Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №2 по теме: «Симметричное и асимметричное шифрование данных средствами криптографического пакета OpenSSL»

**Выполнил:**  
студент группы ПВ-41  
Адаменко И. И.

**Проверил:**  
Смышляев А. Г.

Белгород  
2015

**Цель работы:** ознакомиться со средствами симметричного и асимметричного шифрования, предоставляемыми пакетом OpenSSL и способами доступа к ним из приложений на языке C/C++.

# Задание

## Часть А

Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольное или оконное приложение, выполняющее следующие функции:

1. Зашифровывание/расшифровывание указанного файла блочным шифром AES со 128-разрадным ключом в режиме CBC или CFB-128 на выбор пользователя. Сеансовый ключ и вектор инициализации генерируются псевдослучайно (ключ может импортироваться). Вектор должен сохраняться с файлом шифртекста (формат придумать самостоятельно).
2. Генерация ключевой пары RSA с длиной ключа не менее 2048 бит.
3. Сохранение открытого и закрытого ключей в файлах PEM-формата.
4. Зашифровывание сеансового ключа шифра AES с помощью ранее сгенерированного открытого ключа RSA и сохранение его в файле, извлечение из файла и расшифровывание с помощью закрытого ключа для восстановления ранее зашифрованного им файла.

## Часть Б

Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольное или оконное приложение, выполняющее следующие функции:

1. Зашифровывание/расшифровывание указанного файла блочным шифром ГОСТ 28147-89 в режиме CFB (устанавливается по умолчанию). Сеансовый ключ и вектор инициализации генерируются псевдослучайно (ключ может также импортироваться). Вектор должен сохраняться с файлом шифртекста (формат придумать самостоятельно) и использоваться при расшифровывании.
2. Генерация двух ключевых пар алгоритма ГОСТ Р 34.10-2001 с заранее определённым набором параметров.
3. Сохранение открытого и закрытого ключей в файлах PEM-формата.
4. Выработку общего для двух ключевых пар ключа симметричного шифрования, используемого для обмена сеансовым ключом. При этом величина UKM должна вырабатываться псевдослучайно, сохраняться в файле, а затем считываться при генерации общего ключа «второй стороной».
5. Зашифровывание/расшифровывание сеансового ключа блочным шифром ГОСТ 28147-89 на базе общего ключа, выработанного в п. 4, сохранение его в файле, извлечение из файла. Для зашифровывания сеансового ключа можно использовать тот же вектор инициализации, что и для шифрования файла с открытым текстом.

# Код программы

## Часть А

main.h:

1. /\* \_TEXT, \_T etc \*/
2. #include <atlstr.h>
3. /\* setlocale \*/
4. #include <locale.h>
5. /\* EVP\_\* funcs & consts + BIO \*/
6. #include <openssl\evp.h>
7. /\* all errors funcs & consts \*/
8. #include <openssl\err.h>
9. /\* RAND\_\* funcs \*/
10. #include <openssl\rand.h>
11. /\* RSA\_\* & EVP\_PKEY\_\* \*/
12. #include <openssl\rsa.h>
13. /\* PEM\_\* \*/
14. #include <openssl\pem.h>
16. **BOOL** CreateRSAPair(BIO\* publicBIO, BIO\* privateBIO);
17. **BOOL** EncryptRSA(BIO\* publicBIO, **BYTE**\* inputBuf, BIO\* outputBIO);
18. **BOOL** DecryptRSA(BIO\* privateBIO, **BYTE**\* outputBuf, BIO\* inputBIO);
19. **BOOL** EncryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
     **const** **BYTE**\* key, **const** **BYTE**\* iv);
20. **BOOL** DecryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
     **const** **BYTE**\* key, **const** **BYTE**\* iv);

main.cpp:

