MANUAL INTRODUCTORIO PARA PROGRAMAR CON OPENSSL

LIC. ANA MARÍA ARIAS ROIG

1. Introducción

OpenSSL es una implementación *open source* de SSL (Secure Socket Layer) que puede descargarse de http://www.openssl.org. En algunas distribuciones linux ya viene instalado.

Permite realizar tareas criptográficas tales como:

- calcular **hashes** criptográficos (MD2, MD4, MD5, RIPEMD-160, SHA, SHA-1)
- cifrar con **algoritmos de cifrado simétrico** (AES, CAST, DES, Tiple DES, Blowfish, RC2) o en flujo (RC4), y usar aquéllos en múltiples modos de operación (ECB, CBC, CFB, OFB).
- generar **claves asimétricas** para los algoritmos habituales (Diffie-Hellman, DSA, RSA)
- Utilizar los algoritmos para **firmar, certificar** y revocar claves.
- manejar **formatos de certificados** (X.509, PEM, PKCS7, PKCS8, PKCS12).

El objetivo del presente documento es presentar la forma de codificar en C, a través de la librería de openssl, las operaciones más importantes de hash, cifrado simétrico y cifrado asimétrico, incluyendo generación de claves privada y pública y firma digital. En general se presenta más de una manera de resolver dichas operaciones, para que se observen distintas opciones de implementación.

La cantidad de funciones que ofrece openssl se extiende mucho más de lo que este documento puede abarcar, por lo que se recomienda investigar los archivos citados para más información.

2. Instalación.

Descargar de http://www.openssl.org/source/ la última versión. Por ejemplo, openssl-1.0.0d.tar.gz.

Después de descomprimir, se tendrá una carpeta openssl-1.0.0d, donde se cuenta con información sobre instalación en los distintos sistemas operativos.

Una vez instalado NO conviene borrar esta carpeta, ya que contiene documentación importante para consulta permanente.[1]

En particular, para instalar en windows se corre install.com. Luego se copian en **c:\MinGW\lib** los archivos de **c:\OpenSSL-Win32\lib\Mingw**, (libeay32.a, libeay32.def, ssleay32.a, ssleay32.def) y en **c:\MinGW\include** la carpeta **openssl** que se encuentra en **c:\OpenSSL-Win32\include**.

En linux deberán instalarse los paquetes **openssl** y **libssl-dev**. Al compilar, usar la opción **-lcrypto.**

3. Funciones y estructuras en openssl.

Para cada tarea criptográfica existe una librería de funciones. Así, md5.h tiene la lista de funciones para hacer un hash con md5; aes.h tiene la lista de funciones para encriptar con el algoritmo aes; rsa.h tiene la lista de funciones para trabajar con rsa, etc.

Esas funciones son las funciones básicas para realizar las tareas criptográficas. Sin embargo, presentan una gran falta de uniformidad en su implementación. Por ejemplo, las primitivas de aes.h son muy diferentes a las primitivas de des.h, a pesar de que en ambos casos se trata de cifrados de bloque. Para lograr una mayor uniformidad en el manejo de técnicas criptográficas y evitar tener que incursionar en tareas de más bajo nivel, existe la librería evp.h (envelope). Esta librería contiene funciones para todas las tareas criptográficas que se necesiten.

Para manejar cuestiones de entrada salida, openssl provee bio.h. Bio.h contiene una serie de funciones de abstracción de I/O que permiten manejar conexiones SSL, conexiones de red no encriptadas

[1] En **openssl-1.0.0d\doc\crypto** están los archivos de extensión .pod con documentación de los .h correspondientes y de funciones en particular. En **openssl-1.0.0d\crypto** están los archivos que implementan todas las funciones de las librerías de crypto. En **openssl-1.0.0d\apps** están los archivos fuente de las aplicaciones de openssl por línea de comando.

y archivos. Se usará, por ejemplo, en el caso de guardar y recuperar claves asimétricas, ya que así lo requieren las funciones de pem.h (funciones de escritura/lectura en formato PEM)[2]

En general, la estructura de datos fundamental es el contexto. Dicha estructura almacena información sobre algoritmo de cifrado o hash, si encripta o desencripta, clave actual, bloque actual, etc. En la mayoría de los casos, esas estructuras de contexto son las que primero se inicializan y las últimas que se liberan.

4. Hash

En todos los casos de hash usaremos el mismo ejemplo: hash del texto: "hace mucho calor hoy". Asumiendo que dicho texto está en el archivo in.txt, lo que se codifica es equivalente a:

```
OpenSSL>dgst –md5 –out hash.txt in.txt
OpenSSL>dgst –sha1 –out hash.txt in.txt
```

4.1. md5 con funciones de md5.h

- **Estructuras:**

La estructura para manejar digestos en md5.h es:

```
typedef struct MD5state_st
       {
       MD5_LONG A,B,C,D;
       MD5_LONG Nl,Nh;
       MD5_LONG data[MD5_LBLOCK];
       unsigned int num;
       } MD5_CTX;
```

- **Funciones:**

```
unsigned char *MD5(const unsigned char *d, unsigned long n, unsigned char *md);
```

- salida de 128 bits.
- Retorna un puntero al valor de hash.
- Calculan el digesto de los **n** bytes en la cadena **d**, y lo ubica en **md**. Es importante tener en cuenta que **md** no retorna con ningún símbolo de fin de cadena.
- La longitud de la salida es de 16bytes, o sea que debe reservarse 16 bytes para **md**.
- Si **md** es NULL, el digesto se ubica en un arreglo estático.

Si el mensaje no puede ser completamente almacenado en memoria, se usan las funciones:

```
int MD5_Init(MD5_CTX *c);

int MD5_Update(MD5_CTX *c, const void *data, unsigned long len);

int MD5_Final(unsigned char *md, MD5_CTX *c);
```

MD2_Init()

- inicializa una estructura de tipo **MD5_CTX**.
- Retorna **1 (éxito**) o **0** en caso contrario.

MD2_Update()

- puede ser llamada repetidamente con porciones de mensaje que serán hasheados (len bytes por data)
- Retorna **1 (éxito)** o **0** en caso contrario.

MD2_Final()

- ubica el digesto en **m**d, que debe tener espacio.
- Retorna **1 (éxito)** o **0** en caso contrario.

- **Ejemplos:**

Ejemplo 1: En forma directa, usando la función MD5.[3]

---

[2] En la documentación indica que se pueden usar las funciones de PEM sobre estructura FILE en lugar de BIO, pero suele presentar problemas.

[3] Para abreviar este documento, **no** se escriben todas las validaciones de retorno de funciones. Sin embargo, se **recomienda hacer siempre las validaciones** para controlar errores.

```
int
main()
{
    unsigned char *data = "hace mucho calor hoy";
    unsigned char *md;
    /*Forma directa*/
    md = malloc (MD5_DIGEST_LENGTH);
    MD5(data, strlen(data), md);
    saveDataHexa(md, MD5_DIGEST_LENGTH,"hash.txt");[4]
    free(md);
    return EXIT_SUCCESS;
}
```

Ejemplo 2: Usando la estructura MD5_CTX.

```
int
main()
{
    unsigned char *data = "hace mucho calor hoy";
    MD5_CTX origen;
    unsigned char *md;

    MD5_Init(&origen);
    md = malloc (MD5_DIGEST_LENGTH);
    MD5_Update(&origen, data, strlen(data));
    MD5_Final(md, &origen);
    saveDataHexa(md, MD5_DIGEST_LENGTH, "hash.txt");
    free (md);
    return EXIT_SUCCESS;
}
```

4.2. sha1 con funciones de sha.h

- **Estructuras:**

La estructura para manejar digestos en sha.h es:

```
typedef struct SHAstate_st
      {
      SHA_LONG h0,h1,h2,h3,h4;
      SHA_LONG Nl,Nh;
      SHA_LONG data[SHA_LBLOCK];
      unsigned int num;
      } SHA_CTX;
```

- **Funciones:**

```
unsigned char *SHA1(const unsigned char *d, size_t n, unsigned char *md);
```

- salida de 160 bits.
- Retorna un puntero al valor de hash.
- Calculan el digesto de los **n** bytes en la cadena **d**, y lo ubica en **md**. Es importante tener en cuenta que **md** no retorna con ningún símbolo de fin de cadena.

