

 18f4b592a8 [▼](#)

[...](#)

mTower / tools / ecdsa_keygen.c



tdrozovsky Added more info about mTower binaries and fix some warnings

[History](#)

 1 contributor

385 lines (327 sloc) | 9.91 KB

[...](#)

```

1  /**
2   * @file      arch/arm/m2351/src/numaker_pfm_m2351/secure/main.c
3   * @brief     Provides functionality to start secure world, initialize secure
4   *           and normal worlds, pass to execution to normal world.
5   *
6   * @copyright  Copyright (c) 2019 Samsung Electronics Co., Ltd. All Rights Reserved.
7   * @author    Taras Drozdovskyi t.drozovsky@samsung.com
8   *
9   * Licensed under the Apache License, Version 2.0 (the "License");
10  * you may not use this file except in compliance with the License.
11  * You may obtain a copy of the License at
12  *
13  * http://www.apache.org/licenses/LICENSE-2.0
14  *
15  * Unless required by applicable law or agreed to in writing, software
16  * distributed under the License is distributed on an "AS IS" BASIS,
17  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
18  * See the License for the specific language governing permissions and
19  * limitations under the License.
20  */
21
22  /* Included Files. */
23  #include <stdio.h>
24  #include <stdint.h>
25  #include <stdlib.h>
26  #include <string.h>
27
28  #include <openssl/sha.h>
29  //#include "config.h"

```

```

30  // #include "version.h"
31
32  #include <openssl/ec.h>      // for EC_GROUP_new_by_curve_name, EC_GROUP_free, EC_KEY_new, EC_KEY_
33  #include <openssl/ecdsa.h>  // for ECDSA_do_sign, ECDSA_do_verify
34  #include <openssl/obj_mac.h> // for NID_secp192k1
35
36  #include <stdlib.h>
37  #include <string.h>
38  #include <openssl/conf.h>
39  #include <openssl/evp.h>
40  #include <openssl/rand.h>
41  #include <openssl/err.h>
42
43  /* Pre-processor Definitions. */
44  #define AES_BLOCK_SIZE 16
45  #define AES_KEY_SIZE 32
46
47  /** Start address for non-secure boot image */
48
49  /* Private Types. */
50  /* Any types, enums, structures or unions used by the file are defined here. */
51  /* typedef for NonSecure callback functions */
52
53  /**
54   * @details    ECC ECDSA key structure
55   */
56  typedef struct {
57      uint8_t Qx[32]; /* 256-bits */
58      uint8_t Qy[32]; /* 256-bits */
59      uint8_t d[32]; /* 256-bits */
60  }__attribute__((packed)) ECC_KEY_T;
61
62  typedef struct _AES_DATA {
63      unsigned char key[AES_KEY_SIZE];
64      unsigned char iv[AES_BLOCK_SIZE];
65  } AES_DATA;
66
67  typedef struct Message_Struct {
68      unsigned char *body;
69      int *length;
70      AES_DATA *aes_settings;
71  } Message;
72
73  /* Private Function Prototypes. */
74  /* Prototypes of all static functions in the file are provided here. */
75
76  /* Private Data. */
77  /* All static data definitions appear here. */
78  uint32_t key[] =

```

```

79     { 0x78563412, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
80         0x00000000, 0xefcdac00 };
81
82     uint32_t iv[] =
83     { 0x78563412, 0x00000000, 0x00000000, 0xefcdac00 };
84
85     /* Public Data. */
86     /* All data definitions with global scope appear here. */
87
88     /* Public Function Prototypes */
89     Message *message_init(int);
90     int aes256_init(Message *);
91     Message *aes256_encrypt(Message *);
92     void aes_cleanup(AES_DATA *);
93     void message_cleanup(Message *);
94
95     void sha256(unsigned char *data, unsigned int data_len, unsigned char *hash)
96     {
97         SHA256_CTX sha256;
98         SHA256_Init(&sha256);
99         SHA256_Update(&sha256, data, data_len);
100        SHA256_Final(hash, &sha256);
101    }
102
103    Message *message_init(int length)
104    {
105        Message *ret = malloc(sizeof(Message));
106        ret->body = malloc(length);
107        ret->length = malloc(sizeof(int));
108        *ret->length = length;
109        //initialize aes_data
110        aes256_init(ret);
111        return ret;
112    }
113
114    int aes256_init(Message * input)
115    {
116        AES_DATA *aes_info = malloc(sizeof(AES_DATA));
117        //point to new data
118        input->aes_settings = aes_info;
119        //get rand bytes
120        memcpy(input->aes_settings->key, key, AES_KEY_SIZE);
121        memcpy(input->aes_settings->iv, iv, AES_KEY_SIZE / 2);
122
123        return 0;
124    }
125
126    Message *aes256_encrypt(Message * plaintext)
127    {

