Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №3 по теме: «Создание криптографических сообщений с использованием интерфейса Microsoft CryptoAPI и цифровых сертификатов X.509»

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

**Проверил:**  
Смышляев А. Г.

Белгород  
2015

**Цель работы:** ознакомиться со структурой и форматами представления сертификатов открытых ключей, способами их создания и импортирования в систему, а также получить навыки в создании криптографических сообщений средствами интерфейса Microsoft CryptoAPI.

# Задание

1. С помощью криптографического пакета OpenSSL создать:
   * ключевую пару алгоритма RSA с длиной ключа 2048 бит и соответствующий ей самоподписанный сертификат центра сертификации;
   * две ключевые пары алгоритма RSA с длиной ключа 2048 бит и соответствующие им сертификаты в формате PKCS#12 для двух пользователей — участников процесса обмена криптографическими сообщениями. Сертификаты должны быть подписаны закрытым ключом центра сертификации.
2. Установить в системе созданные сертификаты. В отчёт внести последовательность команд OpenSSL, использованных для создания сертификатов центра сертификации и пользователей.
3. Разработать на языке программирования C/C++ с использованием средств криптографического интерфейса Microsoft CryptoAPI консольное или оконное приложение, выполняющее создание криптографического сообщения по стандарту CMS из указанного пользователем файла и дальнейшего его расшифровывания. Криптографическое сообщение должно содержать данные, зашифрованные алгоритмом AES-128 в режиме CBC и электронную подпись, сделанную с помощью алгоритма RSA. Приложение должно предлагать пользователю перечень имён субъектов сертификатов, установленных в хранилище «Личное», и принимать его выбор имени отправителя и получателя криптографического сообщения. Перед созданием сообщения необходимо верифицировать ЭП в составе выбранных сертификатов и проверить соответствие текущей даты периоду, заданному в их составе. Созданное криптографическое сообщение необходимо выгружать в указанный пользователем файл и загружать из него в память для расшифровывания. Расшифрованное сообщение также необходимо выгружать в файл, указанный пользователем.

# Практическая часть

## Создание сертификатов

Создание ключевой пары алгоритма RSA и самоподписанного сертификата ЦС в PEM-формате, которые будут использоваться для создания сертификатов конечных пользователей:

req -x509 -newkey rsa:2048 -days 730 -keyout ca\_test\_key.pem -out ca\_test\_cert.pem

Создание ключевой пары RSA и сертификата пользователя в PEM-формате, подписанного созданным ранее закрытым ключом ЦС:

req -newkey rsa:2048 -keyout User\_A\_key.pem -out User\_A\_req.pem

ca -md sha256 -keyfile ca\_test\_key.pem -cert ca\_test\_cert.pem -in User\_A\_req.pem -out User\_A\_cert.pem

Создание сертификата пользователя в формате PKCS#12, созданного на базе ключевой пары и сертификата из предыдущего пункта:

pkcs12 -export -in User\_A\_cert.pem -inkey User\_A\_key.pem -out User\_A\_cert.p12

