in and for Common Lisp. This document focus on examples showing typical tasks. The unit tests for each algorithm is a further source of examples. To test the examples yourself load Crypticl and then change into the package with `in-package`:

```
cl-user(2):(load "C:\\crypticl\\src\\load.lisp")
...
cl-user(3): (in-package crypticl)
#<The crypticl package>
crypticl(5):
```

The `crypticl` package also has a short nickname `clc` and you can use it instead of the full name:

```
cl-user(3): (in-package clc)
#<The crypticl package>
clc(5):
```

*email: no.frisurf@tasko

The examples use the two utility functions `hex` and `hexo` to make binary output more readable. `hex` takes an octet vector (byte array) and returns a string representation in hex. `hexo` is the reverse, it takes a hex string and returns the octect vector equivalent. Both functions are part of the library.

2 Hash functions

Create a SHA-1 object:

```
crypticl(4): (setf obj (new-instance 'SHA-1))
#<SHA1 @ #x211285fa>
```

The new-instance function is a factory method used to generate instances of all the algorithms. A SHA-256 object can for example be created with:

```
(new-instance 'SHA-256)
```

Compute SHA-1 hash of a byte vector:

```
crypticl(5): (hash obj #(97 98 99))
#(169 153 62 54 71 6 129 106 186 62 ...)
```

Compute SHA-1 hash of a string and print the 160 bits output as a hexstring:

```
crypticl(8): (hex (hash obj (string-to-octets "abc")))
"a9993e364706816aba3e25717850c26c9cd0d89d"
```

Add bytes to the object multiple times and compute a hash at the end:

```
crypticl(12): (update obj (string-to-octets "ab"))
nil
crypticl(13): (update obj (string-to-octets "c"))
nil
crypticl(14): (hex (hash obj))
"a9993e364706816aba3e25717850c26c9cd0d89d"
```

Implementation note: There is a semantic difference between calling `hash` on a `Hash` object with no data and calling `hash` on an empty byte vector. Calling `hash` on an empty object is more likely to be a user error and hence returns `nil`. Calling `hash` on an empty byte vector on the other hand, may simply mean that we got very short input and hence returns the initial state of the SHA-1 algorithm (which is a valid 160 bits byte vector).

The object oriented interface introduced above is built on top of low level function primitives for each algorithm. Sometimes it's easier to work directly with them. To get the SHA1 hash of a stream (typically a file) use `sha1-on-octet-stream`:

```
crypticl(15): (with-open-file (s "rsa.lisp")
                (hex (sha1-on-octet-stream s)))
"fe5f55ea902e3fd3b6875fb35c87c3f368d37660"
```

3 Random numbers

Handling random numbers correctly is vital for almost all crypto primitives. I recommend studying chapter 10 in [1] before using the random number api of Crypticl (or any crypto library for that matter). Two important factors are a cryptographically secure pseudorandom number generator and a source of high entropy bits for seeding the generator. Crypticl uses 256-bits AES in counter mode as the number generator (based on the Fortuna design from [1]). The function `random-secure-octets` returns an octet vector with random bits:

```
crypticl(16): (random-secure-octets 16)
#(146 37 34 245 50 193 238 169 54 139 ...)
```

Before using any primitives involving keys or other random data you must seed the pseudorandom number generator with high entropy bits. On Linux the generator in Crypticl will seed itself using `/dev/random`, but on Windows you must seed the generator yourself with 256 bits of entropy using the api call `reseed-secure-prng`:

```
crypticl(17): (reseed-secure-prng seed)
#<SecurePRNG-AES @ #x209099f2>
```

The seed must be an octet vector or a bignum. Furthermore you may need to reseed the generator depending on how you use it (see [1]). Note: The handling of entropy and reseeding is weak and brittle in the current (0.2) version of Crypticl and it is very easy to compromise security if you make a mistake. So be extremely careful.

4 Symmetric key encryption

The `Cipher` class provides common functionality for symmetric and asymmetric algorithms used for encryption. Subclasses of the `Cipher` class must support the following methods:

`init-encrypt`: Initializes the `Cipher` object for encryption. Arguments may include the key and mode to use.

`init-decrypt`: Initializes the `Cipher` object for decryption. Arguments may include the key and mode to use.

`update`: Updates the state of the `Cipher` object. This means adding encrypted data for decryption or cleartext for encryption.

`encrypt`: Applies padding and encrypts any leftover data.

`decrypt`: Decrypts any leftover data and removes the padding.

