Cybersecurity MSc: Applied Cryptography 2023-24

Practical Exercises: Asymmetric Cryptography – RSA

September 14, 2023 Due date: no date

Changelog

- v1.0 - Initial version.

Introduction

In order to elaborate this laboratory guide it is required to install the Java Development Environment (JDK), Python 3 or the C compiler and development environment.

The examples provided will use both Java, Python and C. For Python you need to install the `cryptography` module. For C we will use the EVP (Digital Envelope) functions from the Crypto library that is part of OpenSSL, and for that you need to use install the packages `libssl1.0-dev` and `libssl1.1`.

1 Elements of interest for each language

1.1 Java

- javax.crypto.**Cipher** An instance of the `Cipher` allows to encrypt original contents or decrypt cryptograms. Some important methods are:
  - `getInstance`: creates an instance of a encryption/decryption engine for a given algorithm;
  - `init`: initializes the engine instance; typically used to set the key to be used, at least;
  - `update`: performs an incremental encryption/decryption with the engine instance. Engines that work sequentially can keep some internal state across successive calls;
  - `doFinal`: terminates a sequential process (not required for non-sequential processes).
- java.security.**KeyPairGenerator** An instance of the class `KeyPairGenerator` is a generator of asymmetric keys. Some important methods are:
  - `getInstance`: creates an instance of a generator for producing key pair for a given cipher algorithm;
  - `init`: initializes the key generator with some elements for helping the key production;
  - `generateKeyPair` effectively creates an asymmetric key.

1.2 Python

- The cryptography module is both a frontend, high-level interface for dealing with cryptography and a default backend implementation of the fundamental cryptographic primitives.
- The Python interpreter distinguishes bytes from text characters, and thus byte arrays from strings. A string is more than a byte array, it has also an encoding (e.g. UTF-8, Unicode, etc.). You can get the bytes of characters (strings) using the `bytes` function, and you can convert bytes to characters (strings) by using the method `decode` on a byte array (with a target encoding).

  By default, in cryptography we always use bits and bytes, nothing else. So, do not attempt to solve interpreter errors by changing everything to text (characters and strings)!

1.3 C

- The function `EVP_PKEY_encrypt` is the high-level interface for handling public key encryptions, whilst `EVP_PKEY_decrypt` handle private key decryptions. Both functions use an `EVP_PKEY_CTX` context, which needs to be created and set to encrypt or decrypt with functions `EVP_PKEY_-encrypt_init` or `EVP_PKEY_decrypt_init`, respectively.
- A structure `EVP_PKEY` describes an asymmetric key pair that can be an RSA key pair.
- A structure `EVP_PKEY_CTX` describes the current context of a cryptographic transformation, and it is created with `EVP_PKEY_CTX_new` or `EVP_PKEY_CTX_new_id` and released with `EVP_PKEY_-CTX_free`. This context is used to store fundamental data elements (e.g. keys, padding, etc.) used to process cryptographic transformations and deal with padding.

2 Creation and storage of a random asymmetric key pair

Develop a program to generate a random RSA key pair and save it to a file.

2.1 Generation parameters

The fundamental parameter that is required is the dimension of n, usually provided in bits. The dimension of n will be used by the generator to select the appropriate p and q (prime) values. The dimension of n broadly defines the quality of the RSA key pair.

A secondary parameter is the value of e. Most generators allow programs to specify one, but they usually default to suitable values, such as 3 or $2^{16} + 1$ (both are prime values with only two 1 bits).

A third parameter is a randomness source. The values of p and q are usually randomly set, and then tested for primality. Generators usually use a default randomness source, hopefully a good one.

The following code samples exemplify how we can generate a random, 2048-bit RSA key pair and inspect its values using Java:

```java
import java.security.KeyPairGenerator;
import java.security.spec.RSAKeyGenParameterSpec;
import java.security.KeyPair;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

RSAKeyGenParameterSpec spec;
KeyPairGenerator generator;
KeyPair keyPair;
RSAPrivateKey prvKey;
RSAPublicKey pubKey;

// Specify how to generate a new RSA key pair with 2048 bits
// F4 means 65537

spec = new RSAKeyGenParameterSpec( 2048, RSAKeyGenParameterSpec.F4 );

// Create an RSA key pair generator

generator = KeyPairGenerator.getInstance( "RSA" );

// Initialized the generator and use it to generate a new key pair

generator.initialize( spec );
keyPair = generator.generateKeyPair();

prvKey = (RSAPrivateKey) keyPair.getPrivate();
pubKey = (RSAPublicKey) keyPair.getPublic();

// Just for debugging, let's print some details about the generated key pair

System.out.printf( "e = %s\n", pubKey.getPublicExponent().toString( 16 ) );
System.out.printf( "d = %s\n", prvKey.getPrivateExponent().toString( 16 ) );
System.out.printf( "n = %s\n", pubKey.getModulus().toString( 16 ) );
```