[4] Función para guardar en formato hexa y poder ver que da lo mismo que con por linea de comando. Por ej.:

```
int saveDataHexa(unsigned char *data, size_t datalen, unsigned char *where)
{
    FILE *out;
    int i;
    out = fopen(where, "w");
    for (i = 0; i < datalen; i++)
        fprintf(out, "%02x", data[i]);
    fclose(out);
    return EXIT_SUCCESS;
}
```

- La longitud de la salida es de 20bytes, o sea que debe reservarse 16 bytes para **md**.
- Si **md** es NULL, el digesto se ubica en un arreglo estático.

Si el mensaje no puede ser completamente almacenado en memoria, se usan las funciones:

```
int SHA1_Init(SHA_CTX *c);
int SHA1_Update(SHA_CTX *c, const void *data, size_t len);
int SHA1_Final(unsigned char *md, SHA_CTX *c);
```

`int SHA1_Init(SHA_CTX *c);`

- inicializa una estructura de tipo **SHA_CTX**.
- Retorna **1 (éxito**) o **0** en caso contrario.

`int SHA1_Update(SHA_CTX *c, const void *data, size_t len);`

- puede ser llamada repetidamente con porciones de mensaje que serán hasheados (len bytes por data)
- Retorna **1 (éxito)** o **0** en caso contrario.

`int SHA1_Final(unsigned char *md, SHA_CTX *c);`

- ubica el digesto en **m**d, que debe tener espacio.
- Borra el contexto c
- Retorna **1 (éxito)** o **0** en caso contrario.

- **Ejemplos:**

En este caso, las funciones de md5 y sha1 poseen características casi idénticas por lo que no hace falta dar nuevos ejemplos.

4.3. md5 y sha1 con funciones de evp.h

- **Estructuras:**

Para digestos de mensajes se usan dos estructuras: **`EVP_MD`** y **`EVP_MD_CTX.`**

En **`ossl_typ.h`** y en **`evp.h`** están sus detalles:

`typedef struct env_md_st` **`EVP_MD;`**

```
struct env_md_st
	{
	int type;
	int pkey_type;
	int md_size;
	unsigned long flags;
	int (*init)(EVP_MD_CTX *ctx);
	int (*update)(EVP_MD_CTX *ctx,const void *data,size_t count);
	int (*final)(EVP_MD_CTX *ctx,unsigned char *md);
	int (*copy)(EVP_MD_CTX *to,const EVP_MD_CTX *from);
	int (*cleanup)(EVP_MD_CTX *ctx);
	int (*sign)(int type, const unsigned char *m, unsigned int m_length,
		    unsigned char *sigret, unsigned int *siglen, void *key);
	int (*verify)(int type, const unsigned char *m, unsigned int m_length,
		      const unsigned char *sigbuf, unsigned int siglen,
		      void *key);
	int required_pkey_type[5]; /*EVP_PKEY_xxx */
	int block_size;
	int ctx_size; /* how big does the ctx->md_data need to be */
	/* control function */
	int (*md_ctrl)(EVP_MD_CTX *ctx, int cmd, int p1, void *p2);
	};
```

`typedef struct env_md_ctx_st EVP_MD_CTX;`

```
struct env_md_ctx_st
	{
	const EVP_MD *digest;
	ENGINE *engine; /*functional reference if 'digest' is ENGINE-provided*/
	unsigned long flags;
```

```
      void *md_data;

      /* Public key context for sign/verify */
      EVP_PKEY_CTX *pctx;

      /* Update function: usually copied from EVP_MD */
      int (*update)(EVP_MD_CTX *ctx,const void *data,size_t count);
      };
```

El campo de tipo ENGINE aparece en varias oportunidades y se utiliza si se desea usar una implementación de algoritmo distinta de la que ofrece openssl. Como utilizaremos sólo las implementaciones de openssl, cada vez que se requiera un argumento de tipo ENGINE, éste se completará con NULL.

- **Funciones:**

```
void EVP_MD_CTX_init(EVP_MD_CTX *ctx);

EVP_MD_CTX *EVP_MD_CTX_create(void);
```

- **EVP_MD_CTX_init()** inicializa un contexto para digesto.
- **EVP_MD_CTX_create()** reserva espacio, inicializa y retorna un contexto para digesto.

```
int EVP_MD_CTX_cleanup(EVP_MD_CTX *ctx);

void EVP_MD_CTX_destroy(EVP_MD_CTX *ctx);
```

- **EVP_MD_CTX_cleanup()** elimina un contexto si ya no se utiliza más.
- **EVP_MD_CTX_destroy()** elimina el contexto liberando la memoria reservada (usar si se usó **EVP_MD_CTX_create**)

```
int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);

int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, size_t cnt);

int EVP_DigestFinal_ex(EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s);
```

- **EVP_DigestInit_ex()** inicializa un contexto para digesto para que pueda utilizar un tipo especial de algoritmo de digesto, especificado en el argumento **type**. El tipo puede darse por una función del tipo **EVP_md5() o EVP_sha1()**, o bien mediante la utilización de `EVP_get_digestbyname()`, si previamente se inicializó la tabla de todos los algoritmos de digesto mediante **OpenSSL_add_all_digests()**
- **EVP_DigestUpdate()** efectúa el hash de **cnt** bytes de datos, obtenidos de **d**. Esta función puede invocarse varias veces sobre el mismo contexto para hash de datos adicionales.
- **EVP_DigestFinal_ex()** permite obtener el valor de digesto que se almacenó en el contexto y ubicarlo en md. Si el parámetro **s** no está en NULL, se obtendrá ahí el número de bytes de datos escritos, que como máximo es `EVP_MAX_MD_SIZE`

- **Ejemplos:**

Ejemplo 1:

```
int
main()
{
    char    msg[]="hace mucho calor hoy";
    unsigned char   md[EVP_MAX_MD_SIZE];
    EVP_MD_CTX  *mdctx;
    const EVP_MD    *md_type;
    int md_len;

    /*Cargar tabla con algoritmos disponibles para hash*/
    OpenSSL_add_all_digests();
    md_type = EVP_get_digestbyname("md5");

    /*Crea contexto*/
    mdctx = EVP_MD_CTX_create();
```

```
    /*Efectua el hash y finalmente se guarda en md*/
    EVP_DigestInit_ex(mdctx, md_type, NULL);
    EVP_DigestUpdate(mdctx, msg, strlen(msg));
    EVP_DigestFinal_ex(mdctx, md, &md_len);

    saveDataHexa(md, md_len, "hash.txt");

    /*Libera contexto*/
    EVP_MD_CTX_cleanup(mdctx);

    return EXIT_SUCCESS;
}
```