The class hierarchy looks like this:

```
Crypto
    Cipher
        SymmetricCipher
            AES
            IDEA
        AsymmetricCipher
            RSA
    Hash
        SHA-256
        SHA1
        MD5
    Signature
        DSA
```

To use a symmetric encryption scheme like IDEA or AES, start by getting an instance of the algorithm:

```
crypticl(37): (setf obj (new-instance 'AES))
#<AES @ #x20d98f5a>
```

Then generate the key:

```
crypticl(38): (setf aeskey (generate-key 'AES 128))
<SymmetricKey value:#X4de0df0ea7818446b264d0f8ee07965f>
```

Initialize the algorithm object for encryption with the key:

```
crypticl(45): (init-encrypt obj aeskey)
```

Than transform the cleartext to an octet-vector and encrypt it:

```
crypticl(48): (hex (encrypt obj #(1 2 3)))
"b92a901ceb2d6da3f74cfafcc8bc4064"
```

Note that although the input is only a 3 byte vector the output is a 16 byte long vector because of padding. To decrypt, initialize the object for decryption with the same key used for encryption and call `decrypt`:

```
crypticl(49): (init-decrypt obj aeskey)
crypticl(50): (decrypt obj (hexo "b92a901ceb2d6da3f74cfafcc8bc4064"))
#(1 2 3)
```

The next example is more advanced and sets both the iv and mode before doing encryption and decryption in several steps using the update function.

```
crypticl(126): (setf obj (new-instance 'AES))
#<AES @ #x20e58522>
crypticl(129): (init-encrypt obj aeskey :mode 'cbc :iv #16(0))
0
```

The CBC mode is the default mode and the only safe mode implemented. The default iv is #16(0). In the following we encrypt the input stream a bit at a time, a typical thing to do for large files or network streams. Each call to update returns the cryptotext for the section of cleartext given as input, minus possibly a residue modulo the block size. The encryption (or decryption) must be closed with a call to `encrypt`(`decrypt`). This call will empty any buffered cleartext from previous calls to update, add padding and return the last of the cryptotext.

```
crypticl(131): (init-encrypt obj aeskey :mode 'cbc :iv #16(0))
0
crypticl(134): (hex (update obj #16(1)))
"3de24b30d69b6a0a607e0bb74016e0b0"
crypticl(135): (hex (update obj #7(2)))
""
```

Note how this call didn't return any cryptotext because update didn't add enough cleartext to fill a whole block of cryptotext.

```
crypticl(136): (hex (encrypt obj))
"e32e05ea9f3d9c40c12431c3ef77afbb"
crypticl(137): (init-decrypt obj aeskey :mode 'cbc :iv #16(0))
0
crypticl(138): (hex (update obj (hexo "3de24b30d69b6a0a607e0bb74016e0b0")))
"01010101010101010101010101010101"
crypticl(139): (hex (decrypt obj (hexo "e32e05ea9f3d9c40c12431c3ef77afbb")))
"02020202020202"
```

5 Digital signatures

We create a DSA object and generate a keypair. The keypair object contain a private key for signing data and a public key for verifying signatures.

```
crypticl(148): (setf obj (new-instance 'DSA))
#<DSA @ #x215d9faa>
crypticl(149): (setf dsakey (generate-key 'DSA))
Generating DSA keys, this may take some time...
#<DSAKeyPair @ #x218390ba>
crypticl(150): :in dsakey
DSAKeyPair @ #x218390ba = #<DSAKeyPair @ #x218390ba>
0 Class --------> #<standard-class DSAKeyPair>
1 public -------> <DSAPublicKey>))
2 private ------> <DSAPrivateKey>
```

Initialize the DSA object with the private key for signing:

```
crypticl(155): (init-sign obj (private dsakey))
...
```

Sign the data. The sign function returns the signature as a list of two numbers:

```
crypticl(156): (setf signature (sign obj #16(1)))
(610205237490270933520572927751741211804151194981
814606745356376522186211750303275432244290651385)
```

We can verify a singature by initializing the DSA object with the public key and give the data and the signature as input too verify:

```
crypticl(158): (verify obj signature #16(1))
t
```

6 Diffie-Hellman

The following functions illustrates the Diffie-Hellman interface. The result will be list with two secrets which should be equal. Each of the two Diffie-Hellman objects dh1 and dh2 represents the two endpoints in a secure exchange of a common secret.

```
(defun test-dh ()
  (let (half-secret-1
        half-secret-2
        ;; secret-1 and secret-2 should be equal in the end
        secret-1
        secret-2
        dh1
        dh2
        key-1
        key-1-copy)
    (setf dh1 (make-Diffie-Hellman))
    (setf dh2 (make-Diffie-Hellman))
    (setf key-1 (generate-key 'Diffie-Hellman 64))
    ;; Make a copy of the key. We need a copy because the init function
    ;; stores state in the key object.
    (setf key-1-copy (copy key-1))
    (init-Diffie-Hellman dh1 key-1)
    (init-Diffie-Hellman dh2 key-1-copy)
    (setf half-secret-1 (generate-random-Diffie-Hellman dh1))
    (setf half-secret-2 (generate-random-Diffie-Hellman dh2))
    (setf secret-1 (get-secret-Diffie-Hellman dh1 half-secret-2))
    (setf secret-2 (get-secret-Diffie-Hellman dh2 half-secret-1))
    (list secret-1 secret-2)))

crypticl(189): (test-dh)
(8431410348096402792 8431410348096402792)
```

References

[1] Ferguson, Niels and Schneier, Bruce. 2003. Practical Cryptography. Wiley.