***RSA**Euro*
TECHNICAL REFERENCE

RSAEuro Technical Reference

by Nick Barron <nikb@cix.compulink.co.uk>

Release history:

June 1995:	First edition
October 1995:	Second edition
November 1996:	Third edition

TABLE OF CONTENTS

SECTION I GENERAL INFORMATION

DIGITAL SIGNATURE ROUTINES

ENVELOPE PROCESSING

PEM FUNCTIONS

SECTION II ALGORITHMS

Section II Algorithms Public Key Algorithms

KEY GENERATION

RSA

DIFFIE-HELLMAN

Section II Algorithms Secret Key Algorithms

DES

SECTION III TECHNICAL DESCRIPTION

NATURAL NUMBER ARITHMETIC

MEMORY MANIPULATION

TECHNICAL INFORMATION

APPENDICES

SECTION I
GENERAL INFORMATION

INTRODUCTION

What is RSAEURO?

RSAEURO is a cryptographic toolkit providing various functions for the use of digital signatures, data encryption and supporting areas (PEM encoding, random number generation *etc.*). To aid compatibility with existing software, RSAEURO is call-compatible with RSADSI's “RSAREF” toolkit. RSAEURO allows non-US residents to make use of much of the cryptographic software previously only (legally) available in the US.

> **IMPORTANT NOTICE:** Please do not distribute or use this software in the US – it is *illegal* to use this toolkit in the US, as public-key cryptography is covered by US patents (see the Patents and Trademarks section below for details). If you are a US resident, please use the RSAREF toolkit instead.

RSAEURO contains support for the following:

- RSA encryption, decryption and key generation. Compatible with RSA Laboratories' Public-Key Cryptography Standard (PKCS) #1.
- Generation and verification of message digests using MD2, MD4, MD5 and SHS (SHS currently not implemented in higher-level functions to maintain compatibility with PKCS).
- DES encryption and decryption using cipher block chaining (CBC), with 1, 2 or 3 keys using Encrypt-Decrypt-Encrypt, and DESX, RSADSI's secure DES enhancement.
- Diffie-Hellman key agreement as defined in PKCS #3.
- PEM support – support for RFC 1421 encoded ASCII data with all main functions.
- Key routines implemented in assembler for speed (80386, 680x0 and SPARC currently supported).

What is covered by this document?

This document provides a function-by-function description of the RSAEURO toolkit, at a sufficient level of detail to allow the use of the toolkit within other software (example code fragments are included to clarify the use of the various functions). The internal workings of the functions are not described. For full details of the internal workings of the RSAEURO routines, please consult the (well commented) source code.

The sample code included in the documentation is for demonstration purposes only, and does not necessarily illustrate the ideal use of the functions for all applications. In some cases error handling code has not been included in order to simplify the examples. All sample code was produced and tested using the Gnu C compiler.

It is assumed that the reader is familiar with C programming and basic cryptography, although a detailed knowledge is not required.

How is this document organised?

This document is divided into three main sections, described in the following paragraphs:

- **Section I: General Information**
  - *Introduction (This section).* General introduction to RSAEURO.
  - *Random numbers.* Routines for generating cryptographically-secure random numbers, for use by various other cryptographic functions.
  - *Message digests.* Routines for the creation and verification of message digests.
  - *Digital signatures.* Routines for the creation and verification of digital signatures.

- *Envelope processing.* Routines for the creation and use of digital "envelopes" (an "envelope" is a structure containing encrypted data and an optional digital signature).

- *PEM functions.* Routines for processing Interned privacy-enhanced mail (PEM) encoded messages.

- **Section II: Algorithms**

  Public Key Algorithms

  - *Key generation.* Routines for generating key material for RSA encryption

  - *RSA.* Routines for public key encryption and decryption using RSA and PKCS#1.

  - *Diffie-Hellman*. Routines for exchanging keys via Diffie-Hellman agreement.

  Secret Key Algorithms

  - *DES.* Routines for secret-key encryption and decryption using DES in CBC mode, with either single or triple-key operation.

- **Section III: Technical Description**.

  - *Natural number arithmetic.* Low-level routines for performing natural number arithmetic.

  - *Memory manipulation.* Platform-specific memory manipulation routines.

  - *Technical information.* "Technical" programming information

- **Appendices**

  - *Appendix A: Function cross-reference.* An alphabetical list of functions with brief details and a reference to coverage in the main documentation.

  - *Appendix B: References*. A selection of references to further information.

Contact information

All general comments should be sent to `rsaeuro@sourcery.demon.co.uk`. Bug reports should be sent to `rsaeuro-bugs@sourcery.demon.co.uk`. Comments or corrections regarding the documentation should be sent to `nikb@cix.compulink.co.uk`.

As a last resort, the author may be contacted by "snail-mail" at the following address:

Stephen Kapp
The Post Office
Nr. Clitheroe
Lancashire
BB7 3BB
UNITED KINDOM

Typographic conventions

Throughout this document, blocks of C source code are set in `courier`, and in-text references to functions, constants and the like are set in **arial bold.** Conventional C-style mathematical operators are used throughout (*e.g.* * for "times", / for "divided by" *etc.*)

Licence

RSAEURO TOOLKIT LICENSE AGREEMENT 15th December 1995

Copyright (c) J.S.A.Kapp, 1994-1995.

1. LICENSE. J.S.A.Kapp grants you a nonexclusive, non-transferable, perpetual (subject to the conditions of section 7) license for the "RSAEURO" toolkit (the "Toolkit") and its associated documentation, subject to all of the following terms and conditions:

   i. To use the Toolkit on any computer in your possession.

   ii. to make copies of the Toolkit for back-up purposes.

   iii. to modify the Toolkit in any manner for porting or performance improvement purposes (subject to Section 2) or to incorporate the Toolkit into other computer programs for your own personal or internal use, provided that you provide J.S.A.Kapp with a copy of any such modification or Application Program by electronic mail, and grant J.S.A.Kapp a perpetual, royalty-free license to use and distribute such modifications and Application Programs on the terms set forth in this Agreement.

   iv. To copy and distribute the Toolkit and Application Programs in accordance with the limitations set forth in Section 2.

   "Application Programs" are programs that incorporate all or any portion of the Toolkit in any form. The restrictions imposed on Application Programs in this Agreement shall not apply to any software which through the mere aggregation on distribution media, is co-located or stored with the Toolkit.

2. LIMITATIONS ON LICENSE.

   i. J.S.A.Kapp owns the Toolkit and its associated documentation and all copyrights therein. You may only use, copy, modify and distribute the Toolkit as expressly provided for in this Agreement. You must reproduce and include this Agreement, J.S.A.Kapp's copyright notices and disclaimer of warranty on any copy and its associated documentation.

   ii. The Toolkit and its associated documentation are freeware for noncommercial purposes, however for commercial purposes please contact J.S.A.Kapp for licensing details.

   iii. The Toolkit and Application Programs are to be used for noncommercial purposes. However, media costs associated with the distribution of the Program or Application Programs may be recovered.

   iv. The Toolkit, if modified, must carry prominent notices stating that changes have been made, and the dates of any such changes. v. Prior permission from J.S.A.Kapp is required for any modifications that access the Toolkit through ways other than the published Toolkit interface or for modifications to the Toolkit interface, and structures. J.S.A.Kapp will grant all reasonable requests for permission to make such modifications.

3. You are solely responsible for all of your costs and expenses incurred in connection with the distribution of the Toolkit or any Application Program hereunder, and J.S.A.Kapp shall have no liability, obligation or responsibility there of. J.S.A.Kapp shall have no obligation to provide maintenance, support, upgrades or new releases to you or to any distributee of the Toolkit or any Application Program.

4. THE TOOLKIT AND ITS ASSOCIATED DOCUMENTATION ARE LICENSED "AS IS" WITHOUT WARRANTY AS TO THEIR PERFORMANCE, MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE TOOLKIT IS ASSUMED BY YOU AND YOUR DISTRIBUTEES. SHOULD THE TOOLKIT PROVE DEFECTIVE, YOU AND YOUR DISTRIBUTEES (AND NOT J.S.A.KAPP) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

5. LIMITATION OF LIABILITY, NEITHER J.S.A.KAPP NOR ANY OTHER PERSON WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THE TOOLKIT SHALL BE LIABLE TO YOU OR TO ANY OTHER PERSON FOR ANY DIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF J.S.A.KAPP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

6. RSAEURO is a publication of cryptographic techniques. Applications developed with RSAEURO may be subject to export controls in some countries. If you are located in the United States and develop such applications using RSAEURO, you are advised to obtain a copy of RSAREF from RSADSI, as you may using RSAEURO infringe on Patents held by RSA Data Security and Cylink.

7. The license granted hereunder is effective until terminated. You may terminate it at anytime by destroying all components of the Toolkit and its associated documentation. The termination of your license will not result in the termination of the licenses of any distributees who have received rights to the Toolkit through you so long as they are in compliance with the provisions of this license.

8. GENERAL

   i. Address all correspondence regarding this license to J.S.A.Kapp's electronic mail address <rsaeuro@sourcery.demon.co.uk>, or to

      Mr J.S.A.Kapp.
      The Post Office,
      Dunsop Bridge,
      Clitheroe,
      Lancashire,
      England.
      BB7 3BB.

      Tel. (+44) 1200-448241

   ii. For details of Export Controls and other controls regarding the use of cryptographic techniques please contact your country's relevant authority.

Patents and trademarks

The following terms are registered trademarks as indicated. All other trademarks acknowledged. To the author's knowledge, all relevant trademarks have been duly acknowledged. If there are any omissions, please provide the relevant details and the documentation will be amended accordingly.

RSADSI

RSA Data Security Inc (RSADSI) provide consultancy and software engineering service in the field of cryptography. The RSAREF toolkit, on which RSAEURO is modelled, is available free of charge from RSADSI with the USA and Canada, subject to their licensing agreement. For information regarding licensing RSADSI's toolkits, contact Paul Gordon (`paul@rsa.com`) at RSADSI.

Following a recent arbitration between RSADSI and Cylink, it has been determined that RSADSI hold patent rights to the RSA public-key cryptographic algorithm. For details of licensing the RSA algorithm contact Paul Livesay (`pol@rsa.com`) at RSADSI.

RSADSI can be contacted at:

RSA Data Security Inc,
10 Twin Dolphin Drive
Redwood City
CA 94065

Tel. (415) 595-8782.

DESX and RSAREF are registered trademarks of RSADSI.

Cylink

At the time of writing it was not possible to determine the legal situation regarding patents regarding cryptography held by Cylink. Interested parties should contact Bob Fougner on (+1) 408 735 5893, e-mail: `fougner@cylink.com`.

Release history

This section describes the changes made at each release of the software.

Release 1.00

First major release.

Release 1.01

Modifications:

- **R_RandomCreate** and **R_RandomMix** modified to improve random number generation. **R_RandomMix** had a minor flaw as it didn't flush the old output from the object.

Release 1.02

Bug fixes:

- Fixed bug in **R_EncodePEMBlock** reported by Wang Wei Jun <wang@iti.gov.sg>.
- Fixed bug in **R_SealUpdate** reported by Anders Heerfordt <i3683@dc.dk>.
- Fixed bug in **dmult**, reported by Anders Heerfordt <i3683@dc.dk>.
- Fixed PADDING[] bug for R_ENHANC.C, reported by Anders Heerfordt.

Release 1.03

Bug fixes:

- **RSAPublicEncrypt**, checks that **RandomStruct** has been initialised prior to use.
- **SHSFinal**, digest now output "directly" to a passed parameter, context cleared on exit.

Release 1.04

Bug fixes:

- Fixed bug in **NN_Encode** and **NN_Decode** routines.
- Added extra seeding code to **R_RandomCreate.**
- IDOK replaced with ID_OK throughout.

Modifications:

- Added **R_RSAEuroInfo** routine.
- Documentation updated to include sample code for most functions.
- Documentation re-structured into sections.

RANDOM NUMBERS

Introduction

Various functions within RSAEURO require random data (primarily for key generation). A stream of random (strictly, pseudo-random) data is generated using the MD5 digest algorithm and a "seed" value, which is provided in the form of the **random** structure.

Before use, the **random** structure must be initialised and "seeded" itself, by "mixing in" an amount of genuine random data. The procedure for preparing a new random structure is as follows:

1 Reserve **sizeof(R_RANDOM_STRUCT)** memory.

2 Initialise the new structure using **R_RandomInit**. This sets **random->bytesNeeded**, the number of random bytes required to "seed" the structure before use, to RANDOM_BYTES_RQ, and zeroes the data.

3 "Mix in" a suitable quantity of random data using **R_RandomUpdate**. **R_RandomUpdate** takes a caller-supplied block of data and combines it with the existing random structure using MD5. **R_RandomUpdate** also decrements **random->bytesNeeded**, which indicates the amount of random data still required (the **R_GetRandomBytesNeeded** function returns the **bytesNeeded** value of a given **random** structure). **R_RandomUpdate** should be called repeatedly until **R_GetRandomBytesNeeded** returns zero.

R_RandomUpdate may be called once the structure has been initialised (*i.e.* **bytesNeeded** equals zero). RANDOM_BYTES_RQ should be adjusted according to the "purity" of the random data source.

An additional function, **R_RandomCreate**, creates and initialises a "fresh" **random** structure using data from the current system clock, via the ANSI **gmtime** function (this function uses a separate variable, RANDOM_BYTES_INT, to indicate the amount of mix-in bytes, currently set to 512).

On ANSI-compliant systems, **R_RandomCreate** can be used as a "one-stop shop" for producing a ready-to-use **random** structure. Other sources of random data, such as keyboard timings, disk latency and so on are highly system-dependant, and have not yet been implemented.

Once a **random** structure has been created, initialised and seeded, it may be used by **R_GenerateBytes** to produce a stream of pseudo-random data. **R_GenerateBytes** returns an error if an invalid (non-seeded) **random** structure is referenced.

The function **R_RandomMix** uses the ANSI **clock** and **time** functions to randomise the current state of an existing, initialised random structure. Then flush any pending output from the output state.

The function **R_RandomFinal** clears a **random** structure.

Functions

R_RandomInit

```
int R_RandomInit(random)R_RANDOM_STRUCT *random;                   /* random structure */
```

Initialises a new **random** structure. Zeroes the data area and sets **random->bytesNeeded** to the system default (RANDOM_BYTES_RQ). Always returns ID_OK.

R_RandomUpdate

```
int R_RandomUpdate(random, block, len)
R_RANDOM_STRUCT *random;                              /* random structure */
unsigned char *block;                                   /* block of data */
unsigned int len;                                     /* length of block */
```

Updates a previously initialised **random** structure by mixing in a block of caller-supplied data using MD5. Updates **random->bytesNeeded** as appropriate. Always returns ID_OK.

R_GetRandomBytesNeeded

```
int R_GetRandomBytesNeeded(bytesNeeded, random)
unsigned int *bytesNeeded                   /* number of mix-in bytes needed */
R_RANDOM_STRUCT *random                               /* random structure */
```

Returns the number of seed bytes still required for the **random** structure. On exit, **bytesNeeded** contains the number of bytes required by the structure **random**. Always returns ID_OK.

R_GenerateBytes

```
int R_GenerateBytes(block, len, random)
unsigned char *block;                                           /* block */
unsigned int len;                                     /* length of block */
R_RANDOM_STRUCT *random;                              /* random structure */
```

Populates **block** with **len** pseudo-random bytes derived from **random** using MD5. Returns RE_NEED_RANDOM if random has not been fully initialised, ID_OK otherwise.

R_RandomFinal

```
void R_RandomFinal(random)
R_RANDOM_STRUCT *random;                              /* random structure */
```

Clears a **random** structure, setting all values and data to zero.