1. // + libeay32.dll, ssleay.dll & gost.dll from openssl pckg
2. #include "main.h"
4. #define MAX\_STRING\_LENGTH 255
6. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
7. {
8. setlocale(LC\_ALL, "RUS");
9. USES\_CONVERSION; // for T2A
11. /\* evp.h; OPENSSL\_LOAD\_CONF ? \_conf : \_noconf
12. \* loads all realization of cipher & hashing algorithms
13. \*/
14. OpenSSL\_add\_all\_algorithms();
15. /\* err.h; registers the error strings for all funcs \*/
16. ERR\_load\_crypto\_strings();
18. unsigned **char** key[16] = { 0 };
19. unsigned **char** iv[16] = { 0 };
20. **HANDLE** inputFile = NULL;
21. **HANDLE** outputFile = NULL;
22. **HANDLE** keyFile = NULL;
23. **int** result;
25. unsigned **int** todo;
26. \_tprintf(\_T("Выберите что будем делать:\n"));
27. \_tprintf(\_T("1. Зашифровывать\n"));
28. \_tprintf(\_T("2. Расшифровывать\n"));
29. \_tprintf(\_T("[<1>/2]: "));
30. \_tscanf(\_T("%u"), &todo);
32. **if** (todo < 1 || todo > 2) {
33. todo = 1;
34. }
36. unsigned **int** mode;
37. \_tprintf(\_T("Выберите режим:\n"));
38. \_tprintf(\_T("1. CBC\n"));
39. \_tprintf(\_T("2. CFB\n"));
40. \_tprintf(\_T("[<1>/2]: "));
41. \_tscanf(\_T("%u"), &mode);
43. **if** (mode < 1 || mode > 2) {
44. mode = 1;
45. }
47. **TCHAR** input[MAX\_STRING\_LENGTH];
48. \_tprintf(\_T("Введите название входного файла: "));
49. \_tscanf(\_T("%s"), input);
51. **TCHAR** output[MAX\_STRING\_LENGTH];
52. \_tprintf(\_T("Введите название выходного файла: "));
53. \_tscanf(\_T("%s"), output);
55. **if** (todo == 1) {
56. // encrypt
58. unsigned **int** keyType;
59. \_tprintf(\_T("Сеансовый ключ:\n"));
60. \_tprintf(\_T("1. Сгенерировать\n"));
61. \_tprintf(\_T("2. Выбрать из файла\n"));
62. \_tprintf(\_T("[<1>/2]: "));
63. \_tscanf(\_T("%u"), &keyType);
65. **if** (keyType < 1 || keyType > 3) {
66. keyType = 1;
67. }
69. **if** (keyType == 1) {
70. // generate keys
72. /\* init rand generator;
73. \* gets data from hash of screenshot
74. \*/
75. RAND\_screen();
77. /\* returns 1 when success, otherwise returns 0 \*/
78. result = RAND\_bytes(key, **sizeof**(key));
80. **if** (result != 1) {
81. \_tprintf(\_T("Ошибка при генерации сеансового ключа.\n"));
82. getchar();
83. exit(EXIT\_FAILURE);
84. }
86. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
87. \_tprintf(\_T("Введите название для сохранения файла с открытым ключом: "));
88. \_tscanf(\_T("%s"), publicKey);
90. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
91. \_tprintf(\_T("Введите название для сохранения файла с закрытым ключом: "));
92. \_tscanf(\_T("%s"), privateKey);
94. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
95. \_tprintf(\_T("Введите название для сохранения файла с сеансовым ключом: "));
96. \_tscanf(\_T("%s"), sessionKey);
98. /\* bio.h \*/
99. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "w");
100. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "w");
101. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "w");
103. **if** (!CreateRSAPair(publicKeyFile, privateKeyFile)) {
104. \_tprintf(\_T("Ошибка при создании пары ключей.\n"));
106. BIO\_free(publicKeyFile);
107. BIO\_free(privateKeyFile);
109. getchar();
110. exit(EXIT\_FAILURE);
111. }
113. BIO\_free(publicKeyFile);
114. BIO\_free(privateKeyFile);
116. publicKeyFile = BIO\_new\_file(T2A(publicKey), "r");
118. **if** (!EncryptRSA(publicKeyFile, key, sessionKeyFile)) {
119. \_tprintf(\_T("Ошибка при зашифровывании сеансового ключа.\n"));
120. getchar();
121. exit(EXIT\_FAILURE);
122. }
124. BIO\_free(publicKeyFile);
125. BIO\_free(sessionKeyFile);
126. } **else** {
127. // keys from files
129. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
130. \_tprintf(\_T("Введите название для файла с сеансовым ключом: "));
131. \_tscanf(\_T("%s"), sessionKey);
133. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
134. \_tprintf(\_T("Введите название для файла с закрытым ключом: "));
135. \_tscanf(\_T("%s"), privateKey);
137. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
138. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "r");
140. **if** (!DecryptRSA(privateKeyFile, key, sessionKeyFile)) {
141. \_tprintf(\_T("Ошибка при расшифровывании сеансового ключа.\n"));
142. getchar();
143. exit(EXIT\_FAILURE);
144. }
146. BIO\_free(privateKeyFile);
147. BIO\_free(sessionKeyFile);
148. }
150. /\* returns 1 when success,
151. \* 0 when data not crypto-strong
152. \* -1 when fail
153. \*/
154. result = RAND\_pseudo\_bytes(iv, **sizeof**(iv));
155. **if** (result < 0) {
156. \_tprintf(\_T("Ошибка при генерации вектора инициализации.\n"));
157. getchar();
158. exit(EXIT\_FAILURE);
159. }
161. inputFile = CreateFile(input, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0,  
      NULL);
162. outputFile = CreateFile(output, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS,  
      0, NULL);
164. unsigned **long** written;
166. // write iv first
167. WriteFile(outputFile, &iv, **sizeof**(iv), &written, 0);
169. **if** (mode == 1) {
170. /\* third param is EVP\_CIPHER struct which contains
171. \* info about cipher method like: size of block, length of key, mode etc
172. \* for instance here: AES, 128-bit, CBC-mode
173. \*/
174. result = EncryptAES(inputFile, outputFile, EVP\_aes\_128\_cbc(), key, iv);
175. } **else** {
176. result = EncryptAES(inputFile, outputFile, EVP\_aes\_128\_cfb128(), key,  
      iv);
177. }
179. **if** (result == 0) {
180. \_tprintf(\_T("Ошибка при шифровании.\n"));
181. getchar();
182. exit(EXIT\_FAILURE);
183. } **else** {
184. \_tprintf(\_T("Шифрование успешно завершено.\n"));
185. getchar();
186. }
188. CloseHandle(inputFile);
189. CloseHandle(outputFile);
190. } **else** {
191. // decrypt
193. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
194. \_tprintf(\_T("Введите название для файла с сеансовым ключом: "));
195. \_tscanf(\_T("%s"), sessionKey);
197. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
198. \_tprintf(\_T("Введите название для файла с закрытым ключом: "));
199. \_tscanf(\_T("%s"), privateKey);
201. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
202. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "r");
204. **if** (!DecryptRSA(privateKeyFile, key, sessionKeyFile)) {
