**Cryptography Project #3**

소프트웨어학부

2018044720 석예림

1. rsa\_pss.c 소스코드

/\*

\* Copyright 2020. Heekuck Oh, all rights reserved

\* 이 프로그램은 한양대학교 ERICA 소프트웨어학부 재학생을 위한 교육용으로 제작되었습니다.

\*/

#include <stdlib.h>

#include <string.h>

#include <gmp.h>

#include "rsa\_pss.h"

#if defined(SHA224)

void (\*sha)(const unsigned char \*, unsigned int, unsigned char \*) = sha224;

#elif defined(SHA256)

void (\*sha)(const unsigned char \*, unsigned int, unsigned char \*) = sha256;

#elif defined(SHA384)

void (\*sha)(const unsigned char \*, unsigned int, unsigned char \*) = sha384;

#else

void (\*sha)(const unsigned char \*, unsigned int, unsigned char \*) = sha512;

#endif

/\*

\* Copyright 2020. Heekuck Oh, all rights reserved

\* rsa\_generate\_key() - generates RSA keys e, d and n in octet strings.

\* If mode = 0, then e = 65537 is used. Otherwise e will be randomly selected.

\* Carmichael's totient function Lambda(n) is used.

\*/

void rsa\_generate\_key(void \*\_e, void \*\_d, void \*\_n, int mode)

{

mpz\_t p, q, lambda, e, d, n, gcd;

gmp\_randstate\_t state;

/\*

\* Initialize mpz variables

\*/

mpz\_inits(p, q, lambda, e, d, n, gcd, NULL);

gmp\_randinit\_default(state);

gmp\_randseed\_ui(state, arc4random());

/\*

\* Generate prime p and q such that 2^(RSAKEYSIZE-1) <= p\*q < 2^RSAKEYSIZE

\*/

do {

do {

mpz\_urandomb(p, state, RSAKEYSIZE/2);

} while (mpz\_probab\_prime\_p(p, 50) == 0);

do {

mpz\_urandomb(q, state, RSAKEYSIZE/2);

} while (mpz\_probab\_prime\_p(q, 50) == 0);

mpz\_mul(n, p, q);

} while (!mpz\_tstbit(n, RSAKEYSIZE-1));

/\*

\* Generate e and d using Lambda(n)

\*/

mpz\_sub\_ui(p, p, 1);

mpz\_sub\_ui(q, q, 1);

mpz\_lcm(lambda, p, q);

if (mode == 0)

mpz\_set\_ui(e, 65537);

else do {

mpz\_urandomb(e, state, RSAKEYSIZE);

mpz\_gcd(gcd, e, lambda);

} while (mpz\_cmp(e, lambda) >= 0 || mpz\_cmp\_ui(gcd, 1) != 0);

mpz\_invert(d, e, lambda);

/\*

\* Convert mpz\_t values into octet strings

\*/

mpz\_export(\_e, NULL, 1, (RSAKEYSIZE/8), 1, 0, e);

mpz\_export(\_d, NULL, 1, (RSAKEYSIZE/8), 1, 0, d);

mpz\_export(\_n, NULL, 1, (RSAKEYSIZE/8), 1, 0, n);

/\*

\* Free the space occupied by mpz variables

\*/

mpz\_clears(p, q, lambda, e, d, n, gcd, NULL);

}

/\*

\* Copyright 2020. Heekuck Oh, all rights reserved

\* rsa\_cipher() - compute m^k mod n

\* If m >= n then returns EM\_MSG\_OUT\_OF\_RANGE, otherwise returns 0 for success.