The following code does the same in Python:

```python
from cryptography.hazmat.primitives.asymmetric import rsa

# Generate key pair.

private_key = rsa.generate_private_key( public_exponent=65537, key_size=2048 )
public_key = private_key.public_key()

# Just for debugging, let's print some details about the generated key pair

print( "e = %s\n" % (public_key.public_numbers().e) )
print( "d = %s\n" % (private_key.private_numbers().d) )
print( "n = %s\n" % (public_key.public_numbers().n) )
```

The following code does the same in C:

```c
#include <stdio.h>

#include <openssl/evp.h>
#include <openssl/bn.h>

EVP_PKEY_CTX *ctx;
EVP_PKEY *key_pair = 0;
ENGINE * engine = 0;
const OSSL_PARAM * key_pair_params;
int max_size;
BIGNUM * bn;

// Create an RSA key pair context with the default engine

ctx = EVP_PKEY_CTX_new_id( EVP_PKEY_RSA, engine );
if (ctx == 0) {
    // Error
}

if (EVP_PKEY_keygen_init( ctx ) <= 0) {
    // Error
}

// Define required key pair parameters

if (EVP_PKEY_CTX_set_rsa_keygen_bits( ctx, 2048 ) <= 0) {
    // Error
}

// You can define other parameters with other
// EVP_PKEY_CTX_set_rsa_... functions

// Generate key pair.
// The space for it is allocated by the generator

if (EVP_PKEY_keygen( ctx, &key_pair) <= 0) {
    // Error
}

key_pair_params = EVP_PKEY_gettable_params( key_pair );
if (key_pair_params == 0) {
    // Error
}

printf( "Key pair parameter list\n" );

for (int i = 0; key_pair_params[i].key != 0; i++) {
    printf( "\t%s (of type %d)\n", key_pair_params[i].key, key_pair_params[i].data_type );
}

if (EVP_PKEY_get_int_param( key_pair, "max-size", &max_size ) == 1) {
   printf( "max-size (block size without padding) = %d\n", max_size );
}

bn = BN_new();
if (EVP_PKEY_get_bn_param( key_pair, "e", &bn ) == 1) {
   printf( "e = %s\n", BN_bn2hex( bn ) );
}
if (EVP_PKEY_get_bn_param( key_pair, "d", &bn ) == 1) {
   printf( "d = %s\n", BN_bn2hex( bn ) );
}
if (EVP_PKEY_get_bn_param( key_pair, "n", &bn ) == 1) {
   printf( "n = %s\n", BN_bn2hex( bn ) );
}
BN_free( bn );
```

2.2 Randomness sources

RSA key pairs are commonly generated from a randomness source. This is fundamental to ensure the unpredictability of private keys. This said, one should guarantee that key pairs are produced from good randomness sources.

Good, in this context, means that the bits provided by that source are statistically acceptable as random but, more importantly, they are unpredictable. Unpredictability means that each bit does not depend on the ones generated before and does not influence the ones produced afterwards.

Operating system pseudo-devices for producing random byte streams, such as `/dev/random` in Linux, are good randomness choices. For more details on randomness for security check the RFC 4086.

2.3 Key pair storage

RSA key pairs are stored as follows:

- A file that contains both keys of the key pair (and possibly some other parameters, such as p and q, to speed up operations with the private key).

  This is how the key pair owner stores it. The private key can be protected by a password, so that only the key owner could have access to the private key from the file (otherwise, anyone with access to the file could do it).

- A file that contains only the public key.

  This file is used for transmitting the public key to others than the key pair owner.

There are many formats that can be used to store key pairs and public keys:

- Distinguished Encoding Rule (DER), This is a binary format that uses an ASN.1 encoding.
- Public Key Cryptography Standard #12 (PKCS #12), This is also a binary format that uses an ASN.1 encoding. This format does not store directly public keys, but rather certificates of public keys.
- Privacy Enhanced Mail (PEM). This is a textual format used to encode DER or PKCS #12 representations.

PEM is very convenient for transmiting keys through communication channels that at their lowest level only handle textual contents (e-mail, HTTP, etc.) and facilitates the management of keys and key pairs on files, since they can be concatenated.