205. \_tprintf(\_T("Ошибка при расшифровывании сеансового ключа.\n"));
206. getchar();
207. exit(EXIT\_FAILURE);
208. }
210. BIO\_free(privateKeyFile);
211. BIO\_free(sessionKeyFile);
213. inputFile = CreateFile(input, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0,  
      NULL);
214. outputFile = CreateFile(output, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS,  
      0, NULL);
216. unsigned **long** read;
218. ReadFile(inputFile, &iv, **sizeof**(iv), &read, 0);
220. **if** (mode == 1) {
221. result = DecryptAES(inputFile, outputFile, EVP\_aes\_128\_cbc(), key, iv);
222. } **else** {
223. result = DecryptAES(inputFile, outputFile, EVP\_aes\_128\_cfb128(), key, iv);
224. }
226. **if** (result == 0) {
227. \_tprintf(\_T("Ошибка при расшифровывании.\n"));
228. getchar();
229. exit(EXIT\_FAILURE);
230. } **else** {
231. \_tprintf(\_T("Расшифровывание успешно завершено.\n"));
232. getchar();
233. }
235. CloseHandle(inputFile);
236. CloseHandle(outputFile);
237. }
239. getchar();
240. }
242. **BOOL** CreateRSAPair(BIO\* publicBIO, BIO\* privateBIO) {
243. **int** result;
245. /\* NULLS -- func for calc demo & args
246. \* RSA\_F4 == 65537
247. \*/
248. RSA\* rsa = RSA\_generate\_key(2048, RSA\_F4, NULL, NULL);
250. /\* checks that p and q are in fact prime, and that n = p \* q \*/
251. result = RSA\_check\_key(rsa);
252. **if** (result != 1) **return** FALSE;
254. result = PEM\_write\_bio\_RSAPublicKey(publicBIO, rsa);
256. **if** (result != 1) **return** FALSE;
258. /\* bio, rsa, evp\_cipher,
259. \* char array with password, length of it
260. \* cb for input password
261. \* password
262. \*/
263. result = PEM\_write\_bio\_RSAPrivateKey(privateBIO, rsa, EVP\_aes\_128\_cfb128(),  
      NULL, 0, NULL, NULL);
265. **if** (result != 1) **return** FALSE;
267. RSA\_free(rsa);
269. **return** TRUE;
270. }
272. **BOOL** EncryptRSA(BIO\* publicBIO, **BYTE**\* inputBuf, BIO\* outputBIO) {
273. RSA\* rsa = RSA\_new();
275. // NULLS -- cb for pass & pass
276. rsa = PEM\_read\_bio\_RSAPublicKey(publicBIO, &rsa, NULL, NULL);
278. // 'cause rsa key length == 2048, outLength = 256
279. **int** outLength = RSA\_size(rsa);
280. **BYTE**\* outputBuf = **new** **BYTE**[outLength];
282. // 16 -- length (AES128);
283. **int** result = RSA\_public\_encrypt(16, inputBuf, outputBuf, rsa,  
      RSA\_PKCS1\_OAEP\_PADDING);
285. **if** (result == -1) **return** FALSE;
287. /\* returns count of written bytes \*/
288. BIO\_write(outputBIO, outputBuf, outLength);
290. **delete**[] outputBuf;
292. **return** TRUE;
293. }
295. **BOOL** DecryptRSA(BIO\* privateBIO, **BYTE**\* outputBuf, BIO\* inputBIO) {
296. RSA\* rsa = RSA\_new();
298. // NULLS -- cb for pass & pass
299. rsa = PEM\_read\_bio\_RSAPrivateKey(privateBIO, &rsa, NULL, NULL);
301. **int** inLength = RSA\_size(rsa);
302. **BYTE**\* inputBuf = **new** **BYTE**[inLength];
304. /\* returns count of read bytes \*/
305. BIO\_read(inputBIO, inputBuf, inLength);
307. // 16 -- length (AES128);
308. **int** result = RSA\_private\_decrypt(inLength, inputBuf, outputBuf, rsa,  
      RSA\_PKCS1\_OAEP\_PADDING);
310. **if** (result == -1) **return** FALSE;
312. **delete**[] inputBuf;
314. RSA\_free(rsa);
316. **return** TRUE;
317. }
319. **BOOL** EncryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
      **const** **BYTE**\* key, **const** **BYTE**\* iv) {
320. /\* context of cipher algo \*/
321. EVP\_CIPHER\_CTX ctx;
323. EVP\_CIPHER\_CTX\_init(&ctx);
325. **int** result = EVP\_EncryptInit(&ctx, cipher, key, iv);
327. **if** (!result) **return** FALSE;
329. **bool** final = **false**;
331. **if** (cipher != EVP\_aes\_128\_cbc()) {
332. final = **true**;
333. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
334. }
336. **while** (**true**) {
337. **const** **int** length = 16;
338. **int** outLength = 0;
339. **int** finalLength = 0;
341. **BYTE** inputBuf[length];
342. **BYTE** outputBuf[length];
344. unsigned **long** read;
346. ReadFile(inputFile, inputBuf, length, &read, 0);
348. **if** (read < 0) **return** FALSE;
350. **if** (read == 0 && final) **break**;
352. result = EVP\_EncryptUpdate(&ctx, outputBuf, &outLength, inputBuf, read);
354. **if** (!result) **return** FALSE;
356. **if** (read != length && !final) {
357. result = EVP\_EncryptFinal(&ctx, outputBuf + outLength, &finalLength);
359. **if** (!result) **return** FALSE;
361. final = **true**;
362. }
364. unsigned **long** written;
365. WriteFile(outputFile, outputBuf, outLength + finalLength, &written, 0);
366. }
368. EVP\_CIPHER\_CTX\_cleanup(&ctx);
370. **return** TRUE;
371. }
373. **BOOL** DecryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
      **const** **BYTE**\* key, **const** **BYTE**\* iv) {
374. /\* context of cipher algo \*/
375. EVP\_CIPHER\_CTX ctx;
377. EVP\_CIPHER\_CTX\_init(&ctx);
379. **int** result = EVP\_DecryptInit(&ctx, cipher, key, iv);
381. **if** (!result) **return** FALSE;
383. **bool** final = **false**;
384. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
386. **if** (cipher != EVP\_aes\_128\_cbc())
387. {
388. final = **true**;
389. }
391. **long** fileLength = GetFileSize(inputFile, NULL) - 16;
392. **long** fullRead = 0;
394. **while** (**true**) {
395. **const** **int** length = 16;
396. **int** outLength = 0;
397. **int** finalLength = 0;
399. **BYTE** inputBuf[length];
400. **BYTE** outputBuf[length];
402. unsigned **long** read;
404. ReadFile(inputFile, inputBuf, length, &read, 0);
406. **if** (read < 0) **return** FALSE;
408. **if** (read == 0 && final) **break**;
410. fullRead += read;
412. **if** (fullRead == fileLength && !final) EVP\_CIPHER\_CTX\_set\_padding(&ctx, 1);
414. result = EVP\_DecryptUpdate(&ctx, outputBuf, &outLength, inputBuf, read);
416. **if** (!result) **return** FALSE;
418. **if** (fullRead == fileLength && !final) {
419. result = EVP\_DecryptFinal(&ctx, outputBuf + outLength, &finalLength);
421. **if** (!result) **return** FALSE;
423. final = **true**;
424. }
426. unsigned **long** written;
427. WriteFile(outputFile, outputBuf, outLength + finalLength, &written, 0);
428. }
430. EVP\_CIPHER\_CTX\_cleanup(&ctx);
432. **return** TRUE;
433. }

