Assignment 6: Public Key Cryptography - Design

Thadeus Ballmer

November 2021

1 Introduction

In this assignment public key cryptography is implemented utilizing RSA encryption. Three binaries are required:

1. `keygen`
2. `encrypt`
3. `decrypt`

The `keygen` binary generates a RSA public and private key pair. The `encrypt` binary encrypts a file with the public key generated from `keygen`. The `decrypt` binary decrypts the encrypted file using the private key matching the public key.

2 GNU Multiple Precision Arithmetic

The GMP library facilities computation using numbers with arbitrary precision. This library is needed because C does not include types large enough which RSA needs in order to generate a secure encryption. Since GMP's random functions are needed, two utility functions are required: a function to set the random state and a function which clears and frees memory used by the state.

```
void randstate_init(uint64_t seed)
    gmp_randinit_mt(state) // state is an extern variable declared in randstate.h
    gmp_randseed_ui(state, seed)

void randstate_clear(void)
    gmp_randclear(state)
```

It is worth mentioning that gmp variables are inititalized as the first element of an array of a structure. This means the variables act as pointers. Therefore, when passed as parameters they mimic call-by-reference functionality.

3 Number Theoretic Functions

The number theoretic functions declared in `numtheory.h` are necessary for the RSA computations. These functions are used extensively in the RSA library implementation.

3.1 Modular Exponentiation

Although a function to calculate Modular exponentiation can be found in the GMP library, this assignment calls for a custom implementation.

```
void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)
    out = 1
    power = base
    while exponent > 0
        if odd(exponent)
            out = (out * p) mod modulus
        power = (power * power) mod modulus
        exponent = floor(exponent / 2)
```

3.2 Primality Testing

This test implements the Miller-Rabin primality test, even though it is a pseudo-prime test. A deterministic primality test would take too much time with the size of numbers used.

```
bool is_prime(mpz_t n, uint64_t iters)
    write n - 1 = 2^s * r such that r is odd
    for (i = 1; i <= iters; i += 1)
        choose random a in the set {2,3,...,n-2}
        y = pow_mod(a, r, n)
        if y != 1 and y != n - 1
            j = 1
            while (j <= s - 1 and y != n - 1)
                y = pow_mod(y, 2, n)
                if y == 1
                    return false
                j += 1
            if y != n - 1
                return false
    return true
```

Wile `is_prime` tests if a number is prime, the `make_prime` function uses `is_prime` to generate a prime number.

```
void make_prime(mpz_t p, uint64_t bits, uint64_t iters)
    while (!is_prime(randomly generate mpz_t with bits number of bits, iters))
        ;
```

3.3 Modular Inverses

A function to calculate the greatest common divisor using Euclid's algorithm is needed.

```
void gcd(mpz_t d, mpz_t a, mpz_t b)
    while (b != 0)
        t = b
        b = a mod b
        a = t
    d = a
```

The extended Euclidean algorithm is useful when computing the modular inverse.

```
void mod_inverse(mpz_t i, mpz_t a, mpz_t n)
    r = n
    r_inv = a
    while (r_inv != 0)
        q = floor(r / r_inv)
        (r, r_inv) = (r_inv, r - q * r_inv)
        (t, t_inv) = (t_inv, t - q * t_inv)
    if r > 1
        i = 0
        exit
    if t < 0
        t += n
        i = t
```

4 RSA Library

The function rsa_make_pub generates a new RSA public key which includes: two primes p and q, their product n, and a public exponent e.

```
void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)
    make_prime(p, bits = random number from [nbits/4,(3*nbits)/4], iters)
    make_prime(q, nbits -= bits, iters)
    totient = (p - 1)(q - 1)
    do
        mpz_urandomb(rand, state, nbits)
    while ((temp = gcd(totient, rand)) == 1)
    e = temp
```

The function rsa_write_pub writes a public key to `pbfile`.

```
void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
    // The values n, e and, s must be formatted as hextrings before writing.
    printf(pbfile, n)
    printf(pbfile, e)
    printf(pbfile, s)
    fclose(pbfile, username)
```

The function rsa_read_pub reads a public key from `pbfile`

```
void read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
```

The function rsa_make_priv generates a new RSA private key.

```
void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)
    mod_inverse(d, e, (p-1)(q-1))
```

The function rsa_write_priv writes a private RSA key to `pvfile`.

```
void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)
    // The values n and d must be formatted as hextrings before writing.
    printf(pbfile, n)
    printf(pbfile, d)
```

The function rsa_read_priv reads a private RSA key from `pvfile`.

```
void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)
```

The function rsa_encrypts a message.

```
void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)
    pow_mod(c, m, e, n)
```

The function rsa_encrypt_file encrypts a file and writes the contents to `outfile`.

```
void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)
    block_size = floor((log2(n) - 1) / 8)
    Dynamically allocate array block of block_size
    block[0] = 0xFF
    while(Still bytes to read)
        read k-1 bytes into block
        convert read bytes into mpz_t m
        rsa_encrypt()
```

The function rsa_decrypt decrypts an encrypted message.

```
void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)
    pow_mod(m, c, d, n)
```

The function rsa_decrypt_file decrypts a file and writes the contents to outfile

```
void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)
    block_size = floor((log2(n) - 1) / 8)
    Dynamically allocate array block of block_size
    while (Still bytes to read)
        scan in hexstrings as a mpz_t c
        convert c to bytes
        write j - 1 bytes to outfile
```

The function rsa_sign produces an RSA sign

```
void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)
    pow_mod(s, m, d, n)
```

The function rsa_verify verifies a message as RSA encrypted.

```
bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)
    pow_mod(t, s, e, n)
    return t == m
```

5 Key Generator

The key generator should accept the following command-line options:

- -b: specifies the minimum bits needed for the public key.
- -i: specifies the number of Miller-Rabin iterations for testing primes (default: 50).
- -n pbfile: specifies the public key file (default: rsa.pub).
- -d pvfile: specifies the private key file (default: rsa.priv).
- -s: specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch).
- -v: enables verbose output.
- -h: displays program synopsis and usage.

5.1 Keygen Implementation

```
main()
    while (opt = getopt(OPTIONS) != -1)
        switch(opt)
            Include cases for each option
```

```
if (help or getopt error)
    print(Help manual)
    exit

if (pbfile does not exit)
    print(Error message)
    print(Help Manual)
    exit

if (pvfile does not exit)
    print(Error message)
    print(Help Manual)
    close pbfile
    exit

set file permissions
initialize random seed

rsa_make_pub()
rsa_make_priv()

rsa_sign()
rsa_write_pub()
rsa_write_priv()

if (verbose)
    print(Statistics)

close pbfile and pvfile
clear gmp integers and random seed
```

6 Encryptor

The encryptor should accept the following command-line options:

- -i: specifies the input file to encrypt (default: stdin).
- -o: specifies the output file to encrypt (default: stdout).
- -n: specifies the file containing the public key (default: rsa.pub).
- -v: enables verbose output.

- h: displays program synopsis and usage.

6.1 Encrypt implementation

```
main()
    while (opt = getopt(OPTIONS) != -1)
        switch(opt)
            Include cases for each option

    if (error or help)
        print(Help manual)
        close infile and outfile
        exit

    if (infile fails to open)
        print(Error message)
        close outfile
        exit

    if (public key can not be opened)
        print(Error message)
        close infile and outfile
        exit

    rsa_read_pub()

    if (verbose)
        print(Statistics)

    if (rsa_verify == false)
        print(Signature could not be verified)
        close infile, outfile, and public key
        clear
        exit

    rsa_encrypt_file()

    close infile, outfile, and public key
    clear
```

7 Decryptor

The decryptor should accept the following command-line options:

- -i: specifies the input file to decrypt (default: stdin).
- -o: specifies the output file to decrypt (default: stdout).
- n: specifies the file containing the private key (default: rsa.priv).
- -v: enables verbose output.
- -h: displays program synopsis and usage.

```
main()
    while (opt = getopt(OPTIONS) != -1)
        switch(opt)
            Include cases for every option

    if (error or help)
        print(Help manual)
        close infile and outfile
        exit

    if (infile fails to open)
        print(Error message)
        close outfile
        exit

    if (Private key can not be opened)
        print(Error message)
        close infile and outfile
        exit

    rsa_read_priv()

    if (verbose)
        print(Statistics)

    rsa_decrypt_file()
    close infile, outfilem and private key
    clear
```