```

```

128     EVP_CIPHER_CTX *enc_ctx;
129     Message * encrypted_message;
130     int enc_length = *(plaintext->length)
131         + (AES_BLOCK_SIZE - *(plaintext->length) % AES_BLOCK_SIZE);
132
133     encrypted_message = message_init(enc_length);
134     //set up encryption context
135     enc_ctx = EVP_CIPHER_CTX_new();
136     EVP_EncryptInit(enc_ctx, EVP_aes_256_cfb(), plaintext->aes_settings->key,
137         plaintext->aes_settings->iv);
138     //encrypt all the bytes up to but not including the last block
139     if (!EVP_EncryptUpdate(enc_ctx, encrypted_message->body, &enc_length,
140         plaintext->body, *plaintext->length))
141     {
142         EVP_CIPHER_CTX_cleanup(enc_ctx);
143         printf("EVP Error: couldn't update encryption with plain text!\n");
144         return NULL;
145     }
146     //update length with the amount of bytes written
147     *(encrypted_message->length) = enc_length;
148     //EncryptFinal will cipher the last block + Padding
149     if (!EVP_EncryptFinal_ex(enc_ctx, enc_length + encrypted_message->body,
150         &enc_length))
151     {
152         EVP_CIPHER_CTX_cleanup(enc_ctx);
153         printf("EVP Error: couldn't finalize encryption!\n");
154         return NULL;
155     }
156     //add padding to length
157     *(encrypted_message->length) += enc_length;
158     //no errors, copy over key & iv rather than pointing to the plaintext msg
159     memcpy(encrypted_message->aes_settings->key, plaintext->aes_settings->key,
160         AES_KEY_SIZE);
161     memcpy(encrypted_message->aes_settings->iv, plaintext->aes_settings->iv,
162         AES_KEY_SIZE / 2);
163     //Free context and return encrypted message
164     EVP_CIPHER_CTX_cleanup(enc_ctx);
165     return encrypted_message;
166 }
167
168 void aes_cleanup(AES_DATA *aes_data)
169 {
170     // free(aes_data -> iv);
171     // free(aes_data -> key);
172     // free(aes_data);
173 }
174
175 void message_cleanup(Message *message)
176 {

```

```

177     //free message struct
178     aes_cleanup(message->aes_settings);
179     free(message->length);
180     free(message->body);
181     free(message);
182 }
183
184 /**
185  * @brief      main - entry point of mTower: secure world.
186  *
187  * @param      None
188  *
189  * @returns     None (function is not supposed to return)
190  */
191 int main(int argc, char * argv[])
192 {
193     // Initialize openssl
194     // ERR_load_crypto_strings();
195     // OpenSSL_add_all_algorithms();
196     // OPENSSL_config(NULL);
197
198     EC_KEY *eckey = EC_KEY_new();
199     if (NULL == eckey) {
200         printf("Failed to create new EC Key\n");
201         return -1;
202     }
203
204     EC_GROUP *ecgroup = EC_GROUP_new_by_curve_name(NID_X9_62_prime256v1);
205     if (NULL == ecgroup) {
206         printf("Failed to create new EC Group\n");
207         return -1;
208     }
209
210     if (EC_KEY_set_group(eckey, ecgroup) != 1 ) {
211         printf("Failed to set group for EC Key\n");
212         return -1;
213     }
214
215     if (EC_KEY_generate_key(eckey) != 1) {
216         printf("Failed to generate EC Key\n");
217         return -1;
218     }
219
220     const BIGNUM* d = EC_KEY_get0_private_key(eckey);
221     const EC_POINT* Q = EC_KEY_get0_public_key(eckey);
222     const EC_GROUP* group = EC_KEY_get0_group(eckey);
223
224     BIGNUM* x = BN_new();
225     BIGNUM* y = BN_new();

```

```

226
227     if (!EC_POINT_get_affine_coordinates_GFp(group, Q, x, y, NULL)) {
228         return -1;
229     }
230
231     // printf("d: %s\n", BN_bn2hex(d));
232     // printf("X: %s\n", BN_bn2hex(x));
233     // printf("Y: %s\n", BN_bn2hex(y));
234
235     ECC_KEY_T ecc_ecdsa_key;
236
237     BN_bn2bin(d, &ecc_ecdsa_key.d[0]);
238     BN_bn2bin(x, ecc_ecdsa_key.Qx);
239     BN_bn2bin(y, ecc_ecdsa_key.Qy);
240
241     // for (int i = 0; i != 32; i++)
242     //     printf(" %02X", ecc_ecdsa_key.d[i]);
243     // printf("\n");
244
245     char* buff;
246     buff = malloc(strlen(argv[1]) + strlen("ecdsa_keys.bin") + 1);
247     buff[0] = 0;
248     strcat(buff, argv[1]);
249     strcat(buff, "ecdsa_keys.bin");
250
251     FILE* fd = fopen(buff, "wb");
252     if (!fd) {
253         printf("Failed to open file\n");
254         free(buff);
255         return -1;
256     }
257     free(buff);
258
259     if (fwrite((void *) &ecc_ecdsa_key, sizeof(char),
260         (size_t) (sizeof(ECC_KEY_T)), fd) != (size_t) (sizeof(ECC_KEY_T)))
261     {
262         fclose(fd);
263         return -1;
264     }
265     fclose(fd);
266
267     Message *message, *enc_msg;
268
269     message = message_init(64);
270     memcpy((char *) message->body, (unsigned char *) ecc_ecdsa_key.Qx, 64);
271
272     enc_msg = aes256_encrypt(message);
273     //clean up ssl;
274     // EVP_cleanup();