R_RandomCreate

```
void R_RandomCreate(random)
R_RANDOM_STRUCT *random;                              /* random structure */
```

Initialises a **random** structure and seeds it with data derived using the ANSI **gmtime** and **clock** function. The quantity of seeding data is defined by RANDOM_BYTES_RQINT.

R_RandomMix

```
void R_RandomMix(random)
R_RANDOM_STRUCT *random;
```

Randomises the internal state of the supplied **random** structure, using data from the ANSI **clock** and **time** functions, then flushes any pending output.

Data Types

R_RANDOM_STRUCT

```
typedef struct {
  unsigned int bytesNeeded;                        /* seed bytes required */
  unsigned char state[16];                            /* state of object */
  unsigned int outputAvailable;                 /* number byte available */
  unsigned char output[16];                              /* output bytes */
} R_RANDOM_STRUCT;
```

The R_RANDOM_STRUCT type stores the state and characteristics of a random number generator.

bytesNeeded	Number of remaining "mix in" bytes required to initialise the structure (initially defined by RANDOM_BYTES_RQ). Must be zero before the structure may be used.
state	Internal state of the random number generator.
outputAvailable	Indicates the number of unused bytes in the output array. When this value reaches zero, the output array is regenerated.
output	Output of the random number generator.

Examples

Initialising a random structure with supplied data

The following code sample shows the "standard" method of initialising a **random** structure using the R_RandomInit and R_RandomUpdate functions. Note that in the example, zero data is used – this will **not** generate a random stream. The seed data for the random structure should be derived from a genuine random source, such as keyboard timing latency. Methods for obtaining such data are machine-specific and are not detailed in this document.

```
void RandomExample()
{
      /*    For the example, we use a zero value for the "random" data
            This should be replaced by a genuine (platform dependent)
            source of random data
      */
      static unsigned char        seedByte = 0;
      unsigned int                bytesNeeded, i;
      static R_RANDOM_STRUCT      randomStruct;
      unsigned char               randomData[16];

      /*    Initialise a clean random structure           */
      R_RandomInit(&randomStruct);

      /* Initialize with all zero seed bytes, which will not yield an actual
         random number output. */
      bytesNeeded = 1;       /* So we get through the loop */
      while (bytesNeeded != 0) {
            R_RandomUpdate(&randomStruct, &seedByte, 1);
            R_GetRandomBytesNeeded(&bytesNeeded, &randomStruct);
      }

      /* Get some bytes and print random output   */
      R_GenerateBytes(randomData, 16, &randomStruct);
      for( i=0; i<16; i++) {
            printf("%02x:", randomData[i]);
      }
      printf("\n\n");

}
```

Initialising a random structure using the system time

The following code sample shows the use of the **R_RandomCreate** function to create and initialise a random structure, using data from the system clock. Note that it is not necessary to call **R_RandomInit**, all that is required is a blank R_RANDOM_STRUCT.

The system clock is not an ideal source of random data, and as such a larger amount of seed data is required, defined by RANDOM_BYTES_RQINT (default value 512). This value should be adjusted for paranoia.

```
void RandomExample()
{
      unsigned int                      i;
      static R_RANDOM_STRUCT            randomStruct;
      unsigned char                     randomData[16];

      /*    Initialise random structure from system clock        */
      R_RandomCreate(&randomStruct);

      /* Get some bytes and print random output   */
      R_GenerateBytes(randomData, 16, &randomStruct);
      for( i=0; i<16; i++) {
```

```
		printf("%02x:", randomData[i]);	}
	printf("\n\n");
}
```

MESSAGE DIGESTS

Introduction

RSAEURO supports four different message digest algorithms: MD2, MD4, MD5 and Secure Hash Standard (SHS). The current MD2, MD4 and MD5 routines are based on source code made available by RSADSI.

Support for each digest method consists of three basic functions: **init**, which initialises the relevant structures and contexts; **update**, which adds data to the digest, and **final** which "tidies up" and returns the final digest value. To simplify implementation, the digest and signature routines are called via "parent" routines, with the algorithm to be used passed as a parameter.

High-level functions are provided for processing data which is memory-resident. These functions handle all memory allocation, initialisation and processing internally, providing a "one-stop shop" solution. However, as the data to be processed must be resident in memory, the run-time resource requirements of these functions are larger than the init-update-final method.

The procedure for producing a message digest is as follows:

1 Initialise the digest "context", containing the digest generator state, input buffer *etc.*, by calling **R_DigestInit**. The digest type (MD2, MD4, MD5 or SHS) is specified as a parameter to **R_DigestInit**.

2 Process the source data a block at a time, using **R_DigestUpdate**.

3 Produce the final digest value using **R_DigestFinal**.

The **R_Digest** functions act as "wrappers" for the algorithm-specific message digest routines. An additional function, **R_DigestBlock**, may be used for memory-resident data.

Functions

R_DigestInit

```
int R_DigestInit(context, digesttype)
R_DIGEST_CTX *context;                                  /* new context */
int digesttype;                              /* message-digest algorithm */
```

Initialises a context ready for digest production. The R_DIGEST_CTX type is a union structure supporting the different context types required for each message digest algorithm. **context** is a pointer to a "blank" R_DIGEST_CTX structure, **digesttype** indicates the digest algorithm to be used. Currently supported digest types are DA_MD2, DA_MD4, DA_MD5 and DA_SHS. Returns RE_DIGEST_ALGORITHM if an invalid (unsupported) digest algorithm is selected, ID_OK otherwise.

R_DigestUpdate

```
int R_DigestUpdate(context, partIn, partInLen)
R_DIGEST_CTX *context;                                      /* context */
unsigned char *partIn;                               /* next data part */
unsigned int partInLen;                    /* length of next data part */
```

Updates **context** using the appropriate digest algorithm (as indicated by the context) with the supplied data. **partIn** points to the data block, **partInLen** indicates the length of the block in bytes. Returns RE_DIGEST_ALGORITHM if an invalid (unsupported) digest algorithm is selected, ID_OK otherwise.

R_DigestFinal

```
int R_DigestFinal(context, digest, digestLen)
R_DIGEST_CTX *context;                                    /* context */
unsigned char *digest;                             /* message digest */
unsigned int *digestLen;                 /* length of message digest */
```

Produces the final digest value from **context**. On exit, **digest** contains the message digest and **digestLen** indicates length of the digest in bytes. **R_DigestFinal** also zeroes the context to remove any sensitive data from memory. Returns RE_DIGEST_ALGORITHM if an invalid (unsupported) digest algorithm is selected, ID_OK otherwise.

R_DigestBlock

```
int R_DigestBlock(digest, digestLen, block, blockLen, digestAlgorithm)
unsigned char *digest;                             /* message digest */
unsigned int *digestLen;                 /* length of message digest */
unsigned char *block;                                       /* block */
unsigned int blockLen;                            /* length of block */
int digestAlgorithm;                   /* message-digest algorithm */
```

Produces a digest of the data block supplied (pointed to by **block**, **blockLen** bytes long), using the digest algorithm indicated by **digestAlgorithm**. On success, the digest is returned in **digest**, and the length of the digest is indicated by **digestLen**. Context creation, initialisation and clearing is handled internally.

Returns RE_DIGEST_ALGORITHM if an invalid (unsupported) digest algorithm is selected, ID_OK otherwise.

MD2Init

```
void MD2Init(context)
MD2_CTX *context;                                         /* context */
```

Initialises a new MD2 context, ready for digest production. **context** is a pointer to a “blank” MD2_CTX structure.

MD2Update

```
void MD2Update(context, input, inputLen)
MD2_CTX *context;                                         /* context */
unsigned char *input;                                 /* input block */
unsigned int inputLen;                     /* length of input block */
```

Updates the MD2 **context** from the supplied data block (pointed to by **input**, **inputLen** bytes long). **context** must be an MD2_CTX structure which has been initialised using **MD2_init**. No checks are made for context validity.

MD2Final

```
void MD2Final(digest, context)
unsigned char digest[16];                          /* message digest */
MD2_CTX *context;                                         /* context */
```

Produces the final MD2 digest value from **context**. **context** must be an MD2_CTX structure which has been initialised using **MD2_init**. No checks are made for context validity. On exit, **digest** contains the MD2 message digest. **MD2Final** zeroes the context to remove any sensitive data from memory.

MD4Init

```
void MD4Init(context)
MD4_CTX *context;                                         /* context */
```

Initialises a new MD4 context, ready for digest production. **context** is a pointer to a “blank” MD4_CTX structure.

MD4Update

```
void MD4Update(context, input, inputLen)
MD4_CTX *context;                                          /* context */
unsigned char *input;                                  /* input block */
unsigned int inputLen;                       /* length of input block */
```

Updates the MD4 **context** from the supplied data block (pointed to by **input**, **inputLen** bytes long). **context** must be an MD4_CTX structure which has been initialised using **MD4_init**. No checks are made for context validity.

MD4Final

```
void MD4Final(digest, context)
unsigned char digest[16];                           /* message digest */
MD4_CTX *context;                                          /* context */
```

Produces the final MD4 digest value from **context**. **context** must be an MD4_CTX structure which has been initialised using **MD4_init**. No checks are made for context validity. On exit, **digest** contains the MD4 message digest. MD4**Final** zeroes the context to remove any sensitive data from memory.

MD5Init

```
void MD5Init (context)
MD5_CTX *context;                                         /* context  */
```

Initialises a new MD5 context, ready for digest production. **context** is a pointer to a "blank" MD5_CTX structure.

MD5Update

```
void MD5Update(context, input, inputLen)
MD5_CTX *context;                                          /* context */
unsigned char *input;                                  /* input block */
unsigned int inputLen;                       /* length of input block */
```

Updates the MD5 **context** from the supplied data block (pointed to by **input**, **inputLen** bytes long). **context** must be an MD5_CTX structure which has been initialised using **MD5_init**. No checks are made for context validity.

MD5Final

```
void MD5Final (digest, context)
unsigned char digest[16];                           /* message digest */
MD5_CTX *context;                                          /* context */
```

Produces the final MD5 digest value from **context**. **context** must be an MD5_CTX structure which has been initialised using **MD5_init**. No checks are made for context validity. On exit, **digest** contains the MD5 message digest. MD5**Final** zeroes the context to remove any sensitive data from memory.

SHSInit

```
void SHSInit(context)
SHS_CTX *context;                                          /* context */
```

Initialises a new SHS context, ready for digest production. **context** is a pointer to a "blank" SHS_CTX structure.

SHSUpdate

```
void SHSUpdate(context, buffer, count)
SHS_CTX *context;                                          /* context */
BYTE *buffer;                                          /* input block */
int count;                                   /* length of input block */
```

Updates the SHS **context** from the supplied data block (pointed to by **buffer**, **count** bytes long). **context** must be an SHS_CTX structure which has been initialised using **SHS_init**. No checks are made for context validity.

SHSFinal

```
void SHSFinal(digest, context)
char *digest                                          /* digest */
SHS_CTX *context;                                     /* context */
```

Produces the final SHS digest value from **context**, returning the digest in **digest**. **context** must be an SHS_CTX structure which has been initialised using **SHS_init**. No checks are made for context validity. On exit, **digest** contains the SHS message digest, and **context** is cleared.

Data Types

R_DIGEST_CTX

```
typedef struct {
  int digestAlgorithm;                                /* digest type */
  union {                                     /* digest sub-context */
     MD2_CTX md2;
     MD4_CTX md4;
     MD5_CTX md5;
     SHS_CTX shs;
  } context;
} R_DIGEST_CTX;
```

The R_DIGEST_CTX type stores the context for a message digest generation.

digestAlgorithm	The message digest algorithm for the context (DA_MD2, DA_MD4, DA_MD5 or DA_SHS).
context	The algorithm-specific context

MD2_CTX

```
typedef struct {
  unsigned char state[16];                                /* state */
  unsigned char checksum[16];                          /* checksum */
  unsigned int count;                 /* number of bytes, modulo 16 */
  unsigned char buffer[16];                        /* input buffer */
} MD2_CTX;
```

The MD2_CTX type stores the context for an MD2 operation.

state	Internal state machine.
checksum	Checksum (see MD2 source for details).
count	Number of bytes processed, modulo 16.
buffer	Input buffer for data to be processed.

MD4_CTX

```
typedef struct {
  UINT4 state[4];                                  /* state (ABCD) */
  UINT4 count[2];          /* number of bits, modulo 2^64 (lsb first) */
  unsigned char buffer[64];                        /* input buffer */
} MD4_CTX
```

The MD4_CTX type stores the context for an MD4 operation.

state	Internal state machine.
count	Number of bits processed, modulo 2^{64}
buffer	Input buffer for data to be processed.

MD5_CTX

```
typedef struct {
  UINT4 state[4];                                                  /* state (ABCD) */
  UINT4 count[2];                        /* number of bits, modulo 2^64 (lsb first) */
  unsigned char buffer[64];                                        /* input buffer */
} MD5_CTX;
```

The MD5_CTX type stores the context for an MD5 operation.

state	Internal state machine.
count	Number of bits processed, modulo 2^{64}
buffer	Input buffer for data to be processed.

SHS_CTX

```
typedef struct {
  UINT4 digest [5];                                              /* Message digest */
  UINT4 countLo, countHi;                                      /* 64-bit bit count */
  UINT4 data [16];                                              /* SHS data buffer */
} SHS_CTX;
```

The SHS_CTX type stores the context for an MD5 operation.

state	Internal state machine.
countLo	Number of bits processed, least significant part.
countHi	Number of bits processed, most significant part.
data	Input buffer for data to be processed.

Examples

Generating a message digest

The following code sample shows the "standard" method of generating a message digest using **R_DigestInit**, **R_DigestUpdate** and **R_DigestFinal**. The example uses a simple string as input data, processing the digest in 1-byte blocks. In practice, larger blocks would be used, typically the buffer size of a file stream.

Note that the digest algorithm is only specified when initialising the context – the **R_DigestUpdate** and **R_DigestFinal** functions determine the algorithm from the context. The use of these functions provides greater reliability than calling the lower-level functions (e.g. **MD5_Init**, **MD5_Update** and **MD5_Final**).

```
void DigestExample()
{
      R_DIGEST_CTX          md5ctxt;
      char                  demostring[] = "this is   a sample string";
      unsigned char         digestOut[MAX_DIGEST_LEN];
      unsigned int          stringlength, i, digestLen;

      /* Initialise MD5 context   */
      if (R_DigestInit(&md5ctxt, DA_MD5) != ID_OK) {
            printf("Error: Invalid digest type passed to R_DigestInit!\n");
            return; }

      /*    Update the digest context a byte at a time           */
      stringlength = strlen(demostring);
      for (i=0; i<stringlength; i++) {
            R_DigestUpdate(&md5ctxt, &demostring[i], 1);        }

      /*    Finalise digest context and print final value        */
      R_DigestFinal(&md5ctxt,   digestOut, &digestLen);
      for( i=0; i<digestLen; i++) {
            printf("%02x:", digestOut[i]);
      }
      printf("\n\n");
}
```

Generating a message digest of memory-resident data

The following code sample shows the use of the **R_DigestBlock** function to generate a digest from memory-resident data.

```
void DigestExample()
{
      R_DIGEST_CTX           md5ctxt;
      char                   demostring[] = "this is a sample string";
      unsigned char          digestOut[MAX_DIGEST_LEN];
      unsigned int           stringlength, i, digestLen;

      /*    Digest the demo string using DigestBlo   ck and display the result         */
      R_DigestBlock(digestOut, &digestLen, demostring, stringlength, DA_MD5);
      for( i=0; i<digestLen; i++) {
            printf("%02x:", digestOut[i]);
      }
      printf("\n\n");
}
```

DIGITAL SIGNATURE ROUTINES

Introduction

RSAEURO provides support for digital signatures using MD2, MD4 and MD5 digests (to maintain compliance with PKCS #1, SHS cannot be used for digital signatures, as it produces a 160 bit digest).