\*/

static int rsa\_cipher(void \*\_m, const void \*\_k, const void \*\_n)

{

mpz\_t m, k, n;

/\*

\* Initialize mpz variables

\*/

mpz\_inits(m, k, n, NULL);

/\*

\* Convert big-endian octets into mpz\_t values

\*/

mpz\_import(m, (RSAKEYSIZE/8), 1, 1, 1, 0, \_m);

mpz\_import(k, (RSAKEYSIZE/8), 1, 1, 1, 0, \_k);

mpz\_import(n, (RSAKEYSIZE/8), 1, 1, 1, 0, \_n);

/\*

\* Compute m^k mod n

\*/

if (mpz\_cmp(m, n) >= 0) {

mpz\_clears(m, k, n, NULL);

return EM\_MSG\_OUT\_OF\_RANGE;

}

mpz\_powm(m, m, k, n);

/\*

\* Convert mpz\_t m into the octet string \_m

\*/

mpz\_export(\_m, NULL, 1, (RSAKEYSIZE/8), 1, 0, m);

/\*

\* Free the space occupied by mpz variables

\*/

mpz\_clears(m, k, n, NULL);

return 0;

}

/\*

\* Copyright 2020. Heekuck Oh, all rights reserved

\* A mask generation function based on a hash function

\*/

static unsigned char \*mgf(const unsigned char \*mgfSeed, size\_t seedLen, unsigned char \*mask, size\_t maskLen)

{

uint32\_t i, count;

size\_t hLen;

unsigned char \*mgfIn, \*p, \*m;

/\*

\* Check if maskLen > 2^32\*hLen

\*/

hLen = SHASIZE/8;

if (maskLen > 0x0100000000\*hLen)

return NULL;

/\*

\* Generate octet string mask

\*/

if ((mgfIn = (unsigned char \*)malloc(seedLen+4)) == NULL)

return NULL;;

memcpy(mgfIn, mgfSeed, seedLen);

count = maskLen/hLen + (maskLen%hLen ? 1 : 0);

if ((m = (unsigned char \*)malloc(count\*hLen)) == NULL)

return NULL;

p = (unsigned char \*)&i;

for (i = 0; i < count; i++) {

mgfIn[seedLen] = p[3];

mgfIn[seedLen+1] = p[2];

mgfIn[seedLen+2] = p[1];

mgfIn[seedLen+3] = p[0];

(\*sha)(mgfIn, seedLen+4, m+i\*hLen);

}

/\*

\* Copy the mask and free memory

\*/

memcpy(mask, m, maskLen);

free(mgfIn); free(m);

return mask;

}

/\*

\* rsassa\_pss\_sign - RSA Signature Scheme with Appendix

\*/

int rsassa\_pss\_sign(const void \*m, size\_t mLen, const void \*d, const void \*n, void \*s)

{

unsigned char mHash[SHASIZE/8];

unsigned char M[8+2\*(SHASIZE/8)];

unsigned char salt[SHASIZE/8];

unsigned char H[SHASIZE/8];

unsigned char DB[(RSAKEYSIZE/8)-(SHASIZE/8)-1];

unsigned char maskedDB[(RSAKEYSIZE/8)-(SHASIZE/8)-1];

unsigned char mMgf[(RSAKEYSIZE/8)-(SHASIZE/8)-1];

unsigned char EM[(RSAKEYSIZE/8)];

uint8\_t msb = 0x01;

uint8\_t bc = 0xbc;

if(((SHASIZE == 224) || (SHASIZE == 256)) && (mLen > 0x1fffffffffffffff)){

return EM\_MSG\_TOO\_LONG;

}

if(2\*(SHASIZE/8)+2 > (RSAKEYSIZE/8)){

return EM\_HASH\_TOO\_LONG;

}

// M'

for(int i = 0; i < 8; i++){

M[i] = 0x00;

}

// mHash

sha(m, mLen, mHash);

memcpy(M+8, mHash, SHASIZE/8);

// salt

arc4random\_buf(salt, SHASIZE/8);

memcpy(M+8+SHASIZE/8, salt, SHASIZE/8);

// H

sha(M, 8+2\*(SHASIZE/8), H);

// DB

for(int i = 0; i < (RSAKEYSIZE/8)-2\*(SHASIZE/8)-1; i++){

DB[i] = 0x00;

