UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

# DEPARTMENT OF ELECTRICAL, ELECTRONIC AND COMPUTER ENGINEERING

# NETWORK SECURITY (EHN 410)

# PRACTICAL 3 GUIDE

# Contents

# 1 PRACTICAL OVERVIEW

The following practical will implement a RC4 stream cipher to exchange information/files between two users. In order to secure the transfer of the key, the users will make use of the RSA algorithm. It is therefore required that the following executables be implemented:

- **rsakeygen** – generation of RSA key parameters
- **rsaencrypt** – encryption of a $n$-bit value (of **ANY** size) using the RSA algorithm
- **rsadecrypt** – decryption of an $n$-bit encrypted key using the RSA algorithm
- **rc4** - The RC4 stream cipher.

The order of execution for the entire system will be as follows:

1. **rsakeygen** to generate public- and private-keys for RSA
2. **rc4** encrypt a file.
3. **rsaencrypt** to encrypt the RC4 key
4. **rsadecrypt** to decrypt the RC4 key,
5. **rc4** to then decrypt the file using the decrypted key

For simplicity it can be assumed that the encrypted RC4 key and file are sent separately (two files). Furthermore, for a RSA key-size of 128-bits the first byte of the key is assumed to be the MSB. Thus a key with bytes $0x01, 0x23, 0x45$ would thus be padded as

$$0x01234500000000000000000000000000$$

to form the full key. The subsequent sections provide more details for each of the aforementioned executables.

## 1.1 RSA KEY GENERATION

The usage for the key generation utility is as follows:

```
rsakeygen -b bits -KU public_key_file -KR private_key_file -key key
```

The `bits` specify the number of bits that need to be generated for the given key. The `key` that will be used to set the RNG (Random Number Generator) seed is specified as hexadecimal numbers in the command-line parameters. The `public_key_file` is the filename to which the public key should be written. The `private_key_file` is the filename to which the private key should be written. The key files should be text files. The $n$ parameter is written first to the file,

followed by a newline character. After that, the *d* or *e* parameter is written to file. All the keys should be written to file in base-10. An example of a file is given below:

```
744878218330144283309668055455880340415520323366244076540197 8
579845424461157612190571103849169750335041829882058454349171 6
280890434483333688486835975045 71 <newline>
65537 <newline>
```

The e-parameter will thus be 65537. Please note that the generation of the random numbers should be exactly the same as described in the practical lecture, otherwise you will not be able to test your implementation. You may assume that for the demo, the e-parameter will remain fixed (i.e. 65537 )

### 1.1.1 Generating and testing random numbers

When generating the random numbers it is important to avoid creating short primes. This can be accomplished using the following procedure:

- To prevent against short primes, set the first binary digit (MSB) of the number to 1
- Calculate the next digits using your RC4 implementation
- Use `mpz_nextprime` to calculate the first prime number larger than the random number generated

## 1.2 RSA encryption

The usage for the encryption utility is:

```
rsaencrypt -key key -fo outputfile -KU public_key_file
```

The `rsaencrypt` program prompts the user for a key. The key can be a maximum of 16 characters long. (It is assumed that the key is 16 characters of plaintext. This is just to simplify the implementation.) This is the one-time key that is going to be used for the RC4 encryption and decryption. If the key is shorter than 16 bytes, the rest of the key should be zer-padded (i.e. 0x00).

The RSA encryption algorithm will then encrypt the RC4 key using the previously generated public key. The result is written to a text file (again, one line containing the encrypted value, base-10).

**Note:** The plaintext RC4 key can also be stored in a key-file for ease of use (see 1.4).

### 1.3 RSA DECRYPTION

The usage for the RSA decryption utility is:

```
rsadecrypt -fi inputfile -KR private_key_file -fo outputfile
```

The encryption utility reads the encrypted RC4 key from the input file. It then decrypts the RC4 key using the private RSA key and writes the result to the output file.

