Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №1 по теме: «Симметричное шифрование данных с использованием криптографических интерфейсов Microsoft CryptoAPI и Cryptography API: Next Generation»

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

**Проверил:**  
Смышляев А. Г.

Белгород  
2015

**Цель работы:** ознакомиться с понятием криптопровайдера в интерфейсе CryptoAPI и провайдера алгоритма в интерфейсе Cryptography API: Next Generation, способами подключения к ним, получить навыки выполнения базовых криптографических преобразований: хэширование и симметричного шифрования.

# Задание

Разработать на языке программирования C/C++ консольное или оконное приложение, выполняющее зашифровывание и расшифровывание файла произвольного формата с помощью алгоритмам AES-128, AES-192 или AES-256 по выбору пользователя. Программа должна генерировать сеансовый ключ на основании хэшированного пароля, который запрашивается у пользователя. Использовать хэш-функцию SHA-256.

Приложение должно по выбору пользователя использовать функционал двух криптографических интерфейсов:

* CryptoAPI (в этом случае режим блочного шифрования — устанавливаемый по умолчанию CBC с нулевым вектором инициализации, используемый криптопровайдер — «Microsoft Enhanced RSA and AES Cryptographic Provider»).
* Cryptography API: Next Generation (дополнительно предоставляется выбор режима блочного шифрования: ECB, CBC, CFB, причём в последних двух также используется нулевой вектор инициализации).

# Код программы

lab.h (заголовочный файл программы):

1. #include <atlstr.h>
2. #include <Windows.h>
3. #include <bcrypt.h>
4. #include <WinCrypt.h>
5. #include <locale.h>
6. #include <fcntl.h>
7. #include <io.h>
8. #include <stdio.h>
10. #ifndef STATUS\_SUCCESS
11. #define STATUS\_SUCCESS                  ((NTSTATUS)0x00000000L)
12. #define STATUS\_UNSUCCESSFUL             ((NTSTATUS)0xC0000001L)
13. #endif
15. #define MAX\_STRING\_LENGTH 255
17. **void** encrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode);
18. **void** decrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode);
19. **void** encryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode, **UINT** blockmode);
20. **void** decryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode, **UINT** blockmode);

lab.cpp (основной файл программы):