PEM files start and end with a characeristic header and trailer line, respectively:

- `---BEGIN PRIVATE KEY---` and `---END PRIVATE KEY---`. These lines are used to bound a key pair (with a cleartext private key). They surround a textual encoding of an PKCS #8 unencrypted private key information.
- `---BEGIN ENCRYPTED PRIVATE KEY---` and `---END ENCRYPTED PRIVATE KEY---`. These lines are used to bound a key pair (with an encrypted private key). They surround a textual encoding of a PKCS #8 encrypted private key information.
- `---BEGIN PUBLIC KEY---` and `---END PUBLIC KEY---`. These lines are used to bound a public key. They surround a textual encoding of an ANS.1-encoded (using BER or DER) public key information.

When a file stores a password-encrypted private key, it must as well record how the private key was encrypted (which algorithm was used). This includes the algorithm to transform the password to a symmetric key (for a given algorithm) and the algorithm to encrypt the private key with that key. This is implemented by means of a different format, PKCS #8.

The following code samples exemplify how we can print PEM encodings of an RSA key pair using Java:

```
import java.security.KeyPair;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

import java.util.Base64;

prvKey = (RSAPrivateKey) keyPair.getPrivate();
pubKey = (RSAPublicKey) keyPair.getPublic();

// Let's dump the generated keys in different PEM formats (with different contents)

// Create a PEM encoder, which is a Base64 MIME encoder with 64-character lines

pemEncoder = Base64.getMimeEncoder( 64, lineSeparator );

// Write the key pair to stdout
// No private key encryption

asn1Object = prvKey.getEncoded();
out.printf( "-----BEGIN PRIVATE KEY-----\n" +
            pemEncoder.encodeToString( asn1Object ) +
            "\n-----END PRIVATE KEY-----\n" );

//  Write the publick key of a key pair to stdout

asn1Object = pubKey.getEncoded();
out.printf( "-----BEGIN PUBLIC KEY-----\n" +
            pemEncoder.encodeToString( asn1Object ) +
            "\n-----END PUBLIC KEY-----\n" );
```

The following code does the same in Python:

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

public_key = private_key.public_key()

# Let's dump the generated keys in different PEM formats (with different contents)

# Write the key pair to stdout
# No private key encryption

pem = private_key.private_bytes( encoding=serialization.Encoding.PEM,
                                   format=serialization.PrivateFormat.TraditionalOpenSSL,
                                   encryption_algorithm=serialization.NoEncryption() )
print( pem.decode( 'ascii' ) )

# Write the key pair to stdout
# The private key is encrypted with AES+CBC and a key derived from the password "senha"

pem = private_key.private_bytes( encoding=serialization.Encoding.PEM,
                   format=serialization.PrivateFormat.PKCS8,
                   encryption_algorithm=serialization.BestAvailableEncryption( b'senha' ) )
print( pem.decode( 'ascii' ) )

# Write the publick key of a key pair to stdout

pem = public_key.public_bytes( encoding=serialization.Encoding.PEM,
                               format=serialization.PublicFormat.SubjectPublicKeyInfo )
print( pem.decode( 'ascii' ) )
```

The following code does the same in C:

```c
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/pem.h>

EVP_PKEY *key_pair;

// Write the key pair to stdout
// No private key encryption

PEM_write_PrivateKey( stdout, key_pair, 0, 0, 0, 0, 0 );

// Write the key pair to stdout
// The private key is encrypted with AES+CBC and a key derived from the password "senha"

PEM_write_PrivateKey( stdout, key_pair, EVP_aes_128_cbc(), "senha", 5, 0, 0 );

// Write the publick key of a key pair to stdout

PEM_write_PUBKEY( stdout, key_pair );
```

3 Padding and RSA

When we encrypt data with RSA we need to use some kind of padding. The padding varies, one can use several padding strategies. Furthermore, there are different paddings when one uses the private of the public key for encrypting.

Since encryption with private keys is usually explored for signing data, the padding for encryptions with private keys has some impact of the exact data that is encrypted, besides the padding used (it needs to have a special, well-defined structure). Therefore, hereafter we will consider only padding for encryptions with public keys.

Padding is crucial to implement a randomized encryption, by using a fixed and a random parts. This means that if you cipher several times the same input, you get different outputs. This is fundamental to avoid discovering inputs of public key ciphers by trial & error.

Padding reduces the amount of data you can encrypt with an RSA key. RSA implements a block encryption, but unlike with symmetric ciphers, an encryption operation takes less data then it produces, due to padding.

Padding is ordinarily performed by the functions that perform RSA encryptions, and not manually by programmers. But, in some cases, they can take parameters.

There are two kinds of padding for public key encryptions:

- PKCS #1 v1.5. This is the oldest padding and has no parameters. It takes at least 11 bytes.
- OAEP (Optimal Asymmetric Encryption Padding). This padding is preferable to PKCS #1 v1.5. It uses a digest function to implement part of the fixed padding, and its minimum length depends on the output length h of that function: $2 + 2\cdot$. The digest function is used to process a so-called label, which is optional.

  It also uses a so-called MGF (Mask Generation Function), which is able to produce a variable-length byte mask from a variable-length input using a digest function. This last digest function can be different from the previous one.

The following code samples exemplify how we can use an OAEP padding using Java:

```java
import java.security.KeyPair;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.MessageDigest;
import javax.crypto.spec.OAEPParameterSpec;
import java.security.spec.MGF1ParameterSpec;
import javax.crypto.spec.PSource.PSpecified;
import javax.crypto.Cipher;
import java.util.Arrays;

KeyPair keyPair;
RSAPrivateKey prvKey;
RSAPublicKey pubKey;
OAEPParameterSpec padding;
MessageDigest hash1, hash2;
String label = "a label of your choice, possibly empty";
Cipher cipher;
byte[] msg1, msg2, ciphertext;

prvKey = (RSAPrivateKey) keyPair.getPrivate();
pubKey = (RSAPublicKey) keyPair.getPublic();

// Setup the OAEP padding
// (we used 2 different hash functions, but that is optional)

hash1 = MessageDigest.getInstance( "SHA-256" );
hash2 = MessageDigest.getInstance( "SHA-512" );
padding = new OAEPParameterSpec( hash1.getAlgorithm(),
                                 "MGF1",
                                 new MGF1ParameterSpec ( hash2.getAlgorithm() ),
                                 new PSpecified( label.getBytes() ) );

// Generate a byte array msg1 with the maximum length and encrypt it
// Note: the maximum length is given by the maximum size of the
//       public key modulus (n), in bytes, subtracted by the
//       minimum padding length (2 + 2 x len(hash1_digest))

msg1 = new byte[(pubKey.getModulus().bitLength() + 7) / 8 - 2 * hash1.getDigestLength() - 2];

cipher = Cipher.getInstance( "RSA/ECB/OAEPPadding" );
cipher.init( Cipher.ENCRYPT_MODE, pubKey, padding );
ciphertext = cipher.doFinal( msg1 );

// Check to see if the decryption works (you need to use the same padding!)

cipher.init( Cipher.DECRYPT_MODE, prvKey, padding );
msg2 = cipher.doFinal( ciphertext );

if (Arrays.equals( msg1, msg2 ) ) {
    out.println( "True" );
}
else {
    out.println( "False" );
}
```

The following code does the same in Python:

```python
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

public_key = private_key.public_key()

# Define a padder (we used 2 different hash functions, but that is optional)

hash1 = hashes.SHA256()
hash2 = hashes.SHA512()
padder = padding.OAEP( mgf=padding.MGF1( algorithm=hash2 ),
                       algorithm=hash1,
                       label="a label of your choice, possibly empty" )

# Fill a byte array msg1 with the maximum length with zeros and encrypt it
# Note: the maximum length is given by the maximum size of the
#       public key modulus (n), in bytes, subtracted by the
#       minimum padding length (2 + 2 x len(hash1_digest))

msg1 = bytearray( public_key.key_size // 8 - 2 * hash1.digest_size - 2 )
ciphertext = public_key.encrypt( bytes( msg1 ), padder )

# Check to see if the decryption works (you need to use the same padder!)

msg2 = private_key.decrypt( ciphertext, padder )

# msg1 must be equal to msg2

print( msg1 == msg2 )
```

The following code does the same in C:

```
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include <malloc.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX * ctx;
EVP_PKEY * key_pair = 0;
ENGINE * engine = 0;
const EVP_MD * hash1, * hash2;
char * label_contents = "a label of your choice, possibly empty";
char * label;
uint8_t * msg1, * msg2, * ciphertext;
size_t msg1_len, msg2_len, ciphertext_len, label_len;

// Initiate an encryption context with a key pair

ctx = EVP_PKEY_CTX_new( key_pair, engine );
if (ctx == 0) {
    // Error occurred
}

if (EVP_PKEY_encrypt_init ( ctx ) <= 0) {
    // Error
}

// Setup the OAEP padding
// (we used 2 different hash functions, but that is optional)

if (EVP_PKEY_CTX_set_rsa_padding( ctx, RSA_PKCS1_OAEP_PADDING ) <= 0) {
    // Error
}

hash1 = EVP_sha256();
hash2 = EVP_sha512();