Ejemplo 2:

```
int
main()
{
    char    msg[]="hace mucho calor hoy";
    unsigned char   md[EVP_MAX_MD_SIZE];
    EVP_MD_CTX  *mdctx;
    int md_len;

    /*Crea contexto*/
    mdctx = EVP_MD_CTX_create();

    /*Efectua el hash y finalmente se guarda en md_value*/
    EVP_DigestInit_ex(mdctx, EVP_md5(), NULL);
    EVP_DigestUpdate(mdctx, msg, strlen(msg));
    EVP_DigestFinal_ex(mdctx, md, &md_len);

    saveDataHexa(md, md_len, "hash4.txt");

    /*Libera contexto*/
    EVP_MD_CTX_cleanup(mdctx);

    return EXIT_SUCCESS;
}
```

5. Cifrado Simétrico

5.1. des con des.h

- **Estructuras:**

  typedef unsigned char **DES_cblock**[8];

  typedef unsigned char **const_DES_cbloc**k[8];

```
typedef struct DES_ks
    {
    union
       {
       DES_cblock cblock;
       DES_LONG deslong[2];
       } ks[16];
    } DES_key_schedule;
```

- **Funciones:**

```
void DES_random_key(DES_cblock *ret);
```

– Genera una clave aleatoria.

```
int DES_set_key_checked(const_DES_cblock *key, DES_key_schedule *schedule);

void DES_set_key_unchecked(const_DES_cblock *key, DES_key_schedule *schedule);
```

- Para que un bloque de 8 bytes pueda ser usado como clave DES debe convertirse a la estructura DES_key_schedule con estas funciones.
- **`DES_set_key_checked()`** controla que la clave sea de paridad impar y que no sea débil o semidébil.
- Retorna -1 si la paridad es incorrecta, --2 si es una clave débil y si ocurre algún error, el key schedule no se genera.

```
void DES_set_odd_parity(DES_cblock *key);
```

- Setea la paridad de la clave a impar.

```
int DES_is_weak_key(const_DES_cblock *key);
```

- Retorna 1 si es débil y 0 si es apropiada. La probabilidad de que una clave generada al azar sea débil es de 1 en 2^{52}.

```
void DES_ecb_encrypt(const_DES_cblock *input, DES_cblock *output, DES_key_schedule *ks,
int enc);
```

- Encripta o desencripta (según el valor del argumento **`enc`**), un bloque `input` de 8 bytes en modo ECB, obteniendo otro bloque de 8 bytes en **`output`**.
- ECB encripta de a un bloque por vez, asi que si hay que cifrar una cantidad mayor que 8 bytes, deberá hacerse una función que vaya tomando cada bloque, lo encripte y los concatene en un cifrado final.
- Si los datos a encriptar no son múltiplos de 8 bytes, hay que rellenar con algún tipo de padding.
- El argumento **`enc`** puede ser DES_ENCRYPT o DES_DECRYPT.

```
void DES_ecb2_encrypt(const_DES_cblock *input, DES_cblock *output, DES_key_schedule *ks1,
DES_key_schedule *ks2, int enc);

void DES_ecb3_encrypt(const_DES_cblock *input, DES_cblock *output, DES_key_schedule *ks1,
DES_key_schedule *ks2, DES_key_schedule *ks3, int enc);
```

- para Triple-DES.

```
void DES_ncbc_encrypt(const unsigned char *input, unsigned char *output, long length,
DES_key_schedule *schedule, DES_cblock *ivec, int enc);
```

- Encripta o desencripta en modo CBC usando el key schedule provisto y el vector de inicialización **`ivec`**, que es de 8 bytes también.
- Si la longitud especificada en **`length`** no es múltiplo de 8 bytes, el último bloque se copia en un área temporal y se completa con ceros.
- La salida siempre es múltiplo de 8.
- El valor del vector de inicialización queda modificado.

```
void DES_cfb_encrypt(const unsigned char *in, unsigned char *out, int numbits, long
    length, DES_key_schedule *schedule, DES_cblock *ivec, int enc);

void DES_ofb_encrypt(const unsigned char *in, unsigned char *out, int numbits, long
    length, DES_key_schedule *schedule, DES_cblock *ivec);
```

- Encriptan o desencriptan en modo CFB y OFB respectivamente, usando el key schedule provisto y el vector de inicialización **`ivec`**, que es de 8 bytes también.
- No requiere padding.
- Operan sobre input de 8 bits.
- La salida siempre es múltiplo de 8.
- El valor del vector de inicialización queda modificado.

```
void DES_cfb64_encrypt(const unsigned char *in, unsigned char *out, long length,
DES_key_schedule *schedule, DES_cblock *ivec, int *num, int enc);

void DES_ofb64_encrypt(const unsigned char *in, unsigned char *out, long length,
DES_key_schedule *schedule, DES_cblock *ivec, int *num);
```

- Encriptan o desencriptan en modo CFB y OFB respectivamente, con 64 bits de feedback.

- **Ejemplos:**

Ejemplo 1. Encripción del texto: "*Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente.*", en DES modo ECB con clave "12345678", mostrando el cifrado en codificación base 64. Asumiendo que dicho texto está en el archivo in.txt, lo que se codifica es equivalente a:

```
openssl enc –des-ecb -in in.txt –K 3132333435363738[5] –e –a
```

```
#define BLOCK_SIZE      8
int
main()
{
    unsigned char *nomArch = "des-ecb.txt";
    unsigned char *in = "Inteligencia, dame el nombre exacto de las cosas... Que
mi palabra sea la cosa misma, creada por mi alma nuevamente.";
    unsigned char *inPad;
    unsigned char *out;
    int inl, outl;

    DES_cblock  k = "12345678";
    DES_key_schedule    ks;

    /* Setear paridad impar*/
    DES_set_odd_parity(&k);
    /* Setear key schedule*/
    DES_set_key_checked(&k, &ks);
    /* Completar con padding*/
    inl = strlen(in);
    inPad = padding(in, &inl, BLOCK_SIZE);
    /* Modo ecb encripta de a BLOQUES*/
    out = malloc (inl);
    outl = inl;
    Encrypt(inPad, inl, out, &ks);
    saveEncryptedData(out, outl,nomArch);
    free(inPad);
    free(out);
    return EXIT_SUCCESS;
}
```

Antes de encriptar, hay que rellenar con algún método hasta completar un tamaño de entrada múltiplo de bloque (padding) Como método se puede usar, por ejemplo, el de llenar con ceros o bien el PKCS5[6], que usa como valor de relleno el mismo número de bytes que hay que completar. Es decir, si se deben completar con relleno 5 bytes, después del último byte de la entrada original se guarda 55555.

```
unsigned char *
padding(unsigned char *in, int *inl, size_t blocksize)
{
    int pad;
    int i;
    unsigned char *inPad;

    pad = blocksize - (*inl) % blocksize;
    inPad = malloc(*inl + pad);
    memcpy(inPad, in, *inl);
    for (i = (*inl); i < (*inl + pad); i++)
        inPad[i] = pad;
    *inl +=pad;
    return (inPad);
}
```

[5] Tener en cuenta que la clave tiene 8 bytes y el valor que se recibe por línea de comandos está codificado en hexa.
[6] Este es el padding por defecto de openssl enc

Como en DES modo ECB se encripta de a un bloque por vez, se debe invocar varias veces la función **`DES_ecb_encrypt`** hasta terminar de encriptar toda la entrada.

```
int
Encrypt(unsigned char *in, int inl, unsigned char *out, DES_key_schedule *ks)
{
    DES_cblock inB;
    DES_cblock outB;
    int numB;
    int i;

    numB = inl / BLOCK_SIZE;
    for (i = 0; i < numB; i++)
    {
        memcpy(inB, in + i*BLOCK_SIZE, BLOCK_SIZE);
        DES_ecb_encrypt(&inB,&outB,ks,DES_ENCRYPT);
        memcpy(out + i*BLOCK_SIZE, outB, BLOCK_SIZE);
    }
    return SUCCESS;
}
```