## Часть Б

main.h:

1. /\* \_TEXT, \_T etc \*/
2. #include <atlstr.h>
3. /\* setlocale \*/
4. #include <locale.h>
5. /\* EVP\_\* funcs & consts + BIO \*/
6. #include <openssl\evp.h>
7. /\* all errors funcs & consts \*/
8. #include <openssl\err.h>
9. /\* RAND\_\* funcs \*/
10. #include <openssl\rand.h>
11. /\* RSA\_\* & EVP\_PKEY\_\* \*/
12. #include <openssl\rsa.h>
13. /\* PEM\_\* \*/
14. #include <openssl\pem.h>
15. /\* ENGINE\_\* \*/
16. #include <openssl\engine.h>
18. **BOOL** EncryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
     BIO\* sessionKeyFile, unsigned **char**\* key);
19. **BOOL** DecryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
     BIO\* sessionKeyFile, unsigned **char**\* key);
20. **int** EncryptGOSTBuffer(unsigned **char**\* key, **int** keyLength, unsigned **char**\* encryptedKey,  
     **int**\* encryptedKeyLength, unsigned **char**\* genericKey, unsigned **char**\* iv);
21. **int** DecryptGOSTBuffer(unsigned **char**\* key, **int** keyLength, unsigned **char**\* encryptedKey,  
     **int**\* encryptedKeyLength, unsigned **char**\* genericKey, unsigned **char**\* iv);
22. **int** EncryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key, unsigned **char**\* iv);
23. **int** DecryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key, unsigned **char**\* iv);
24. **BOOL** CreateGOSTPair(BIO\* publicBIO, BIO\* privateBIO);

main.cpp:

1. #include "main.h"
3. #define MAX\_STRING\_LENGTH 255
4. #define KEY\_LENGTH 32
5. #define IV\_LENGTH 8
6. #define UKM\_LENGTH 8
8. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
9. {
10. setlocale(LC\_ALL, "RUS");
11. USES\_CONVERSION;
13. /\* evp.h
14. \* loads all realization of cipher & hashing algorithms + read conf
15. \*/
16. OPENSSL\_add\_all\_algorithms\_conf();
17. /\* err.h; registers the error strings for all funcs \*/
18. ERR\_load\_crypto\_strings();
20. unsigned **char** key[KEY\_LENGTH];
21. unsigned **char** iv[IV\_LENGTH];
23. **TCHAR** input[MAX\_STRING\_LENGTH];
24. **TCHAR** output[MAX\_STRING\_LENGTH];
25. **HANDLE** keyFile = NULL;
26. **int** result;
28. unsigned **int** todo;
29. \_tprintf(\_T("Выберите что будем делать:\n"));
30. \_tprintf(\_T("1. Зашифровывать\n"));
31. \_tprintf(\_T("2. Расшифровывать\n"));
32. \_tprintf(\_T("3. Генерировать ключевые пары\n"));
33. \_tprintf(\_T("[<1>/2/3]: "));
34. \_tscanf(\_T("%u"), &todo);
36. **if** (todo < 1 || todo > 3) {
37. todo = 1;
38. }
40. **if** (todo != 3) {
41. \_tprintf(\_T("Введите название входного файла: "));
42. \_tscanf(\_T("%s"), input);
44. \_tprintf(\_T("Введите название выходного файла: "));
45. \_tscanf(\_T("%s"), output);
46. }
48. **if** (todo == 1) {
49. // encrypt
51. /\* init rand generator;
52. \* gets data from hash of screenshot
53. \*/
54. RAND\_screen();
56. /\* returns 1 when success, otherwise returns 0 \*/
57. result = RAND\_bytes(key, KEY\_LENGTH);
59. **if** (result != 1) {
60. \_tprintf(\_T("Ошибка при генерации сеансового ключа.\n"));
61. getchar();
62. exit(EXIT\_FAILURE);
63. }
65. /\* returns 1 when success,
66. \* 0 when data not crypto-strong
67. \* -1 when fail
68. \*/
69. result = RAND\_pseudo\_bytes(iv, IV\_LENGTH);
71. **if** (result < 0) {
72. \_tprintf(\_T("Ошибка при генерации вектора инициализации.\n"));
73. getchar();
74. exit(EXIT\_FAILURE);
75. }
77. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
78. \_tprintf(\_T("Введите название файла с открытым ключом: "));
79. \_tscanf(\_T("%s"), publicKey);
81. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
82. \_tprintf(\_T("Введите название файла с закрытым ключом: "));
83. \_tscanf(\_T("%s"), privateKey);
85. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
86. \_tprintf(\_T("Введите название для сохранения файла с сеансовым ключом: "));
87. \_tscanf(\_T("%s"), sessionKey);
89. **TCHAR** ukmKey[MAX\_STRING\_LENGTH];
90. \_tprintf(\_T("Введите название для сохранения файла с UKM-ключом: "));
91. \_tscanf(\_T("%s"), ukmKey);
93. BIO \*inputFile = BIO\_new\_file(T2A(input), "r");
94. BIO \*outputFile = BIO\_new\_file(T2A(output), "w");
95. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
96. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "r");
97. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "w");
98. BIO \*ukmKeyFile = BIO\_new\_file(T2A(ukmKey), "w");
100. // write iv first
101. BIO\_write(outputFile, iv, IV\_LENGTH);
103. result = EncryptGOSTBIO(inputFile, outputFile, key, iv);
105. result = EncryptSessionKey(privateKeyFile, publicKeyFile, ukmKeyFile,  
      sessionKeyFile, key);
107. **if** (result != 1) {
108. \_tprintf(\_T("Ошибка зашифровывании сеансового ключа.\n"));
110. BIO\_free(privateKeyFile);
111. BIO\_free(publicKeyFile);
112. BIO\_free(sessionKeyFile);
113. BIO\_free(ukmKeyFile);
114. BIO\_free(inputFile);
115. BIO\_free(outputFile);
117. getchar();
118. exit(EXIT\_FAILURE);
119. }
121. BIO\_free(privateKeyFile);
122. BIO\_free(publicKeyFile);
123. BIO\_free(sessionKeyFile);
124. BIO\_free(ukmKeyFile);
125. BIO\_free(inputFile);
126. BIO\_free(outputFile);
128. **if** (!result) {
129. \_tprintf(\_T("Ошибка при зашифровывании текста.\n"));
130. getchar();
131. exit(EXIT\_FAILURE);
132. }
134. \_tprintf(\_T("Шифрование успешно завершено.\n"));
135. getchar();
136. } **else** **if** (todo == 2) {
137. // decrypt
139. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
140. \_tprintf(\_T("Введите название файла с открытым ключом: "));
141. \_tscanf(\_T("%s"), publicKey);
143. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
144. \_tprintf(\_T("Введите название файла с закрытым ключом: "));
145. \_tscanf(\_T("%s"), privateKey);
147. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
148. \_tprintf(\_T("Введите название файла с сеансовым ключом: "));
149. \_tscanf(\_T("%s"), sessionKey);
151. **TCHAR** ukmKey[MAX\_STRING\_LENGTH];
152. \_tprintf(\_T("Введите название файла с UKM-ключом: "));
153. \_tscanf(\_T("%s"), ukmKey);
155. BIO \*inputFile = BIO\_new\_file(T2A(input), "r");
156. BIO \*outputFile = BIO\_new\_file(T2A(output), "w+");
157. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
158. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "r");
159. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "r");
160. BIO \*ukmKeyFile = BIO\_new\_file(T2A(ukmKey), "r");
162. // read iv first
163. BIO\_read(inputFile, iv, IV\_LENGTH);
165. unsigned **char** keyBuffer[KEY\_LENGTH];
167. result = DecryptSessionKey(privateKeyFile, publicKeyFile, ukmKeyFile,  
      sessionKeyFile, keyBuffer);
169. **if** (result != 1) {
170. \_tprintf(\_T("Ошибка при расшифровывании сеансового ключа.\n"));
172. BIO\_free(privateKeyFile);
173. BIO\_free(publicKeyFile);
174. BIO\_free(sessionKeyFile);
175. BIO\_free(ukmKeyFile);
176. BIO\_free(inputFile);
177. BIO\_free(outputFile);
179. getchar();
180. exit(EXIT\_FAILURE);
181. }