## Код программы

1. // + crypt32.lib
2. #include <Windows.h>
3. #include <WinCrypt.h>
4. #include <locale.h>
5. #include <stdio.h>
6. #include <tchar.h>
8. #define MY\_ENCODING\_TYPE (PKCS\_7\_ASN\_ENCODING | X509\_ASN\_ENCODING)
9. #define SENDER\_CERT 0
10. #define RECIPIENT\_CERT 1
11. #define INPUT\_FILE 0
12. #define OUTPUT\_FILE 1
14. **void** Fail(**char**\* string);
15. **void** ShowStoreCerts(HCERTSTORE hStore);
16. PCCERT\_CONTEXT FindCert(HCERTSTORE hStore, **LPTSTR** findParams);
17. **int** CheckCert(HCERTSTORE hStore, PCCERT\_CONTEXT pSubjectContext);
18. PCCERT\_CONTEXT RequestCert(HCERTSTORE hStoreMy, HCERTSTORE hStoreRoot, **int** mode);
19. **FILE**\* RequestFile(**int** mode);
20. **int** EncryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
     PCCERT\_CONTEXT pRecipientCert);
21. **int** DecryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
     PCCERT\_CONTEXT pRecipientCert, HCERTSTORE hStore);
23. **int** main(**int** argc, **char**\* argv[]) {
24. setlocale(LC\_ALL, "RUS");
26. unsigned **int** mode = 0;
27. HCERTSTORE hStoreMy, hStoreRoot;
28. **FILE** \*inputFile, \*outputFile;
30. printf("Что будем делать?\n1. Зашифровывать\n2. Расшифровывать\n<1>/2: ");
31. scanf("%u", &mode);
33. **if** (mode < 1 || mode > 2) {
34. mode = 1;
35. }
37. /\* first param is a handle of a cryptographic service provider
38. \* but it's not used since XP
39. \*/
40. **if** (! (hStoreMy = CertOpenSystemStore(NULL, TEXT("MY")))) {
41. Fail("Не удалось открыть личное хранилище сертификатов.");
42. }
44. **if** (! (hStoreRoot = CertOpenSystemStore(NULL, TEXT("ROOT")))) {
45. Fail("Не удалось открыть хранилище сертификатов доверенных корневых центров \  
     сертификации.");
46. }
48. printf("\nСертификаты, доступные в личном хранилище:\n");
49. ShowStoreCerts(hStoreMy);
51. PCCERT\_CONTEXT pCertSender = RequestCert(hStoreMy, hStoreRoot, SENDER\_CERT);
52. PCCERT\_CONTEXT pCertRecipient = RequestCert(hStoreMy, hStoreRoot, RECIPIENT\_CERT);
53. inputFile = RequestFile(INPUT\_FILE);
54. outputFile = RequestFile(OUTPUT\_FILE);
56. **if** (mode == 1) {
57. **if** (!EncryptMessage(inputFile, outputFile, pCertSender, pCertRecipient)) {
58. Fail("Ошибка при зашифровывании сообщения.");
59. }
60. } **else** {
61. **if** (!DecryptMessage(inputFile, outputFile, pCertSender, pCertRecipient,  
     hStoreMy)) {
62. Fail("Ошибка при расшифровывании сообщения.");
63. }
64. }
66. // last param is flags
67. **if** (!CertCloseStore(hStoreMy, 0)) {
68. Fail("Не удалось закрыть личное хранилище сертификатов.");
69. }
71. **if** (!CertCloseStore(hStoreRoot, 0)) {
72. Fail("Не удалось закрыть хранилище сертификатов доверенных корневых центров \  
     сертификации.");
73. }
75. CertFreeCertificateContext(pCertSender);
76. CertFreeCertificateContext(pCertRecipient);
77. fclose(inputFile);
78. fclose(outputFile);
80. printf("\nРабота программы успешно завершена.\n");
81. system("pause");
82. }
84. **void** Fail(**char**\* string) {
85. printf("%s\n", string);
86. system("pause");
87. exit(EXIT\_FAILURE);
88. }
90. **void** ShowStoreCerts(HCERTSTORE hStore) {
91. PCCERT\_CONTEXT pCert = NULL;
92. **const** unsigned **int** nameLength = 128;