Por último, la función **`saveEncryptedData`** almacena la salida en un archivo en base 64. Para almacenar la salida en base 64 se puede desarrollar una función de transformación a esa base o bien usar funciones de bio:

```
int saveEncryptedData(unsigned char *out, int len,unsigned char *where)
{
    BIO *b64;
    BIO *bio;
    b64 = BIO_new(BIO_f_base64());

    bio = BIO_new(BIO_s_file());
    if(bio == NULL)
        return FAILURE;
    if(!BIO_write_filename(bio, where))
        return FAILURE;

    bio = BIO_push(b64, bio);

    BIO_write(bio, out, len);
    BIO_flush(bio);

    BIO_free_all(bio);

    return SUCCESS;
}
```

Ejemplo 2. Para la encripción del texto: "*Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente*.", con clave "12345678", y vector de inicialización "87654321"mostrando el cifrado en codificación base 64. Asumiendo que dicho texto está en el archivo in.txt, lo que se codifica es equivalente a:

```
openssl enc –des-ecb -in in.txt –K 3132333435363738 -IV 3837363534333231 –e –a
```

Una vez hecho el padding, la función misma divide y procesa por bloques la entrada. Es decir, se la invoca con la entrada completa y no de a un bloque por vez.

```
int
main()
{
    unsigned char *nomArch = "des-cbc.txt";
    unsigned char *in = "Inteligencia, dame el nombre exacto de las cosas... Que
mi palabra sea la cosa misma, creada por mi alma nuevamente.";
    unsigned char *inPad;
    int inl;
```

```
    unsigned char out[MAX_ENCR_LENGTH];
    int outl;
    DES_cblock k = "12345678";
    DES_cblock iv ="87654321";
    DES_key_schedule    ks;

    /* Setear paridad impar*/
    DES_set_odd_parity(&k);
    /* Setear key schedule*/
    DES_set_key_checked(&k, &ks);

    /* Completar con padding*/
    inl = strlen(in);
    inPad = padding(in, &inl);
    /* Encripción modo CBC*/
    DES_ncbc_encrypt(inPad, out, inl, &ks, &iv, DES_ENCRYPT);
    outl = inl;
    saveEncryptedData(out, outl,nomArch);
    free(inPad);
    return EXIT_SUCCESS;
}
```

5.2. aes con aes.h

- **Estructuras:**

```
typedef struct aes_key_st AES_KEY;
```

```
struct aes_key_st {
#ifdef AES_LONG
    unsigned long rd_key[4 *(AES_MAXNR + 1)];
#else
    unsigned int rd_key[4 *(AES_MAXNR + 1)];
#endif
    int rounds;
};
```

- **Funciones:**

```
int AES_set_encrypt_key(const unsigned char *userKey, const int bits, AES_KEY *key);
```

– Transforma la cadena **userKey** (de longitud **bits**) en un key schedule apropiado para **encripción** con AES.

```
int AES_set_decrypt_key(const unsigned char *userKey, const int bits, AES_KEY *key);
```

– Transforma la cadena **userKey** (de longitud **bits**) en un key schedule apropiado para **desencripción** con AES.

```
void AES_ecb_encrypt(const unsigned char *in, unsigned char *out, const AES_KEY *key,
const int enc);
```

– Función para encripción / desencripción con AES. Encripta o desencripta los primeros 128 bits de la entrada **in. (un bloque).**

```
void AES_encrypt(const unsigned char *in, unsigned char *out, const AES_KEY *key);

void AES_decrypt(const unsigned char *in, unsigned char *out, const AES_KEY *key);
```

– `AES_encrypt()` es equivalente a `AES_ecb_encrypt()` con **enc** seteado en AES_ENCRYPT.

– `AES_decrypt()` es equivalent to `AES_ecb_encrypt()` con **enc** seteado en AES_DECRYPT.

```
void AES_cbc_encrypt(const unsigned char *in, unsigned char *out, size_t length, const
AES_KEY *key, unsigned char *ivec, const int enc);
```

– Implementa el modo CBC.

– Encripta o desencripta la entrada **in** cuya longitud está dada por **length** en bytes.

```
void AES_cfb1_encrypt(const unsigned char *in, unsigned char *out, size_t length, const
AES_KEY *key, unsigned char *ivec, int *num, const int enc);

void AES_cfb8_encrypt(const unsigned char *in, unsigned char *out, size_t length, const
AES_KEY *key, unsigned char *ivec, int *num, const int enc);

void AES_cfb128_encrypt(const unsigned char *in, unsigned char *out, size_t length, const
AES_KEY *key, unsigned char *ivec, int *num, const int enc);
```

- Implementan el modo CFB para 1, 8 o 128 bits de feedback respectivamente.
- En `AES_cfb1_encrypt()` se opera sobre datos cuya longitud está expresada en bits: el valor de **length** es el número de bits.
- `AES_cfb8_encrypt()` y `AES_cfb128_encrypt()` operan sobre datos cuya longitude está expresada en bytes. El valor de **length** es el número de bytes.

```
void AES_ofb128_encrypt(const unsigned char *in, unsigned char *out, size_t length, const
AES_KEY *key, unsigned char *ivec, int *num);
```

- Implementa el modo OFB con 128 bits de feedback.
- Opera sobre datos cuya longitud está expresada en bytes.
- Como en modo OFB se usa el mismo algoritmo para encripción que para desencripción el argumento **enc** no es necesario.

- **Ejemplos:**

Ejemplo 1. Encripción del texto: "*Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente.*", en AES 128 modo ECB con clave "0123456789012345", mostrando el cifrado en codificación base 64. Asumiendo que dicho texto está en el archivo in.txt, lo que se codifica es equivalente a:

```
openssl enc –aes-128-ecb -in in.txt –K 30313233343536373839303132333435 –e –a
```

```
int
main()
{
    unsigned char *nomArch = "aes-128-ecb.txt";
    unsigned char *in = "Inteligencia, dame el nombre exacto de las cosas... Que
mi palabra sea la cosa misma, creada por mi alma nuevamente.";
    unsigned char *inPad;
    unsigned char *out;
    int inl, outl;
    unsigned char *k = "0123456789012345";/*128 bits = 16 bytes*/
    AES_KEY ks;

    /* Setear key schedule*/
    AES_set_encrypt_key(k, 128, &ks);
    /* Completar con padding*/
    inl = strlen(in);
    inPad = padding(in, &inl, AES_BLOCK_SIZE);
    /* Modo ecb encripta de a BLOQUES*/
    out = malloc (inl);
    outl = inl;
    Encrypt(inPad, inl, out, &ks);
    saveEncryptedData(out, outl,nomArch);
    free(in);
    free(out);
    return EXIT_SUCCESS;
}
```

Donde la función de Encrypt es:

```
int
Encrypt(unsigned char *in, int inl, unsigned char *out, AES_KEY *ks)
{
    unsigned char inB[AES_BLOCK_SIZE];
```

```
    unsigned char outB[AES_BLOCK_SIZE];
    int numB;
    int i;

    numB = inl / AES_BLOCK_SIZE;
    for (i = 0; i < numB; i++)
    {
        memcpy(inB, in + i * AES_BLOCK_SIZE, AES_BLOCK_SIZE);
        AES_ecb_encrypt(inB, outB, ks, AES_ENCRYPT);
        memcpy(out + i * AES_BLOCK_SIZE, outB, AES_BLOCK_SIZE);
    }
    return SUCCESS;
}
```

