```
(base) andre@DESKTOP-UM1B7BM:/mnt/c/Users/andre/OneDrive/Systems/lab10-1/lab10$ ./prog1
asd
Cipher text:
0000 - 21 c9 26 7c c5 69 b7 33-40 d5 97 2e 45 78 49 fa   !.&|.i.3@...ExI.
(base) andre@DESKTOP-UM1B7BM:/mnt/c/Users/andre/OneDrive/Systems/lab10-1/lab10$ ./prog2
Plain text: asd
(base) andre@DESKTOP-UM1B7BM:/mnt/c/Users/andre/OneDrive/Systems/lab10-1/lab10$
```

```
(base) andre@DESKTOP-UM1B7BM:/mnt/c/Users/andre/OneDrive/Systems/lab10$ ./prog1
asd
Cipher text:
0000 - c2 ba 41 2b 07 cb b0 a8-83 19 10 30 02 d9 eb c7   ..A+.......0....
(base) andre@DESKTOP-UM1B7BM:/mnt/c/Users/andre/OneDrive/Systems/lab10$ ./prog2
Plain text: asd
(base) andre@DESKTOP-UM1B7BM:/mnt/c/Users/andre/OneDrive/Systems/lab10$
```

```c
/*****************************************
*
*                prog1.c
*
*  Example of the use of OpenSSL to encode a
*  message that is stored on a file.
*
*  Encrypt a message that is read from standard
*  input.  The encrypted message is written on
*  the file message and they key information is
*  written on the file secret.
*
*****************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/random.h>
#include <stdio.h>
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <unistd.h>
```

```c
/*
 *  Struct to store the key and the initialization vector
 */
struct secretStruct {
   unsigned char key[32];
   unsigned char iv[16];
} secret;

/*
 *  Simple error reporting procedure
 */
void handleErrors() {
   ERR_print_errors_fp(stderr);
   abort();
}

/*
 *  Create a key and initialization vector using the
 *  kernel random number generator.  The results are
 *  stored on the file secret.
 */
void make_key() {
   int fout;
   ssize_t ret;

   fout = open("secret", O_WRONLY | O_CREAT | O_TRUNC, 0600);
   ret = getrandom(&secret.key, 32, 0);
   if(ret != 32) {
       printf("random key generation failed\n");
       abort();
   }
   ret = getrandom(&secret.iv, 16, 0);
   if(ret != 16) {
       printf("intialization vector generation failed\n");
       abort();
   }
   write(fout, &secret, sizeof(secret));
   close(fout);
}
```

```
 /*
  *  This procedure encrypts a message using the AES-CBC encryption
  *  method.  The first parameters is the plain text to be encrypted.
  *  The second parameter is the length of the plain text.  The third
  *  parameter is a pointer to a block of memory where the cipher text
  *  will be returned
  */
 int encrypt(unsigned char *plaintext, int length, unsigned char
*ciphertext) {
    int len;
    int ciphertext_len;
    EVP_CIPHER_CTX *ctx;

    /*
     *  Start by creating a new cipher context
     */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /*
     *  Initialize the encryption process.  The second parameter
     *  is the encryption algorithms to be used.  The third
     *  parameter is set to NULL to indicate that the default
     *  implementation of this algorithm will be used.
     */
    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, secret.key,
secret.iv))
        handleErrors();

    /*
     *  Start encrypting the message, place it in ciphertext
     */
    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, length))
        handleErrors();

    ciphertext_len = len;

    /*
     *  Finalize the encryption.  Finish off the last block
```

```
     *  of encrypted text.
     */
    if(1 != EVP_EncryptFinal_ex(ctx, ciphertext+len, &len))
        handleErrors();

    ciphertext_len += len;

    /*
     *  Free up the storage used by the cipher context
     */
    EVP_CIPHER_CTX_free(ctx);

    return(ciphertext_len);

}

int main(int argc, char **argv) {
    unsigned char plain[256];
    unsigned char *cipher;
    int len;
    int n;
    int fout;

    make_key();

    /*
     *  Read the plain text from standard input
     */
    bzero(&plain, 256);
    fgets((char *)plain, 256, stdin);
    n = strlen((char *)plain);
    // Remove the \n at the end of the line
    plain[n-1] = 0;

    /*
     *  Compute the size of the cipher text
     *  and allocate memory for it.  It must
     *  be a block size.
     */
    n = (n/32+1)*32;
```

```
    cipher = (unsigned char *) malloc(n);

    /*
     *  Do the encryption, print the result and
     *  save it to the message file.  Rememeber to
     *  include the \0 in the message size.
     */
    len = encrypt((unsigned char *)&plain, strlen((char *)plain)+1,
cipher);

    printf("Cipher text: \n");
    BIO_dump_fp(stdout, (const char *) cipher, len);

    fout = open("message", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    n = write(fout, cipher, len);
    close(fout);

 }

/*****************************************
 *
 *             prog2
 *
 *  This program reads the secret key information
 *  from the secret file and uses it to decode
 *  the message in the message file.
 *
 *****************************************/

 #include <sys/types.h>
 #include <sys/stat.h>
 #include <fcntl.h>
 #include <sys/random.h>
 #include <stdio.h>
 #include <openssl/conf.h>
 #include <openssl/evp.h>
 #include <openssl/err.h>
 #include <string.h>
 #include <unistd.h>
```

```
/*
 *  Structure to store the key and the initialization vector
 */
 struct secretStruct {
   unsigned char key[32];
   unsigned char iv[16];
} secret;

/*
 * Standard error handling procedure.
 */
void handleErrors() {
   ERR_print_errors_fp(stderr);
   abort();
}

/*
 *  Read the key and initialization vector from the secret file
 */
void read_key() {
   int fin;

   fin = open("secret", O_RDONLY, 0600);
   read(fin, &secret, sizeof(secret));
   close(fin);

}

/*
 *  Procedure that decodes the encrypted message.  The first parameter
 *  is the cipher text.  The second parameter is the length of the
 *  cipher text.  The third parameter is a pointer to a block of
 *  memory where the plain text will be stored.
 */
int decode(unsigned char *cipher, int length, unsigned char *plain) {
   EVP_CIPHER_CTX *ctx;
   int len;
   int plaintext_len;

   /*
```

```
     *  Create a context for the decrypt operation
     */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /*
     *  Initialize the decryption operation.  The second parameter is
     *  the decryption algorithm to be used.  The NULL third parameter
     *  specifies that the default implementation for this algorithm
     *  will be used.
     */
    if(1 != EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, secret.key,
secret.iv))
        handleErrors();

    /*
     *  Start descrypting the message.
     */
    if(1 != EVP_DecryptUpdate(ctx, plain, &len, cipher, length))
        handleErrors();

    plaintext_len = len;

    /*
     *  Finish decrypting the message, make sure that the last
     *  block of the message is processed.
     */
    if(1 != EVP_DecryptFinal_ex(ctx, plain+len, &len))
        handleErrors();

    plaintext_len += len;

    /*
     *  Free the memory used by the cipher context.
     */
    EVP_CIPHER_CTX_free(ctx);

    return(plaintext_len);

 }
```

```c
int main(int argc, char ** argv) {
   unsigned char buffer[512];
   unsigned char *plain;
   int len;
   int n;
   int fin;

   read_key();

   /*
    *  Read the encrypted message from the message file.
    */
   fin = open("message", O_RDONLY, 0644);
   n = read(fin, buffer, 512);
   close(fin);

   /*
    *  Allocate memory for the plain text based on the size
    *  of the encrypted message.
    */
   plain = (unsigned char *) malloc(n);

   /*
    *  Decode the message and print the result
    */
   len = decode(buffer, n, plain);

   printf("Plain text: %s\n",plain);

}
```