93. **TCHAR** szNameString[nameLength];
94. **int** result = 0;
96. **while** (pCert = CertEnumCertificatesInStore(hStore, pCert)) {
97. result = CertGetNameString(
98. pCert,
99. // part of name; this is for CN:
100. CERT\_NAME\_SIMPLE\_DISPLAY\_TYPE,
101. // type of processing; there is not processing:
102. 0,
103. // helper for 2nd; we don't need it:
104. NULL,
105. szNameString,
106. nameLength
107. );
109. **if** (result > 1) {
110. \_tprintf(TEXT("%s\n"), szNameString);
111. }
112. }
113. }
115. PCCERT\_CONTEXT FindCert(HCERTSTORE hStore, **LPTSTR** searchString) {
116. **return** CertFindCertificateInStore(
117. hStore,
118. MY\_ENCODING\_TYPE,
119. // flags-modificators for next param
120. 0,
121. // type of search; we search by CN
122. CERT\_FIND\_SUBJECT\_STR,
123. searchString,
124. // prev cert by same search; we have only 1 cert, so:
125. NULL
126. );
127. }
129. **int** CheckCert(HCERTSTORE hStore, PCCERT\_CONTEXT pSubjectContext) {
130. **DWORD** dwFlags = CERT\_STORE\_SIGNATURE\_FLAG | CERT\_STORE\_TIME\_VALIDITY\_FLAG;
131. PCCERT\_CONTEXT pCAContext = CertGetIssuerCertificateFromStore(
132. hStore,
133. pSubjectContext,
134. NULL,
135. // flags set what we need to check: signature & validity period
136. &dwFlags
137. );
139. **if** (!pCAContext) {
140. Fail("Сертификат ЦС отсутствует в указанном хранилище.");
141. }
143. **return** dwFlags == 0;
144. }
146. /\*
147. \* mode: 0 == sender, !0 == recipient
148. \*/
149. PCCERT\_CONTEXT RequestCert(HCERTSTORE hStoreMy, HCERTSTORE hStoreRoot, **int** mode) {
150. **TCHAR** certName[64];
152. **if** (mode == 0) {
153. printf("\nВведите имя сертификата отправителя: ");
154. } **else** {
155. printf("\nВведите имя сертификата получателя: ");
156. }
158. \_tscanf(\_T("%s"), certName);
160. PCCERT\_CONTEXT pCert = FindCert(hStoreMy, certName);
162. **if** (!pCert) {
163. Fail("Не удалось найти указанный сертификат.");
164. }
166. **if** (!CheckCert(hStoreRoot, pCert)) {
167. Fail("Проверка сертификата завершилась с ошибкой.");
168. }
170. **return** pCert;
171. }
173. /\*
174. \* mode: 0 == input, !0 == output
175. \*/
176. **FILE**\* RequestFile(**int** mode) {
177. **char** fileName[255];
178. **FILE**\* file;
180. **if** (mode == 0) {
181. printf("\nВведите имя входного файла: ");
182. scanf("%s", fileName);
184. file = fopen(fileName, "rb");
185. } **else** {
186. printf("\nВведите имя выходного файла: ");
187. scanf("%s", fileName);
189. file = fopen(fileName, "wb");
190. }
192. **return** file;
193. }
195. **int** GetFileSize(**FILE** \*file) {
196. **int** currentPosition = ftell(file), length = 0;
198. fseek(file, 0, SEEK\_END);
199. length = ftell(file);
200. fseek(file, currentPosition, SEEK\_SET);
202. **return** length;
203. }
205. **int** EncryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
      PCCERT\_CONTEXT pRecipientCert) {
206. **DWORD** messageLength = GetFileSize(inputFile), outputMessageLength;
207. **PBYTE** message = (**PBYTE**)malloc(messageLength), outputMessage;
209. // struct with signature params (+ set cbSize)
210. CRYPT\_SIGN\_MESSAGE\_PARA pSignPara = { **sizeof**(CRYPT\_SIGN\_MESSAGE\_PARA) };