Ejemplo 2. Encripción del texto: "*Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente.*", en AES 128 modo CBC con clave "0123456789012345" y vector de inicialización "5432109876543210" mostrando el cifrado en codificación base 64. Asumiendo que dicho texto está en el archivo in.txt, lo que se codifica es equivalente a:

```
   openssl enc –aes-128-cbc -in in.txt –K 30313233343536373839303132333435 –iv
35343332313039383736353433323130 –e –a
```

```
int
main()
{
    unsigned char *nomArch = "aes-128-cbc.txt";
    unsigned char *in = "Inteligencia, dame el nombre exacto de las cosas... Que
mi palabra sea la cosa misma, creada por mi alma nuevamente.";
    unsigned char *inPad;
    unsigned char *out;
    int inl, outl;
    unsigned char *k = "0123456789012345";/*128 bits = 16 bytes*/
    unsigned char iv[] = "5432109876543210";
    AES_KEY ks;

    /* Setear key schedule*/
    AES_set_encrypt_key(k, 128, &ks);
    /* Completar con padding*/
    inl = strlen(in);
    inPad = padding(in, &inl, AES_BLOCK_SIZE);

    out = malloc (inl);
    outl = inl;

    AES_cbc_encrypt(inPad, out, inl, &ks, iv, AES_ENCRYPT);

    saveEncryptedData(out, outl,nomArch);

    free(in);
    free(out);
    return EXIT_SUCCESS;
}
```

5.3. des y aes con evp.h

- **Estructuras:**

Para cifrado simétrico se usan dos estructuras: **EVP_CIPHER** y **EVP_CIPHER_CTX.**

```
typedef struct evp_cipher_st EVP_CIPHER;
```

```
struct evp_cipher_st
       {
       int nid;
       int block_size;
       int key_len;        /* Default value for variable length ciphers */
```

```
       int iv_len;
       unsigned long flags;      /* Various flags */
       int (*init)(EVP_CIPHER_CTX *ctx, const unsigned char *key,
                 const unsigned char *iv, int enc); /* init key */
       int (*do_cipher)(EVP_CIPHER_CTX *ctx, unsigned char *out,
                    const unsigned char *in, size_t inl);/* encrypt/decrypt data */
       int (*cleanup)(EVP_CIPHER_CTX *); /* cleanup ctx */
       int ctx_size;       /* how big ctx->cipher_data needs to be */
       /* Populate a ASN1_TYPE with parameters */
       int (*set_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *);
       /* Get parameters from a ASN1_TYPE */
       int (*get_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *);
       /* Miscellaneous operations */
       int (*ctrl)(EVP_CIPHER_CTX *, int type, int arg, void *ptr);
       /* Application data */
       void *app_data;
       } /* EVP_CIPHER */;
```

```
typedef struct evp_cipher_ctx_st EVP_CIPHER_CTX;
struct evp_cipher_ctx_st
       {
       const EVP_CIPHER *cipher;
       ENGINE *engine;     /* functional reference if 'cipher' is ENGINE-provided */
       int encrypt;        /* encrypt or decrypt */
       int buf_len;        /* number we have left */

       unsigned char  oiv[EVP_MAX_IV_LENGTH]; /* original iv */
       unsigned char  iv[EVP_MAX_IV_LENGTH];  /* working iv */
       unsigned char buf[EVP_MAX_BLOCK_LENGTH];/* saved partial block */
       int num;     /* used by cfb/ofb mode */

       void *app_data;           /* application stuff */
       int key_len;        /* May change for variable length cipher */
       unsigned long flags;      /* Various flags */
       void *cipher_data; /* per EVP data */
       int final_used;
       int block_mask;
       unsigned char final[EVP_MAX_BLOCK_LENGTH];/* possible final block */
       } /* EVP_CIPHER_CTX */;
```

- **Funciones:**

```
void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *ctx);
```

  - Inicializa contexto para cifrado.

```
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);
```

  - elimina el contexto liberando la memoria reservada.

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl,
unsigned char *key, unsigned char *iv);

int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char
*in, int inl);

int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);
```

- **EVP_EncryptInit_ex** inicializa un contexto de cifrado para que pueda utilizar un tipo especial de algoritmo de cifrado, especificado en el argumento **type**. El tipo puede darse por una función del tipo **EVP_des_cbc() O EVP_idea_cbc()**, o bien mediante la utilización de `EVP_get_cipherbyname()`, si previamente se inicializó la tabla de todos los algoritmos de cifrado mediante **OpenSSL_add_all_ciphers()**
- Si **impl** es NULL es porque se usa la implementación por default.
- **key** es la clave pública y el vector de inicialización es **iv**. Sus tamaños dependen del cifrado. En el caso de modo ECB, IV es NULL.
- Se puede invocar EncryptInit con NULL en todos los campos excepto en **type** y luego actualizarlos.

- **EVP_EncryptUpdate** encripta la entrada **in**, de **inl** bytes y lo vuelca en **out**. La variable outl registra el total de bytes escritos en **out**, que debe tener espacio suficiente para guardar los bytes encriptados.
- **EVP_EncryptFinal_ex** Encripta el ultimo bloque. Si el padding está habilitado (default), entonces encripta lo que queda del último bloque. El padding Standard es PKCS. El número de bytes escritos se guarda en outl y será entonces igual al tamaño de un bloque si el padding está habilitado. Si no está habilitado, retorna error si quedaron bytes en un bloque parcial.

```
int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl,
unsigned char *key, unsigned char *iv);

int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char
*in, int inl);

int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);
```

- Son las operaciones correspondientes a desencripción.

```
int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl, unsigned
char *key, unsigned char *iv, int enc);

int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char
*in, int inl);

int EVP_CipherFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);
```

- Son funciones que pueden usarse para encriptar o desencriptar, dependiendo del valor de enc.

```
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
```

- Habilita o deshabilita el padding. Por default, las operaciones de encripción usan padding standard. El padding es descartado cuando se desencripta. Si el padding es 0, no se hace padding pero habrá error si el total de datos a procesar no es múltiplo de un bloque.

```
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);

#define EVP_get_cipherbynid(a) EVP_get_cipherbyname(OBJ_nid2sn(a))

#define EVP_get_cipherbyobj(a) EVP_get_cipherbynid(OBJ_obj2nid(a))
```

- Retornan una estructura EVP_CIPHER