EVP_PKEY_CTX_set_rsa_oaep_md( ctx, hash1 );
EVP_PKEY_CTX_set_rsa_mgf1_md( ctx, hash2 );
label = strdup( label_contents ); // The encryption engine takes control of this memory buffer
EVP_PKEY_CTX_set0_rsa_oaep_label( ctx, label, strlen(label) );

// Allocate the longest possible message to encrypt
// Note: the maximum length is given by the maximum size of the
//       public key modulus (n), in bytes, subtracted by the
//       minimum padding length (2 + 2 x len(hash1_digest))

msg1_len = EVP_PKEY_get_size( key_pair ) - 2 * EVP_MD_size( hash1 ) - 2;
msg1 = alloca( msg1_len );

// Determine the output ciphertext length

if (EVP_PKEY_encrypt( ctx, 0, &ciphertext_len, msg1, msg1_len ) <= 0) {
    // Error
}

ciphertext = alloca( ciphertext_len );

// Encrypt msg1

if (EVP_PKEY_encrypt( ctx, ciphertext, &ciphertext_len, msg1, msg1_len ) <= 0) {
    // Error
}

EVP_PKEY_CTX_free( ctx );

// Check to see if the decryption works (you need to use the same padder!)

// Initiate a decryption context with the same key pair

ctx = EVP_PKEY_CTX_new( key_pair, engine );
if (ctx == 0) {
    // Error occurred
}

if (EVP_PKEY_decrypt_init ( ctx ) <= 0) {
    // Error
}

// Setup the OAEP padding

if (EVP_PKEY_CTX_set_rsa_padding( ctx, RSA_PKCS1_OAEP_PADDING ) <= 0) {
    // Error
}

EVP_PKEY_CTX_set_rsa_oaep_md( ctx, hash1 );
EVP_PKEY_CTX_set_rsa_mgf1_md( ctx, hash2 );
label = strdup( label_contents ); // The decryption engine takes control of this memory buffer
EVP_PKEY_CTX_set0_rsa_oaep_label( ctx, label, strlen(label) );

// Determine the plaintext buffer length

if (EVP_PKEY_decrypt( ctx, 0, &msg2_len, ciphertext, ciphertext_len ) <= 0) {
    // Error
}

msg2 = alloca( msg2_len );

// Retrieve msg2

if (EVP_PKEY_decrypt( ctx, msg2, &msg2_len, ciphertext, ciphertext_len ) <= 0) {
    // Error
}

// msg1 and msg2 must be equal

if (msg1_len == msg2_len && memcmp( msg1, msg2, msg1_len ) == 0) {
    printf( "True\n" );
}
else {
    printf( "False\n" );
}
```

4 Bulk data encryption and decryption using an RSA key pair

Bulk data encryption with RSA can be performed by using a conventional ECB approach, since RSA implements a block cipher. However, it is not convenient for several reasons: it is very slow (asymmetric techniques are much slower than symmetric) and each block takes a portion of padding, which creates a non-neglectable space overhead on the cryptogram.

A more convenient approach consists on using the so-called hybrid, or mixed encryption. First, a random symmetric key is generated (usually called the message key). Then, that key is used to encrypt the message data, using some symmetric algorithm (e.g. AES) and cipher mode, and the cryptogram is complemented (usually concatenated) with another cryptogram resulting from the encryption of the message key with the public key of the recipient. This makes the opposite: decrypts first the cryptogram hiding the message key, and then uses it to decrypt cryptogram hiding the message data.

Write a program to encrypt the contents of a file using an hybrid encryption. The program should accept three parameters: the key file, the input and the output files.

Tip: you can use less arguments and consider the use of `stdin` and `stdout` in the absence of file specifications.

Check the size of the resulting encrypted files for each input file. Explain why they are always bigger and the actual size increment.

Develop a program similar to the previous but able to decipher a file. Three arguments should be sent to the program: the key file, file to decipher and the output file.

Tip: Try to use a program written in a different language than the one with which you did the encryption.

References

- Randomness Requirements for Security, RFC 4086, June 2005
  `https://www.rfc-editor.org/rfc/pdfrfc/rfc4086.txt.pdf`
- PKCS #12: Personal Information Exchange Syntax v1.1, RFC 7292, July 2014
  `https://www.rfc-editor.org/rfc/pdfrfc/rfc7292.txt.pdf`)
- The Java Tutorial: Security Features in Java SE
  `https://docs.oracle.com/javase/tutorial/security/index.html`
- Java Documentation: Security
  `http://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`
- Welcome to pyca/cryptography
  (`https://cryptography.io`)
- Libcrypto API
  (`https://wiki.openssl.org/index.php/Libcrypto_API`)