Signature generation consists of three basic functions: **init**, which initialises the relevant structures and contexts, **update**, which adds data to the digest, and **final** which "tidies up", generates the final digest value, and encrypts it using the sender's secret key to produce the signature. To simplify implementation, the signature routines are called via "parent" routines, with the digest algorithm required passed as a parameter.

High-level functions are provided for processing data which is memory-resident. These functions handle all memory allocation, initialisation and processing internally, providing a "one-stop shop" solution. However, as the data to be processed must be resident in memory, the run-time resource requirements of these functions are larger than the init-update-final method.

The procedure for producing a digital signature is as follows:

1. Initialise the signature context by calling **R_SignInit**. The message digest type required is passed as a parameter. To maintain PKCS #1 compatibility, SHS is rejected by **R_SignInit** as a digest type.

2. Process the source data a block at atime using **R_SignUpdate**.

3. Produce the final signature using **R_SignFinal**. The sender's private key is passed as a parameter. RSA is the only algorithm supported for signatures.

An additional function, **R_SignBlock**, may be used to produce a signature for memory-resident data.

RSAEURO also provides routines for verifying a supplied signature. The procedure is as follows:

1. Initialise the signature context by calling **R_VerifyInit**. The message digest type required is passed as a parameter. To maintain PKCS #1 compatibility, SHS is rejected by **R_SignInit** as a digest type.

2. Process the source data a block at a time using **R_VerifyUpdate**.

3. Produce the final digest and verify it against a supplied signature using **R_VerifyFinal**. The sender's public key is passed as a parameter, to allow the decryption of the supplied signature. RSA is the only algorithm supported for signatures.

An additional function, **R_VerifyBlock**, may be used to verify a signature for memory-resident data.

Functions

R_SignInit

```
int R_SignInit(context, digesttype)
R_SIGNATURE_CTX *context;                                   /* new context */
int digesttype;                                 /* message-digest algorithm */
```

Initialises a digest context ready for signature production. The R_SIGNATURE_CTX type is a union structure supporting the different context types required for each message digest algorithm. **context** is a pointer to a "blank" R_SIGNATURE_CTX structure, **digesttype** indicates the digest algorithm to be used. Returns RE_DIGEST_ALGORITHM if an invalid digest type is specified (such as SHS), ID_OK otherwise.

R_SignUpdate

```
int R_SignUpdate(context, partIn, partInLen)
R_SIGNATURE_CTX *context;                                          /* context */
unsigned char *partIn;                                       /* next data part */
unsigned int partInLen;                            /* length of next data part */
```

Updates **context** using the appropriate digest algorithm (as indicated by the context) with the supplied data (**partInLen** bytes from **partIn**). Returns RE_DIGEST_ALGORITHM if an invalid digest type is specified (such as SHS), ID_OK otherwise.

R_SignFinal

```
int R_SignFinal(context, signature, signatureLen, privateKey)
R_SIGNATURE_CTX *context;                                          /* context */
unsigned char *signature;                                        /* signature */
unsigned int *signatureLen;                            /* length of signature */
R_RSA_PRIVATE_KEY *privateKey;                     /* signer's RSA private key */
```

Produces a signature from the supplied context and private key. The digest value is first calculated using **R_DigestFinal** and the **context**, and this value is then encrypted using RSA with **privatekey**. The encrypted value is returned in **signature**, and the length of the signature is returned in **signatureLen**. Returns RE_PRIVATE_KEY if the private key is invalid, RE_DIGEST_ALGORITHM if an invalid digest type is specified (such as SHS), ID_OK otherwise.

R_SignFinal "restarts" the signature context, ready for re-use, and clears all sensitive information.

R_SignBlock

```
int R_SignBlock(signature, signatureLen, block, blockLen,digestAlgorithm, privateKey)
unsigned char *signature;                                        /* signature */
unsigned int *signatureLen;                            /* length of signature */
unsigned char *block;                                                /* block */
unsigned int blockLen;                                     /* length of  block */
int digestAlgorithm;                               /* message-digest algorithm */
R_RSA_PRIVATE_KEY *privateKey;                     /* signer's RSA private key */
```

Produces a signature for the data block supplied (pointed to by **block**, **blockLen** bytes long). **digestAlgorithm** indicates the required message digest algorithm, **privateKey** is the sender's RSA private key. On success, returns ID_OK, **signature** contains the generated signature, **signatureLen** indicates the length in bytes of the signature. On error, returns RE_DIGEST_ALGORITHM if an invalid digest algorithm is selected or RE_PRIVATE_KEY if the private key is invalid.

R_VerifyInit

```
int R_VerifyInit(context, digesttype)
R_SIGNATURE_CTX *context;                                      /* new context */
int digesttype;                                    /* message-digest algorithm */
```

Initialises **context** ready for signature verification. The R_SIGNATURE_CTX type is a union structure supporting the different context types required for each message digest algorithm. **context** is a pointer to a "blank" R_SIGNATURE_CTX structure, **digesttype** indicates the digest algorithm to be used. Returns RE_DIGEST_ALGORITHM if an invalid digest type is specified (such as SHS), ID_OK otherwise.

R_VerifyUpdate

```
int R_VerifyUpdate(context, partIn, partInLen)
R_SIGNATURE_CTX *context;                                          /* context */
unsigned char *partIn;                                       /* next data part */
unsigned int partInLen;                            /* length of next data part */
```

Updates **context** using the appropriate digest algorithm (as indicated by the context) with the supplied data (**partInLen** bytes from **partIn**). Returns RE_DIGEST_ALGORITHM if an invalid digest type is specified (such as SHS), ID_OK otherwise.

R_VerifyFinal

```
int R_VerifyFinal(context, signature, signatureLen, publicKey)
R_SIGNATURE_CTX *context;                                          /* context */
unsigned char *signature;                                        /* signature */
unsigned int *signatureLen;                            /* length of signature */
R_RSA_PRIVATE_KEY *privateKey;                     /* signer's RSA public key */
```

Verifies the supplied signature against the digest produced from the supplied context. Returns zero for success, RE_LEN if the supplied signature is too long (greater than MAX_SIGNATURE_LEN), RE_PUBLIC_KEY if the supplied public key cannot decrypt the signature correctly, RE_SIGNATURE if the message digests do not match or RE_DIGEST_ALGORITHM if an invalid digest type is specified (such as SHS).

R_VerifyBlockSignature

```
int R_VerifyBlockSignature( block, blockLen, signature, signatureLen,
                            digestAlgorithm, publicKey)
unsigned char *block;                                                /* block */
unsigned int blockLen;                                     /* length of block */
unsigned char *signature;                                        /* signature */
unsigned int signatureLen;                            /* length of sig nature */
int digestAlgorithm;                              /* message-digest algorithm */
R_RSA_PUBLIC_KEY *publicKey;                       /* signer's RSA public key */
```

Verifies the signature of a memory-resident data block (pointed to by **block**, **blockLen** bytes long). **digestAlgorithm** indicates the required message digest algorithm, **publicKey** is the sender's RSA public key, **signature** points to the signature to verify and **signatureLen** indicates the length of the signature in bytes. RE_DIGEST_ALGORITHM if an invalid digest algorithm is selected. On success, returns zero. On error, returns RE_DIGEST_ALGORITHM if an invalid digest algorithm is selected, RE_LEN if the supplied signature is too long (greater than MAX_SIGNATURE_LEN), RE_PUBLIC_KEY if the supplied public key cannot decrypt the signature correctly or RE_SIGNATURE if the message digests do not match.

Data Types

R_SIGNATURE_CTX

```
typedef struct {
     R_DIGEST_CTX digestContext;
} R_SIGNATURE_CTX;
```

The R_SIGNATURE_CTX type stores the context for a signature generation. Currently, R_SIGNATURE_CTX is the same as R_DIGEST_CTX, although it has been separately typed for future revisions.

R_RSA_PRIVATE_KEY

The R_RSA_PRIVATE_KEY type stores an RSA private key. See page 43 for a detailed description.

R_RSA_PUBLIC_KEY

The R_RSA_PUBLIC_KEY type stores an RSA public key. See page 42 for a detailed description.

Examples

Signing and verifying a block of data

The following code sample shows the “standard” method of generating a digital signature using **R_SignInit**, **R_SignUpdate** and **R_SignFinal**. The example uses a simple string as input data, processing the digest in 1-byte blocks. In practice, larger blocks would be used, typically the buffer size of a file stream. The function **KeyGenExample** generates an RSA key pair, and is described in detail on page 39.

```
void DigSigExample()
{
	R_RANDOM_STRUCT		randomStruct;
	R_RSA_PROTO_KEY		protoKey;
	R_RSA_PUBLIC _KEY		publicKey;
	R_RSA_PRIVATE_KEY		privateKey;
	R_SIGNATURE_CTX		sigctx;
	char			demostring[] = "This is a sample string to sign";
	char			signature[MAX_SIGNATURE_LEN];
	int			stringlength, signaturelength, status, i;

	/*	Generate keys		*/
	KeyGenExample(&publicKey, &privateKey);

	/*	Sign test string using privateKey and MD5	*/

	/*	Initialise signature structure		*/
	if (R_SignInit(&sigctx, DA_MD5) != 0)
	{
		printf("R_SignInit failed with invalid digest type\n");
		return;
	}

	/*	Add dat a a block (in this case, 1 byte) at a time		*/
	stringlength = strlen(demostring);
	for (i=0; i<stringlength; i++) {
		R_SignUpdate(&sigctx, &demostring[i], 1);
	}

	/*	Finalise signature					*/
	status = R_SignFinal(&sigctx, signature, &signaturelength, &privateKey);
	if (status) {
		printf("R_SignFinal failed with %x\n", status);
		return;
	}

	/*	Now verify */

	/*	Initialise context for verification			*/
	if (R_VerifyInit(&sigctx, DA_MD5) != 0)
	{
		printf("R_VerifyInit failed with invalid digest ty	pe\n");
		return;
	}

	/*	Add data a block (in this case, 1 byte) at a time		*/
	for (i=0; i<stringlength; i++) {
		R_VerifyUpdate(&sigctx, &demostring[i], 1);
	}

	/*	Finalise verification					*/
	status = R_VerifyFinal(&sigctx, signature, signaturelength, &publicKey);
	if (status) {
		printf("R_VerifyFinal failed with %x\n", status);
		return;
	}

	printf("Signature verified!!\n");

}
```

Signing and verifying a block of memory-resident data

The following code sample shows the use of **R_SignBlock** and **R_VerifyBlockSignature** to sign a block of memory-resident data and then verify the signature.

```
void DigSigExample()
{
      R_RANDOM_STRUCT             randomStruct;
      R_RSA_PROTO_KEY             protoKey;
      R_RSA_PUBLIC_KEY            publicKey;
      R_RSA_PRIVATE_KEY           privateKey;
      R_SIGNATURE_CTX             sigctx;
      char                        demostring[] = "This is a sample string to sign";
      char                        signature[MAX_SIGNATURE_LEN];
      int                         stringlength, signaturelength, status, i;

      /*    Generate keys         */
      KeyGenExample(&publicKey, &privateKey);

      /*    Sign block                                              */

      stringlength = strlen(demostring);
      status = R_SignBlock(signature, &signaturelength, demostring, stringlength,
                           DA_MD5, &privateKey);
      if (status) {
            printf("R_SignBlock failed with %x\n", status);
            return;
      }

      /*    Verify signature                                        */
      status = R_VerifyBlockSignature(demostring, stringlength, signature,
                                      signaturelength, DA_MD5, &publicKey);
      if (status) {
            printf("R_VerifyBlockSignature failed with %x\n", status);
            return;
      }

      printf("Signatu re verified!!\n");
```

ENVELOPE PROCESSING

Introduction

RSAEURO uses the concept of a “digital envelope” for handling encrypted data. Data is first encrypted using a secret-key algorithm, using a random session key. The session key is then encrypted using the public keys of the intended recipients. The encrypted versions of the session key and the secret-key encrypted message form the digital envelope. “Opening” a digital envelope requires the decryption of the session key, using the recipient's private key (assuming the recipient is one of the intended recipients!), then using the session key to decrypt the message.

RSAEURO provides six varieties of secret-key encryption, all based on the US Data Encryption Standard (DES):

EA_DES_CBC	DES in cipher-block chaining (CBC) mode, using a single key.
EA_DESX_CBC	RSADSI's “enhanced” DES (CBC with an additional XOR with a secret value).
EA_DES_EDE3_CBC	Triple-DES, using three keys, in CBC mode. EDE is “Encrypt-Decrypt-Encrypt”, where data is encrypted with key1, decrypted with key2, then encrypted with key3. EDE avoids certain weaknesses of “plain” multiple encryption.
EA_DES_EDE2_CBC	Triple-DES using two keys (key1 and key 3 are identical).

EA_DES_EDE3_CBC is the most secure, and the slowest, method of encryption supported by RSAEURO.

Sealing data in digital envelopes

The procedure for “sealing” data in a digital envelope is as follows:

1 Initialise the envelope context, by calling **R_SealInit**. **R_SealInit** generates the random session key and returns the public-key encrypted versions of the session key (the session key itself, together with other intermediate data, is stored in the context). The secret-key encryption method to use is specified as a parameter to **R_SealInit**.

2 Process the source data a block at a time, using **R_SealUpdate**.

3 “Close” the envelope using **R_SealFinal**, and clear the encryption context.

Opening digital envelopes

The procedure for “opening” a digital envelope is as follows:

1 Initialise a new envelope context using **R_OpenInit**. This decrypts the session key using the recipient's private key, and sets up the context ready for decryption of the main message.

2 Process the encrypted data a block at a time using **R_OpenUpdate**.

3 Process the final encrypted data block using **R_OpenFinal**, which also removes any padding data.

Functions

R_SealInit

```
int R_SealInit(context, encryptedKeys, encryptedKeyLens, iv, publicKeyCount,
               publicKeys, encryptionAlgorithm, randomStruct
R_ENVELOPE_CTX *context;                                         /* new context */
unsigned char **encryptedKeys;                                /* encrypted keys */
unsigned int *encryptedKeyLens;                    /* lengths of encrypted keys */
unsigned char iv[8];                                  /* initialization vector */
unsigned int publicKeyCount;                          /* number of public keys */
R_RSA_PUBLIC_KEY **publicKeys;                                   /* public keys */
int encryptionAlgorithm;                           /* data encryption algorithm */
R_RANDOM_STRUCT *randomStruct;                              /* random structure */
```

Initialises an envelope sealing operation. **context** points to an allocated blank R_ENVELOPE_CTX structure and **randomStruct** points to a pre-initialised R_RANDOM_STRUCT. **encryptionAlgorithm** indicates which method of secret encryption is required (see the Introduction to this section on page 23 for valid values).

R_SealInit uses the **random** structure to generate a session key and initialisation vector for the secret-key encryption (DES in CBC mode requires a 64-bit initialisation vector).

The public keys of the intended recipients are placed in the **publicKeys** array, with the total number of public keys indicated by **publicKeyCount** (at least one public key must be supplied). An invalid public key results in an RE_PUBLIC_KEY error, and no further keys will be processed.

On success, returns zero, the **encryptedKeys** array contains the public-key encrypted session key (for each supplied public key) and **encryptedKeyLens** contains the respective encrypted key lengths. **iv** contains the DES initialisation vector.

On error, returns RE_NEED_RANDOM if **randomStruct** has not been initialised, RE_PUBLIC_KEY if an invalid public key has been supplied or RE_ENCRYPTION_ALGORITHM if an invalid encryption algorithm has been selected.