```
#define EVP_CIPHER_nid(e)              ((e)->nid)
#define EVP_CIPHER_CTX_nid(e)          ((e)->cipher->nid)
#define EVP_CIPHER_key_length(e)       ((e)->key_len)
#define EVP_CIPHER_CTX_key_length(e)   ((e)->key_len)
#define EVP_CIPHER_iv_length(e)              ((e)->iv_len)
#define EVP_CIPHER_CTX_iv_length(e)    ((e)->cipher->iv_len)
int EVP_CIPHER_type(const EVP_CIPHER *ctx);
#define EVP_CIPHER_CTX_type(c)         EVP_CIPHER_type(EVP_CIPHER_CTX_cipher(c))
#define EVP_CIPHER_CTX_cipher(e)       ((e)->cipher)
#define EVP_CIPHER_block_size(e)       ((e)->block_size)
#define EVP_CIPHER_CTX_block_size(e)   ((e)->cipher->block_size)
#define EVP_CIPHER_mode(e)             ((e)->flags) & EVP_CIPH_MODE)
#define EVP_CIPHER_CTX_mode(e)         ((e)->cipher->flags & EVP_CIPH_MODE)
```

- Para obtener datos internos de la estructura de cifrado.

Algunos algoritmos de cifrado disponibles son:

DES: **EVP_des_cbc(void), EVP_des_ecb(void), EVP_des_cfb1(void), EVP_des_cfb8(void), EVP_des_ede_cfb64(void);, EVP_des_ofb(void), EVP_des_ede_cbc(void), EVP_des_ede(), EVP_des_ede_ofb(void), EVP_des_ede_cfb(void), EVP_des_ede3_cbc(void), EVP_des_ede3(), EVP_des_ede3_ofb(void), EVP_des_ede3_cfb(void), EVP_desx_cbc(void)**

AES: **EVP_aes_128_ecb(void); EVP_aes_128_cbc(void); EVP_aes_128_cfb1(void); EVP_aes_128_cfb8(void); EVP_aes_128_cfb128(void); EVP_aes_128_cfb128 EVP_aes_128_ofb(void); EVP_aes_192_ecb(void); EVP_aes_192_cbc(void); EVP_aes_192_cfb1(void); EVP_aes_192_cfb8(void); EVP_aes_192_cfb128(void); EVP_aes_192_cfb128 EVP_aes_192_ofb(void);**

RC4: **EVP_rc4(void), EVP_rc4_40(void)**

IDEA: **`EVP_idea_cbc() EVP_idea_ecb(void), EVP_idea_cfb(void), EVP_idea_ofb(void), EVP_idea_cbc(void)`**

BLOWFISH: **`EVP_bf_cbc(void), EVP_bf_ecb(void), EVP_bf_cfb(void), EVP_bf_ofb(void);`**

CAST: **`EVP_cast5_cbc(void), EVP_cast5_ecb(void), EVP_cast5_cfb(void), EVP_cast5_ofb(void)`**

- **Ejemplo:**

Ejemplo 1. Encripción del texto: "*Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente.*", en AES 128 modo CBC con clave "0123456789012345" y vector de inicialización "5432109876543210" mostrando el cifrado en codificación base 64. Asumiendo que dicho texto está en el archivo in.txt, lo que se codifica es equivalente a:

```
openssl enc –aes-128-cbc -in in.txt –K 303132333435363738393031323334353 –iv
353433323130393837363534333231303 –e –a
```

```
int
main()
{
    unsigned char *nomArch = "aes-128-cbc-evp.txt";
    unsigned char *in = "Inteligencia, dame el nombre exacto de las cosas... Que
mi palabra sea la cosa misma, creada por mi alma nuevamente.";
    unsigned char out[MAX_ENCR_LENGTH];
    int inl, outl, templ;
    unsigned char *k = "0123456789012345";/*128 bits = 16 bytes*/
    unsigned char iv[] = "5432109876543210";

    EVP_CIPHER_CTX ctx;

    /* Inicializar contexto */
    EVP_CIPHER_CTX_init(&ctx);
    /* Establecer parámetros de encripción en el contexto*/
    EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, k, iv);
    inl = strlen(in);
    EVP_EncryptUpdate(&ctx, out, &outl, in, inl);
    /* Encripta 112 */
    printf("Encripta primero %d bytes\n",outl);
    /* Encripta la parte final, lo que queda del bloque + padding,
    en este caso 4 bytes + 12 bytes de padding */
    EVP_EncryptFinal(&ctx, out + outl, &templ);
    printf("Finalmente se encriptan %d bytes\n", templ);/*últimos 16 bytes*/
    saveEncryptedData(out, outl + templ,nomArch);
    /* Borrar estructura de contexto */
    EVP_CIPHER_CTX_cleanup(&ctx);
    return EXIT_SUCCESS;
}
```

6. Passwords

- **Funciones:**

Para generar clave e iv a partir de una password, se puede utilizar la función **`EVP_BytesToKey().`**

```
int EVP_BytesToKey (const EVP_CIPHER *type, EVP_MD *md, const unsigned char *salt, const
unsigned char *data, int datal, int count, unsigned char *key, unsigned char *iv)
```

`EVP_BytesToKey()` devuelve la longitud de la clave correspondiente al algoritmo de cifrado especificado en el argumento **`type`**.

Para obtener la clave y el vector de inicialización, a partir del password indicado en el parámetro **`data`**, de longitud **`datal`**, la función realiza un algoritmo que involucra una o más iteraciones (según el valor del argumento **`count`**) de transformación en las que un algoritmo de hash especificado en el argumento **`md`** va obteniendo una clave del tamaño deseado. El parámetro **`salt`** es opcional (puede ser NULL). El algoritmo es:

```
md_buf = H (data + salt)
```

```
MIENTRAS cierto HACER

   PARA i = 1 HASTA count-1 HACER
      md_buf = H (md_buf)
   FINPARA

   SI key no completa ENTONCES
      key = bytes de md_buf
   FINSI

   SI iv no completo ENTONCES
      iv = bytes de md_buf
   FINSI

   SI iv Y key completos ENTONCES
      SALIR del MIENTRAS
   SINO
      md_buf = H (md_buf + data + salt)
   FINSI

FIN MIENTRAS
```

- **Ejemplos:**

Ejemplo 1: Encripción del texto: "*Hoy encripto.*", en AES 128 modo CBC con password "margarita", sin **salt**, mostrando el cifrado en codificación base 64.

Equivale a:[7]

OpenSSL>enc –aes128 –e –in hoy.txt –out hoyE.txt –k margarita –a –p -nosalt

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/evp.h>
#include "base64.h"
#define MAX_ENCR_LENGTH 1024
#define SUCCESS 0
#define FAILURE !SUCCESS
#define TAM_CLAVE   16
int saveEncryptedData(unsigned char *out, int len,unsigned char *where);
void mostrarKey(unsigned char key[]);
int
main()
{
    EVP_CIPHER_CTX ctx;
    unsigned char *nomArchE= "aes128cbc-evp-pwdE.txt";
    unsigned char in[MAX_ENCR_LENGTH] = "Hoy encripto";
    int inl;
    unsigned char out[MAX_ENCR_LENGTH];
    int outl, templ;

    unsigned char *pwd = "margarita";
    unsigned char key[TAM_CLAVE];
    unsigned char iv[TAM_CLAVE];
    printf("Clave AES: %d bytes.\n", EVP_CIPHER_key_length(EVP_aes_128_cbc()));
    printf("IV AES: %d bytes.\n", EVP_CIPHER_iv_length(EVP_aes_128_cbc()));

    EVP_BytesToKey(EVP_aes_128_cbc(), EVP_md5(), NULL, pwd, strlen(pwd),1, key, iv);
    printf("\nKey derivada:");
```

[7] El default de count es 1, y el default de hash puede ser md5 o sha1. Para saber cuál es, hay que ver el archivo openssl.cnf

```
    mostrarKey(key);
    printf("\nIV derivada:");
    mostrarKey(iv);

    /* Inicializar contexto */
    EVP_CIPHER_CTX_init(&ctx);
    EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);
    inl = strlen(in);
    EVP_EncryptUpdate(&ctx, out, &outl, in, inl);
    EVP_EncryptFinal_ex(&ctx, out + outl, &templ);
    outl +=templ;
    saveEncryptedData(out, outl,nomArchE);
    /* Borrar estructura de contexto */
    EVP_CIPHER_CTX_cleanup(&ctx);
    return EXIT_SUCCESS;
}
void mostrarKey(unsigned char key[])
{
    int i;
    for (i = 0; i < 16; i++)
    {
        printf("%0x", key[i]);
    }
}
```