212. pSignPara.dwMsgEncodingType = MY\_ENCODING\_TYPE;
213. pSignPara.pSigningCert = pSenderCert;
215. /\* HashAlgorithm is CRYPT\_ALGORITHM\_IDENTIFIER
216. \* pszObjId means algorithms for signature, hashing & crypting
217. \*/
218. pSignPara.HashAlgorithm.pszObjId = szOID\_RSA\_SHA256RSA;
220. // length of rgpMsgCert
221. pSignPara.cMsgCert = 1;
223. // users' ctxs; can be array
224. pSignPara.rgpMsgCert = &pSenderCert;
226. // params of message crypting
227. CRYPT\_ENCRYPT\_MESSAGE\_PARA pEncryptPara = { **sizeof**(CRYPT\_ENCRYPT\_MESSAGE\_PARA) };
228. pEncryptPara.dwMsgEncodingType = MY\_ENCODING\_TYPE;
230. // cipher algo
231. pEncryptPara.ContentEncryptionAlgorithm.pszObjId = szOID\_NIST\_AES128\_CBC;
233. fread(message, messageLength, 1, inputFile);
234. **int** result = CryptSignAndEncryptMessage(
235. &pSignPara,
236. &pEncryptPara,
237. // length of pRecipientCert
238. 1,
239. &pRecipientCert,
240. message,
241. messageLength,
242. // if null then returns bufferSize
243. NULL,
244. &outputMessageLength
245. );
247. **if** (!result) {
248. Fail("Ошибка при определении размера зашифрованного сообщения.");
249. }
251. outputMessage = (**PBYTE**)malloc(outputMessageLength);
252. result = CryptSignAndEncryptMessage(
253. &pSignPara,
254. &pEncryptPara,
255. 1,
256. &pRecipientCert,
257. message,
258. messageLength,
259. outputMessage,
260. &outputMessageLength
261. );
263. **if** (!result) {
264. **return** 0;
265. }
267. fwrite(outputMessage, outputMessageLength, 1, outputFile);
269. **return** 1;
270. }
272. **int** DecryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
      PCCERT\_CONTEXT pRecipientCert, HCERTSTORE hStore) {
273. **DWORD** messageLength = GetFileSize(inputFile), outputMessageLength;
274. **PBYTE** message = (**PBYTE**)malloc(messageLength), outputMessage;
276. // struct with decrypt params (+ set cbSize)
277. CRYPT\_DECRYPT\_MESSAGE\_PARA pDecryptPara = { **sizeof**(CRYPT\_DECRYPT\_MESSAGE\_PARA) };
279. pDecryptPara.dwMsgAndCertEncodingType = MY\_ENCODING\_TYPE;
280. // can be array
281. pDecryptPara.rghCertStore = &hStore;
282. // length of ^
283. pDecryptPara.cCertStore = 1;
285. CRYPT\_VERIFY\_MESSAGE\_PARA pVerifyPara = { **sizeof**(CRYPT\_VERIFY\_MESSAGE\_PARA) };
286. pVerifyPara.dwMsgAndCertEncodingType = MY\_ENCODING\_TYPE;
288. fread(message, messageLength, 1, inputFile);
289. **int** result = CryptDecryptAndVerifyMessageSignature(
290. &pDecryptPara,
291. &pVerifyPara,
292. // signer index:
293. 0,
294. message,
295. messageLength,
296. // if null then returns bufferSize
297. NULL,
298. &outputMessageLength,
299. &pRecipientCert,
300. &pSenderCert
301. );
303. **if** (!result) {
304. Fail("Ошибка при определении размера исходного сообщения.");
305. }
307. outputMessage = (**PBYTE**)malloc(outputMessageLength);
308. result = CryptDecryptAndVerifyMessageSignature(
309. &pDecryptPara,
310. &pVerifyPara,
311. 0,
312. message,
313. messageLength,
314. outputMessage,
315. &outputMessageLength,
316. &pRecipientCert,
317. &pSenderCert
318. );
320. **if** (!result) {
321. **return** 0;
322. }
324. fwrite(outputMessage, outputMessageLength, 1, outputFile);
326. **return** 1;
327. }