1. #include "stdafx.h"
2. #include "lab1.h"
4. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
5. {
6. setlocale(LC\_ALL, "RUS");
8. HCRYPTPROV hProv;
10. **if** (! CryptAcquireContext(&hProv, NULL, NULL, PROV\_RSA\_AES, CRYPT\_VERIFYCONTEXT))
11. {
12. \_tprintf(\_T("Ошибка подключения к криптопровайдеру.\n"));
13. system("PAUSE");
14. exit(EXIT\_FAILURE);
15. } **else** {
16. \_tprintf(\_T("Криптопровайдер подключён успешно.\n\n"));
17. }
19. **UINT** crypt;
20. \_tprintf(\_T("Зашифровать или расшифровать?\n"));
21. \_tprintf(\_T("1. Зашифровать\n"));
22. \_tprintf(\_T("2. Расшифровать\n"));
23. \_tprintf(\_T("[<1>/2]: "));
24. \_tscanf(\_T("%u"), &crypt);
26. **if** (crypt < 1 || crypt > 2) {
27. crypt = 1;
28. }
30. **TCHAR** input[MAX\_STRING\_LENGTH], output[MAX\_STRING\_LENGTH];
32. **if** (crypt == 1) {
33. \_tprintf(\_T("Введите название файла, который нужно зашифровать: "));
34. \_tscanf(\_T("%s"), input);
36. \_tprintf(\_T("Введите название для зашифрованного файла: "));
37. \_tscanf(\_T("%s"), output);
38. } **else** {
39. \_tprintf(\_T("Введите название файла, который нужно расшифровать: "));
40. \_tscanf(\_T("%s"), input);
42. \_tprintf(\_T("Введите название для расшифрованного файла: "));
43. \_tscanf(\_T("%s"), output);
44. }
46. **TCHAR** password[MAX\_STRING\_LENGTH];
47. \_tprintf(\_T("Введите пароль: "));
48. \_tscanf(\_T("%s"), password);
50. **UINT** mode;
51. \_tprintf(\_T("Выберите режим шифрования:\n"));
52. \_tprintf(\_T("1. AES-128\n"));
53. \_tprintf(\_T("2. AES-192\n"));
54. \_tprintf(\_T("3. AES-256\n"));
55. \_tprintf(\_T("[<1>/2/3]: "));
56. \_tscanf(\_T("%u"), &mode);
58. **if** (mode < 1 || mode > 3) {
59. mode = 1;
60. }
62. **UINT** crypttype;
63. \_tprintf(\_T("Выберите криптографический интерфейс:\n"));
64. \_tprintf(\_T("1. CryptoAPI\n"));
65. \_tprintf(\_T("2. Cryptography API: Next Generation\n"));
66. \_tprintf(\_T("[<1>/2]: "));
67. \_tscanf(\_T("%u"), &crypttype);
69. **if** (crypttype < 1 || crypttype > 2) {
70. crypttype = 1;
71. }
73. **UINT** blockmode;
74. **if** (crypttype == 2) {
75. \_tprintf(\_T("Выберите режим блочного шифрования:\n"));
76. \_tprintf(\_T("1. ECB\n"));
77. \_tprintf(\_T("2. CBC\n"));
78. \_tprintf(\_T("3. CFB\n"));
79. \_tprintf(\_T("[<1>/2/3]: "));
80. \_tscanf(\_T("%u"), &blockmode);
82. **if** (blockmode < 1 || blockmode > 3) {
83. blockmode = 1;
84. }
85. }
87. **if** (crypt == 1) {
88. **if** (crypttype == 1) {
89. encrypt(hProv, input, output, password, mode);
90. } **else** {
91. encryptng(hProv, input, output, password, mode, blockmode);
92. }
93. } **else** {
94. **if** (crypttype == 1) {
95. decrypt(hProv, output, input, password, mode);
96. } **else** {
97. decryptng(hProv, output, input, password, mode, blockmode);
98. }
99. }
101. CryptReleaseContext(hProv, 0);
103. **return** 0;
104. }
106. **void** encrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
      **UINT** mode) {
107. HCRYPTHASH hHash;
109. **BOOL** createdHash = CryptCreateHash(provider, CALG\_SHA\_256, 0, 0, &hHash);
111. **if** (! createdHash) {
112. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
113. system("PAUSE");
114. exit(EXIT\_FAILURE);
115. }
117. createdHash = CryptHashData(hHash, (**PBYTE**)password,  
      \_tcslen(password)\***sizeof**(**TCHAR**), 0);
119. **if** (! createdHash) {
120. \_tprintf(\_T("Ошибка при генерации хэш-объекта.\n"));
121. system("PAUSE");
122. exit(EXIT\_FAILURE);
123. }
125. ALG\_ID AESType;
127. **switch** (mode) {
128. **case** 1: AESType = CALG\_AES\_128;
129. **break**;
130. **case** 2: AESType = CALG\_AES\_192;
131. **break**;
132. **case** 3: AESType = CALG\_AES\_256;
133. **break**;