183. result = DecryptGOSTBIO(inputFile, outputFile, keyBuffer, iv);
185. BIO\_free(privateKeyFile);
186. BIO\_free(publicKeyFile);
187. BIO\_free(sessionKeyFile);
188. BIO\_free(ukmKeyFile);
189. BIO\_free(inputFile);
190. BIO\_free(outputFile);
192. **if** (!result) {
193. \_tprintf(\_T("Ошибка при расшифровывании текста.\n"));
194. getchar();
195. exit(EXIT\_FAILURE);
196. }
198. \_tprintf(\_T("Расшифровывание успешно завершено.\n"));
199. getchar();
200. } **else** {
201. // generate key pairs
203. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
204. \_tprintf(\_T("Введите название файла с открытым ключом: "));
205. \_tscanf(\_T("%s"), publicKey);
207. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
208. \_tprintf(\_T("Введите название файла с закрытым ключом: "));
209. \_tscanf(\_T("%s"), privateKey);
211. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "wb");
212. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "wb");
214. result = CreateGOSTPair(publicKeyFile, privateKeyFile);
216. BIO\_free(publicKeyFile);
217. BIO\_free(privateKeyFile);
219. **if** (!result) {
220. \_tprintf(\_T("Ошибка при создании PEM-пары.\n"));
221. getchar();
222. exit(EXIT\_FAILURE);
223. }
225. \_tprintf(\_T("PEM-пара создана успешно.\n"));
226. getchar();
227. }
229. getchar();
230. }
232. **BOOL** EncryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
      BIO\* sessionKeyFile, unsigned **char**\* key) {
233. **int** result = 0;
235. // ctx for private key
236. EVP\_PKEY \*privatePKey = EVP\_PKEY\_new();
238. privatePKey = PEM\_read\_bio\_PrivateKey(privateKeyFile, &privatePKey, NULL, NULL);
240. // ctx for public key
241. EVP\_PKEY \*publicPKey = EVP\_PKEY\_new();
243. publicPKey = PEM\_read\_bio\_PUBKEY(publicKeyFile, &publicPKey, NULL, NULL);
245. **if** (publicPKey == NULL || privatePKey == NULL) **return** FALSE;
247. ENGINE \*engine = ENGINE\_by\_id("gost");
249. **if** (engine == NULL) **return** FALSE;
251. // ctx for key pairs
252. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new(privatePKey, engine);
254. **if** (ctx == NULL) **return** FALSE;
256. result = EVP\_PKEY\_derive\_init(ctx);
258. **if** (result != 1) **return** FALSE;
260. RAND\_screen();
262. unsigned **\_\_int64** ukm;
263. unsigned **\_\_int64** iv;
264. RAND\_bytes((unsigned **char** \*)&ukm, UKM\_LENGTH);
265. RAND\_bytes((unsigned **char** \*)&iv, IV\_LENGTH);
267. // ctx, key type, operation type, cmd code, buff size for cmd, &buff
268. result = EVP\_PKEY\_CTX\_ctrl(ctx, -1, -1, EVP\_PKEY\_CTRL\_SET\_IV,  
      UKM\_LENGTH, &ukm);
270. **if** (result != 1) **return** FALSE;
272. result = EVP\_PKEY\_derive\_set\_peer(ctx, publicPKey);
274. **if** (result != 1) **return** FALSE;
276. **size\_t** keyLength = 0;
278. // returns key length
279. result = EVP\_PKEY\_derive(ctx, NULL, &keyLength);
281. unsigned **char**\* genericKey = **new** unsigned **char**[keyLength];
283. // returns generic key
284. result = EVP\_PKEY\_derive(ctx, genericKey, &keyLength);
286. **if** (result != 1) {
287. **delete**[] genericKey;
288. **return** FALSE;
289. }
291. **int** encryptedKeyLength = KEY\_LENGTH;
292. unsigned **char**\* encryptedKey = **new** unsigned **char**[KEY\_LENGTH];
294. result = EncryptGOSTBuffer(key, KEY\_LENGTH, encryptedKey, &encryptedKeyLength,  
      genericKey, (unsigned **char**\*)&iv);
296. **if** (!result)
297. {
298. **delete**[] encryptedKey;
299. **delete**[] genericKey;
300. **return** FALSE;
301. }
303. BIO\_write(sessionKeyFile, &iv, IV\_LENGTH);
304. BIO\_write(sessionKeyFile, encryptedKey, KEY\_LENGTH);
306. BIO\_write(ukmKeyFile, &ukm, UKM\_LENGTH);
308. **delete**[] encryptedKey;
309. **delete**[] genericKey;
311. EVP\_PKEY\_free(privatePKey);
312. EVP\_PKEY\_free(publicPKey);
313. EVP\_PKEY\_CTX\_free(ctx);
315. **return** TRUE;
316. }
318. **BOOL** DecryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
      BIO\* sessionKeyFile, unsigned **char**\* key) {
319. **int** result = 0;
321. EVP\_PKEY \*privatePKey = EVP\_PKEY\_new();
323. privatePKey = PEM\_read\_bio\_PrivateKey(privateKeyFile, &privatePKey, NULL, NULL);
325. EVP\_PKEY \*publicPKey = EVP\_PKEY\_new();
327. publicPKey = PEM\_read\_bio\_PUBKEY(publicKeyFile, &publicPKey, NULL, NULL);
329. **if** (publicPKey == NULL || privatePKey == NULL) **return** FALSE;
331. ENGINE \*engine = ENGINE\_by\_id("gost");
333. **if** (engine == NULL) **return** FALSE;
335. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new(privatePKey, engine);
337. **if** (ctx == NULL) **return** FALSE;
339. result = EVP\_PKEY\_derive\_init(ctx);
341. **if** (result != 1) **return** FALSE;
343. RAND\_screen();
345. unsigned **\_\_int64** ukm;
346. unsigned **char** iv[IV\_LENGTH];
348. BIO\_read(ukmKeyFile, &ukm, UKM\_LENGTH);
349. BIO\_read(sessionKeyFile, iv, IV\_LENGTH);
351. // key type, operation type, cmd code, buff size for cmd, &buff
352. result = EVP\_PKEY\_CTX\_ctrl(ctx, -1, -1, EVP\_PKEY\_CTRL\_SET\_IV,  
      UKM\_LENGTH, &ukm);
354. **if** (result != 1) **return** FALSE;
356. result = EVP\_PKEY\_derive\_set\_peer(ctx, publicPKey);
358. **if** (result != 1) **return** FALSE;
360. **size\_t** keyLength = 0;
362. result = EVP\_PKEY\_derive(ctx, NULL, &keyLength);
364. unsigned **char**\* genericKey = **new** unsigned **char**[keyLength];
366. result = EVP\_PKEY\_derive(ctx, genericKey, &keyLength);
368. **if** (result != 1) {
369. **delete**[] genericKey;
370. **return** FALSE;
371. }
373. **int** decryptedKeyLength = KEY\_LENGTH;