R_SealUpdate

```
int R_SealUpdate (context, partOut, partOutLen, partIn, partInLen)
R_ENVELOPE_CTX *context;                                             /* context */
unsigned char *partOut;                             /* next encrypted data part */
unsigned int *partOutLen;                 /* length of next encrypted data part */
unsigned char *partIn;                                        /* next data part */
unsigned int partInLen;                             /* length of next data part */
```

Continues a sealing operation, encrypting a block of data using the supplied context. **context** is a R_ENVELOPE_CTX structure which has been successfully initialised using **R_SealInit**. **partInLen** bytes of **partIn** are encrypted and returned in **partOut**. Due to data padding, some expansion may occur, and **partOut** should be at least eight bytes larger than **partIn**.

Always returns ID_OK.

R_SealFinal

```
int R_SealFinal(context, partOut, partOutLen)
R_ENVELOPE_CTX *context;                                             /* context */
unsigned char *partOut;                             /* last encrypted data part */
unsigned int *partOutLen;                 /* length of last encrypted data part */
```

Finalises a sealing operation, flushing the context buffer and resetting the context (to allow further use of the session key if required). **context** is the R_ENVELOPE_CTX structure in use for the current sealing operation. On exit **partOut** contains **partOutLen** bytes to be appended to the encrypted data (the contents of the context buffer). **partOutLen** will be no more than eight. Always returns ID_OK.

Note that although the context is restarted, sensitive information is not cleared. If the context is no longer required, it is the caller's responsibility to clear it for security.

R_OpenInit

```
int R_OpenInit(context, encryptionAlgorithm, encryptedKey, encryptedKeyLen, iv,
               privateKey)
R_ENVELOPE_CTX *context;                                        /* new context */
int encryptionAlgorithm;                           /* data encryption algorithm */
unsigned char *encryptedKey;                  /* encrypted data encryption key */
unsigned int encryptedKeyLen;                      /* length of encrypted key */
unsigned char iv[8];                                 /* initialization vector */
R_RSA_PRIVATE_KEY *privateKey;                /* recipient's RSA private key   */
```

Initialises an envelope context ready for an "opening" (decryption) operation. The encrypted session key is decrypted using **privateKey**, and placed in the context. The context is then initialised with the initialisation vector (supplied unencrypted "in" the envelope, and passed to **R_OpenInit** as **iv**), ready for decryption. **encryptionAlgorithm** indicates the encryption algorithm to be used.

On success, returns zero. On error, returns RE_LEN if **encryptedKey** is too long (**encryptedKeyLen** > MAX_ENCRYPTED_KEY_LEN), RE_PRIVATE_KEY if the private key is invalid (*i.e.* the correct session key cannot be retrieved) or RE_ENCRYPTION_ALGORITHM if an invalid encryption algorithm is selected.

R_OpenUpdate

```
int R_OpenUpdate(context, partOut, partOutLen, partIn, partInLen)
R_ENVELOPE_CTX *context;                                            /* context */
unsigned char *partOut;                           /* next recovered data part */
unsigned int *partOutLen;              /* length of next recovered data part */
unsigned char *partIn;                            /* next encrypted data part */
unsigned int partInLen;                /* length of next  encrypted data part */
```

Continues an opening operation, decrypting a block of data using the supplied context. **context** is a R_ENVELOPE_CTX structure which has been successfully initialised using **R_OpenInit**. **partInLen** bytes of **partIn** are decrypted and returned in **partOut**. Due to data padding, some expansion may occur, and **partOut** should be at least eight bytes larger than **partIn**.

Always returns ID_OK.

R_OpenFinal

```
int R_OpenFinal(context, partOut, partOutLen)
R_ENVELOPE_CTX *context;                                            /* context */
unsigned char *partOut;                           /* last recovered data part */
unsigned int *partOutLen;              /* length of last recovered data part */
```

Finalises an opening operation, flushing the context buffer and re-initialising the context. **context** is the R_ENVELOPE_CTX structure in use for the current opening operation.

On success, returns zero and **partOut** contains **partOutLen** bytes to be appended to the decrypted data (the contents of the context buffer). **partOutLen** will be no more than eight. On error, returns RE_KEY if the session key is invalid.

Note that although the context is restarted, sensitive information is not cleared. If the context is no longer required, it is the caller's responsibility to clear it for security.

Data Types

R_ENVELOPE_CTX

```
typedef struct {
  int encryptionAlgorithm;                              /* encryption type */
  union {                                          /* encryption sub-context */
     DES_CBC_CTX des;
     DES3_CBC_CTX des3;
     DESX_CBC_CTX desx;
  } cipherContext;
  unsigned char buffer[8];                                   /* data buffer */
  unsigned int bufferLen;                                  /* buffer length */
} R_ENVELOPE_CTX;
```

The R_ENVELOPE_CTX type stores the context for a “sealing” (encryption) operation.

encryptionAlgorithm	The encryption algorithm for the context (EA_DES_CBC, EA_DES_EDE2_CBC, EA_DES_EDE3_CBC or EA_DESX_CBC).
cipherContext	The cipher-specific context.
buffer	The input buffer for the sealing operation (DES encrypts in 64-bit blocks, so incoming data is buffered until 8 bytes are available).
bufferLen	The number of bytes in the buffer.

R_RSA_PRIVATE_KEY

The R_RSA_PRIVATE_KEY type stores an RSA private keySee page 43 for a detailed description.

R_RSA_PUBLIC_KEY

The R_RSA_PUBLIC_KEY type stores an RSA public key.See page 42 for a detailed description.

R_RANDOM_STRUCT

The R_RANDOM_STRUCT type stores the state and characteristics of a random number generatorSee page 8 for a detailed description.

Examples

Sealing and opening an envelope

The following code sample shows an example of “sealing” a block of data in an envelope, and subsequently “opening” the envelope and decrypting the data. Two public keys are used for the sealing operation, illustrating the use of the envelope functions with multiple keys (e.g. multiple recipients).

```
void EnvelopeExample()
{
      R_RANDOM_STRUCT        randomStruct;
      R_RSA_PUBLIC_KEY       publicKey[2], *publicKeys[2];
      R_RSA_PRIVATE_KEY      privateKey[2], *privateKeys[2];
      R_ENVELOPE_ CTX        envctx, envctx2;
      unsigned char          encryptedKey[2][MAX_ENCRYPTED_KEY_LEN],
                             *encryptedKeys[2];
      unsigned char          iv[8];
      char                   demostring[] = "This is a demo";
      char                   encryptedString[50], decryptedString[50], foo[50];
      int                    status, paddingLen, stringlength, encryptedKeyLen[2],
                             encryptedStringLen, decryptedStringLen;

      /*    Initialise Random structure      */
      R_RandomCreate(&randomStruct);

      /*    Initialise encryptedKeys array        */
      encryptedKeys[0] = encryptedKey[0];
      encryptedKeys[1] = encry  ptedKey[1];
```

```
	/*	Generate two keypairs				*/
	KeyGenExample(&publicKey[0], &privateKey[0]);
	publicKeys[0] = &publicKey[0];
	privateKeys[0] = &privateKey[0];
	KeyGenExample(&publicKey[1], &privateKey[1]);
	publicKeys[1] = &publicKey[1];
	privateKeys[1] = &privateKey[1];

	/*	Initialise envelope using both keys	*/
	status = R_SealInit( &envctx, encryptedKeys, &encryptedKeyLen, iv, 2,
				&publicKeys, EA_DES_CBC, &randomStruct);
	if (status) {
		printf("R_SealInit failed with %x\n", stat	us);
		return;
	}

	/*	Process string using R_SealUpdate		*/
	stringlength = strlen(demostring);
	R_SealUpdate( &envctx, encryptedString, &encryptedStringLen, demostring,
			stringlength);

	/*	Finalise context			*/
	R_SealFinal(&envctx, &encryptedString[encryptedStringLen], &paddingLen);
	encryptedStringLen += paddingLen;

	/*	"Open" the envelope, using fresh context	*/
	status = R_OpenInit( &envctx2, EA_DES_CBC, encryptedKeys[1],
				encryptedKeyLen[1], iv, privateKeys[1]);
	if (sta tus) {
		printf("R_OpenInit failed with %x\n", status);
		return;
	}

	/*	Process using update				*/
	R_OpenUpdate( &envctx2, decryptedString, &decryptedStringLen,
			encryptedString, encryptedStringLen);

	/*	Finalise				*/
	status = R_OpenFinal(&envctx2, &decryptedString[decryptedStringLen],
				&paddingLen);
	if (status) {
		printf("R_OpenFinal failed with %x\n", status);
		return;
	}

	decryptedStringLen += paddingLen;
	decryptedString[decryptedStringLen] = (char) 0;
	printf("Re sult: %s\n", decryptedString);
}
```

PEM FUNCTIONS

Introduction

RSAEURO provides a number functions to process data in Privacy Enhanced Mail (PEM) format, ASCII-encoded according to RFC 1421. In addition to simple encoding and decoding functions, PEM "versions" of several other functions are also provided. The following list of PEM functions provides brief details, and the function descriptions which follow provide more detailed information.

R_EncodePEMBlock	Encodes data into ASCII according to RFC 1421.
R_DecodePEMBlock	Decodes data in RFC 1421 format into "raw" data.
R_SignPEMBlock	Produces an RFC 1421 encoded signature of a data block, optionally RFC 1421 encoding the data following signature generation.
R_VerifyPEMSignature	Verifies an RFC 1421 encoded signature, optionally decoding the content prior to signature generation.
R_SealPEMBlock	Signs and seals a block of data in and RFC 1421 encoded "envelope", using single-key DES CBC. Only supports single recipients.
R_OpenPEMBlock	"Opens" an RFC 1421 encoded "envelope", verifying the signature and decrypting the data.
R_EncryptOpenPEMBlock	Encrypts a data block, returning encrypted RFC 1421 encoded data.
R_DecryptOpenPEMBlock	Decrypts and decodes an encrypted, RFC 1421 encoded data block.

Throughout this section, any reference to "ASCII encoded" should be read as "ASCII encoded according to RFC 1421", and so on.

Functions

R_EncodePEMBlock

```
int R_EncodePEMBlock(encodedBlock, encodedBlockLen, block, blockLen)
unsigned char *encodedBlock;                                  /* encoded block  */
unsigned int *encodedBlockLen;                      /* length of encoded block */
unsigned char *block;                                                /* block */
unsigned int blockLen;                                     /* length of block */
```

Encodes a block of binary data into ASCII for transmission through 7-bit channels such as Internet electronic mail. **blockLen** bytes of **block** are encoded and returned in **encodedBlock**, of length **encodedBlock** (in bytes).

Data expansion occurs as four ASCII characters are used to encode three data bytes. Therefore, the **encodedBlock** buffer should be allocated at least 33% larger than **block**.

Always returns ID_OK.

R_DecodePEMBlock

```
int R_DecodePEMBlock (outbuf, outlength, inbuf, inlength)
unsigned char *outbuf;                                    /* block */
unsigned int *outlength;                        /* length of block */
unsigned char *inbuf;                            /* encoded block */
unsigned int inlength;                 /* length of encoded block */
```

Decodes a block of ASCII into binary data. **Inbuf** holds the input data, **inlength** indicates the number of ASCII bytes to process, and therefore must be an integer multiple of four.

On success, returns ID_OK and **outbuf** contains **outlength** bytes of decoded data. On error, returns RE_ENCODING in the event of an encoding error (or if **inlength** is not an integer multiple of four).

R_SignPEMBlock

```
int R_SignPEMBlock( encodedContent, encodedContentLen, encodedSignature,
                    encodedSignatureLen, content, contentLen, recode, digestAlgorithm,
                    privateKey)
unsigned char *encodedContent;                         /* encoded content */
unsigned int *encodedContentLen;             /* length of encoded content */
unsigned char *encodedSignature;                     /* encoded signature */
unsigned int *encodedSignatureLen;         /* length of encoded signature */
unsigned char *content;                                        /* content */
unsigned int contentLen;                             /* length of content */
int recode;                                              /* recoding flag */
int digestAlgorithm;                                  /* digest algorithm */
R_RSA_PRIVATE_KEY *privateKey;              /* signer's RSA private key */
```

Produces a digital signature of the supplied data, and returns a ASCII-encoded version of the signature. Optionally ASCII-encodes the data block.

Content contains the data to be signed, **contentLen** indicates the length of the data. **digestAlgorithm** indicates the message digest algorithm to use, **privateKey** is the signer's RSA private key, used to encrypt the digest to produce a signature.

If **recode** is TRUE, the data block (**content**) is ASCII encoded following the message digest generation, and the encoded data is returned in **encodedContent**, and its length is returned in **encodedContentLen**.

On success, returns ID_OK, **encodedSignature** contains the ASCII encoded signature and **encodedSignatureLen** indicates the length of the encoded signature. On error, returns RE_DIGEST_ALGORITHM if an invalid digest algorithm is specified or RE_PRIVATE_KEY if the private key is invalid.

R_VerifyPEMSignature

```
int R_VerifyPEMSignature( content, contentLen, encodedContent, encodedContentLen,
                          encodedSignature, encodedSignatureLen, recode,
                          digestAlgorithm, publicKey)
unsigned char *content;                                        /* content */
unsigned int *contentLen;                            /* length of content */
unsigned char *encodedContent;             /* (possibly) encoded content */
unsigned int encodedContentLen;              /* length of encoded content */
unsigned char *encodedSignature;                     /* encoded signature */
unsigned int encodedSignatureLen;          /* length of encoded signature */
int recode;                                              /* recoding flag */
int digestAlgorithm;                                  /* digest algorithm */
R_RSA_PUBLIC_KEY *publicKey;                     /* signer's public key */
```

Decodes and verifies an ASCII-encoded signature. Optionally decodes the data block prior to message digest generation and verification.

content contains the data against which the signature is to be verified, **contentLen** indicates the length of the data. If **recode** is TRUE, **encodedContent** (**encodedContentLen** bytes long) is decoded into **content** prior to signature verification.

publicKey is used to decrypt the signature, and the resulting message digest is compared with the digest generated from **content** using the digest algorithm indicated by **digestAlgorithm** (it is the caller's responsibility to identify the appropriate digest algorithm).

Returns zero for success (the digests match), RE_SIGNATURE_ENCODING if the signature cannot be decoded correctly, RE_CONTENT_ENCODING if the content cannot be decoded correctly, RE_LEN if the signature length is invalid, RE_DIGEST_ALGORITHM if an invalid digest algorithm is selected, RE_PUBLIC_KEY if the supplied public key is invalid or RE_SIGNATURE if the signature is incorrect (*i.e.* the digests do not match).

R_SealPEMBlock

```
int R_SealPEMBlock( encryptedContent, encryptedContentLen, encryptedKey,
                    encryptedKeyLen, encryptedSignature, encryptedSignatureLen, iv,
                    content, contentLen, digestAlgorithm, publicKey, privateKey,
                    randomStruct)
unsigned char *encryptedContent;                    /* encoded, encrypted content */
unsigned int *encryptedContentLen;                                      /* length */
unsigned char *encryptedKey;                            /* encoded, encrypted key */
unsigned int *encryptedKeyLen;                                          /* length */
unsigned char *encryptedSignature;                /* encoded, encrypted signature */
unsigned int *encryptedSignatureLen;                                    /* length */
unsigned char iv[8];                                 /* DES initialization vector */
unsigned char *content;                                                /* content */
unsigned int contentLen;                                     /* length of content */
int digestAlgorithm;                                 /* message-digest algorithms */
R_RSA_PUBLIC_KEY *publicKey;                        /* recipient's RSA public key */
R_RSA_PRIVATE_KEY *privateKey;                        /* signer's RSA private key */
R_RANDOM_STRUCT *randomStruct;                                /* random structure */
```

Seals data in a digital envelope, with EA_DES_CBC encryption and digital signature, and returns PEM ASCII-encoded data.

content contains the data to be sealed, **contentLen** indicates the length of the data. A signature of **content** is produced using the digest algorithm indicated by **digestAlgorithm** and the sender's private key, **privateKey**. **content** is then encrypted using EA_DES_CBC, with a random session key generated from **randomStruct** (it is the caller's responsibility to ensure that **randomStruct** has been initialised).

