암호학

Project #5

소프트웨어학부
2017012251
윤영훈

1. source code

```
#
CC=gcc
CFLAGS=-Wall
GMP=-lgmp

all: test.o rsa_pss.o sha2.o
	$(CC) $(CFLAGS) -o test test.o rsa_pss.o sha2.o $(GMP) -lbsd

test.o: test.c rsa_pss.h
	$(CC) $(CFLAGS) -c test.c

rsa_pss.o: rsa_pss.c rsa_pss.h
	$(CC) $(CFLAGS) -c rsa_pss.c

sha2.o: sha2.c sha2.h
	$(CC) $(CFLAGS) -c sha2.c

clean:
	rm -rf *.o
	rm -rf test
```

makefile -lbsd 추가

```
#include <stdlib.h>
#include <string.h>
#include <gmp.h>
#include "rsa_pss.h"
#include <stdint.h>

#include <bsd/stdlib.h>
```

rsa_pss.c bsd 추가

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "rsa_pss.h"

#include <bsd/stdlib.h>
```

test.c bsd 추가

source code를 설명하기에 앞서 주어진 makefile과 test.c, rsa_pss.c로는 linux 환경에서 arc4random 함수를 정상적으로 사용할 수 없으므로 다음과 같이 변경하였다.

```
#define DB_LEN RSAKEYSIZE/8 - SHASIZE/8 - 1
#define PS_LEN DB_LEN - SHASIZE/8
```

rsa_pss.h DB_LEN, PS_LEN 추가

또한 계산의 편의성을 위해 DB_LEN, PS_LEN을 rsa_pss.h에 미리 define했다.

① RSASSA_PSS_SIGN

```
168  int rsassa_pss_sign(const void *m, size_t mLen, const void *d, const void *n, void *s)
169  {
170      // SHA224, SHA256에서 hash function의 input이 너무 길면 EM_MSG_TOO_LONG return
171      if (((SHASIZE || 224) || (SHASIZE || 256)) && mLen > 0x1fffffffffffffff)
172          return EM_MSG_TOO_LONG;
173
174      unsigned char mHash[SHASIZE/8];
175      unsigned char mgf_Hash[DB_LEN];
176      unsigned char MPrime[2*(SHASIZE/8)+8];
177      unsigned char salt[SHASIZE/8];
178      unsigned char H[SHASIZE/8];
179      unsigned char DB[DB_LEN];
180      unsigned char EM[RSAKEYSIZE/8];
181
182      // m을 hash하여 mHash 획득
183      sha(m, mLen, mHash);
184
185      // salt를 random number로 채움
186      for (int i=0; i<SHASIZE/8; i++){
187          salt[i] = arc4random() & 255;
188      }
189
190      // MPrime의 처음 8 bytes는 0x00로 채우고 이후 mHash, salt를 이어붙임
191      memset(MPrime, 0x00, 8);
192      memcpy(MPrime + 8, mHash, SHASIZE/8);
193      memcpy(MPrime + 8 + SHASIZE/8, salt, SHASIZE/8);
194
195      // MPrime을 hash하여 H 획득
196      sha(MPrime, 2*(SHASIZE/8)+8, H);
197
198      // hash H의 길이가 너무 커서 EM에 넣을 수 없는 경우 EM_HASH_TOO_LONG return
199      if ((sizeof(H) / sizeof(H[0]) > RSAKEYSIZE/2))
200          return EM_HASH_TOO_LONG;
```

RSASSA_PSS_SIGN 1

SHA224, SHA256에서는 hash function의 input이 너무 길면 오류가 발생한다. 따라서 해당 케이스 발생 시 오류코드 EM_MSG_TOO_LONG을 return한다.

이후 m을 hash하여 mHash를 획득한 뒤 arc4random 함수를 사용하여 salt를 0x00 형태의 random한 16진수로 채운다.

MPrime은 8 bytes의 zero-padding, mHash, salt로 구성되어 있으므로 memset, memcpy를 이용하여 MPrime을 채운다.

이후 MPrime을 hash하여 H를 획득한다. 이때 H의 길이가 너무 커서 EM에 넣을 수 없는 경우 오류코드 EM_HASH_TOO_LONG을 return한다.