134. }
136. HCRYPTKEY hKeyH;
138. createdHash = CryptDeriveKey(provider, AESType, hHash, 0, &hKeyH);
140. **if** (! createdHash) {
141. \_tprintf(\_T("Ошибка создании сессионного ключа.\n"));
142. system("PAUSE");
143. exit(EXIT\_FAILURE);
144. }
146. CryptDestroyHash(hHash);
148. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
150. **if** (input == NULL) {
151. \_tprintf(\_T("Ошибка при открытии файла с текстом.\n"));
152. system("PAUSE");
153. exit(EXIT\_FAILURE);
154. }
156. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
158. **if** (output == NULL) {
159. \_tprintf(\_T("Ошибка при создании файла для шифртекста.\n"));
160. system("PAUSE");
161. exit(EXIT\_FAILURE);
162. }
164. unsigned **long** read, written;
165. **const** unsigned **int** length = 496;
166. **BOOL** eof = **false**;
168. **do** {
169. byte s[length];
171. eof = ! ReadFile(input, s, length, &read, NULL);
172. **if** (eof || read == 0) **break**;
174. **bool** final = read != length;
176. **BOOL** crypted = CryptEncrypt(hKeyH, 0, final, 0, s, &read, read + 16);
178. **if** (! crypted && ! final) {
179. \_tprintf(\_T("Ошибка при зашифровывании файла.\n"));
180. system("PAUSE");
181. exit(EXIT\_FAILURE);
182. }
184. WriteFile(output, s, read, &written, NULL);
186. } **while** (**true**);
188. CloseHandle(input);
189. CloseHandle(output);
191. CryptDestroyHash(hKeyH);
193. \_tprintf(\_T("Файл зашифрован успешно.\n"));
194. system("PAUSE");
195. }
197. **void** decrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
      **UINT** mode) {
198. HCRYPTHASH hHash;
200. **BOOL** createdHash = CryptCreateHash(provider, CALG\_SHA\_256, 0, 0, &hHash);
202. **if** (! createdHash) {
203. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
204. system("PAUSE");
205. exit(EXIT\_FAILURE);
206. }
208. createdHash = CryptHashData(hHash, (**PBYTE**)password,  
      \_tcslen(password)\***sizeof**(**TCHAR**), 0);
210. **if** (! createdHash) {
211. \_tprintf(\_T("Ошибка при генерации хэш-объекта.\n"));
212. system("PAUSE");
213. exit(EXIT\_FAILURE);
214. }
216. ALG\_ID AESType;
218. **switch** (mode) {
219. **case** 1: AESType = CALG\_AES\_128;
220. **break**;
221. **case** 2: AESType = CALG\_AES\_192;
222. **break**;
223. **case** 3: AESType = CALG\_AES\_256;
224. **break**;
225. }
227. HCRYPTKEY hKeyH;
229. createdHash = CryptDeriveKey(provider, AESType, hHash, 0, &hKeyH);
231. **if** (! createdHash) {
232. \_tprintf(\_T("Ошибка при создании сессионного ключа.\n"));
233. system("PAUSE");
234. exit(EXIT\_FAILURE);
235. }
237. CryptDestroyHash(hHash);
239. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
241. **if** (input == NULL) {
242. \_tprintf(\_T("Ошибка при открытии файла с шифртекстом.\n"));
243. system("PAUSE");
244. exit(EXIT\_FAILURE);
245. }
247. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
249. **if** (output == NULL) {
250. \_tprintf(\_T("Ошибка при создании файла с исходным текстом.\n"));
251. exit(EXIT\_FAILURE);
252. }
254. unsigned **long** read, written;
255. **const** unsigned **int** length = 496;
256. **BOOL** eof = **false**;
258. **do** {
259. byte s[length];
261. eof = ! ReadFile(input, s, length, &read, NULL);
262. **if** (eof || read == 0) **break**;
264. **bool** final = read != length;
266. **BOOL** crypted = CryptDecrypt(hKeyH, 0, final, 0, s, &read);
268. **if** (! crypted && ! final) {
269. \_tprintf(\_T("Ошибка при расшифровывании файла.\n"));
270. system("PAUSE");
271. exit(EXIT\_FAILURE);
272. }
274. WriteFile(output, s, read, &written, NULL);
276. } **while** (**true**);
278. CloseHandle(input);
279. CloseHandle(output);
281. CryptDestroyHash(hKeyH);
283. \_tprintf(\_T("Файл успешно расшифрован.\n"));
284. system("PAUSE");
285. }
287. **void** encryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password, **UINT** mode, **UINT** blockmode) {
288. NTSTATUS status = STATUS\_UNSUCCESSFUL;
290. BCRYPT\_ALG\_HANDLE phAlgorithmSHA256;
291. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmSHA256,  
      BCRYPT\_SHA256\_ALGORITHM, NULL, 0))) {
292. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма SHA-256.\n"));
293. system("PAUSE");
294. exit(EXIT\_FAILURE);
295. }
297. **DWORD** cbObject = 0, cbSHA256ObjectLength = 0;
298. **PBYTE** pbSHA256Object = NULL;
299. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_OBJECT\_LENGTH,  
      (**PBYTE**)&cbSHA256ObjectLength, **sizeof**(**DWORD**), &cbObject, 0);
300. pbSHA256Object = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbSHA256ObjectLength);
302. BCRYPT\_HASH\_HANDLE phHashSHA256;
303. **if** (! BCRYPT\_SUCCESS(status = BCryptCreateHash(phAlgorithmSHA256, &phHashSHA256,  
      pbSHA256Object, cbSHA256ObjectLength, NULL, 0, 0))) {
304. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
305. system("PAUSE");
306. exit(EXIT\_FAILURE);
307. }
309. **if** (! BCRYPT\_SUCCESS(status = BCryptHashData(phHashSHA256, (**PBYTE**)password,  
      \_tcslen(password) \* **sizeof**(**TCHAR**), 0))) {
310. \_tprintf(\_T("Ошибка при хэшировании пароля.\n"));
311. system("PAUSE");
312. exit(EXIT\_FAILURE);
313. }
315. **PBYTE** pbHash = NULL;
316. **DWORD** cbData = 0, cbHash = 0;
317. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_HASH\_LENGTH, (**PBYTE**)&cbHash,  
      **sizeof**(**DWORD**), &cbData, 0);
318. pbHash = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbHash);
320. **if** (! BCRYPT\_SUCCESS(status = BCryptFinishHash(phHashSHA256, pbHash, cbHash, 0))) {
321. \_tprintf(\_T("Ошибка при извлечении хэш-кода из хэш-объекта.\n"));
322. system("PAUSE");
323. exit(EXIT\_FAILURE);
324. }
326. BCRYPT\_ALG\_HANDLE phAlgorithmAES;
327. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmAES,  
      BCRYPT\_AES\_ALGORITHM, NULL, 0))) {
328. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
329. system("PAUSE");
330. exit(EXIT\_FAILURE);
331. }
333. **switch** (blockmode) {
334. **case** 1: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_ECB, **sizeof**(BCRYPT\_CHAIN\_MODE\_ECB), 0);
335. **break**;
336. **case** 2: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CBC, **sizeof**(BCRYPT\_CHAIN\_MODE\_CBC), 0);
337. **break**;
338. **case** 3: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CFB, **sizeof**(BCRYPT\_CHAIN\_MODE\_CFB), 0);
339. **break**;
340. }
342. **DWORD** cbAESLength = 0;
343. **PBYTE** pbAESObject = NULL;
344. BCryptGetProperty(phAlgorithmAES, BCRYPT\_OBJECT\_LENGTH, (**PBYTE**)&cbAESLength,  
      **sizeof**(**DWORD**), &cbObject, 0);
345. pbAESObject = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbAESLength);
347. **ULONG** AESType;
349. **switch** (mode) {
350. **case** 1: AESType = 16;
351. **break**;
352. **case** 2: AESType = 24;
353. **break**;
354. **case** 3: AESType = 32;
355. **break**;
356. }
358. BCRYPT\_KEY\_HANDLE phKey;
359. **if** (! BCRYPT\_SUCCESS(status = BCryptGenerateSymmetricKey(phAlgorithmAES, &phKey,  
      pbAESObject, cbAESLength, (**PBYTE**)pbHash, AESType, 0))) {
360. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
361. system("PAUSE");
362. exit(EXIT\_FAILURE);
363. }
365. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
367. **if** (input == NULL) {
368. \_tprintf(\_T("Ошибка при открытии файла с исходным текстом.\n"));
369. system("PAUSE");
370. exit(EXIT\_FAILURE);
371. }
373. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
375. **if** (output == NULL) {
376. \_tprintf(\_T("Ошибка при создании файла с шифртекстом.\n"));
377. system("PAUSE");
378. exit(EXIT\_FAILURE);
379. }
381. unsigned **long** read, written;
382. **const** unsigned **int** length = 496;
383. **bool** eof = **false**;
385. **do** {
386. byte s[length];
388. eof = ! ReadFile(input, s, length, &read, NULL);
389. **if** (eof || read == 0) **break**;
391. **ULONG** dwFlags = 0;
393. **if** (read != length) {
394. dwFlags = BCRYPT\_BLOCK\_PADDING;
395. }
397. **DWORD** cbOutput = 0, cbCipherText = 0;
398. **PBYTE** pbOutput = NULL;
399. **if** (! BCRYPT\_SUCCESS(status = BCryptEncrypt(phKey, s, read, NULL, NULL, 0, NULL,  
       0, &cbOutput, dwFlags)) && read == length) {
400. \_tprintf(\_T("Ошибка при вычислении размера шифркода.\n"));
401. system("PAUSE");
402. exit(EXIT\_FAILURE);
403. }
405. pbOutput = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbOutput);
407. **if** (! BCRYPT\_SUCCESS(status = BCryptEncrypt(phKey, s, read, NULL, NULL, 0,  
      pbOutput, cbOutput, &cbCipherText, dwFlags)) && read == length) {
408. \_tprintf(\_T("Ошибка при зашифровывании файла.\n"));
409. system("PAUSE");
410. exit(EXIT\_FAILURE);
411. }
413. WriteFile(output, pbOutput, cbCipherText, &written, NULL);
415. HeapFree(GetProcessHeap(), 0, pbOutput);
417. } **while** (**true**);
419. CloseHandle(input);
420. CloseHandle(output);
422. BCryptDestroyHash(phHashSHA256);
423. BCryptDestroyKey(phKey);
424. BCryptCloseAlgorithmProvider(phAlgorithmAES, 0);
425. BCryptCloseAlgorithmProvider(phAlgorithmSHA256, 0);
427. HeapFree(GetProcessHeap(), 0, pbSHA256Object);
428. HeapFree(GetProcessHeap(), 0, pbAESObject);
429. HeapFree(GetProcessHeap(), 0, pbHash);
431. \_tprintf(\_T("Файл успешно зашифрован.\n"));
432. system("PAUSE");
433. }
435. **void** decryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password, **UINT** mode, **UINT** blockmode) {
436. NTSTATUS status = STATUS\_UNSUCCESSFUL;
438. BCRYPT\_ALG\_HANDLE phAlgorithmSHA256;
439. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmSHA256,  
      BCRYPT\_SHA256\_ALGORITHM, NULL, 0))) {
440. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма SHA-256.\n"));
441. system("PAUSE");
442. exit(EXIT\_FAILURE);
443. }
445. **DWORD** cbObject = 0, cbSHA256ObjectLength = 0;
446. **PBYTE** pbSHA256Object = NULL;
447. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_OBJECT\_LENGTH,  
      (**PBYTE**)&cbSHA256ObjectLength, **sizeof**(**DWORD**), &cbObject, 0);
448. pbSHA256Object = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbSHA256ObjectLength);
450. BCRYPT\_HASH\_HANDLE phHashSHA256;
451. **if** (! BCRYPT\_SUCCESS(status = BCryptCreateHash(phAlgorithmSHA256, &phHashSHA256,  
      pbSHA256Object, cbSHA256ObjectLength, NULL, 0, 0))) {
452. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
453. system("PAUSE");
454. exit(EXIT\_FAILURE);
455. }
457. **if** (! BCRYPT\_SUCCESS(status = BCryptHashData(phHashSHA256, (**PBYTE**)password,  
      \_tcslen(password) \* **sizeof**(**TCHAR**), 0))) {
458. \_tprintf(\_T("Ошибка при хэшировании пароля.\n"));
459. system("PAUSE");
460. exit(EXIT\_FAILURE);
461. }
463. **PBYTE** pbHash = NULL;
464. **DWORD** cbData = 0, cbHash = 0;
465. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_HASH\_LENGTH, (**PBYTE**)&cbHash,  
      **sizeof**(**DWORD**), &cbData, 0);
466. pbHash = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbHash);
468. **if** (! BCRYPT\_SUCCESS(status = BCryptFinishHash(phHashSHA256, pbHash, cbHash, 0))) {
469. \_tprintf(\_T("Ошибка при извлечении хэш-кода из хэш-объекта.\n"));
470. system("PAUSE");
471. exit(EXIT\_FAILURE);
472. }
474. BCRYPT\_ALG\_HANDLE phAlgorithmAES;
475. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmAES,  
      BCRYPT\_AES\_ALGORITHM, NULL, 0))) {
476. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
477. system("PAUSE");
478. exit(EXIT\_FAILURE);
479. }
481. **switch** (blockmode) {
482. **case** 1: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_ECB, **sizeof**(BCRYPT\_CHAIN\_MODE\_ECB), 0);
483. **break**;
484. **case** 2: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,   
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CBC, **sizeof**(BCRYPT\_CHAIN\_MODE\_CBC), 0);
485. **break**;
486. **case** 3: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CFB, **sizeof**(BCRYPT\_CHAIN\_MODE\_CFB), 0);
487. **break**;
488. }
490. **DWORD** cbAESLength = 0;
491. **PBYTE** pbAESObject = NULL;
492. BCryptGetProperty(phAlgorithmAES, BCRYPT\_OBJECT\_LENGTH, (**PBYTE**)&cbAESLength,  
      **sizeof**(**DWORD**), &cbObject, 0);
493. pbAESObject = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbAESLength);
495. **ULONG** AESType;
497. **switch** (mode) {
498. **case** 1: AESType = 16;
499. **break**;
500. **case** 2: AESType = 24;
501. **break**;
502. **case** 3: AESType = 32;
503. **break**;
504. }
506. BCRYPT\_KEY\_HANDLE phKey;
507. **if** (! BCRYPT\_SUCCESS(status = BCryptGenerateSymmetricKey(phAlgorithmAES, &phKey,  
      pbAESObject, cbAESLength, (**PBYTE**)pbHash, AESType, 0))) {
508. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
509. system("PAUSE");
510. exit(EXIT\_FAILURE);
511. }
513. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
515. **if** (input == NULL) {
516. \_tprintf(\_T("Ошибка при открытии файла с шифртекстом.\n"));
517. system("PAUSE");
518. exit(EXIT\_FAILURE);
519. }
521. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
523. **if** (output == NULL) {
524. \_tprintf(\_T("Ошибка при создании файла с исходным текстом.\n"));
525. system("PAUSE");
526. exit(EXIT\_FAILURE);
527. }
529. unsigned **long** read, written;
530. **const** unsigned **int** length = 496;
531. **bool** eof = **false**;
533. **do** {
534. byte s[length];
536. eof = ! ReadFile(input, s, length, &read, NULL);
537. **if** (eof || read == 0) **break**;
539. **ULONG** dwFlags = 0;
541. **if** (read != length) {
542. dwFlags = BCRYPT\_BLOCK\_PADDING;
543. }
545. **DWORD** cbOutput = 0, cbCipherText = 0;
546. **PBYTE** pbOutput = NULL;
547. **if** (! BCRYPT\_SUCCESS(status = BCryptDecrypt(phKey, s, read, NULL, NULL, 0, NULL,   
      0, &cbCipherText, dwFlags)) && read == length) {
548. \_tprintf(\_T("Ошибка при вычислении размера исходного текста.\n"));
549. system("PAUSE");
550. exit(EXIT\_FAILURE);
551. }
553. pbOutput = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbCipherText);
555. **if** (! BCRYPT\_SUCCESS(status = BCryptDecrypt(phKey, s, read, NULL, NULL, 0,  
      pbOutput, cbCipherText, &cbOutput, dwFlags)) && read == length) {
556. \_tprintf(\_T("Ошибка при расшифрованиии файла с шифртекстом.\n"));
557. system("PAUSE");
558. exit(EXIT\_FAILURE);
559. }
561. WriteFile(output, pbOutput, cbOutput, &written, NULL);
563. HeapFree(GetProcessHeap(), 0, pbOutput);
565. } **while** (**true**);
567. CloseHandle(input);
568. CloseHandle(output);
570. BCryptDestroyHash(phHashSHA256);
571. BCryptDestroyKey(phKey);
572. BCryptCloseAlgorithmProvider(phAlgorithmAES, 0);
573. BCryptCloseAlgorithmProvider(phAlgorithmSHA256, 0);
575. HeapFree(GetProcessHeap(), 0, pbSHA256Object);
576. HeapFree(GetProcessHeap(), 0, pbAESObject);
577. HeapFree(GetProcessHeap(), 0, pbHash);
579. \_tprintf(\_T("Файл успешно расшифрован.\n"));
580. system("PAUSE");
581. }