On success, returns zero, **encryptedContent** contains **encryptedContentLen** bytes of ASCII encoded encrypted content, **encryptedKey** contains the ASCII encoded session key (**encryptedKeyLen** bytes long), encrypted with **publicKey**, and **encryptedSignature** contains the ASCII encoded signature. All secret-key encryption is performed using EA_DES_CBC and the session key.

It is the caller's responsibility to clear the context if it is no longer required.

On error, returns RE_DIGEST_ALGORITHM if an invalid digest algorithm is selected, RE_PRIVATE_KEY if the private key is invalid, RE_PUBLIC_KEY if the public key is invalid or RE_NEED_RANDOM if the random structure is not initialised.

R_OpenPEMBlock

```
int R_OpenPEMBlock( content, contentLen, encryptedContent,
                    encryptedContentLen, encryptedKey, encryptedKeyLen,
                    encryptedSignature, encryptedSignatureLen, iv,
                    digestAlgorithm, privateKey, publicKey)
unsigned char *content;                                                /* content */
unsigned int *contentLen;                                    /* length of content */
unsigned char *encryptedContent;                    /* encoded, encrypted content */
unsigned int encryptedContentLen;                                       /* length */
unsigned char *encryptedKey;                            /* encoded, encrypted key */
unsigned int encryptedKeyLen;                                           /* length */
unsigned char *encryptedSignature;                /* encoded, encrypted signature */
unsigned int encryptedSignatureLen;                                     /* length */
unsigned char iv[8];                                 /* DES initialization vector */
int digestAlgorithm;                                 /* message-digest algorithms */
R_RSA_PRIVATE_KEY *privateKey;                     /* recipient's RSA private key */
R_RSA_PUBLIC_KEY *publicKey;                           /* signer's RSA public key */
```

“Opens” a ASCII encoded digital envelope, verifies the signature, decodes and decrypts the content of the envelope.

encryptedContent contains the encoded, encrypted content, and **encryptedContent** indicates its length in bytes. The session key is retrieved from **encryptedKey** (**encryptedKeyLen** bytes long) using **publicKey**, then used to decrypt the **encryptedContent**. Once the content has been decrypted, the signature is retrieved, decoded and verified against the content using the recipient's private key (**privateKey**) and the message digest algorithm indicated by **digestAlgorithm**.

On success, returns zero and **content** contains **contentLen** bytes of plaintext data.

On error, returns RE_KEY_ENCODING if the key cannot be decoded, RE_SIGNATURE_ENCODING if the signature cannot be decoded, RE_CONTENT_ENCODING if the content cannot be decoded, RE_LEN if the encrypted session key is too long, RE_PRIVATE_KEY if the private key is invalid, RE_KEY if the retrieved session key is invalid, RE_DIGEST_ALGORTIHM if an invalid digest algorithm is selected, RE_PUBLIC_KEY if the public key is invalid or RE_SIGNATURE if the signature is incorrect (*i.e.* the message digests do not match).

R_EncryptOpenPEMBlock

```
int R_EncryptOpenPEMBlock(context, output, outputLen, input, inputLen)
R_ENVELOPE_CTX *context;                                    /* context */
unsigned char *output;                        /* encrypted, encoded block */
unsigned int *outputLen;                              /* length of output */
unsigned char *input;                                /* block to encrypt */
unsigned int inputLen;                                        /* length */
```

Encrypts a block of data and returns the ciphertext in ASCII encoded format. **context** is the current envelope context, which must have been initialised correctly by the caller. **inputLen** bytes from input are encrypted and encoded into ASCII, then returned in **output**. On exit, **outputLen** may be up to 33% larger than **inputLen** (three source bytes become four output bytes), so **output** should be allocated to account for the data expansion. Always returns ID_OK.

R_DecryptOpenPEMBlock

```
int R_DecryptOpenPEMBlock(context, output, outputLen, input, inputLen)
R_ENVELOPE_CTX *context;                                    /* context */
unsigned char *output;                        /* decoded, decrypted block */
unsigned int *outputLen;                              /* length of output */
unsigned char *input;                         /* encrypted, encoded block */
unsigned int inputLen;                                        /* length */
```

Decrypts a block of ASCII encoded ciphertext and returns the plaintext in **output**. **context** is the current envelope context, which must have been initialised correctly by the caller. **inputLen** bytes from **input** are decoded, decrypted, then returned in **output**. Always returns ID_OK.

Data Types

R_RSA_PRIVATE_KEY

The R_RSA_PRIVATE_KEY type stores an RSA private key. See page 43 for a detailed description.

R_RSA_PUBLIC_KEY

The R_RSA_PUBLIC_KEY type stores an RSA public key. See page 42 for a detailed description.

R_RANDOM_STRUCT

The R_RANDOM_STRUCT type stores the state and characteristics of a random number generator. See page 8 for a detailed description.

Examples

Sealing and opening a PEM envelope

The following code sample shows an example of “sealing” a block of data in a PEM envelope, and subsequently “opening” the envelope and decrypting the data.

```
void PEMExample()
{
      R_RANDOM_STRUCT            randomStruct;
      R_RSA_PROTO_KEY            protokey;
      R_RSA_PRIVATE_KEY          myprivatekey;
      R_RSA_PRIVATE_KEY          theirprivatekey;
```

```
    R_RSA_PUBLIC_KEY            mypublickey;
    R_RSA_PUBLIC_KEY            theirpublickey;
    unsigned char               encryptedContent[1024];
    unsigned char               encrypted Signature[1024];
    unsigned char               encryptedKey[1024];
    unsigned char               content[1024];
    unsigned char               iv[8];
    unsigned int                encryptedContentLen;
    unsigned int                encryptedSignatureLen;
    unsigned int                encryptedKeyLen;
    int                         status, contentLen;

    /*    Generate keypairs     */
    R_RandomCreate(&randomStruct);

    /* set key attribute */
    protokey.bits = 512;
    protokey.useFermat4 = 1;

    /* Generate "my" and "their" keys      */
    status = R_GeneratePEMKeys(      &theirpublickey, &theirprivatekey,
                                     &protokey, &randomSt ruct);
    if (status) {
          printf("R_GeneratePEMKeys failed with %d\n", status);
          return;
    }

    R_RandomMix(&random);
    status = R_GeneratePEMKeys(      &mypublickey, &myprivatekey,
                                     &protokey, &randomStruct);
    if (status) {
          printf("R_GeneratePEMKeys failed with %d\n", status);
          return;
    }



    /*    Initialise content    */
    strcpy((char *)content, "This is a PEM test Message");

    /*    Seal block using my private key       */
    status = R_SealPEMBlock(encryptedContent, &encryptedContentLen,
                            encryptedKey, &encrypte dKeyLen,
                            encryptedSignature, &encryptedSignatureLen,
                            iv, content, strlen((char *) content),
                            DA_MD5, &theirpublickey, &myprivatekey,
                            &randomStruct);

    if (status) {
          printf("R_SealPEMBlock failed with %d\n", status);
          return;
    }

    /*    Now "open" the block, using my public key and their
          private key                                       */

    /*    Clear content         */
    R_memset((POINTER)&content, 0, sizeof(content));

    printf("Openblock\n");
    status = R_OpenPEMBlock(content, &contentLen, encryptedContent,
                            encrypted ContentLen, encryptedKey,
                            encryptedKeyLen, encryptedSignature,
                            encryptedSignatureLen, iv, DA_MD5,
                            &theirprivatekey, &mypublickey);

    if (status) {
          printf("R_OpenPEMBlock failed with %d\n", status);
          return;
    }

    printf("Envelope contents: %s\n", content);
}
```

Signing and verifying a PEM block

The following code sample shows an example of signing a PEM block, and subsequently verifying the envelope and decrypting the data.

```
void PEMExample2()
{
      R_RANDOM_STRUCT        randomStruct;
      R_RSA_PROTO_KEY        protokey;
      R_RSA_PRIVATE_KEY      myprivatekey;
      R_RSA_PUBLIC_KEY       mypublickey;
      unsigned char          content[1024];
      unsigned char          encodedSignature[1024];
      int                    status, contentLen, encodedSignatureLen;

      /*    Generate keypair       */
      R_RandomCreate(&randomStruct);
      protokey.bits = 512;
      protokey.useFermat4 = 1;
      status = R_GeneratePEMKeys(       &mypublickey, &myprivatekey,
                                        &protokey, &randomStruct);
      if (status) {
            printf("R_GeneratePEMKeys failed with %d\n", status);
            return;
      }

      /*Initialise co ntent*/
      strcpy((char *)content, "This is a PEM signature test Message");
      contentLen = strlen((char *)content);

      status = R_SignPEMBlock(NULL, NULL, encodedSignature,
                              &encodedSignatureLen, content,
                              contentLen, 0, DA_MD5, &myprivatekey);
      if (status) {
            printf("R_SignPEMlock failed with %d\n", status);
            return;
      }

      printf("Signature created OK\n");

      /*Now verify signature*/
      status = R_VerifyPEMSignature(    content, &contentLen, content,
                                        contentLen, encodedSignature,
                                        encodedSignatureLen, 0, DA_MD5,
                                        &mypublickey);

      if (status) {
            printf("R_VerifyPEMSignature failed with %x\n%s", status, content);
            return;
      }

      printf("Signature verified OK\n");

}
```

SECTION II
ALGORITHMS

SECTION II
ALGORITHMS

PUBLIC KEY ALGORITHMS

KEY GENERATION

Introduction

This section describes the functions in RSAEURO for the generation of RSA key pairs.

A single function, **R_GeneratePEMKeys**, provides RSA key generation for RSAEURO. A "prototype key" is passed to the function, indicating the length of the modulus in bits and the public exponent. Two values are supported for the public exponent: 3 or "Fermat 4" (65537). A pre-initialised random structure is required for key generation.

Functions

R_GeneratePEMKeys

```
int R_GeneratePEMKeys(publicKey, privateKey, protoKey, randomStruct)
R_RSA_PUBLIC_KEY *publicKey;                                /* new RSA public key */
R_RSA_PRIVATE_KEY *privateKey;                             /* new RSA private key */
R_RSA_PROTO_KEY *protoKey;                                   /* RSA prototype key */
R_RANDOM_STRUCT *randomStruct;                                /* random structure */
```

Generates an RSA public/private key pair, based on the supplied prototype key.

On success, returns zero, with the new public and private keys returned in **publicKey** and **privateKey**.

On error, returns RE_MODULUS_LEN if the modulus length specified in **protoKey** is invalid (either less than MIN_RSA_MODULUS_BITS or greater than MAX_RSA_MODULUS_BITS), RE_NEED_RANDOM if **randomStruct** has not been initialised or RE_DATA if a problem occurred generating primes.

Data Types

R_RSA_PUBLIC_KEY

The R_RSA_PUBLIC_KEY type stores an RSA public key. See page 42 for a detailed description.

R_RSA_PRIVATE_KEY

The R_RSA_PRIVATE_KEY type stores an RSA private key. See page 43 for a detailed description.

R_RSA_PROTO_KEY

The R_RSA_PROTO_KEY type provides a template for RSA keypair generation. See page 43 for a detailed description.

R_RANDOM_STRUCT

The R_RANDOM_STRUCT type stores the state and characteristics of a random number generator. See page 8 for a detailed description.

Examples

Generating an RSA keypair

The following code sample shows the use of **R_GeneratePEMKeys** to create an RSA keypair.

```
void KeyGenExample(publicKey, privateKey)
R_RSA_PUBLIC_KEY *publicKey;
R_RSA_PRIVATE_KEY *privateKey;
{
      R_RANDOM_STRUCT       randomStruc t;
      R_RSA_PROTO_KEY       protoKey;
      int                   status;

      /*    Initialise random structure ready for keygen         */
      R_RandomCreate(&randomStruct);

      /*    Initialise prototype key structure            */
      protoKey.bits=512;
      protoKey.useFermat4 = 1;

      /*    Generate keys                          */
      status = R_GeneratePEMKeys(publicKey, privateKey, &protoKey, randomStruct);
      if (status)
      {
            printf("R_GeneratePEMKeys failed with %d\n", status);
            return;
      }

}
```

RSA

Introduction

This section describes the RSA processing routines provided by RSAEURO for RSA encryption and decryption. The RSA functions are listed below:

RSAPrivateEncrypt	Encrypts a block of data using an RSA private key, according to PKCS#1: RSA Encryption Standard.
RSAPrivateDecrypt	Decrypts a block of data using an RSA private key, according to PKCS#1: RSA Encryption Standard.
RSAPublicEncrypt	Encrypts a block of data using an RSA public key, according to PKCS#1: RSA Encryption Standard.
RSAPublicDecrypt	Decrypts a block of data using an RSA public key, according to PKCS#1: RSA Encryption Standard.

NOTE: Paul Kocher recently published a timing-based attack which could be used against RSA encryption providing the attacker has access to the machine performing the encryption – this is **not** usually the case with email encryption. Future versions of RSAEURO will incorporate protection against such attacks. For further details, see the preliminary draft of the paper, available on the Internet at `http://www.cryptography.com` and RSADSI's response at `http://www.rsa.com` .

Functions

RSAPrivateEncrypt

```
int RSAPrivateEncrypt(output, outputLen, input, inputLen, privateKey)
unsigned char *output;                                        /* output block */
unsigned int *outputLen;                            /* length of output block */
unsigned char *input;                                          /* input block */
unsigned int inputLen;                               /* length of input block */
R_RSA_PRIVATE_KEY *privateKey;                             /* RSA private key */
```

Performs a PKCS#1-compliant RSA private-key encryption. **inputLen** bytes from **input** are encrypted using **privateKey**, and the result returned in **ouput**, **outputLen** bytes long. **output** should be large enough to hold the result of the calculation, which will be one byte longer than the private key (*i.e.* not larger than MAX_RSA_MODULUS_LEN + 1). **inputLen** must be at least eleven bytes smaller than the modulus size (the additional eleven bytes are required for PKCS#1 encoding).

On exit, **outputLen** bytes of encrypted data are returned in **output**. Returns RE_LEN if the input block is too large for the supplied key, ID_OK otherwise.

RSAPrivateDecrypt

```
int RSAPrivateDecrypt(output, outputLen, input, inputLen, privateKey)
unsigned char *output;                                        /* output block */
unsigned int *outputLen;                             /* length of output block *
unsigned char *input;                                          /* input block */
unsigned int inputLen;                               /* length of input block */
R_RSA_PRIVATE_KEY *privateKey;                             /* RSA private key */
```

Performs an RSA private-key decryption of a PKCS#1-compliant input block. **inputLen** bytes from **input** are decrypted using **privateKey**, and the result returned in **output**, **outputLen** bytes long. **outputLen** will be no larger than MAX_RSA_MODULUS_LEN + 1.

On exit, **outputLen** bytes of decrypted data are returned in **output**. Returns RE_LEN if the input block size is incorrect for the supplied key, RE_DATA if the decrypted data is not a valid PKCS#1 data block, ID_OK otherwise.