374. //\*key = new unsigned char[KEY\_LENGTH];
375. unsigned **char**\* encryptedKey = **new** unsigned **char**[KEY\_LENGTH];
377. BIO\_read(sessionKeyFile, encryptedKey, KEY\_LENGTH);
379. result = DecryptGOSTBuffer(encryptedKey, KEY\_LENGTH, key, &decryptedKeyLength,  
      genericKey, iv);
381. **delete**[] encryptedKey;
382. **delete**[] genericKey;
384. **if** (!result) **return** FALSE;
386. EVP\_PKEY\_free(privatePKey);
387. EVP\_PKEY\_free(publicPKey);
388. EVP\_PKEY\_CTX\_free(ctx);
390. **return** TRUE;
391. }
393. **int** EncryptGOSTBuffer(unsigned **char**\* key, **int** keyLength,  
      unsigned **char**\* encryptedKey, **int**\* encryptedKeyLength,  
      unsigned **char**\* genericKey, unsigned **char**\* iv) {
394. **int** result = 0;
396. ENGINE \*engine = ENGINE\_by\_id("gost");
398. **if** (engine == NULL) **return** 0;
400. EVP\_CIPHER\_CTX ctx;
401. EVP\_CIPHER\_CTX\_init(&ctx);
402. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
404. result = EVP\_EncryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, genericKey, iv);
406. **if** (result != 1) {
407. EVP\_CIPHER\_CTX\_cleanup(&ctx);
408. **return** result;
409. }
411. result = EVP\_EncryptUpdate(&ctx, encryptedKey, encryptedKeyLength, key,  
      keyLength);
413. EVP\_CIPHER\_CTX\_cleanup(&ctx);
415. **return** result;
416. }
418. **int** DecryptGOSTBuffer(unsigned **char**\* encryptedKey, **int** encryptedKeyLength,  
      unsigned **char**\* key, **int**\* keyLength,  
      unsigned **char**\* genericKey, unsigned **char**\* iv) {
419. **int** result = 0;
421. ENGINE \*engine = ENGINE\_by\_id("gost");
423. **if** (engine == NULL)
424. **return** 0;
426. EVP\_CIPHER\_CTX ctx;
427. EVP\_CIPHER\_CTX\_init(&ctx);
428. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
430. result = EVP\_DecryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, genericKey, iv);
432. **if** (result != 1) {
433. EVP\_CIPHER\_CTX\_cleanup(&ctx);
434. **return** result;
435. }
437. result = EVP\_DecryptUpdate(&ctx, key, keyLength, encryptedKey,  
      encryptedKeyLength);
439. EVP\_CIPHER\_CTX\_cleanup(&ctx);
441. **return** result;
442. }
444. **int** EncryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key,  
      unsigned **char**\* iv) {
445. **int** result = 0;
447. ENGINE \*engine = ENGINE\_by\_id("gost");
449. **if** (engine == NULL) **return** FALSE;
451. EVP\_CIPHER\_CTX ctx;
452. EVP\_CIPHER\_CTX\_init(&ctx);
453. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
455. result = EVP\_EncryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, key, iv);
457. **if** (result != 1) {
458. EVP\_CIPHER\_CTX\_cleanup(&ctx);
459. **return** FALSE;
460. }
462. **while** (**true**)
463. {
464. **int** outputLength = 8;
465. **int** inputLength = 8;
467. unsigned **char** input[8];
468. unsigned **char** output[8];
470. **int** read = BIO\_read(inputFile, input, 8);
472. **if** (read == 0) {
473. EVP\_CIPHER\_CTX\_cleanup(&ctx);
474. **break**;
475. }
477. result = EVP\_EncryptUpdate(&ctx, output, &outputLength, input, read);
479. **if** (!result) {
480. EVP\_CIPHER\_CTX\_cleanup(&ctx);
481. **return** FALSE;
482. }
484. BIO\_write(outputFile, output, outputLength);
485. }
487. EVP\_CIPHER\_CTX\_cleanup(&ctx);
489. **return** TRUE;
490. }
492. **int** DecryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key,  
      unsigned **char**\* iv) {
493. **int** result = 0;
495. ENGINE \*engine = ENGINE\_by\_id("gost");
497. **if** (engine == NULL) **return** FALSE;
499. EVP\_CIPHER\_CTX ctx;
500. EVP\_CIPHER\_CTX\_init(&ctx);
501. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
503. result = EVP\_DecryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, key, iv);
505. **if** (result != 1) {
506. EVP\_CIPHER\_CTX\_cleanup(&ctx);
507. **return** FALSE;
508. }
510. **while** (**true**)
511. {
512. **int** outputLength = 8;
513. **int** inputLength = 8;
515. unsigned **char** input[8];
516. unsigned **char** output[8];
518. **int** read = BIO\_read(inputFile, input, 8);
520. **if** (read == 0) {
521. EVP\_CIPHER\_CTX\_cleanup(&ctx);
522. **break**;
523. }
525. result = EVP\_DecryptUpdate(&ctx, output, &outputLength, input, read);
527. **if** (!result) {
528. EVP\_CIPHER\_CTX\_cleanup(&ctx);
529. **return** FALSE;
530. }
532. BIO\_write(outputFile, output, outputLength);
533. }
535. EVP\_CIPHER\_CTX\_cleanup(&ctx);
537. **return** TRUE;
538. }
540. **BOOL** CreateGOSTPair(BIO\* publicBIO, BIO\* privateBIO) {
541. **int** result = 0;
543. ENGINE \*engine = ENGINE\_by\_id("gost");
545. **if** (engine == NULL) **return** FALSE;
547. EVP\_PKEY \*pkey = EVP\_PKEY\_new();
549. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new\_id(NID\_id\_GostR3410\_2001, engine);
551. **if** (ctx == 0) **return** FALSE;
553. result = EVP\_PKEY\_CTX\_ctrl\_str(ctx, "paramset", "A");
555. **if** (result != 1) **return** FALSE;
557. result = EVP\_PKEY\_keygen\_init(ctx);
559. **if** (result != 1) **return** FALSE;
561. result = EVP\_PKEY\_keygen(ctx, &pkey);
563. **if** (result != 1) **return** FALSE;
565. result = PEM\_write\_bio\_PrivateKey(privateBIO, pkey,  
      EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      NULL, 0, NULL, NULL);
567. **if** (result != 1) **return** FALSE;
569. result = PEM\_write\_bio\_PUBKEY(publicBIO, pkey);
571. **if** (result != 1) **return** FALSE;
573. EVP\_PKEY\_free(pkey);
574. EVP\_PKEY\_CTX\_free(ctx);
576. **return** TRUE;
577. }