# Примеры работы программы

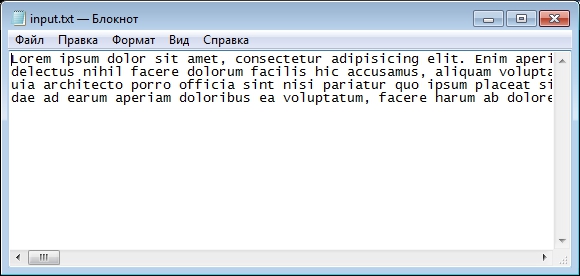


Рисунок 1. Файл с исходным текстом

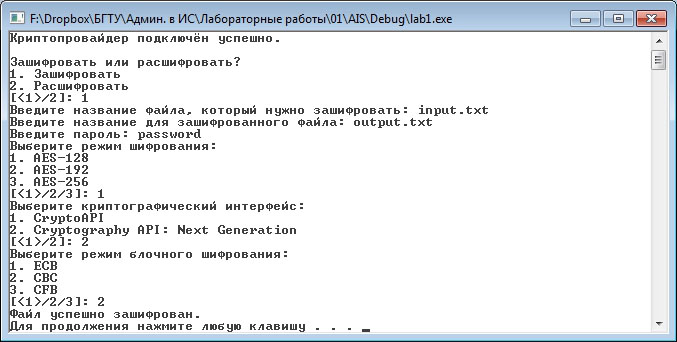


Рисунок 2. Процесс зашифровывания файла

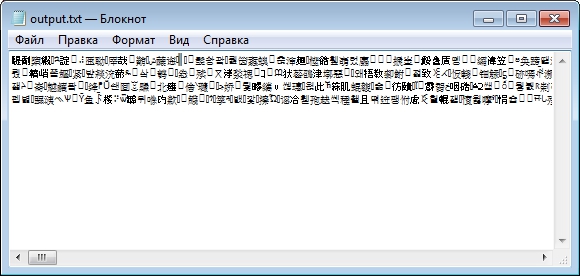


Рисунок 3. Зашифрованный файл

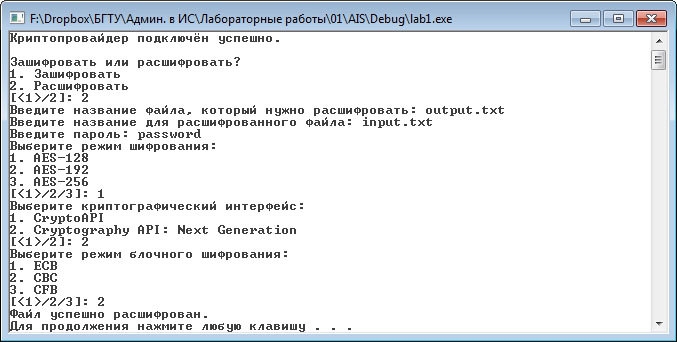


Рисунок 4. Процесс расшифровывания файла