Ejemplo 2: Descifrado de la cadena encriptada en AES 128 modo CBC con password “margarita” con salt, codificada en base64: **"U2FsdGVkX1+64WZZ0ZAl9LN1oywyeXTHqcZx/VXz5Kc="**

Equivale a:

OpenSSL>enc –aes128 –e –in margaE.txt –out margaD.txt –k margarita –a –p

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/evp.h>
#include "base64.h"
#define MAX_ENCR_LENGTH 1024
#define SUCCESS 0
#define FAILURE !SUCCESS
#define TAM_CLAVE   16
#define TAM_SALT    8
int obtenerEntrada(unsigned char *in, unsigned char *inB64);
int obtenerSalt(unsigned char *salt, int slen, unsigned char *in, int inl);
int saveDecryptedData(unsigned char *out, int len,unsigned char *where);
void mostrarKey(unsigned char key[]);
int
main()
{
    EVP_CIPHER_CTX ctx;
    unsigned char *nomArch = "aes128cbc-evp-pwd.txt";
    unsigned char inB64[] = "U2FsdGVkX1+64WZZ0ZAl9LN1oywyeXTHqcZx/VXz5Kc=";
    unsigned char in[MAX_ENCR_LENGTH];
    int inl;

    unsigned char out[MAX_ENCR_LENGTH];
    int outl, templ;

    unsigned char *pwd = "margarita";
    unsigned char key[TAM_CLAVE];
    unsigned char iv[TAM_CLAVE];
    unsigned char salt[TAM_SALT];

    inl = obtenerEntrada(in, inB64);
```

```
    if ((inl = obtenerSalt(salt, TAM_SALT, in, inl))==0)
    {
        printf("Entrada sin Salt");
        return EXIT_FAILURE;
    }
    EVP_BytesToKey(EVP_aes_128_cbc(), EVP_md5(), salt, pwd, strlen(pwd),1, key, iv);
    printf("\nKey derivada:");
    mostrarKey(key);
    printf("\nIV derivada:");
    mostrarKey(iv);

    /* Inicializar contexto*/
    EVP_CIPHER_CTX_init(&ctx);
    EVP_DecryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);
    EVP_DecryptUpdate(&ctx, out, &outl, in, inl);
    EVP_DecryptFinal_ex(&ctx, out + outl, &templ);
    outl +=templ;
    saveDecryptedData(out, outl,nomArch);
    /* Borrar estructura de contexto*/
    EVP_CIPHER_CTX_cleanup(&ctx);
    return EXIT_SUCCESS;
}
int obtenerEntrada(unsigned char *in, unsigned char *inB64)
{
    int inl, inl64;
    inl64 = strlen (inB64);
    inl = inl64 * 3 /4;
    fromBase64(in, inB64, inl64);
    inl = strlen(in);
    return inl;
}
int saveDecryptedData(unsigned char *out, int len,unsigned char *where)
{
    FILE *fileOut;

    if((fileOut = fopen(where, "w"))==NULL)
        return FAILURE;

    out[len]='\0';
    fprintf(fileOut,"%s", out);
    fclose(fileOut);
    return SUCCESS;
}
int obtenerSalt(unsigned char *salt, int slen, unsigned char *in, int inl)
{
    int finlen;
    if (strncmp(in, "Salted__",8)!=0)
        return 0;
    strncpy(salt, in + 8, slen);
    finlen = inl - 8 - slen;
    strncpy(in, in + 8 + slen, finlen);
    return (inl - 8 - slen);
}
```

7. Cifrado Asimétrico

7.1. rsa con rsa.h

- **Estructuras:**

  Se usa la estructura **RSA**

  typedef struct **rsa_st RSA;**

```
struct rsa_st
```

```
	{
	/* The first parameter is used to pickup errors where
	 * this is passed instead of aEVP_PKEY, it is set to 0 */
	int pad;
	long version;
	const RSA_METHOD *meth;
	/* functional reference if 'meth' is ENGINE-provided */
	ENGINE *engine;
	BIGNUM *n;          // public modulus
	BIGNUM *e;          // public exponent
	BIGNUM *d;          // private exponent
	BIGNUM *p;          // secret prime factor
	BIGNUM *q;          // secret prime factor
	BIGNUM *dmp1;// d mod (p-1)
	BIGNUM *dmq1;// d mod (q-1)
	BIGNUM *iqmp;// q^-1 mod p
	/* be careful using this if the RSA structure is shared */
	CRYPTO_EX_DATA ex_data;
	int references;
	int flags;

	/* Used to cache montgomery values */
	BN_MONT_CTX *_method_mod_n;
	BN_MONT_CTX *_method_mod_p;
	BN_MONT_CTX *_method_mod_q;

	/* all BIGNUM values are actually in the following data, if it is not
	 * NULL */
	char *bignum_data;
	BN_BLINDING *blinding;
	BN_BLINDING *mt_blinding;
	};
```

La estructura RSA puede contener tanto la clave privada como la pública. Está compuesta por varios campos de tipo BIGNUM[8], para guardar los valores que se utilizan para rsa: exponente, módulo, números p y q, etc. En las claves públicas, el exponente privado y el campo secreto relacionado están en NULL.

- **Funciones:**

```
RSA * RSA_new(void);
```

- Reserva espacio, inicializa y retorna una estructura RSA. Es equivalente a la invocación `RSA_new_method(NULL);`

```
RSA *RSA_new_method(ENGINE *engine);
```

- Reserva espacio, inicializa y retorna una estructura RSA, de forma que pueda usarse `engine` para las operaciones. Si `engine` es NULL; se usa el default.

```
void RSA_free(RSA *rsa);
```

- Libera la estructura RSA. La clave se elimina.

```
RSA *RSA_generate_key(int num, unsigned long e,
    void (*callback)(int,int,void *), void *cb_arg);
```

- Genera un par de claves y las retorna en una estructura RSA recientemente inicializada.

[8] Estructura de arreglo dinámico definido en bn.h:

```
typedef struct bignum_st BIGNUM;
struct bignum_st
	{
	BN_ULONG *d; /* Pointer to an array of 'BN_BITS2' bit chunks. */
	int top;     /* Index of last used d +1. */
	/* The next are internal book keeping for bn_expand. */
	int dmax;    /* Size of the d array. */
	int neg;     /* one if the number is negative */
	int flags;
	};
```

- El generador de números aleatorios debería ser seteado antes de llamar a `RSA_generate_key()`
- El tamaño del módulo se da en bits, y por seguridad debe elegirse un número mayor que 1024.
- El exponente público es e, un número impar, generalmente 3, 17 o 65537.
- La función de callback es para proveer un feedback que permita ver el progreso en la generación de la clave. Puede ser NULL.

```
int RSA_check_key(RSA *rsa);
```

- Valida las claves RSA generadas por `RSA_generate_key()`. Chequea que **p** y **q** son en efecto números primos, y que **n = p*q**. También chequea que **d*e = 1 mod (p-1*q-1)**, y que **dmp1**, **dmq1** y **iqmp** están seteados correctamente o son **NULL**.
- Retorna 1 si es una clave válida y 0 en caso contrario.

```
int RSA_print(BIO *bp, RSA *x, int offset);
int RSA_print_fp(FILE *fp, RSA *x, int offset);
```

- Imprimen los componentes de la clave RSA en hexadecimal, de manera legible. Las líneas de salida se indentan según el valor de offset indicado.