### 1.4 RC4

The usage for the rc4 utility is:

```
rc4 -fi inputfile -fo outputfile -kf keyfile
```

If no key file is specified, the encryption utility prompts the user for a key. It then encrypts the input file to the output file using RC-4. Optionally a key file can be specified that contains the key. It is again assumed that the key is a text string. The password can be up to 16 bytes long (the password is not padded in this case, as was the case with RSA). Test your RC4 implementation against the test vectors given in the IETF draft standard for ARCFOUR. Your implementation must be capable of using the output file from your rsadecrypt program as keyfile. Please implement the following functions in your RC4 implementation:

```
void rc4_init(rc4info_t *rc4i, unsigned char *key, int keylen);
unsigned char rc4_getbyte(rc4info_t* rc4i);
```

You are free to define the `rc4info_t` structure as you wish.

## 2 TEST VECTORS

### 2.1 TESTING RC4/RGN

Below are some test vectors that can be used to initialise RC4. Make sure your implementation also works with other key lengths. Find RC4 test vectors to test your implementation.

**Input key (64-bits):**

`0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD,0xEF`

**Output:**

`0x74,0x94, 0xC2, 0xE7, 0x10, 0x4B, 0x08, 0x79, ...`

**Input key (128-bits):**

`0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10`

**Output:**

```
0x9a, 0xc7, 0xcc 0x9a, 0x60, 0x9d, 0x1e, 0xf7, 0xb2, 0x93, 0x28, 0x99, ...
```

## 2.2 TESTING RSA KEY GENERATION

### 2.2.1 EXAMPLE 1:

To test the RSA key generation, make use of the following procedure:

- Initialize your random number generator with: $0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF$
- The least significant bit of each output byte from the stream is used to generate on digit of the random number
- 18-bit key-length used in the test data (to make comparison easier)
- Encryption/decrytion with such a short key-length will not work since M must be smaller than n

**First half:**

```
bitnum 1 rn: 0x74 bitval 0
bitnum 2 rn: 0x94 bitval 0
bitnum 3 rn: 0xC2 bitval 0
bitnum 4 rn: 0xE7 bitval 1
bitnum 5 rn: 0x10 bitval 0
bitnum 6 rn: 0x4B bitval 1
bitnum 7 rn: 0x8 bitval 0
bitnum 8 rn: 0x79 bitval 1
random value (base-2) : 100010101
```

**Second half:**

```
bitnum 1 rn: 0x0D bitval 1
bitnum 2 rn: 0x4B bitval 1
bitnum 3 rn: 0xD5 bitval 1
bitnum 4 rn: 0x53 bitval 1
bitnum 5 rn: 0x32 bitval 0
bitnum 6 rn: 0x8F bitval 1
bitnum 7 rn: 0x1E bitval 0
bitnum 8 rn: 0xFC bitval 0
random value (base-2) : 111110100
```

**Output:**

```
p : 281
q : 503
n : 141343
e : 65537
d : 111473
```

### 2.2.2 EXAMPLE 2:

This example provides the information for generating a 128-bit RSA key pair using a initialisation key (characters) of *abcdefhgi*

**RNG values:**

```
e4
7b
b2
97
2b
6f
e1
7a
.
.
.
eb
14
f2
92
4d
a1
```

**Output**

```
first random value (base-10): 12622624516681506658
second random value (base-10): 10325958134448386374
p = 12622624516681506749
q = 10325958134448386513
n = 130340692306115037894155577508770076237
Q(n) = 130340692306115037871206994857640182976
e = 65537
d = 955007050040028991796616977595712947 85
```

# 3 ADDITIONAL RESOURCES

1. GNU Multi-precision library

    - GNU Multi-precision library: http://gmplib.org
    - GMP library documentation: http://gmplib.org/gmp-man-4.2.1.pdf

2. RSA encryption algorithm

    - Stallings, W., Network security essentials, 3rd edition.
    - Schneier, B., Applied cryptography : protocols, algorithms, and source code in C.

3. RC4 algorithm

    - Stallings, W., Network security essentials, 3rd edition.
    - IETF draft standard: http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt