```
/*
RC4 is a stream cipher.
That means RC4 produces a stream of bits that are xor'ed
with the bits in the message.
In a stream cipher it is essential that the same key never be used twice.
If Sam the skilled snoop ever encounters two different messages
encrypted with the same RC4 key, then Sam can xor their cipher text
to eliminate the RC4 cipher bits
because xor repeated twice is a null operation.

If Sam knows the plaintext of one of the messages,
he can easily recover the other.
Worse, if the messages are in a natural language such as English,
he can usually recover both messages -- even if he knew neither.

While revelation of encoded messages is bad news for any cipher,
in neither case is Sam able to recover the user key.

CipherSaber addresses the requirement that the same key
never be used twice by appending a ten byte quantity,
called the initialization vector (IV), to the users key.
If the same IV is never used twice then the same RC4 key can never be used
 twice.


  What follows is some of the code used in the encryption decryption process
  Structures are used for global variables because structures can be
  easily zeroed at convenient points to destroy all traces of user inputs

  There should be enough code here for the user to check that the encryption is secure



#include "stdafx.h"
#include "crypt.h"
#include "resource.h"

struct GlobalVar
{
    HWND hWnd;
    HWND hWnd1;
    HWND hDlg;
    HWND hDlg2;

    HINSTANCE   MyInstance;
    HINSTANCE   ProcessInstance;
    unsigned int i;
    int nSel;
    char tstring[MAX_PATH21];
    char astring[MAX_PATH21];
    char line[MAX_PATH21];
    char confirm[20];
    int NoBars; //  used for progress bar if > 32k
    int BarRemaining;
    int TotalBars;
    int count;
};

struct LocalVar
{
    char *c, *d;
    unsigned char *a, *b;
    unsigned int i,j, k, nRet;
    unsigned int progress;
    unsigned long l;
    unsigned long ck, ci, ts;
    unsigned int min;
    __int64 tsize;
    __int64 fsize;
    __int64 fsizeslack;
    double f, fy;
    char password[70];
    char password2[70];
```

```
    char passworda[70];
    char password2a[70];
    char password3[60];
    char filename[MAX_PATH21];
    unsigned int FileSize;          //  unsigned long
    int passwordtitletype;
    int DOUBLECLICK;
    BOOL ENCRYPT;
    BOOL FIRSTTIME;
    BOOL SELECTPASSWORD;
    BOOL COPYPASSWORD;
    BOOL PRINTPASSWORD;
    BOOL CONFIRMHASH;
    BOOL SHOWPASSWORD;
    BOOL IDENTITY;
    BOOL LOADPASSWORD;
    BOOL CONFIRMEMAIL;
    int SAVEPASSWORD;
    BOOL NEWPASSWORD;
    BOOL EDIT1;
    BOOL DELETE1;
    BOOL FILEMAKEBARSTEP;
    BOOL HASHMAKEBARSTEP;
    BOOL SAVEMESSAGE;
    BOOL LARGEFILE;
    BOOL PUTPASSWORDONCLIPBOARD;
    BOOL SHA1;
    int ADDPASSWORD;
    int SHADBASE;
    BOOL COPYALL;
    short MOD;
    struct passwordlist list;
    unsigned int NoPasswords;
    unsigned int time1, time2, timecount, process;
    BOOL HIDESHOWDISKSIZE;
    BOOL SHOWSELECTION;
    BOOL HIDEDIR2;
    BOOL HAVEROOM;
    BOOL INDEX;
    int Logtype;
    char Logstring[100];
    BOOL FAT32LOCK;
    char *AboutData;
    HANDLE hAboutData;
    SYSTEMTIME  ST;
};

struct FileVar
{
    unsigned long fsize, fsize2;
    unsigned long fsizein, fsizeout;
    unsigned long AmountRead;
    unsigned long AmountWrite;
    unsigned long FileRead;
    unsigned long FileWrite;
    __int64 TotalFileSize;
    unsigned int fsizesig;
    __int64 TotalFileSize1;
    __int64 TotalFileSize2;
    unsigned int ClusterSize;
    long position;
    long ltemp;
    char BufferJee[200];
    unsigned int count1;
    unsigned int count2;
    unsigned int hfSHAsize;
    int i, j, k, l;
    char *a, *b;
    HANDLE hFile, hFile1, hFileNew;
    HANDLE hGmem, hGmem1, hGmem2, hGmem3, hGmem4;   //, hGmemTable;
    char *PathAddress;
    WIN32_FIND_DATA *fd;
    unsigned long GmemSize, Gmem2Size, Gmem3Size;   //, Gmem4Size, CryptSize;
```

```
    unsigned long PasswordSize, PasswordSize2;
    unsigned char   *hpAddress, *hpAddress2, *hpAddress3, *hpAddress4, *h1, *h2, *h3, *h4;
    char dirname1[MAX_PATH21];
    char dirname2[MAX_PATH21];
    char fdirname1[MAX_PATH21];
    char fdirname2[MAX_PATH21];
    char fdirname3[MAX_PATH21];
    int DoubleClickCheck;
    char deletefile[MAX_PATH21];
    char passwordfile[MAX_PATH21];
    BOOL bRet;
    HANDLE hFind;
    int FULLNAME;
};

struct fileSHA
{
    char SHA[20];
};

extern HWND hProgBar;
extern HWND hDlgModeless;

#ifndef DIFF
#define DIFF 1  //  3
#endif DIFF

struct GlobalVar Gvar;
struct LocalVar lVar;
struct FileVar Fvar;
extern DWORD dwPlatformId;  //  this determines what operating system present
extern char *errstring;     //  "Not Enough Memory for Program"
extern char *CannotgetHashofencryptedfile;
extern char *Hashoforiginaldecomfilediff;
extern char *YoucannotuseinPassword;
extern  char *String;
extern unsigned int *IndexF;
extern HANDLE hMem;


struct passwordlist *PasswordList;
struct PasswordDetails
{
    unsigned int PasswordNo;
    int EditBoxSelection;
    unsigned int temp;
    unsigned int i, j;
    HWND hDlg;
    HWND focus;
    HWND editbox;
    HWND box1;
    HWND box2;
    HWND box3;
    HHOOK hookdata;
    LPMOUSEHOOKSTRUCT lpMouseHookStruct;
    HHOOK hookmousedata;
    //WORD position;
    DWORD cursor;
    DWORD cursorold1;
    DWORD cursorold2;
    DWORD cursorold3;
    WORD positionLow;
    WORD positionHigh;
    WORD oldpositionLow;
    WORD oldpositionHigh;
};
struct PasswordDetails *Pass;
HANDLE hPass;

BOOL FAR PASCAL encrypt(unsigned char *fileBuffer,
    unsigned long fileBufferLength)
{
```

```
    lVar.progress = 0;
    Initialise10ByteVector(Vector10Byte);
    RemoveHpAddress();
    //  the fileBuffer is a compressed data stream in zip format
    //  data is always compressed before it reaches encrypt function
    ProcessEncrypt(fileBuffer, fileBufferLength, lVar.password);

    //  added encrypt ver 5.99
    AES(fileBuffer, fileBufferLength, lVar.password3, TRUE);

    ProcessEncrypt(fileBuffer, fileBufferLength, lVar.password2);



    //  it now has to be hashed
    ProgressBarMessage("getting Digital Signature of Encrypted Data", "");
    //  SHA0 used here
    if (!SHA(Fvar.BufferJee, fileBuffer, fileBufferLength))
    {
        ErrBox(CannotgetHashofencryptedfile);   //  "Cannot get Hash of encrypted file");
        return FALSE;
    }

    //  we now need to add this to end of file
    //  add vector10Byte first
    //  then add SHA0

    //  vector10Byte
    lVar.a = &fileBuffer[fileBufferLength];
    lVar.b = Vector10Byte;
    for (lVar.i=0; lVar.i < 10; lVar.i++)
    {
        *lVar.a = *lVar.b;
        lVar.a++;
        lVar.b++;
    }

    //  SHA
    lVar.b = (unsigned char *) Fvar.BufferJee;

    for (lVar.i=0; lVar.i < sizeof(long)*5; lVar.i++)
    {
        *lVar.a = *lVar.b;
        lVar.a++;
        lVar.b++;
    }

    Fvar.AmountWrite +=  (unsigned long) (sizeof(long)*5+10);
    return TRUE;

}

BOOL FAR PASCAL ProcessEncrypt(unsigned char *fileBuffer,
    unsigned long fileBufferLength, char *passphrase)
{
    //  now ready for encryption
    //  will encrypt with lVar.password[0] == 0
    if (*passphrase == 0 && passphrase == lVar.password2)
    {
        return FALSE;
    }

    AppendToPassphrase(passphrase,
        Vector10Byte, key);
    memset(passphrase, 0, strlen(passphrase));  //  KEYLENGTH+10
    DoBlowfishEncrypt(fileBuffer,
        fileBufferLength, TRUE);

    InitialiseStateArray(state,
        key);
    memset(key, 0, sizeof(KEYLENGTH+10));
    ProgressBarMessage("XOR Encrypt", "");
    rc4(fileBuffer,
```

```
        fileBufferLength, state, TRUE);
    //  file has now been encrypted

    //  further XOR to disguise chars before mixing
    //  we are using pkzip with header to increase size of smaller files
    //  to aid mixing
    //  to destroy known header and prevent XOR with
    //  other text with same password
    //  XOR known text with unknown to hide values so they
    //  cannot be known
    //  this prevents reconstrututing of mix from knowing a value
    ProgressBarMessage("XORing neighbour", "");

    //  with zip we have a known header
    //  we have a known header which provides a method of attack
    //  if we XOR from the end of the file to the beginning
    //  we will end up with the header not known
    XORfromTop(fileBuffer, fileBufferLength);

    ProgressBarMessage("Position Mix", "");
    rc4Mix(fileBuffer,
        fileBufferLength, TRUE);

    memset(state, 0, sizeof(state));
  return TRUE;

}

void FAR PASCAL rc4Mix(unsigned char *fileBuffer,
    unsigned long fileBufferLength, BOOL ENCRYPT)
{
    lVar.nRet = 2;
    if (fileBufferLength < 50000) lVar.nRet = 5;
    if (fileBufferLength < 10000) lVar.nRet = 10;
    if (fileBufferLength < 500) lVar.nRet = 20;
    lVar.progress = 0;
    lVar.k = 0;

    unsigned char *a, *b;
    a = fileBuffer;
    b = fileBuffer;

    if (ENCRYPT)
    {
        for (lVar.i=0; lVar.i<lVar.nRet; lVar.i++)
        {
            lVar.MOD = 2;
            for (lVar.l = 3; lVar.l < fileBufferLength-1; lVar.l++)
            {
                //  range 2 to fileBufferLength-2
                unsigned char temp;
                lVar.MOD++;
                if (lVar.MOD == 256) lVar.MOD = 0;
            //  lVar.j = (lVar.l+1) % 256;
                lVar.j = lVar.MOD;
            //  if (MOD != lVar.j)
            //  {
            //      lVar.progress++;
            //  }

                //  lVar.k = (state[lVar.j] + lVar.k) % 256;
                lVar.k = (state[lVar.j] + lVar.k);
                //  lVar.k = 255 && state[lVar.j] = 255
                if (lVar.k >= 256)
                {
                    lVar.k -= 256;
                    if (lVar.k >= 256) lVar.k -= 256;
                }


                lVar.ck = (lVar.k + lVar.l);

                lVar.ci = lVar.l;
```

```
            //  we are going to swap array aroung
            if (lVar.ci > fileBufferLength - DIFF ||
                lVar.ck > fileBufferLength - DIFF)
            {
                //  number of chars in filebuffer > 4
                //  this takes chars at end of file
                //  and places them at beginning
                //  thus the end of file may swap with beginning of file

                if (lVar.ci > fileBufferLength - DIFF)
                {
                    while (lVar.ci > fileBufferLength - DIFF)
                    {
                        lVar.ci -= (fileBufferLength-DIFF);
                    }
                    if (lVar.ci < 3) lVar.ci = 3;

                }
                if (lVar.ck > fileBufferLength - DIFF)
                {
                    while (lVar.ck > fileBufferLength - DIFF)
                    {
                        lVar.ck -= (fileBufferLength-DIFF);
                    }
                }
                if (lVar.ck < 3) lVar.ck = 3;
            }

            temp = *(fileBuffer+lVar.ck);
            *(fileBuffer+lVar.ck) = *(fileBuffer+lVar.ci);
            *(fileBuffer+lVar.ci) = temp;

            temp = state[lVar.j];
            state[lVar.j] = state[lVar.k];
            state[lVar.k] = temp;
            lVar.progress ++;
            if (lVar.progress > PROGRESSCOUNTMAX)
            {
                SendMessage(hProgBar, PBM_STEPIT, 0, 0);
                lVar.progress = 0;
            }

        }
    }
}
else
{

    Fvar.h3 = Fvar.hpAddress3;
    for (lVar.i=0; lVar.i<lVar.nRet; lVar.i++)
    {
        lVar.MOD = 2;
        for (lVar.l = 3; lVar.l < fileBufferLength-1; lVar.l++)
        {
            //  range 2 to fileBufferLength-2
            unsigned char temp;
            lVar.MOD++;
            if (lVar.MOD == 256) lVar.MOD = 0;

            lVar.j = lVar.MOD;

            lVar.k = (state[lVar.j] + lVar.k);

            if (lVar.k >= 256)
            {
                lVar.k -= 256;
                if (lVar.k >= 256) lVar.k -= 256;
            }
            *Fvar.h3 = lVar.k;
            Fvar.h3++;
```

```
                temp = state[lVar.j];
                state[lVar.j] = state[lVar.k];
                state[lVar.k] = temp;
                lVar.progress ++;
                if (lVar.progress > PROGRESSCOUNTMAX)
                {
                    SendMessage(hProgBar, PBM_STEPIT, 0, 0);
                    lVar.progress = 0;
                }

            }
        }

        Fvar.h3--;
        for (lVar.i=0; lVar.i<lVar.nRet; lVar.i++)
        {

            for (lVar.l = fileBufferLength-2; lVar.l > 2; lVar.l--)
            {
                //  range 2 to fileBufferLength-2
                unsigned char temp;

                lVar.ck = (*Fvar.h3 + lVar.l);

                lVar.ci = lVar.l;

                //  we are going to swap array aroung
                if (lVar.ci > fileBufferLength - DIFF ||
                    lVar.ck > fileBufferLength - DIFF)
                {
                    if (lVar.ci > fileBufferLength - DIFF)
                    {
                        while (lVar.ci > fileBufferLength - DIFF)
                        {
                            lVar.ci -= fileBufferLength-DIFF;
                        }
                        if (lVar.ci < 3) lVar.ci = 3;

                    }
                    if (lVar.ck > fileBufferLength - DIFF)
                    {
                        while (lVar.ck > fileBufferLength - DIFF)
                        {
                            lVar.ck -= fileBufferLength-DIFF;
                        }
                    }
                    if (lVar.ck < 3) lVar.ck = 3;

                }

                temp = *(fileBuffer+lVar.ck);
                *(fileBuffer+lVar.ck) = *(fileBuffer+lVar.ci);
                *(fileBuffer+lVar.ci) = temp;
                lVar.progress ++;
                if (lVar.progress > PROGRESSCOUNTMAX)
                {
                    SendMessage(hProgBar, PBM_STEPIT, 0, 0);
                    lVar.progress = 0;
                }
                *Fvar.h3 = '\0';    //  clearmemory
                Fvar.h3--;
            }
        }
    }
}

void FAR PASCAL Initialise10ByteVector(unsigned char *Vector10Byte)
{
    GetSHAforInitialise10ByteVector();
    //  this function takes and uses Gvar.astring & Gvar.tstring
    //  SHA in Gvar.astring
    //  we have 20 chars we now need to cut this down to ten
    //  XOR two chars will reduce 2 down to one and be unique
```

```
    lVar.c = Gvar.astring;
    lVar.d = &Gvar.astring[1];
    for (lVar.i = 0; lVar.i < 10; lVar.i ++)
    {
        *lVar.c ^= *lVar.d;
        Vector10Byte[lVar.i] = *lVar.c;
        lVar.c += 2;
        lVar.d += 2;
    }
}


void FAR PASCAL GetSHAforInitialise10ByteVector()
{
    //  If encrypting, generate 10 random numbers (0-255 range).
    //  Write these as the first 10 bytes of the output file

    //  GetTickCount function retrieves the number of milliseconds
    //  that have elapsed since Windows was started.

    OSVERSIONINFO ov;

    HWND h;
    h = GetActiveWindow();
    SYSTEMTIME  ST;
    GetSystemTime(&ST);
    MEMORYSTATUS MS;
    GlobalMemoryStatus(&MS);
    POINT P;
    GetCursorPos(&P);
    DWORD time2, l;
    DWORD z;
    z = GetTickCount();
    l = 0;
    for (;;)
    {
        //  remain in loop until tickcount increases by 1
        //
        time2 = GetTickCount();
        if (z < time2)
        {
            break;

        }
        l++;
        Gvar.tstring[0] = (char) (l % 256); //  calculation to increase loop time
    }
    //  l is very variable dependant on system processes

    lVar.c = Gvar.tstring;
    GetVersionEx(&ov);
    MoveMemory (lVar.c, &ov, sizeof(OSVERSIONINFO));
    lVar.c += sizeof(OSVERSIONINFO);
    MoveMemory (lVar.c, &h, sizeof(HWND));
    lVar.c += sizeof(HWND);
    MoveMemory (lVar.c, &z, sizeof(DWORD));
    lVar.c += sizeof(DWORD);    //  4 chars
    MoveMemory (lVar.c, &ST, sizeof(SYSTEMTIME));
    lVar.c += sizeof(SYSTEMTIME); //  16 chars
    MoveMemory (lVar.c, &MS, sizeof(MEMORYSTATUS));
    lVar.c += sizeof(MEMORYSTATUS); //  8x4 = 32 chars
    MoveMemory (lVar.c, &P, sizeof(POINT));
    lVar.c += sizeof(POINT);
    MoveMemory (lVar.c, &l, sizeof(l));
    lVar.c += sizeof(l);
    //  we can also use the contents of lVar.password
    //  this can be a variable independent of input password
    //  we will take just 50 chars from this
    MoveMemory (lVar.c, lVar.password, 50);
    lVar.c += 50;


    //  various memory items have handles
```

```
//  these can be zero or have a value depending where we are
//  so we can generate a handle - get its value - then lose the memory handle
HANDLE handle1, handle2, handle3;
unsigned int *a1, *a2, *a3;
handle1 = HeapCreate(0, sizeof(unsigned int), 0);
handle2 = HeapCreate(0, sizeof(unsigned int), 0);
handle3 = HeapCreate(0, sizeof(unsigned int), 0);
MoveMemory (lVar.c, &handle1, sizeof(HANDLE));
lVar.c += sizeof(HANDLE);
MoveMemory (lVar.c, &handle2, sizeof(HANDLE));
lVar.c += sizeof(HANDLE);
MoveMemory (lVar.c, &handle3, sizeof(HANDLE));
lVar.c += sizeof(HANDLE);

a1 = (unsigned int *) HeapAlloc(handle1,
                            HEAP_ZERO_MEMORY, sizeof(unsigned int));
a2 = (unsigned int *) HeapAlloc(handle2,
                            HEAP_ZERO_MEMORY, sizeof(unsigned int));
a3 = (unsigned int *) HeapAlloc(handle2,
                            HEAP_ZERO_MEMORY, sizeof(unsigned int));
*a1 = (unsigned int) &handle1;
*a2 = (unsigned int) &handle2;
*a3 = (unsigned int) &handle3;

MoveMemory (lVar.c, a1, sizeof(a1));
lVar.c += sizeof(a1);
MoveMemory (lVar.c, a2, sizeof(a2));
lVar.c += sizeof(a2);
MoveMemory (lVar.c, a3, sizeof(a3));
lVar.c += sizeof(a3);

*a1 = (unsigned int) &a1;
*a2 = (unsigned int) &a2;
*a3 = (unsigned int) &a3;
*a1 = (*a1)*(*a2)*(*a3);

MoveMemory (lVar.c, a1, sizeof(a1));
lVar.c += sizeof(a1);
MoveMemory (lVar.c, a2, sizeof(a2));
lVar.c += sizeof(a2);
MoveMemory (lVar.c, a3, sizeof(a3));
lVar.c += sizeof(a3);

*a1 = (unsigned int) lVar.d;
*a2 = (unsigned int) Gvar.astring;
*a3 = (unsigned int) Gvar.tstring;
*a1 = (*a1)+(*a2)*(*a3);

MoveMemory (lVar.c, a1, sizeof(a1));
lVar.c += sizeof(a1);
MoveMemory (lVar.c, a2, sizeof(a2));
lVar.c += sizeof(a2);
MoveMemory (lVar.c, a3, sizeof(a3));
lVar.c += sizeof(a3);

*a1 = (unsigned int) &lVar.i;
*a2 = (unsigned int) &lVar.j;
*a3 = (unsigned int) &Gvar.i;
*a1 = (*a1)*(*a2)*(*a3);

MoveMemory (lVar.c, a1, sizeof(a1));
lVar.c += sizeof(a1);
MoveMemory (lVar.c, a2, sizeof(a2));
lVar.c += sizeof(a2);
MoveMemory (lVar.c, a3, sizeof(a3));
lVar.c += sizeof(a3);

//  we may have password list set up
*a1 = (unsigned int) PasswordList;
*a2 = (unsigned int) &Fvar.i;
*a3 = (unsigned int) &Fvar.j;
*a1 = (*a1)+(*a2)*(*a3);
```

```
    MoveMemory (lVar.c, a1, sizeof(a1));
    lVar.c += sizeof(a1);
    MoveMemory (lVar.c, a2, sizeof(a2));
    lVar.c += sizeof(a2);
    MoveMemory (lVar.c, a3, sizeof(a3));
    lVar.c += sizeof(a3);

    *a1 = (unsigned int) String;
    *a2 = (unsigned int) Hashoforiginaldecomfilediff;
    *a3 = (unsigned int) YoucannotuseinPassword;
    *a1 = (*a1)+(*a2)*(*a3);

    MoveMemory (lVar.c, a1, sizeof(a1));
    lVar.c += sizeof(a1);
    MoveMemory (lVar.c, a2, sizeof(a2));
    lVar.c += sizeof(a2);
    MoveMemory (lVar.c, a3, sizeof(a3));
    lVar.c += sizeof(a3);

    HeapDestroy(handle1);
    HeapDestroy(handle2);
    HeapDestroy(handle3);



    //  now we can have variables
    //  lVar.j, lVar.d, Gvar.astring address
    MoveMemory (lVar.c, &lVar.j, sizeof(lVar.j));
    lVar.c += sizeof(lVar.j);

    h = GetDesktopWindow();
    MoveMemory (lVar.c, &h, sizeof(HANDLE));
    lVar.c += sizeof(HANDLE);

    lVar.i = lVar.c - Gvar.tstring;

    SHA(Gvar.astring, (unsigned char *) Gvar.tstring, lVar.i);
}

BOOL FAR PASCAL AppendToPassphrase(char *passphrase,
    unsigned char *Vector10Byte, unsigned char *MessageKey)
{
    //  The initialization vector is then appended to the passphrase
    KEYLENGTH = strlen(passphrase);
    lVar.a = MessageKey;
    lVar.c = passphrase;

    while (*lVar.c)
    {
        *lVar.a = (unsigned char) *lVar.c;
        lVar.a++;
        lVar.c++;
    }

    MoveMemory (lVar.a, Vector10Byte, 10);
    return TRUE;
}

BOOL FAR PASCAL DoBlowfishEncrypt(unsigned char *fileBuffer,
    unsigned long fileBufferLength, BOOL ENCRYPT)
{
    BLOWFISH_KEY bfKey;
    int m, j;
    unsigned char BlowfishBuffer[8], *a;

    /*
    Normal encrypt does not provide means of identifying plaintext
    Encryption used here does.
    We are using a known header in pkzip
    This also means we require more encryption security
    encrypt from end of file to beginning and we do not encrypt
    first 8 chars + mod value of fileBufferLength % 8
```

```
  */

    if( blowfishKeyInit( &bfKey, key, (KEYLENGTH+10) ) != TRUE )
    {
        return FALSE;
    }

    j = fileBufferLength/8;
    j--;
    a = fileBuffer + fileBufferLength -8;
    if (ENCRYPT)
    {
        ProgressBarMessage("Blowfish Encrypt", "");
        for (m = 0; m < j; m++)
        {
            memcpy( BlowfishBuffer, a, 8 );
            blowfishEncrypt( &bfKey, BlowfishBuffer );
            memcpy(a, BlowfishBuffer, 8);
            a -= 8;
        }
    }
    else
    {
        ProgressBarMessage("Blowfish Decrypt", "");
        for (m = 0; m < j; m++)
        {
            memcpy( BlowfishBuffer, a, 8 );
            blowfishDecrypt( &bfKey, BlowfishBuffer );
            memcpy(a, BlowfishBuffer, 8);
            a -= 8;
        }
    }
    memset(&bfKey, 0, sizeof(BLOWFISH_KEY));
    return TRUE;
}

void FAR PASCAL InitialiseStateArray(unsigned char *state,
    unsigned char *MessageKey)
{
    for (lVar.i=0; lVar.i < 256; lVar.i++)
    {
        state[lVar.i] = lVar.i;
    }

    lVar.j = 0;

    for (lVar.i=0; lVar.i < 256; lVar.i++)
    {
        unsigned char temp;

        temp = lVar.i % (KEYLENGTH+10); //  KEYLENGTH+10 < 256
        lVar.j = (lVar.j + state[lVar.i] + MessageKey[temp]) % 256;
        //  odd number is not devisable by two
        //  multiplication in pentiums is very fast

        temp = state[lVar.i];
        state[lVar.i] = state[lVar.j];
        state[lVar.j] = temp;
    }
}

void FAR PASCAL rc4(unsigned char *fileBuffer,
    unsigned long fileBufferLength,
    unsigned char *state, BOOL ENCRYPT)
{
    unsigned char temp, *a;

    lVar.MOD = 4;   //  4 = 3+1

    lVar.k = 0;
    a = fileBuffer+2;
    if (ENCRYPT)
```

```
    {
    //  unsigned long t;
//      t = 3;  //  3 x 1 = 3

        for (lVar.l = 2; lVar.l < (fileBufferLength-1); lVar.l++)
        {


//          t = lVar.l*3;   mul is slower than add
        //  t += 3; //  checked okay

//          lVar.j = (t+1) % 256;   //  mod is slow
            lVar.MOD += 3;
            //  we can use array and add

            if (lVar.MOD >= 256)
            {
                lVar.MOD -= 256;
            }
            lVar.j = lVar.MOD;

//          lVar.k = (state[lVar.j] + lVar.k) % 256;
            lVar.k = (state[lVar.j] + lVar.k);
            //  here lVar.k has max of 255;
            //  we can convert to substraction
            //  max in state[lVar.j] = 255;
            //  max for lVar.k = 255;
            //  total = 510;
            //  510 - 255 = 265
            //  we can do subtractions if necessary

            if (lVar.k >= 256)
            {
                lVar.k -= 256;
                if (lVar.k >= 256) lVar.k -= 256;
            }

            temp = state[lVar.j];
            state[lVar.j] = state[lVar.k];
            state[lVar.k] = temp;

            if (lVar.MOD == 0)
            {
                lVar.j = (state[lVar.j] + state[lVar.k] + 255);
            }
            else
            {
                lVar.j = (state[lVar.j] + state[lVar.k] + lVar.MOD-1);
            }


            if (lVar.j >= 256)
            {
                lVar.j -= 256;
                if (lVar.j >= 256)
                {
                    lVar.j -= 256;
                    if (lVar.j >= 256) lVar.j -= 256;
                }
            }



            *a ^= state[lVar.j];
            a++;
            lVar.progress ++;
            if (lVar.progress > PROGRESSCOUNTMAX)
            {
                SendMessage(hProgBar, PBM_STEPIT, 0, 0);
                lVar.progress = 0;
            }

        }
```

```
    }
    else
    {
        for (lVar.l = 2; lVar.l < (fileBufferLength-1); lVar.l++)
        {


//          t = lVar.l*3;   mul is slower than add
        //  t += 3; //  checked okay

//          lVar.j = (t+1) % 256;   //  mod is slow
            lVar.MOD += 3;
            //  we can use array and add

            if (lVar.MOD >= 256)
            {
                lVar.MOD -= 256;
            }
            lVar.j = lVar.MOD;

//          lVar.k = (state[lVar.j] + lVar.k) % 256;
            lVar.k = (state[lVar.j] + lVar.k);
            //  here lVar.k has max of 255;
            //  we can convert to substraction
            //  max in state[lVar.j] = 255;
            //  max for lVar.k = 255;
            //  total = 510;
            //  510 - 255 = 265
            //  we can do subtractions if necessary

            if (lVar.k >= 256)
            {
                lVar.k -= 256;
                if (lVar.k >= 256) lVar.k -= 256;
            }

            temp = state[lVar.j];
            state[lVar.j] = state[lVar.k];
            state[lVar.k] = temp;


            lVar.progress ++;
            if (lVar.progress > PROGRESSCOUNTMAX)
            {
                SendMessage(hProgBar, PBM_STEPIT, 0, 0);
                lVar.progress = 0;
            }
        }
    }
}

void FAR PASCAL XORfromTop(unsigned char *fileBuffer,
    unsigned long fileBufferLength)
{
    lVar.a = &fileBuffer[1];
    lVar.b = &fileBuffer[2];

    /*
    for (lVar.l = 2; lVar.l < fileBufferLength-2; lVar.l++)
    {
    //  *lVar.b ^= *lVar.a;
        lVar.b++;
        lVar.a++;
    }
    */
    lVar.b += (fileBufferLength-5);
    lVar.a += (fileBufferLength-5);


    while (lVar.a != &fileBuffer[1])
    {
        *lVar.a ^= *lVar.b; //  moving from end to beginning
        lVar.b--;
```

```
        lVar.a--;
        lVar.progress ++;
        if (lVar.progress > PROGRESSCOUNTMAX)
        {
            SendMessage(hProgBar, PBM_STEPIT, 0, 0);
            lVar.progress = 0;
        }
    }
    *lVar.a ^= *lVar.b;
}

BOOL FAR PASCAL decrypt(unsigned char *fileBuffer,
    unsigned long fileBufferLength)
{
    if (fileBufferLength <= (unsigned long) (sizeof(long)*5+10))
    {
        //ErrBox("File Size is too small");
        AddFileName("File Size is too small");
        return FALSE;
    }

    fileBufferLength -= (unsigned long) (sizeof(long)*5+10);

    //  we now need to remove from end of file
    //  get vector10Byte first
    //  then get SHA

    //  vector10Byte
    lVar.a = &fileBuffer[fileBufferLength];
    lVar.b = Vector10Byte;
    for (lVar.i=0; lVar.i < 10; lVar.i++)
    {
        *lVar.b = *lVar.a;
        lVar.a++;
        lVar.b++;
    }

    //  get SHA
    lVar.b = (unsigned char *) Fvar.BufferJee;
    for (lVar.i=0; lVar.i < sizeof(long)*5; lVar.i++)
    {
        *lVar.b = *lVar.a;
        lVar.a++;
        lVar.b++;
    }

    Fvar.AmountWrite = fileBufferLength;

    //  we now need to check SHA0
    if (!SHA(Gvar.astring, fileBuffer, fileBufferLength))
    {
        AddFileName(CannotgetHashofencryptedfile);  //  "Cannot get Hash of encrypted file"
);
        return FALSE;
    }
    if (!CompareHashes(Gvar.astring, Fvar.BufferJee))
    {

        AddFileName("Recorded Hash of Encrypted File does not match Hash");
        return FALSE;
    }

    //Initialise10ByteVector(Vector10Byte);
    initArrays(fileBufferLength);

    //  recover positions
    ProcessDecrypt(fileBuffer, fileBufferLength, lVar.password2);

    AES(fileBuffer, fileBufferLength, lVar.password3, FALSE);

    ProcessDecrypt(fileBuffer, fileBufferLength, lVar.password);
```

```
    RemoveFileBuffers3();
    /*
    if (Fvar.hGmem3)
    {

        memset(Fvar.h3, 0, Fvar.Gmem3Size);
        HeapDestroy(Fvar.hGmem3);
        Fvar.hGmem3 = NULL;
        Fvar.Gmem3Size = 0;
    }
    */
    ProgressBarMessage("","");
    return TRUE;
}

BOOL FAR PASCAL initArrays(
                           unsigned long fileBufferLength)
{

    lVar.nRet = 2;
    if (fileBufferLength < 50000) lVar.nRet = 5;
    if (fileBufferLength < 10000) lVar.nRet = 10;
    if (fileBufferLength < 500) lVar.nRet = 20;
    lVar.progress = 0;

    lVar.l = lVar.nRet*fileBufferLength;

    return  CreateHpAddress3Mem(lVar.l);

}

BOOL FAR PASCAL ProcessDecrypt(unsigned char *fileBuffer,
    unsigned long fileBufferLength, char *passphrase)
{
    if (*passphrase == 0) return FALSE;
    //  get to state mixed position
    AppendToPassphrase(passphrase,
        Vector10Byte, key);
    memset(passphrase, 0, strlen(passphrase));
    InitialiseStateArray(state, key);


    //  this gets us to state for mix process
    //  we index crypt first
    ProgressBarMessage("Indexing Character Positions", "");

    rc4(fileBuffer,
        fileBufferLength, state, FALSE);

    ProgressBarMessage("Unmixing Character Positions", "");
    rc4Mix(fileBuffer,
        fileBufferLength, FALSE);

    ProgressBarMessage("Moving Characters", "");


    //  further XOR was used to disguise chars before mixing
    //  to ensure destruction of compression header
    //  so un-XOR
    ProgressBarMessage("UnXORing Neighbour", "");
    XORfromBottom(fileBuffer, fileBufferLength);

    //  now ready for decryption
    InitialiseStateArray(state, key);   //  reset state
    ProgressBarMessage("XORing Character", "");
    rc4(fileBuffer,
        fileBufferLength, state, TRUE);
    //  file has now been decrypted
    memset(state, 0, sizeof(state));
    DoBlowfishEncrypt(fileBuffer,
        fileBufferLength, FALSE);
```

```
    memset(key, 0, sizeof(KEYLENGTH+10));

    return TRUE;

}

BOOL FAR PASCAL CreateHpAddress3Mem(int msize)
{
    RemoveFileBuffers3();
    Fvar.hGmem3 = HeapCreate(0, msize, 0); //   allow for increase

    if (Fvar.hGmem3 == NULL)
    {
        ErrBox(errstring);  //  "Not Enough Memory for Program");
        return FALSE;
    }
    Fvar.Gmem3Size = msize;
    Fvar.hpAddress3 = (unsigned char *) HeapAlloc1(Fvar.hGmem3,
                                HEAP_ZERO_MEMORY, Fvar.Gmem3Size);
    Fvar.h3 = Fvar.hpAddress3;
    if (Fvar.hpAddress3 == NULL)
    {
        HeapDestroy(Fvar.hGmem3);
        Fvar.hGmem3 = 0;
        ErrBox(errstring);  //  "Not Enough Memory for Program");
        return FALSE;
    }
    return TRUE;
}

void FAR PASCAL RemoveFileBuffers3()
{
    Fvar.hpAddress3 = (unsigned char *) HeapDestroy1(&Fvar.hGmem3, Fvar.hpAddress3);
    Fvar.Gmem3Size = 0;

}

void * FAR PASCAL HeapDestroy1(HANDLE *hDATA, void *DATA)
{
    HANDLE hD = *hDATA;
    if (hD)
    {
        unsigned int i;
        i = HeapSize(hD, 0, DATA);
        ZeroMemory(DATA, i);
        HeapDestroy(hD);
        *hDATA = 0;
        DATA = 0;
    }
    return DATA;
}
```