}

memcpy(DB+(RSAKEYSIZE/8)-2\*(SHASIZE/8)-2, &msb, 1);

memcpy(DB+(RSAKEYSIZE/8)-2\*(SHASIZE/8)-1, salt, SHASIZE/8);

// mgf(H)

mgf(H, SHASIZE/8, mMgf, (RSAKEYSIZE/8)-SHASIZE/8-1);

// maskedDB

for(int i = 0; i < (RSAKEYSIZE/8)-(SHASIZE/8)-1; i++){

maskedDB[i] = DB[i] ^ mMgf[i];

EM[i] = maskedDB[i];

}

// EM

if((EM[0]>>7) & 1){

EM[0] = 0x00;

}

memcpy(EM+(RSAKEYSIZE/8)-SHASIZE/8-1, H, SHASIZE/8);

memcpy(EM+(RSAKEYSIZE/8)-1, &bc, 1);

// s

if(rsa\_cipher(EM, d, n)){

return EM\_MSG\_OUT\_OF\_RANGE;

}

memcpy(s, EM, (RSAKEYSIZE/8));

return 0;

}

/\*

\* rsassa\_pss\_verify - RSA Signature Scheme with Appendix

\*/

int rsassa\_pss\_verify(const void \*m, size\_t mLen, const void \*e, const void \*n, const void \*s)

{

unsigned char M[8+(SHASIZE/8)\*2];

unsigned char salt[SHASIZE/8];

unsigned char mHash[SHASIZE/8];

unsigned char MHash[SHASIZE/8];

unsigned char H[SHASIZE/8];

unsigned char DB[(RSAKEYSIZE/8)-(SHASIZE/8)-1];

unsigned char maskedDB[(RSAKEYSIZE/8)-(SHASIZE/8)-1];

unsigned char EM[(RSAKEYSIZE/8)];

unsigned char hMgf[SHASIZE/8];

uint8\_t bc = 0xbc;

// EM

memcpy(EM, s, (RSAKEYSIZE/8));

if(rsa\_cipher(EM, e, n)){

return EM\_MSG\_OUT\_OF\_RANGE;

}

if((EM[0]>>7) & 1){

return EM\_INVALID\_INIT;

}

if((EM[(RSAKEYSIZE/8)-1]) ^ bc){

return EM\_INVALID\_LAST;

}

// maskedDB

memcpy(maskedDB, EM, (RSAKEYSIZE/8)-(SHASIZE/8)-1);

// H

memcpy(H, EM+(RSAKEYSIZE/8)-(SHASIZE/8)-1, SHASIZE/8);

// mgf(H)

mgf(H, SHASIZE/8, hMgf, (RSAKEYSIZE/8)-(SHASIZE/8)-1);

// DB

DB[0] = 0x00;

for(int i = 1; i < (RSAKEYSIZE/8)-(SHASIZE/8)-1; i++){

DB[i] = maskedDB[i] ^ hMgf[i];

}

for(int i = 0; i < (RSAKEYSIZE/8)-2\*(SHASIZE/8)-2; i++){

if(DB[i] ^ 0x00){

return EM\_INVALID\_PD2;

}

}

if(DB[(RSAKEYSIZE/8)-2\*(SHASIZE/8)-2] ^ 0x01){

return EM\_INVALID\_PD2;

}

// salt

memcpy(salt, DB+(RSAKEYSIZE/8)-2\*(SHASIZE/8)-1, SHASIZE/8);

// mHash

sha(m, mLen, mHash);

// M'

for(int i = 0; i < 8; i++){

M[i] = 0x00;

}

memcpy(M+8, mHash, SHASIZE/8);

memcpy(M+8+SHASIZE/8, salt, SHASIZE/8);

// Hash(M')

sha(M, 8+2\*(SHASIZE/8), MHash);

for(int i = 0; i < SHASIZE/8 ; i++){

if(H[i] ^ MHash[i]){

return EM\_HASH\_MISMATCH;

}

}

return 0;

}

2. 코드 내 함수 설명

* Diagram

  Description automatically generatedrsassa\_pss\_sign 서명 과정

unsigned char로 선언된 변수들은 바이트 단위이고, RSAKEYSIZE와 SHASIZE는 bit단위이기 때문에 8로 나눠 바이트 단위로 맞춰 변수 길이를 설정해주었다.