RSAPublicEncrypt

```
int RSAPublicEncrypt(output, outputLen, input, inputLen, publicKey,
                     randomStruct)
unsigned char *output;                                  /* output block */
unsigned int *outputLen;                      /* length of output block */
unsigned char *input;                                    /* input block */
unsigned int inputLen;                        /* length of input b lock */
R_RSA_PUBLIC_KEY *publicKey;                          /* RSA public key */
R_RANDOM_STRUCT *randomStruct;                      /* random structure */
```

Performs a PKCS#1 compliant RSA public key encryption. **inputLen** bytes from **input** are encrypted using **publicKey**, and the result returned in **ouput**, **outputLen** bytes long. **output** should be large enough to hold the result of the calculation, which will be one byte longer than the public key (*i.e.* not larger than MAX_RSA_MODULUS_LEN + 1). **inputLen** must be at least eleven bytes smaller than the modulus size (the additional eleven bytes are required for PKCS#1 encoding). **randomStruct** must be an initialised random structure (random data is required for the PKCS#1 data block).

On exit, **outputLen** bytes of encrypted data are returned in **output**. Returns RE_LEN if the input block size is incorrect for the supplied key, ID_OK otherwise.

RSAPublicDecrypt

```
int RSAPublicDecrypt(output, outputLen, input, inputLen, publicKey)
unsigned char *output;                                  /* output block */
unsigned int *outputLen;                      /* length of output block   */
unsigned char *input;                                    /* input block */
unsigned int inputLen;                         /* length of input block */
R_RSA_PUBLIC_KEY *publicKey;                          /* RSA public key */
```

Performs an RSA public-key decryption of a PKCS#1-compliant input block. **inputLen** bytes from **input** are decrypted using **publicKey**, and the result returned in **output**, **outputLen** bytes long. **outputLen** will be no larger than MAX_RSA_MODULUS_LEN + 1.

On exit, **outputLen** bytes of decrypted data are returned in **output**. Returns RE_LEN if the input block size is incorrect for the supplied key, RE_DATA if the decrypted data is not a valid PKCS#1 data block, ID_OK otherwise.

Data Types

R_RSA_PUBLIC_KEY

```
typedef struct {
  unsigned int bits;                    /* length in bits of modulus */
  unsigned char modulus[MAX_RSA_MODULUS_LEN];               /* modulus */
  unsigned char exponent[MAX_RSA_MODULUS_LEN];      /* public exponent */
} R_RSA_PUBLIC_KEY;
```

The R_RSA_PUBLIC_KEY type stores an RSA public key.

bits	The length of the modulus in bits (MIN_RSA_MODULUS_BITS < bits ≤ MAX_RSA_MODULUS_BITS).
modulus	The modulus, stored as a MAX_RSA_MODULUS_LEN byte number, most significant byte first, padded with zero bytes.
exponent	The public exponent, stored in the same manner as the modulus.

R_RSA_PRIVATE_KEY

```
typedef struct {
  unsigned int bits;                                  /* length in bits of modulus */
  unsigned char modulus[MAX_RSA_MODULUS_LEN];                              /* modulus */
  unsigned char publicExponent[MAX_RSA_MODULUS_LEN];               /* public exponent */
  unsigned char exponent[MAX_RSA_MODULUS_LEN];                    /* private exponent */
  unsigned char prime[2][MAX_RSA_PRIME_LEN];                        /* prime factors */
  unsigned char primeExponent[2][MAX_RSA_PRIME_LEN];           /* exponents for CRT */
  unsigned char coefficient[MAX_RSA_PRIME_LEN];                  /* CRT coefficient */
} R_RSA_PRIVATE_KEY;
```

The R_RSA_PRIVATE_KEY type stores an RSA private key.

bits	The length of the modulus in bits (MIN_RSA_MODULUS_BITS < bits ≤ MAX_RSA_MODULUS_BITS).
modulus	The modulus, stored as a MAX_RSA_MODULUS_LEN byte number, most significant byte first, zero padded.
publicExponent	The public exponent, stored in the same manner as the modulus.
exponent	The private exponent, stored in the same manner as the modulus.
prime	The prime factors (p and q) of the modulus, stored as two MAX_RSA_PRIME_LEN long numbers in the same manner as the modulus (p > q).
primeExponent	The exponents for Chinese Remainder Theorem operations (d mod p-1 and d mod q-1), stored in the same manner as prime.
coefficient	The coefficient (1/q mod p) for Chinese Remainder Theorem operations, stored in the same manner as prime.

R_RSA_PROTO_KEY

```
typedef struct {
  unsigned int bits;                                  /* length in bits of modulus */
  int useFermat4;                             /* public exponent (1 = F4, 0 = 3) */
} R_RSA_PROTO_KEY;
```

The R_RSA_PROTO_KEY type provides a template for RSA keypair generation.

bits	Length of the modulus in bits (MIN_RSA_MODULUS_BITS < bits ≤ MAX_RSA_MODULUS_BITS).
useFermat4	Public exponent, either Fermat4 or 3.

Examples

Private key encryption and public key decryption

The following code sample shows the use of **RSAPrivateEncrypt** to encrypt data using a private key, followed by the use of **RSAPublicDecrypt** to decrypt the data using the corresponding public key. The function **KeyGenExample** generates an RSA key pair, and is described in detail on page 39.

```
void RSAExample()

{
      R_RSA_PUBLIC_KEY        publicKey;
      R_RSA_PRIVATE_KEY       privateKey;
      char                    demostring[] = "Test string for RSA functions #1";
      char                    encryptedString[MAX_RSA_MODULUS_LEN+2];
      char                    decryptedString[256];
      int                     status, encryptedLength, decryptedLength;

      /*    Generate keys          */
      KeyGenExampl e(&publicKey, &privateKey);

      /*    Encrypt string with private key        */
```

```
    status = RSAPrivateEncrypt(encryptedString, &encryptedLength, demostring,
                               strlen(demostring), &privateKey);
    if (status)
    {
        printf("RSAPrivateEncrypt failed with %x\n", status);
        return;
    }

    /*  Decrypt with public key                */
    status = RSAPublicDecrypt(decryptedString, &decryptedLength, encryptedString,
                              encryptedLength, &publicKey);
    if (status)
    {
        printf("RSAPublicDecrypt failed   with %x\n", status);
        return;
    }

    /*  Display decrypted string    */
    decryptedString[decryptedLength+1] = (char) "\0";
    printf("Decrypted string: %s\n", decryptedString);

}
```

Public key encryption and private key decryption

The following code sample shows the use of **RSAPublicEncrypt** to encrypt data using a private key, followed by the use of **RSAPrivateDecrypt** to decrypt the data using the corresponding public key. The function **KeyGenExample** generates an RSA key pair, and is described in detail on page 39.

```
void RSAExample2()

{
    R_RANDOM_STRUCT        randomStruct;
    R_RSA_PUBLIC_KEY       publicKey;
    R_RSA_PRIVATE_KEY      privateKey;
    char                   demostring[] = "Test string for RSA functions #2";
    char                   encryptedString[MAX_RSA_MODULUS_LEN+2];
    char                   decryptedString[256];
    int                    status, encryptedLength, decryptedLength;

    /*  Generate keys         */
    KeyGenExample(&publicKey, &privateKey);


    /*  Initialise Random structure       */
    R_RandomCreate(&randomStruct);

    /*  Encrypt string with public key         */
    status = RSAPublicEncrypt(encryptedString, &encryptedLength, demostring,
                              strlen(demostring), &publicKey, &randomStruct);
    if (status)
    {
        printf("RSAPublicEncrypt failed with %x\n", status);
        return;
    }

    /*  Decrypt with public key                */
    status = RSAPrivateDecrypt(decryptedString, &decryptedLength,
                               encryptedString, encryptedLength, &privateKey);
    if (status)
    {
        printf("RSAPrivateDecrypt failed with %x\n", status);
        return;
    }

    /*  Display decryp ted string   */
    decryptedString[decryptedLength+1] = (char) "\0";
    printf("Decrypted string: %s\n", decryptedString);

}
```

DIFFIE-HELLMAN

Introduction

Diffie-Hellman key agreement provides a method for exchanging session keys without using RSA. Diffie-Hellman gains its security form the difficulty of calculating discrete logarithms in a finite field. The procedure for generating a session key using Diffie-Hellman is as follows:

1. The Diffie-Hellman parameters are generated using **R_GenerateDHParams**, and passed to the relevant parties (this exchange can take place over an insecure communications path, as knowledge of the Diffie-Hellman parameters does not assist an attacker).

2. The two parties wishing to communicate each generate public and private values using **R_SetupDHAgreement**, using the agreed on parameters.

3. Both parties exchange public values, and compute the session key using **R_ComputeDHAgreedKey**.

Functions

R_GenerateDHParams

```
int R_GenerateDHParams(params, primeBits, subPrimeBits, randomStruct)
R_DH_PARAMS *params;                                    /* new Diffie-Hellman parameters */
unsigned int primeBits;                                       /* length of prime in bits */
unsigned int subPrimeBits;                                 /* length of subprime in bits */
R_RANDOM_STRUCT *randomStruct;                                        /* random structure */
```

Generates a set of Diffie-Hellman parameters (prime/modulus and generator). **primeBits** indicates the length of the prime required (in bits), and **subPrimeBits** indicates the length of a prime "q" that divides p-1. The resulting Diffie-Hellman "generator" is of order q. **randomStruct** points to an initialised random structure.

It is the caller's responsibility to use sensible values for **primeBits**; there are no sanity checks.

On success, returns zero, and the Diffie-Hellman parameters in **params**.

On error, returns RE_NEED_RANDOM if **randomStruct** has not been initialised or RE_DATA if a problem occurred generating primes.

R_SetupDHAgreement

```
int R_SetupDHAgreement(publicValue, privateValue, privateValueLen, params,
                       randomStruct)
unsigned char *publicValue;                                           /* new public value */
unsigned char *privateValue;                                         /* new private value */
unsigned int privateValueLen;                                  /* length of private value */
R_DH_PARAMS *params;                                         /* Diffie-Hellman parameters */
R_RANDOM_STRUCT *randomStruct;                                        /* random structure */
```

Generates a set of public and private Diffie-Hellman values, using the supplied prime and generator (from **params**).

params is a previously initialised R_DH_PARAMS structure, **randomStruct** points to an initialised random structure. **privateValueLen** indicates the length of the required private value in bytes (typically, the same size as the **subPrimeBits** value supplied to **R_GenerateDHParameters**).

On success, returns ID_OK, with the Diffie-Hellman private and public values in **privateValue** and **publicValue** respectively (**publicValue** is the same length as **params->prime**).

On error, returns RE_NEED_RANDOM if **randomStruct** has not been initialised or RE_DATA if a problem occurred generating primes.

R_ComputeDHAgreedKey

```
int R_ComputeDHAgreedKey(agreedKey, otherPublicValue, privateValue,
                         privateValueLen, params)
unsigned char *agreedKey;                                   /* new agreed key */
unsigned char *otherPublicValue;                      /* other's public value */
unsigned char *privateValue;                                 /* private value */
unsigned int privateValueLen;                      /* length of private value */
R_DH_PARAMS *params;                             /* Diffie-Hellman parameters */
```

Computes a session key from supplied Diffie-Hellman parameters.

params is a previously initialised R_DH_PARAMS structure, **randomStruct** points to an initialised **random** structure. **privateValue** points to the caller's Diffie-Hellman private value (**privateValueLen** bytes long). **OtherPublicValue** points to the other party's Diffie-Hellman public value (which is **params->primeLen** long).

On success, returns ID_OK, with the generated session key in **agreedKey** (**params->primeLen bytes** long).

On error, returns RE_DATA for a mathematical error (such as incorrect public values or invalid **params** structure).

Data Types

R_DH_PARAMS

```
typedef struct {
  unsigned char *prime;                                              /* prime */
  unsigned int primeLen;                                   /* length of prime */
  unsigned char *generator;                                      /* generator */
  unsigned int generatorLen;                           /* length of generator */
} R_DH_PARAMS;
```

The R_DH_PARAMS type stores a set of parameters for a Diffie-Hellman key exchange.

prime	The prime p, stored as a primeLen-byte long number, most significant byte first, zero padded.
primeLen	The length in bytes of prime.
generator	The generator g, stored in the same manner as prime.
generatorLen	The length in bytes of generator.

R_RANDOM_STRUCT

The R_RANDOM_STRUCT type stores the state and characteristics of a random number generator. See page 8 for a detailed description.

Examples

To be provided.

SECTION II
ALGORITHMS

SECRET KEY ALGORITHMS

DES

Introduction

This section describes the core DES processing routines provided by RSAEURO for DES encryption and decryption in various modes of operation.

RSAEURO supports three different DES modes: single-key DES in cipher-block-chaining (CBC) mode, three-key DES in CBC mode using “encrypt-decrypt-encrypt” and DESX, RSADSI’s “enhanced” DES (CBC with an additional XOR with a secret value). Dual-key DES is provided for envelope processing by using three-key DES with key1 equal to key3.

DES support in each mode consists of three basic functions: **init**, which initialises the relevant structure and loads the key (as supplied to the **init** function); **update**, which processes a block of input data using an initialised context, either encrypting or decrypting, and **restart** which restarts a context, resetting the initialisation vector, allowing the re-use of the same key for further CBC operations. The context contains the key (in the form of subkeys) during the DES operation, and as such should be treated as sensitive data. It is the caller’s responsibility to clear the context once the DES operation is complete.

Functions

DES_CBCInit

```
void DES_CBCInit(context, key, iv, encrypt)
DES_CBC_CTX *context;                                           /* context */
unsigned char *key;                                                 /* key */
unsigned char *iv;                                     /* initializing vector */
int encrypt;                      /* encrypt flag (1 = encrypt, 0 = decrypt) */
```

Initialises a DES CBC context, loading the context with the subkeys. **context** is a blank DES_CBC_CTX structure, **key** is the DES key, **iv** is the initialising vector and **encrypt** is a flag indicating encryption or decryption (zero for decryption, any other value for encryption). Both **key** and **iv** are unsigned char arrays of eight bytes each.

Note that on exit **context** contains the key supplied, and should be handled as security sensitive data. It is the caller’s responsibility to clear **context** once the encryption or decryption operation has been completed.

DES_CBCUpdate

```
int DES_CBCUpdate(context, output, input, len)
DES_CBC_CTX *context;                                           /* context */
unsigned char *output;                                     /* output block */
unsigned char *input;                                       /* input block */
unsigned int len;
```

Continues a DES_CBC operation, encrypting **len** bytes from **input** using the supplied **context**, placing the results in **output**. **context** must be a DES_CBC_CTX structure which has been initialised using **DES_CBCInit**. **len** must be a multiple of eight bytes. **output** must have at least **len** bytes available.

On exit, **output** contains **len** bytes of encrypted data. Returns RE_LEN if **len** is not an integer multiple of eight bytes, ID_OK otherwise.

DES_CBCRestart

```
void DES_CBCRestart(context)
DES_CBC_CTX *context;
```

Restarts the supplied DES_CBC context, resetting the initialisation vector to the original value, allowing the use of the same context (and, consequently, the same DES key) on a new block of data. Note that the key information is not cleared, so **context** should be handled as security sensitive data.

DES3_CBCInit

```
void DES3_CBCInit(context, key, iv, encrypt)
DES3_CBC_CTX *context;                                        /* context */
unsigned char *key;                                               /* key */
unsigned char *iv;                                 /* initializing vector */
int encrypt;                    /* encrypt flag (1 = encrypt, 0 = decrypt) */
```

Initialises an Encrypt-Decrypt-Encrypt (EDE) DES CBC context, loading the context with the subkeys from the three supplied DES keys. **context** is a blank DES3_CBC_CTX structure, **key** is the DES key, **iv** is the initialising vector and **encrypt** is a flag indicating encryption or decryption (zero for decryption, any other value for encryption). Both **key** and **iv** are unsigned byte arrays, of twenty-four and eight bytes respectively (the **key** array consists of the three DES keys concatenated).

Note that on exit **context** contains the key supplied, and should be handled as security sensitive data. It is the caller's responsibility to clear **context** once the encryption or decryption operation has been completed.

DES3_CBCRestart

```
void DES3_CBCRestart (context)
DES3_CBC_CTX *context;                                        /* context */
```

Restarts the supplied DES3_CBC context, resetting the initialisation vector to the original value, allowing the use of the same context (and, consequently, the same DES key) on a new block of data. Note that the key information is not cleared, so **context** should be handled as security sensitive data.

DES3_CBCUpdate

```
int DES3_CBCUpdate(context, output, input, len)
DES3_CBC_CTX *context;                                        /* context */
unsigned char *output;                                   /* output block */
unsigned char *input;                                     /* input block */
unsigned int len;                   /* length of input and output blocks */
```

Continues a DES3_CBC operation, encrypting **len** bytes from **input** using the supplied **context**, placing the results in **output**. **context** must be a DES3_CBC_CTX structure which has been initialised using **DES3_CBCInit**. **len** must be a multiple of eight bytes. **output** must have at least **len** bytes available.

On exit, **output** contains **len** bytes of encrypted data. Returns RE_LEN if **len** is not an integer multiple of eight bytes, ID_OK otherwise.

DESX_CBCInit

```
void DESX_CBCInit(context, key, iv, encrypt)
DESX_CBC_CTX *context;                                        /* context */
unsigned char *key;                             /* DES key and whiteners */
unsigned char *iv;                            /* DES initializing vector */
int encrypt;                    /* encrypt flag (1 = encrypt, 0 = decrypt) */
```

Initialises an DESX CBC context, loading the context with the subkeys and "whiteners" from the supplied key. **context** is a blank DESX_CBC_CTX structure, **key** is the DESX key (the DES key, input whitener and output whitener concatenated), **iv** is the initialising vector and **encrypt** is a flag indicating encryption or decryption (zero for decryption, any other value for encryption). Both **key** and **iv** are unsigned byte arrays, of twenty-four and eight bytes respectively (the **key** array consists of the three DES keys concatenated).

Note that on exit **context** contains the key supplied, and should be handled as security sensitive data. It is the caller's responsibility to clear **context** once the encryption or decryption operation has been completed.

DESX_CBCRestart

```
void DESX_CBCRestart(context)
DESX_CBC_CTX *context;                                          /* context */
```

Restarts the supplied DESX_CBC context, resetting the initialisation vector to the original value, allowing the use of the same context (and, consequently, the same DES key) on a new block of data. Note that the key information is not cleared, so **context** should be handled as security sensitive data.

DESX_CBCUpdate

```
int DESX_CBCUpdate (context, output, input, len)
DESX_CBC_CTX *context;                                          /* context */
unsigned char *output;                                      /* output bloc k */
unsigned char *input;                                        /* input block */
unsigned int len;                    /* length of input and output blocks */
```

Continues a DESX_CBC operation, encrypting **len** bytes from **input** using the supplied **context**, placing the results in **output**. **context** must be a DESX_CBC_CTX structure which has been initialised using **DESX_CBCInit**. **len** must be a multiple of eight bytes. **output** must have at least **len** bytes available.

On exit, **output** contains **len** bytes of encrypted data. Returns RE_LEN if **len** is not an integer multiple of eight bytes, ID_OK otherwise.

Data Types

DES_CBC_CTX

```
typedef struct {
  UINT4 subkeys[32];                                            /* subkeys */
  UINT4 iv[2];                                      /* initializing vector */
  UINT4 originalIV[2];                        /* for restarting the context */
  int encrypt;                                             /* encrypt flag */
} DES_CBC_CTX;
```

The DES_CBC_CTX type stores the context for an single-key DES CBC operation.

subkeys	Array of DES subkeys, ordered according to initialisation (in “normal” order for encryption, “reverse” order for decryption).
iv	64-bit initialising vector (current state).
originalIV	64-bit initialising vector (initial state).
encrypt	“Direction” indicator; one for encrypt, zero for decrypt.

DESX_CBC_CTX

```
typedef struct {
  UINT4 subkeys[32];                                            /* subkeys */
  UINT4 iv[2];                                      /* initializing vector */
  UINT4 inputWhitener[2];                                /* input whitener */
  UINT4 outputWhitener[2];                              /* output whitener */
  UINT4 originalIV[2];                        /* for restarting the context */
  int encrypt;                                             /* encrypt flag */
} DESX_CBC_CTX;
```

The DESX_CBC_CTX type stores the context for an single-key DESX CBC operation.

subkeys	Array of DES subkeys, ordered according to initialisation (in “normal” order for encryption, “reverse” order for decryption).
iv	64-bit initialising vector (current state).
inputWhitener	64-bit input “whitener”, XORed with data during encryption.
outputWhitener	64-bit input “whitener”, XORed with data during encryption.

originalIV — 64-bit initialising vector (initial state).

encrypt — “Direction” indicator; one for encrypt, zero for decrypt.

DES3_CBC_CTX

```
typedef struct {
  UINT4 subkeys[3][32];                              /* subkeys for t hree operations */
  UINT4 iv[2];                                                 /* initializing vector */
  UINT4 originalIV[2];                                 /* for restarting the context */
  int encrypt;                                                        /* encrypt flag */
} DES3_CBC_CTX;
```

The DES3_CBC_CTX type stores the context for an triple-key DES CBC operation.

subkeys — Two-dimensional array of DES subkeys, ordered according to initialisation (in “normal” order for encryption, “reverse” order for decryption).

iv — 64-bit initialising vector (current state).

originalIV — 64-bit initialising vector (initial state).

encrypt — “Direction” indicator; one for encrypt, zero for decrypt.

Examples

Encryption and decryption using DES CBC

The following code sample shows the use of **DES_CBCInit** and **DES_CBCUpdate** to encrypt and decrypt a block of data.

```
void DESExample()
{
      R_RANDOM_STRUCT       randomStruct;
      DES_CBC_CTX           CBCcontext;
      unsigned char         key[8], iv[8];
      char                  demostring[] = "Simple DES demonstration string";
      char                  plaintext[256], ciphertext[256];
      int                   plaintextlength, i;

      /*    Initialise random structure      */
      R_RandomCreate(&ran domStruct);



      /*    Generate random key and iv  */
      R_GenerateBytes((unsigned char *) &key, 8, &randomStruct);
      R_GenerateBytes((unsigned char *) &iv, 8, &randomStruct);

      /*    Initialise fresh context    */
      DES_CBCInit(&CBCcontext, key, iv, 1);

      /*    Check to ensure buffer space      */
      /*    In reality, more elegant error handling is advised!         */
      plaintextlength = strlen(demostring);
      if (plaintextlength > 256) {
            plaintextlength = 256;
      }

      /*    Copy into plaintext buffer and pad if necessary             */
      strncpy(plaintext, de mostring, plaintextlength);
      while (plaintextlength % 8) {
            plaintext[plaintextlength++] = (char) '\0';
      };

      /*    Update context, eight bytes at a time         */
      for(  i=0;
            i<plaintextlength;
            i+=8 ) {
            DES_CBCUpdate(&CBCcontext, &ciphertext[i], &plaintext[i], 8);
      }
```

```
    /*    Clear context and plaintext buffer    */
    R_memset((POINTER)&CBCcontext, 0, sizeof(CBCcontext));
    R_memset((POINTER)plaintext, 0, sizeof(plaintext));

    /*    Decryption process    */

    /*    Initialise fresh context for decryption              */
    DES_CBCInit(&CBCco ntext, key, iv, 0);


    /*    Decrypt data, eight bytes at a time    */
    for(  i=0;
          i<plaintextlength;
          i+=8 ) {
          DES_CBCUpdate(&CBCcontext, &plaintext[i], &ciphertext[i], 8);
    }

    /*    Clear context    */
    R_memset((POINTER)&CBCcontext, 0, sizeof(CBCcontext));

    /*    Display decrpyted data            */
    printf("Decrypted text: %s\n", plaintext);

}
```

SECTION III
TECHNICAL DESCRIPTION

NATURAL NUMBER ARITHMETIC

Introduction

This section describes the natural number arithmetic "primitives" used by various functions within RSAEURO. The following table provides brief details, and the function descriptions which follow provide more detailed information.

NN_Decode(a, digits, b, len)	Decodes a character array representation 'a' into a "raw" value 'b'.
NN_Encode(a, len, b, digits)	Encodes a "raw" value 'a' into a character array 'b'.
NN_Assign(a, b, digits)	Assigns a = b..
NN_AssignZero(a, digits)	Zeroises 'a'.
NN_Assign2Exp(a, b, digits)	Assigns $a = 2^b$
NN_Add(a, b, c, digits)	Computes a = b + c.
NN_Sub(a, b, c, digits)	Computes a = b - c.
NN_Mult(a, b, c, digits)	Computes a = b * c
NN_LShift(a, b, c, digits)	Computes $a = b * 2^c$ (*i.e.* shifts **b** left **c** bits, returning the result in **a**).
NN_RShift(a, b, c, digits)	Computes $a = b / 2^c$ (*i.e.* shifts b right c bits, returning the result in a).
NN_Div(a, b, c, cDigits, d, dDigits)	Computes a = c div d and b = c mod d.
NN_Mod(a, b, bDigits, c, cDigits)	Computes a = b mod c
NN_ModMult(a, b, c, d, digits)	Computes a = b * c mod d
NN_ModExp(a, b, c, cDigits, d, dDigits)	Computes $a = b^c$ mod d
NN_ModInv(a, b, c, digits)	Computes a = 1/b mod c
NN_Gcd(a, b, c, digits)	Assigns a to the greatest common divisor of b and c.
NN_Cmp(a, b, digits)	Returns the sign of a - b
NN_Zero(a, digits)	Returns 1 iff a = 0
NN_Digits(a, digits)	Returns the significant length of natural number a in digits
NN_Bits(a, bits)	Returns the significant length of the natural number a in bits.
NN_DigitBits (a)	Returns the significant length of the digit a in bits.

Representation of natural numbers

Natural numbers are represented internally in RSAEURO as arbitrary-length NN_DIGIT arrays, where the NN_DIGIT type is by default an unsigned 32-bit value (UINT4). The maximum size of an NN_DIGIT array is set by MAX_NN_DIGITS which is derived from MAX_RSA_MODULUS_LEN, to ensure that all values are generated within an appropriate range. The mathematical functions effectively treat NN_DIGIT arrays as integers of an arbitrary length. In the context of natural numbers within RSAEURO, a "digit" is an NN_DIGIT element of a natural number, as opposed to the normal meaning (*i.e.* a single numerical digit). For example, a ten-digit NN_DIGIT array consists of forty bytes of data (ten UINT4s).

NN_DIGIT arrays are packed into unsigned character arrays, most significant bit first, when values are returned to higher-level functions. This behaviour is primarily to maintain compatibility with existing RSAREF code.

For the remainder of this section, the term “natural number” is used to describe an NN_DIGIT array.

Functions

NN_Decode

```
void NN_Decode (a, digits, b, len)
NN_DIGIT *a;
unsigned char *b;
unsigned int digits, len;
```

Decodes a character array into a natural number. **b** is a pointer to the character array, which is **len** bytes long. **a** is a pointer to the destination natural number, which is **digits** digits long.

digits must be large enough to accommodate **len** bytes; if it is not, the most significant bytes of **a** are truncated.

NN_Encode

```
void NN_Encode (a, len, b, digits)
NN_DIGIT *b;
unsigned char *a;
unsigned int digits, len;
```

Encodes a natural number into a character array. **b** is a pointer to the natural number, which is **digits** digits long. **a** points to the destination character array, which is **len** bytes long.

len must be long enough to accommodate **digits** digits of **b**; if it is not, the most significant digits of the natural number are truncated.

NN_Assign

```
void NN_Assign (a, b, digits)
NN_DIGIT *a, *b;
unsigned int digits;
```

Assigns a = b, where **a** and **b** are natural numbers. Note that only **digits** digits of **a** are assigned, so if NNDigits(a) > NNDigits(b), the most significant digits of **a** will not be cleared.

NN_AssignZero

```
void NN_AssignZero (a, digits)
NN_DIGIT *a;
unsigned int digits;
```

Zeroises **digits** digits of the natural number **a**.

NN_Assign2Exp

```
void NN_Assign2Exp (a, b, digits)
NN_DIGIT *a;
unsigned int b, digits;
```

Assigns $a = 2^b$. **a** is the destination natural number which has **digits** digits and **b** is the exponent. The result is undefined if **b** is greater than **digits** * NN_DIGIT_BITS.

NN_Add

```
NN_DIGIT NN_Add (a, b, c, digits)
NN_DIGIT *a, *b, *c;
unsigned int digits;
```

Computes a = b + c, and returns the carry. **a**, **b**, **c** and the return value are natural numbers, all **digits** digits long.

NN_Sub

```
NN_DIGIT NN_Sub (a, b, c, digits)
NN_DIGIT *a, *b, *c;
unsigned int digits;
```

Computes a = b - c, and returns the borrow. **a**, **b**, **c** and the return value are natural numbers, all **digits** digits long.

NN_Mult

```
void NN_Mult (a, b, c, digits)
NN_DIGIT *a, *b, *c;
unsigned int digits;
```

Computes a = b * c. **a**, **b**, **c** and the return value are natural numbers, all **digits** digits long. The result is undefined if **digits** > MAX_NN_DIGITS.

NN_LShift

```
NN_DIGIT NN_LShift (a, b, c, digits)
NN_DIGIT *a, *b;
unsigned int c, digits;
```

Computes a = b * 2^c (*i.e.* shifts **b** left **c** bits, returning the result in **a**). **a**, **b**, **c** and the return value are natural numbers, all **digits** digits long. The result is undefined if **c** > NN_DIGIT_BITS.

NN_RShift

```
NN_DIGIT NN_RShift (a, b, c, digits)
NN_DIGIT *a, *b;
unsigned int c, digits;
```

Computes a = b div 2^c (*i.e.* shifts b right c bits, returning the result in a). Returns the carry **a**, **b**, **c** and the return value are natural numbers, all **digits** digits long. The result is undefined if **c** > NN_DIGIT_BITS.

NN_Div

```
void NN_Div (a, b, c, cDigits, d, dDigits)
NN_DIGIT *a, *b, *c, *d;
unsigned int cDigits, dDigits;
```

Computes a = c div d and b = c mod d. **a**, **b**, **c** and **d** are natural numbers. **a** and **c** are **cDigits** digits long, **b** and **d** are **dDigits** digits long. The result is undefined if **d** = 0, **cDigits** >= 2 * MAX_NN_DIGITS or **dDigits** > MAX_NN_DIGITS.

NN_Mod

```
void NN_Mod (a, b, bDigits, c, cDigits)
NN_DIGIT *a, *b, *c;
unsigned int bDigits, cDigits;
```

Computes a = b mod c. **a**, **b**, and **c** are natural numbers. **a** and **c** are **cDigits** long, **b** is **bDigits** long. The result is undefined if c = 0, **bDigits** >= 2 * MAX_NN_DIGITS or **cDigits** > MAX_NN_DIGITS

NN_ModMult

```
void NN_ModMult (a, b, c, d, digits)
NN_DIGIT *a, *b, *c, *d;
unsigned int digits;
```

Computes a = b * c mod d. **a**, **b**, **c** and **d** are natural numbers, all **digits** digits long. The result is undefined if **d** = 0 or **digits** > MAX_NN_DIGITS.

NN_ModExp

```
void NN_ModExp (a, b, c, cDigits, d, dDigits)
NN_DIGIT *a, *b, *c, *d;
unsigned int cDigits, dDigits;
```

Computes a = b^c mod d. **a**, **b**, **c** and **d** are natural numbers. **a**, **b** and **d** are **dDigits** digits long, **c** is **cDigits** digits long.. The result is undefined if **d** = 0, **cDigits** = 0 or **dDigits** > MAX_NN_DIGITS.