```
202     // DB를 제외한 나머지 EM의 요소를 채움
203     memcpy(EM + DB_LEN, H, SHASIZE/8);
204     memset(EM + DB_LEN + SHASIZE/8, 0xbc, 1);
205
206     // DB 구성, 0으로 계속 padding하다가 salt 바로 앞 bit를 1로 설정하여 salt 구분
207     memset(DB, 0x00, PS_LEN-1);
208     memset(DB+PS_LEN-1, 0x01, 1);
209     memcpy(DB+PS_LEN, salt, SHASIZE/8);
210
211     // mgf로 mgf_Hash 생성
212     mgf(H, SHASIZE/8, mgf_Hash, DB_LEN);
213
214     // DB와 mgf_Hash를 XOR하여 EM 구성
215     for (int i=0; i<DB_LEN; i++){
216         EM[i] = DB[i] ^ mgf_Hash[i];
217     }
218
219     // EM의 맨 처음 bit가 1일 시 0으로 변경
220     if ((EM[0] >> 7) == 1)
221         EM[0] &= 0x7f;
222
223     // EM을 (d, n)으로 서명
224     // 이때 RSA 데이터 값이 modulus n보다 크거나 같다면 EM_MSG_OUT_OF_RANGE return
225     if (rsa_cipher(EM, d, n))
226         return EM_MSG_OUT_OF_RANGE;
227
228     // EM을 s에 복사
229     memcpy(s, EM, RSAKEYSIZE/8);
230
231     return 0;
232 }
```

RSASSA_PSS_SIGN 2

EM은 DB, H, 1 byte의 0xbc로 이루어져 있으므로 우선 DB를 제외한 H, 0xbc를 채운다.

이후 앞서 구한 salt를 그대로 사용하여 DB를 구성한다. 이때 salt의 바로 앞 bit는 1로, 나머지 bit는 0으로 padding하여 salt를 구분한 채로 DB를 구성한다.

mgf를 사용하여 길이 DB_LEN의 mgf_Hash를 생성한다. 이후 DB와 mgf_Hash를 XOR 연산하여 EM을 구성한다. 이때 EM의 맨 처음 bit (MSB)가 1일 시 0으로 변경한다.

EM을 개인키 (d, n)으로 서명한다. 이때 RSA 데이터 값이 modulus n보다 크거나 같다면 오류코드 EM_MSG_OUT_OF_RANGE를 return한다.

최종적으로 EM을 s에 복사한다.

② RSASSA_PSS_VERIFY

```
237  int rsassa_pss_verify(const void *m, size_t mLen, const void *e, const void *n, const void *s)
238  {
239      unsigned char EM[RSAKEYSIZE/8];
240      unsigned char maskedDB[DB_LEN];
241      unsigned char DB[DB_LEN];
242      unsigned char mgf_Hash[DB_LEN];
243      unsigned char H[SHASIZE/8];
244      unsigned char salt[SHASIZE/8];
245      unsigned char mHash[SHASIZE/8];
246      unsigned char HPrime[SHASIZE/8];
247      unsigned char MPrime[2*(SHASIZE/8)+8];
248
249      // s를 EM에 복사
250      memcpy(EM, s, RSAKEYSIZE/8);
251
252      // EM을 (e, n)으로 검증
253      // 이때 RSA 데이터 값이 modulus n보다 크거나 같다면 EM_MSG_OUT_OF_RANGE return
254      if (rsa_cipher(EM, e, n))
255          return EM_MSG_OUT_OF_RANGE;
256
257      // EM의 마지막 byte가 0xbc가 아니면 EM_INVALID_LAST return
258      if (EM[RSAKEYSIZE/8 - 1] != 0xbc)
259          return EM_INVALID_LAST;
260
261      // EM의 첫 bit가 0이 아니면 EM_INVALID_INIT return
262      if ((EM[0] >> 7) != 0)
263          return EM_INVALID_INIT;
264
265      // EM에서 maskedDB, H를 추출
266      memcpy(maskedDB, EM, DB_LEN);
267      memcpy(H, EM + DB_LEN, SHASIZE/8);
268
269      // mgf로 mgf_Hash 복구
270      mgf(H, SHASIZE/8, mgf_Hash, DB_LEN);
```

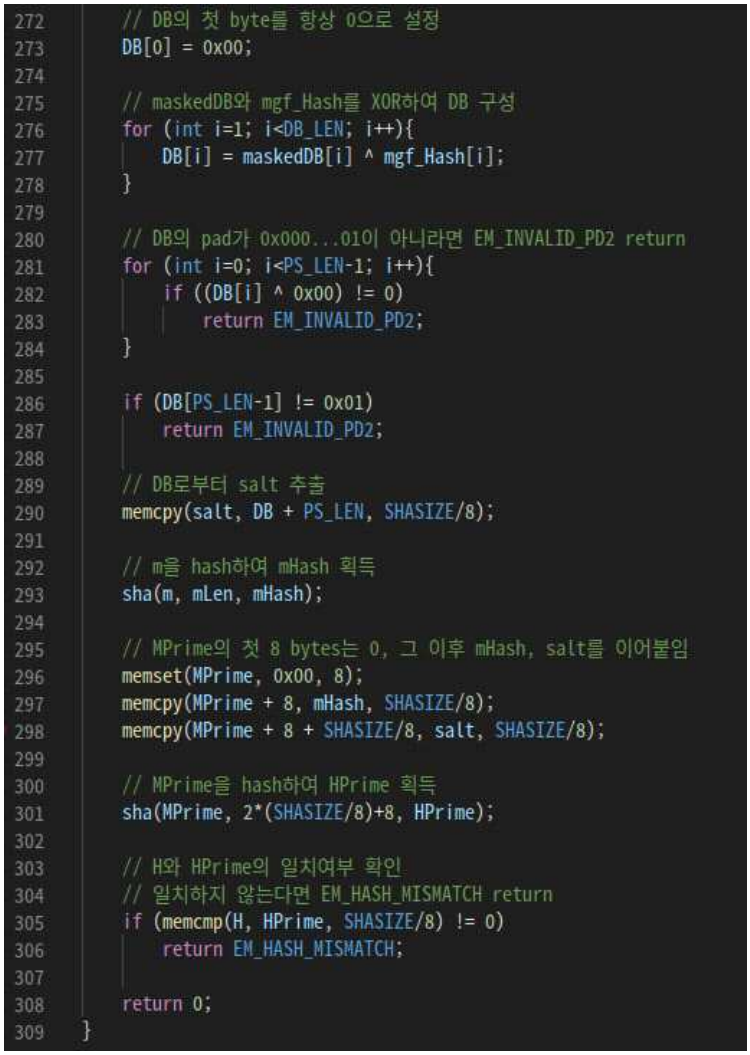
RSASSA_PSS_VERIFY 1

s를 EM에 복사한 뒤 공개키 (e, n)으로 검증한다. 이때 RSA 데이터 값이 modulus n보다 크거나 같다면 오류코드 EM_MSG_OUT_OF_RANGE를 return한다.

EM의 마지막 byte가 0xbc가 아니면 오류코드 EM_INVALID_LAST를 return하고, EM의 첫 bit (MSB)가 0이 아니면 EM_INVALID_INIT를 return한다.

이후 EM에서 maskedDB와 H를 추출한다.

mgf를 사용하여 mgf_Hash를 복구한다.