```

```

275 // CRYPTO_cleanup_all_ex_data(); //Stop data leaks
276 // ERR_free_strings();
277
278 buff = malloc(strlen(argv[1]) + strlen("NuBL32PubKeyEncrypted.bin") + 1);
279 buff[0] = 0;
280 strcat(buff, argv[1]);
281 strcat(buff, "NuBL32PubKeyEncrypted.bin");
282
283 fd = fopen(buff, "wb");
284 if (!fd) {
285     printf("Failed to open file\n");
286     free(buff);
287     return -1;
288 }
289 free(buff);
290
291 if (fwrite((void *) enc_msg->body, sizeof(char), 64, fd) != 64) {
292     fclose(fd);
293     return -1;
294 }
295 fclose(fd);
296 // free(buff);
297 // message_cleanup(message);
298 // message_cleanup(enc_msg);
299
300 buff = malloc(strlen(argv[1]) + strlen("NuBL32PubKeyEncryptedHash.bin") + 1);
301 buff[0] = 0;
302 strcat(buff, argv[1]);
303 strcat(buff, "NuBL32PubKeyEncryptedHash.bin");
304
305 fd = fopen(buff, "wb");
306 if (!fd) {
307     printf("Failed to open file\n");
308     free(buff);
309     return -1;
310 }
311 free(buff);
312 unsigned char PubKeyEncryptedHash[64];
313 sha256((void *) enc_msg->body, 64, (unsigned char *) PubKeyEncryptedHash);
314
315 if (fwrite((void *) PubKeyEncryptedHash, sizeof(char), 32, fd) != 32) {
316     fclose(fd);
317     return -1;
318 }
319 fclose(fd);
320
321 message = message_init(64);
322 memcpy((char *) message->body, (unsigned char *) ecc_ecdsa_key.Qx, 64);
323

```

```
324     enc_msg = aes256_encrypt(message);
325     //clean up ssl;
326     //  EVP_cleanup();
327     //  CRYPTO_cleanup_all_ex_data(); //Stop data leaks
328     //  ERR_free_strings();
329
330     buff = malloc(strlen(argv[1]) + strlen("NuBL33PubKeyEncrypted.bin") + 1);
331     buff[0] = 0;
332     strcat(buff, argv[1]);
333     strcat(buff, "NuBL33PubKeyEncrypted.bin");
334
335     fd = fopen(buff, "wb");
336     if (!fd) {
337         printf("Failed to open file\n");
338         free(buff);
339         return -1;
340     }
341     free(buff);
342
343     if (fwrite((void *) enc_msg->body, sizeof(char), 64, fd) != 64) {
344         fclose(fd);
345         return -1;
346     }
347     fclose(fd);
348     // free(buff);
349     // message_cleanup(message);
350     // message_cleanup(enc_msg);
351
352     buff = malloc(strlen(argv[1]) + strlen("NuBL33PubKeyEncryptedHash.bin") + 1);
353     buff[0] = 0;
354     strcat(buff, argv[1]);
355     strcat(buff, "NuBL33PubKeyEncryptedHash.bin");
356
357     fd = fopen(buff, "wb");
358     if (!fd) {
359         printf("Failed to open file\n");
360         free(buff);
361         return -1;
362     }
363     free(buff);
364     // unsigned char PubKeyEncryptedHash[64];
365     sha256((void *) enc_msg->body, 64, (unsigned char *) PubKeyEncryptedHash);
366
367     if (fwrite((void *) PubKeyEncryptedHash, sizeof(char), 32, fd) != 32) {
368         fclose(fd);
369         return -1;
370     }
371     fclose(fd);
372
```



```
373  exit:
374      BN_free(x);
375      BN_free(y);
376      EC_GROUP_free(ecgroup);
377      EC_KEY_free(eckey);
378
379      // clean up ssl;
380      //  EVP_cleanup();
381      //  CRYPTO_cleanup_all_ex_data(); //Stop data leaks
382      //  ERR_free_strings();
383
384      return 0;
385  }
```