먼저, SHASIZE가 224 또는 256일 때에는 메시지 길이(mLen)가 64비트를 넘어가지 않아야 하기 때문에 초과한다면 에러 처리(EM\_MSG\_TOO\_LONG)를 해주었다.

EM에 수용할 수 있는 해시의 길이인지 확인해 주기 위해서 maskedDB의 길이와 H의 길이와 TF의 길이의 최소값이 EM의 크기보다 작으면 된다. maskedDB의 길이는 DB의 길이와 같기 때문에 DB를 사용하여

최소 길이를 계산해 보면 PS(패딩)값은 길이에 맞춰 채우기 때문에 최소 값은 0이고, 0x01 : 1바이트, salt의 길이 : SAHSIZE/8으로 최소길이는 SHASIZE/8+1이다. H의 길이는 SHASIZE/8로 EM의 최소 길이를 구해보면 2\*(SHASIZE/8) + 2이 된다. 이 값이 EM의 길이인 RSAKEYSIZE/8을 넘게 되면 수용할 수 없기 때문에 에러 처리 (EM\_HASH\_TOO

\_LONG) 를 해준다.

M’을 만들어 주기 위해 처음 8 바이트를 0x00으로 채워주고 sha-2계열의 함수를 사용하여 m을 Hash한 mHash를 채워주고 arc4random\_buf를 사용하여 난수salt를 생성해 M’에 채워주었다.

DB는 salt와 그 앞에 0x01을 채워주고 나머지를 모두 0x00으로 채워준다.

만들어진 M’을 sha함수를 사용해 Hash해준 H를 만들어 주었다. 만든 H를 가지고 MGF함수에 넣어 mMgf를 생성해내고 이것을 DB와 XOR하여 EM에 넣어준다.

EM의 처음 비트가 1이면 강제로 0으로 바꾸어 줘야 하고, H를 채우고, 마지막에 TF 1바이트를 0xBC로 채운다. 이렇게 만든 EM을 가지고 EM^d mod n을 계산하여 s에 저장한다.

* Diagram

  Description automatically generatedrsassa\_pss\_verify 검증 과정

s^e mod n = EM이므로, 계산하여 EM에 저장해 주고, EM의 첫번째 비트가 1이면 EM\_INVALID\_INIT에러 처리를 해 주고, EM의 마지막이 0xbc가 아니면 EM\_INVALID\_LAST에러 처리를 해주고 종료시킨다. EM에서 maskedDB와 H를 구해낸다. H를 MGF함수에 넣고 나온 값과 maskedDB를 XOR하여 DB를 구해내고, 구한 DB에서 salt를 구한다. 서명과정에서 DB와 H를 XOR했을 때 맨 처음 비트를 0으로 강제로 바꾸어 주었기 때문에 검증과정에서 maskedDB와 H를 XOR해 주면 첫 번째 비트가 1이 나올 수 있다. 그래서 DB의 첫 번째 비트를 0으로 강제로 바꾸어 주고, DB의 앞부분이 0x00으로 채워져 있고, salt앞 부분에는 0x01로 채워져 있는지 확인하였다(EM\_INVALID\_PD2). DB에서 구한 salt와 m을 Hash한 후 구한 mHash와 앞 8바이트(0x00)을 가지고 M’을 구한다. 구한 M’을 Hash하여 H와 비교하여 검증한다(EM\_HASH\_MISMATCH).

3. 실행 결과

Text

Description automatically generated