# Примеры работы программы

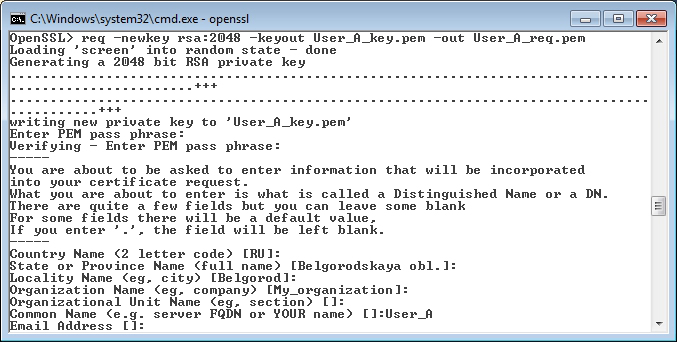


Рисунок 1. Процесс создания ключевой пары RSA и сертификата пользователя в PEM-формате

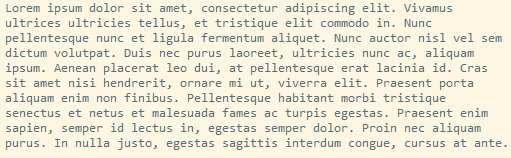


Рисунок 2. Исходный текст

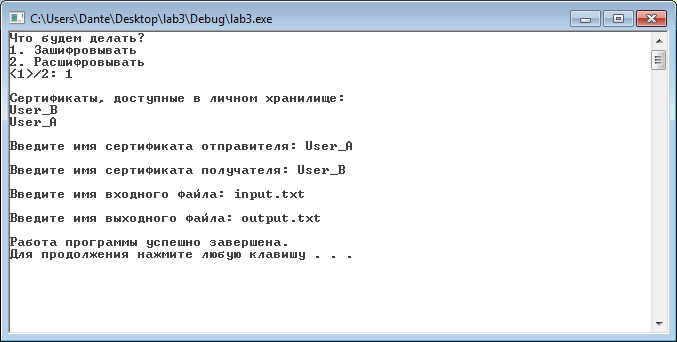


Рисунок 3. Зашифровывание

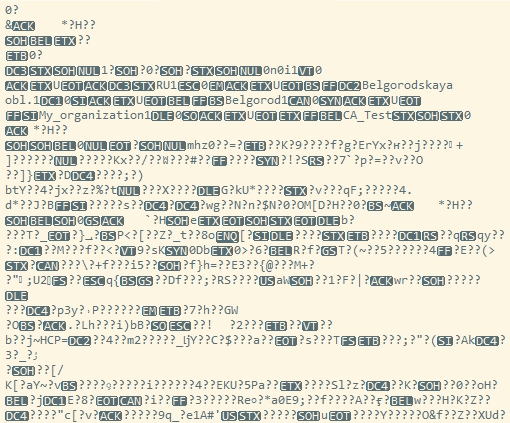


Рисунок 4. Часть шифртекста

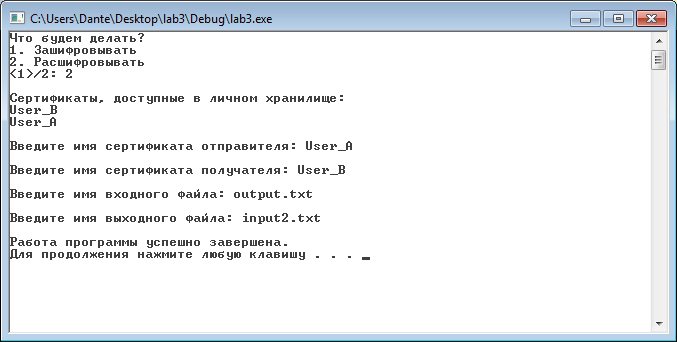


Рисунок 5. Расшифровывание

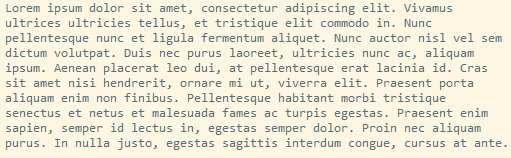


Рисунок 6. Расшифрованный текст