NN_ModInv

```
void NN_ModInv (a, b, c, digits)
NN_DIGIT *a, *b, *c;
unsigned int digits;
```

Computes a = 1/b mod c. **a**, **b** and **c** are natural numbers, all **digits** digits long. The result is undefined if **b** and **c** are not relatively prime (*e.g.* gcd(b, c) is not 1) or **digits** > MAX_NN_DIGITS.

NN_Gcd

```
void NN_Gcd(a ,b ,c, digits)
NN_DIGIT *a, *b, *c;
unsigned int digits;
```

Calculates the greatest common divisor of **b** and **c**, returning the result in **a**. **a**, **b** and **c** are natural numbers, all **digits** digits long. The result is undefined if **c** < **b** or **digits** > MAX_NN_DIGITS.

NN_Cmp

```
int NN_Cmp (a, b, digits)
NN_DIGIT *a, *b;
unsigned int digits;
```

Compares a and b, returns -1 if **a**<**b**, 0 if **a**=**b** or 1 if **a**>**b**. . **a**, **b** and **c** are natural numbers, all **digits** digits long.

NN_Zero

```
int NN_Zero (a, digits)
NN_DIGIT *a;
unsigned int digits;
```

Returns 1 iff a = 0, otherwise returns 1. **a** is a natural number, **digits** digits long.

NN_Digits

```
unsigned int NN_Digits (a, digits)
NN_DIGIT *a;
unsigned int digits;
```

Returns the significant length in digits of the natural number **a** (*e.g.* the position of the first non-zero digit). **digits** is the length of **a** in digits.

NN_Bits

```
unsigned int NN_Bits (a, digits)
NN_DIGIT *a;
unsigned int digits;
```

Returns the significant length in bits of the natural number **a** (*e.g.* the position of the first non-zero bit). **digits** is the total length of **a** in digits.

GeneratePrime

```
int GeneratePrime(a, b, c, d, digits, randomStruct)
NN_DIGIT *a, *b, *c, *d;
unsigned int digits;
R_RANDOM_STRUCT *randomStruct; /* random structure */
```

Generates a random probable prime **a**, where **b** < **a** < **c** and **a**-1 is divisible by **d**. **a**, **b**, **c** and **d** are natural numbers, all **digits** digits long. **randomStruct** is an initialised R_RANDOM_STRUCT.

On exit, the generated prime is returned in **a**. Returns RE_NEED_RANDOM if **randomStruct** has not been fully initialised, RE_DATA if a suitable prime could not be found, ID_OK otherwise.

MEMORY MANIPULATION

Introduction

There are three memory manipulation functions used within RSAEURO:

R_memset	Sets a range of memory to a specified value.
R_memcpy	Copies a block of memory to another address.
R_memcmp	Compares two blocks of memory.

All of these routines are "secure", in that no intermediate storage is used during their operation.

Functions

R_memset

```
void R_memset(output, value, len)
POINTER output;                                   /* output block */
int value;                                        /* value */
unsigned int len;                                 /* length of block */
```

Sets **len** bytes starting at **output** to **value**.

R_memcpy

```
void R_memcpy(output, input, len)
POINTER output;                                   /* output block */
POINTER input;                                    /* input block */
unsigned int len;                                 /* length of blocks */
```

Copies **len** bytes from **input** to **output**.

R_memcmp

```
int R_memcmp(Block1, Block2, len)
POINTER Block1;                                   /* first block */
POINTER Block2;                                   /* second block */
unsigned int len;                                 /* length of blocks */
```

Compares **len** bytes starting at **Block1** with **Block2**.

Returns zero if the blocks are identical. If the blocks are different, returns the difference between the first two non-identical bytes (returns **Block1[difference] - Block2[difference]**, where difference is the offset of the first non-identical byte.

TECHNICAL INFORMATION

Introduction

This section contains miscellaneous technical information regarding RSEURO. The following subjects are covered:

Version information. Details of the R_RSAEuroInfo version information function.

Error Types. A complete list of RSAEURO error types and possible explanations.

Platform-specific configuration Platform-specific configuration information, including compiler settings and data structures.

References Sources of further reading for related subjects and standards

Version information

```
void R_RSAEuroInfo(info)
RSAEUROINFO info;              /* info structure */
```

The **R_RSAEuroInfo** function provides a run-time method of determining the version and supported features of RSAEuro. The function returns an RSAEUROINFO structure:

```
typedef struct {
    unsigned short int Version;                /* RSAEuro Version */
    unsigned int flags;                        /* Version Flags */
    unsigned char ManufacturerID[32];          /* Toolkit ID */
    unsigned int Algorithms;                   /* Algorithms Supported */
} RSAEUROINFO;
```

The RSAEUROINFO type provides version and algorithm information to identify the version and facilities of the RSAEuro toolkit.

Version	RSAEuro version number
flags	Reserved for future use; currently returned as zero.
ManufacturerID	Toolkit identification in ASCII. Currently set to “RSAEURO Toolkit”.
Algorithms	Bit field indicating supported algorithms, based on the following constants:

```
#define IA_MD2 0x00000001
#define IA_MD4 0x00000002
#define IA_MD5 0x00000004
#define IA_SHS 0x00000008
#define IA_DES_CBC 0x00000010
#define IA_DES_EDE2_CBC 0x00000020
#define IA_DES_EDE3_CBC 0x00000040
#define IA_DESX_CBC 0x00000080
#define IA_RSA 0x0001000
#define IA_DH 0x0020000
```

Error Types

RE_CONTENT_ENCODING	An ASCII encoding error occurred during the decoding of a content block.
RE_DATA	An error occurred during one of the mathematical routines. Usually caused by incorrect or invalid data, such as an unmatched set of Diffie-Hellman values.
RE_DIGEST_ALGORITHM	An invalid digest algorithm was selected; either an unsupported digest was selected (*i.e.* not one of the DA_xx values from RSAEURO.H), or SHS was selected for signature generation.
RE_ENCODING	An ASCII encoding error occurred during the decoding of a data block.
RE_KEY	The recovered session key cannot decrypt the associated content or signature.
RE_KEY_ENCODING	An ASCII encoding error occurred during the decoding of a session key.
RE_LEN	An out-of-range signature or session key was encountered, or the data supplied to an RSA function was too large for the key provided.
RE_MODULUS_LEN	An invalid RSA modulus length was specified (either too long or too short)
RE_NEED_RANDOM	An attempt was made to generate random data using an uninitialised random structure.
RE_PRIVATE_KEY	The supplied private key was invalid/incorrect.
RE_PUBLIC_KEY	The supplied public key was invalid/ incorrect.
RE_SIGNATURE	The signature does not match the associated data block.
RE_SIGNATURE_ENCODING	An ASCII encoding error occurred during the decoding of a signature.
RE_ENCRYPTION_ALGORITHM	An invalid encryption algorithm was specified.

Configuration parameters

The following table describes some of the values defined in the RSAEURO toolkit header files which may be modified to customise the behaviour of certain routines. Although the toolkit has been designed with portability in mind, no guarantee is made that the code will work with different settings – please report any difficulties.

Parameter	**Defined in**	**Description**	**Default value**
MIN_RSA_MODULUS_BITS	RSAEURO.H	Minimum size permitted for RSA modulus.	508 bits
MAX_RSA_MODULUS_BITS	RSAEURO.H	Maximum size permitted for RSA modulus. (**Note**: change this value to allow the use of larger keys *etc*)	1024 bits
MAX_DIGEST_LEN	RSAEURO.H	Maximum digest size, in bytes, for any of the supported algorithms.	20 bytes (SHS)
RSAEURO_VER_MAJ	RSAEURO.H	Major version number of the toolkit.	1

Parameter	Defined in	Description	Default value
RSAEURO_VER_MIN	RSAEURO.H	Minor version number of the toolkit.	04
RSAEURO_IDENT	RSAEURO.H	Identifier string for the toolkit (to support variants)	RSAEURO Toolkit
RSAEURO_DATE	RSAEURO.H	Release date of the major version of the toolkit.	See source
NN_DIGIT	NN.H	Type for natural number "digit"	UINT4 (32-bit word)
NN_DIGIT_BITS	NN.H	Number of bits in an NN_DIGIT	32
MAX_NN_DIGIT	RSAEURO.H	Maximum permitted value for an NN_DIGIT	0xFFFFFFFF
RANDOM_BYTES_RQ	R_RANDOM.C	Number of random bytes required to "seed" a **random** structure prior to use.	256
RANDOM_BYTES_RQINT	R_RANDOM.C	Number of random bytes from ANSI time functions required to "seed" a **random** structure prior to use.	512
SHS_BLOCKSIZE	SHS.H	SHS block size, in bytes.	60
SHS_DIGESTSIZE	SHS.H	SHS digest size, in bytes.	20

Platform-specific Configuration

Types

There are three platform-specific types used in RSAEURO, defined in GLOBAL.H and described in the following paragraphs.

POINTER	A generic pointer to memory. It should be possible to cast any other pointer to POINTER.
BYTE	An 8-bit byte.
UINT2	A 16-bit unsigned integer.
UINT4	A 32-bit unsigned integer.

Defined macros

RSAEURO uses three #defined macros:

PROTOTYPES — The PROTOTYPES macro indicates the form of C function declarations. If it is non-zero, functions are declared as:

```
type function (type, ...., type);
```

Otherwise, they take the form:

```
type function ();
```

USEASM — If USEASM is defined then assembler routines are used where available. Assembler versions of the key performance bottlenecks are under development. Please check the source list to see which functions have been implemented in assembler.

USE_ANSI — If USE_ANSI is defined, the ANSI-standard **memcpy**, **memcmp** and **memset** routines are used in place of the RSAEURO **stdlib** routines.

APPENDICES

APPENDIX A: FUNCTION CROSS-REFERENCE

This section provides a cross-reference for each function, indicating where it is described in the documentation. The functions are listed in alphabetical order.

APPENDIX B: REFERENCES

References

General

For general information about cryptography and its applications, consult the following:

- Bruce Schneier's "Applied Cryptography– Protocols, Algorithms, and Source Code in C"(John Wiley & Sons, ISBN 0-471-11709-9) is an excellent introduction to cryptography both from the theoretical and "real world" perspective. It contains full coverage of all the commonly-used algorithms, and a detailed examination of the accompanying protocols.

- The frequently-asked questions (FAQ) file for the Usenet newsgroup sci.crypt provides a good basic coverage of the issues involved in cryptography, and is available free of charge over the Internet (available on the Internet via anonymous ftp from `rtfm.mit.edu` in /pub/usenet/news.answers/cryptography-faq).

- RSADSI provide a good introduction to cryptography in the form of "Frequently Asked Questions About Today's Cryptography", a document available free of charge by anonymous FTP from `ftp.rsa.com`.

- The author of RSAEURO maintains a good cryptography-based World Wide Web page at `http://www.sourcery.demon.co.uk`. The page contains links to many other cryptography and security–related sites.

- RSADSI provide a number of standards relating to "real-life" usage of cryptographic algorithms and protocols known as the Public Key Cryptography Standards (PKCS), as follows:

  - *PKCS #1: RSA Encryption Standard*. PKCS #1 describes a method, called rsaEncryption, for encrypting data using the RSA public-key cryptosystem.

  - *PKCS #3: Diffie-Hellman Key Agreement Standard*. PKCS #3 describes a method for implementing Diffie-Hellman key agreement, whereby two parties, without any prior arrangements, can agree upon a secret key that is known only to them (and, in particular, is not known to an eavesdropper listening to the dialogue by which the parties agree on the key).

  - *PKCS #5: Password-Based Encryption Standard*. PKCS #5 describes a method for encrypting an octet string with a secret key derived from a password. The result of the method is an octet string.

  - *PKCS #6: Extended-Certificate Syntax Standard*. PKCS #6 describes a syntax for extended certificates. An extended certificate consists of an X.509 public-key certificate and a set of attributes, collectively signed by the issuer of the X.509 public-key certificate.

  - *PKCS #7: Cryptographic Message Syntax Standard*. PKCS #7 describes a general syntax for data that may have cryptography applied to it, such as digital signatures and digital envelopes.

  - *PKCS #8: Private-Key Information Syntax Standard*. PKCS #8 describes a syntax for private-key information. Private-key information includes a private key for some public-key algorithm and a set of attributes.

  - *PKCS #9: Selected Attribute Types*. PKCS #9 defines selected attribute types for use in PKCS #6 extended certificates, PKCS #7 digitally signed messages, and PKCS #8 private-key information.

  - *PKCS #10: Certification Request Syntax Standard*. PKCS #10 describes a syntax for certification requests.

  Details of the PKCS standards can be obtained by mailing `pkcs@rsa.com` or via anonymous ftp from `ftp.rsa.com`.

- National Bureau of Standards. FIPS PUB 113: Computer data authentication, 30 May 1985.
- Privacy and Authentication: An Introduction to Cryptography, W. Diffie, M.E. Hellman, Proc. of the IEEE, Vol 67, No. 3, March 1979.

RSA

For further details of the RSA algorithm, consult the following:

- R. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Communications of the ACM, v. 21, n. 2, 2 Feb 1978, pp. 120–126.
- R. Rivest, A. Shamir, and L. Adleman, “On Digital Signatures and Public-Key Cryptosystems”, MIT Laboratory for Computer Science, Technical Report, MIT/LCS/TR-212, Jan 1979
- R. Rivest, A. Shamir, and L. Adleman, “Cryptographic Communications System and Method”, US Patent 4,405,829, 20/9/83.

Diffie-Hellman

For further details of the Diffie-Hellman key exchange algorithm, consult the following:

- W. Diffie and M.E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, IT-22:644-654, 1976.
- RSA Laboratories. PKCS #3: Diffie-Hellman Key-Agreement Standard.

Digest Algorithms

For further details of the digest algorithms used in RSAEURO, consult the following:

- B. Kaliski. RFC 1319: The MD2 Message-Digest Algorithm. April 1992.
- R. Rivest. RFC 1320: The MD4 Message-Digest Algorithm. April 1992.
- R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. April 1992.
- NIST FIPS PUB 180, “Secure Hash Standard”, National Institute of Standards and Technology, US Department of commerce, April 1993 [draft].

DES

For further details of the Data Encryption Standard, consult the following:

- National Bureau of Standards. FIPS Publication 46-1: Data Encryption Standard. January 1988.
- National Bureau of Standards. FIPS Publication 81: DES Modes of Operation. December 1980.
- National Bureau of Standards. FIPS PUB 74: Guidelines for implementing and Using the NBS Data Encryption Standard. 1 April 1981.
- Exhaustive Cryptanalysis of the NBS Data Encryption Standard, W.Diffie & M.E.Hellman, IEEE Computer, June 1977.
- An Application of a Fast Data Encryption Standard Implementation, Matt Bishop, Computing Systems, Vol. 1, No. 3, Summer 1988.
- Differential Cryptanalysis of DES-like Cryptosystems, E.Biham and A.Shamir, Journal of Cryptology.
- A High-Speed Software DES Implementation, D.C.Feldmeier, Computer Communications Research Group, Bellcore, June 1989.

Privacy-enhanced mail

For further details of Internet privacy-enhanced mail and its applications, consult the following:

- J. Linn. RFC 1421: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. February 1993.
- S. Kent. RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. February 1993.
- D. Balenson. RFC 1423: Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. February 1993.
- B. Kaliski. RFC 1424: Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. February 1993.
- Privacy-Enhanced Electronic Mail, Matt Bishop, Dept. of Maths and Computer Science, Dartmouth College.
- Recent Changes to Privacy Enhanced Electronic Mail, Matt Bishop.