# Примеры работы программы

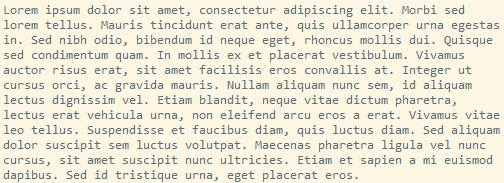


Рисунок 1. Исходный текст

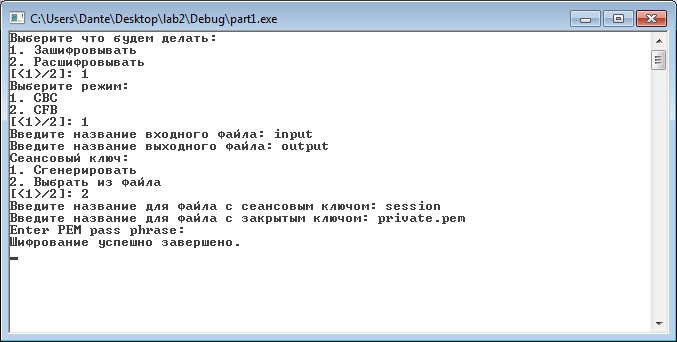


Рисунок 2. Зашифровывание

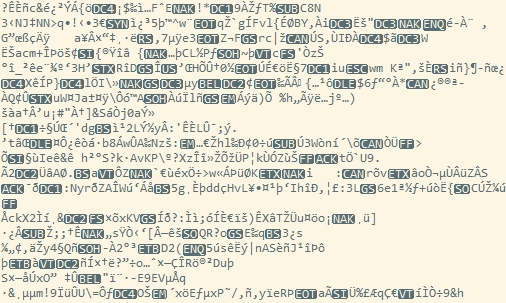


Рисунок 3. Шифртекст

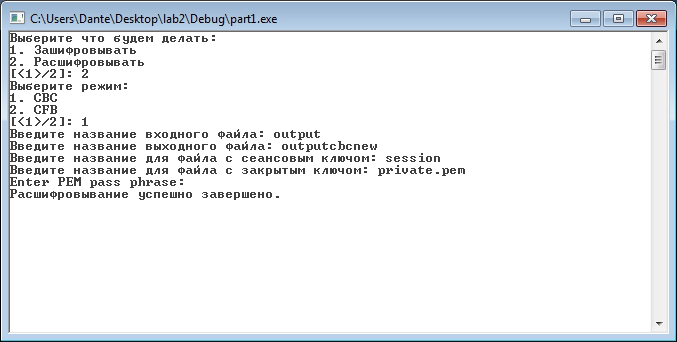


Рисунок 4. Расшифровывание

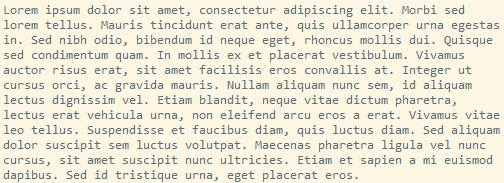


Рисунок 5. Расшифрованное сообщение