```
272     // DB의 첫 byte를 항상 0으로 설정
273     DB[0] = 0x00;
274
275     // maskedDB와 mgf_Hash를 XOR하여 DB 구성
276     for (int i=1; i<DB_LEN; i++){
277         DB[i] = maskedDB[i] ^ mgf_Hash[i];
278     }
279
280     // DB의 pad가 0x000...01이 아니라면 EM_INVALID_PD2 return
281     for (int i=0; i<PS_LEN-1; i++){
282         if ((DB[i] ^ 0x00) != 0)
283             return EM_INVALID_PD2;
284     }
285
286     if (DB[PS_LEN-1] != 0x01)
287         return EM_INVALID_PD2;
288
289     // DB로부터 salt 추출
290     memcpy(salt, DB + PS_LEN, SHASIZE/8);
291
292     // m을 hash하여 mHash 획득
293     sha(m, mLen, mHash);
294
295     // MPrime의 첫 8 bytes는 0, 그 이후 mHash, salt를 이어붙임
296     memset(MPrime, 0x00, 8);
297     memcpy(MPrime + 8, mHash, SHASIZE/8);
298     memcpy(MPrime + 8 + SHASIZE/8, salt, SHASIZE/8);
299
300     // MPrime을 hash하여 HPrime 획득
301     sha(MPrime, 2*(SHASIZE/8)+8, HPrime);
302
303     // H와 HPrime의 일치여부 확인
304     // 일치하지 않는다면 EM_HASH_MISMATCH return
305     if (memcmp(H, HPrime, SHASIZE/8) != 0)
306         return EM_HASH_MISMATCH;
307
308     return 0;
309 }
```

RSASSA_PSS_VERIFY 2

DB[0]의 값이 종종 변경되는 경우를 발견하였다. 따라서 DB[0]을 0x00으로 고정시킨 뒤 maskedDB와 mgf_Hash를 XOR 연산하여 DB를 구성한다. 이때 DB의 pad가 0x000...01이 아니라면 오류코드 EM_INVALID_PD2를 return한다.

이후 DB로부터 salt를 추출하고, m을 hash하여 mHash를 획득한다.

MPrime은 8 bytes의 0, mHash, salt로 이루어져 있으므로 memset, memcpy를 사용하여 채운다.

완성된 MPrime을 hash하여 HPrime을 획득한다.

EM에서 추출한 H와 HPrime이 일치하는지 확인하고 일치하지 않는다면 오류코드 EM_HASH_MISMATC를 return, 일치한다면 최종적으로 0을 return한다.

2. compile & result

정상적으로 compile 되었으며 모든 case 및 random testing에서 PASSED가 출력되었다. '>' 터미널 명령어를 통해 결과를 rsa_pss.txt에 저장하여 별도로 제출한다.

3. feedback

bit 단위가 아닌 byte 단위로 배열을 다루어야 한다는 것을 깨닫기까지 시간을 많이 소모했다. 또한 for문 위주로 코드를 구현하다가 더 효율적인 memcpy, memset으로 변환하는데에도 시간을 소모했다. 또한 오류코드를 처리하는 과정에서 대부분의 오류를 직접 발생시켜보고 해당 오류가 발생하는 위치와 원인을 파악하여 오류코드를 배정했는데, 일부 오류의 경우 직접 자연적으로 발생시키기 어려운 오류가 존재했다. 해당 오류들의 경우 직접 발생시킬 수 없어 오로지 생각에 의존하여 처리했기에 이론과 그에 대한 이해가 더욱 필요했다.