```
int RSA_public_encrypt(int flen, unsigned char *from, unsigned char *to, RSA
*rsa, int padding);
```

- Encripta los **flen** bytes de **from** usando la clave pública almacenada en **rsa** y guarda el texto cifrado en **to**.
- **to** debe apuntar a RSA_size(rsa) bytes.
- Padding puede ser: RSA_PKCS1_PADDING (PKCS #1 v1.5 padding), RSA_PKCS1_OAEP_PADDING, RSA_SSLV23_PADDING, RSA_NO_PADDING
- Retorna el tamaño de los datos encriptados o -1 en caso de error.

```
int RSA_private_decrypt(int flen, unsigned char *from, unsigned char *to, RSA
*rsa, int padding);
```

- Desencripta los **flen** bytes de **from** usando la clave privada almacenada en **rsa** y guarda el texto plano en **to**.
- Retorna la cantidad de bytes del texto recuperado o -1 en caso de error.

```
int RSA_size(const RSA *rsa);
```

- Retorna el tamaño del modulo RSA en bytes. Puede usarse para determiner cuánta memoria debe reservarse para un valor encriptado de RSA.

- **Ejemplos:**

  Ejemplo 1: Generación de clave privada, de 1024 bits de tamaño y exponente 65537, almacenada en hexadecimal.

```
int
main()
{
    RSA *rsaSt;

    rsaSt = RSA_new();
    rsaSt = RSA_generate_key(1024, 65537, NULL, NULL);
    if (RSA_check_key(rsaSt)==1)
    {
        printf("Key is ok");
    }
    saveKeyData(rsaSt, "privKeyRsa.txt");
    RSA_free(rsaSt);
    return EXIT_SUCCESS;
}
int saveKeyData(RSA *rsaSt, unsigned char *where)
```

```
    {
        FILE *out;
        out = fopen(where, "w");
        RSA_print_fp(out, rsaSt, 0);
        fclose(out);
        return SUCCESS;
    }
```

Ejemplo 2: Generación de clave privada, de 1024 bits de tamaño y exponente 65537, encripción y desencripción de la cadena *"Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente."*.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <openssl/rsa.h>
    #include <openssl/bio.h>
    #include <openssl/pem.h>

    #define SUCCESS 0
    #define FAILURE !SUCCESS
    #define MAX_LENGTH  1024
    int saveData(unsigned char *out, int len, unsigned char *where);
    int
    main()
    {
        RSA *rsaSt;
        unsigned char *msg = "Inteligencia, dame el nombre exacto de las cosas...
Que mi palabra sea la cosa misma, creada por mi alma nuevamente.";
        unsigned char *ciph;
        unsigned char *deciph;
        int msgLen, ciphLen, deciphLen;

        rsaSt = RSA_new();
        rsaSt = RSA_generate_key(1024, 65537, NULL, NULL);
        ciph = malloc(RSA_size(rsaSt));
        msgLen = strlen(msg);
        ciphLen = RSA_public_encrypt(msgLen, msg, ciph, rsaSt,
RSA_PKCS1_PADDING);
        saveData(ciph, ciphLen, "cifradoRsa.txt");

        deciph = malloc(MAX_LENGTH);
        deciphLen = RSA_private_decrypt(ciphLen, ciph, deciph, rsaSt,
RSA_PKCS1_PADDING);
        saveData(deciph, deciphLen, "decifradoRsa.txt");
        RSA_free(rsaSt);
        free(ciph);
        free(deciph);
        return EXIT_SUCCESS;
    }
```

6.1. rsa con evp.h

- **Estructuras:**

  Para claves RSA en evp se usan dos estructuras: **`EVP_PKEY`** y **`EVP_PKEY_CTX`**[9]**.**

  `typedef struct evp_pkey_st` **`EVP_PKEY`**`;`

```
struct evp_pkey_st
      {
      int type;
      int save_type;
```

[9] definida en evp_locl.h

```
       int references;
       const EVP_PKEY_ASN1_METHOD *ameth;
       ENGINE *engine;
       union  {
              char *ptr;
#ifndef OPENSSL_NO_RSA
              struct rsa_st *rsa; /* RSA */
#endif
#ifndef OPENSSL_NO_DSA
              struct dsa_st *dsa; /* DSA */
#endif
#ifndef OPENSSL_NO_DH
              struct dh_st *dh;   /* DH */
#endif
#ifndef OPENSSL_NO_EC
              struct ec_key_st *ec;     /* ECC */
#endif
              } pkey;
       int save_parameters;
       STACK_OF(X509_ATTRIBUTE) *attributes; /* [ 0 ] */
       } /* EVP_PKEY */;
```

```
typedef struct evp_pkey_ctx_st EVP_PKEY_CTX;
```

```
struct evp_pkey_ctx_st
       {
       /* Method associated with this operation */
       const EVP_PKEY_METHOD *pmeth;
       /* Engine that implements this method or NULL if builtin */
       ENGINE *engine;
       /* Key: may be NULL */
       EVP_PKEY *pkey;
       /* Peer key for key agreement, may be NULL */
       EVP_PKEY *peerkey;
       /* Actual operation */
       int operation;
       /* Algorithm specific data */
       void *data;
       /* Application specific data */
       void *app_data;
       /* Keygen callback */
       EVP_PKEY_gen_cb *pkey_gencb;
       /* implementation specific keygen data */
       int *keygen_info;
       int keygen_info_count;
       } /* EVP_PKEY_CTX */;
```

- **Funciones:**

```
EVP_PKEY_CTX *EVP_PKEY_CTX_new_id(int id, ENGINE *e);
```

- Reserva memoria e inicializa un contexto para algoritmo de clave pública especificado por id y e.
- Debe invocarse antes de `EVP_PKEY_new()` al generar una clave.

```
void EVP_PKEY_CTX_free(EVP_PKEY_CTX *ctx);
```

- Libera el contexto

```
EVP_PKEY *EVP_PKEY_new(void);
```

- Reserva memoria para una estructura **`EVP_PKEY`**

```
void EVP_PKEY_free(EVP_PKEY *key);
```

- Libera la memoria utilizada por la clave.

```
int EVP_PKEY_keygen_init(EVP_PKEY_CTX *ctx);
```

- Inicializa un contexto de algoritmo de clave pública para una operación de generación de clave.
- Retorna 1 si se efectúa con éxito ó 0 en caso contrario.

```
int EVP_PKEY_CTX_ctrl_str(EVP_PKEY_CTX *ctx, const char *type, const char *value);
```

- Permite aplicar alguna opción específica al contexto, expresada en forma de texto, tal como podría producirse al invocar por línea de comandos. Por ejemplo, si `type` es `"rsa_keygen_bits"` y `value` es `"1024"`, aplica al contexto la opción de que al generar la clave ésta sea de 1024 bits.[10] Si no se desea usar esta función, deben usarse las funciones `EVP_PKEY_CTX_set_rsa_...` que están descriptas en **rsa.h**.

```
int EVP_PKEY_keygen(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey);
```

- Efectúa la operación de generación de clave.

```
int EVP_PKEY_encrypt_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_decrypt_init(EVP_PKEY_CTX *ctx);
```

- Inicializa un contexto de algoritmo de clave pública para una operación de encripción/ desencripción.
- Retorna 1 si se efectúa con éxito ó 0 en caso contrario.

```
int EVP_PKEY_encrypt(EVP_PKEY_CTX *ctx, unsigned char *out, size_t *outlen, const
unsigned char *in, size_t inlen);
int EVP_PKEY_decrypt(EVP_PKEY_CTX *ctx, unsigned char *out, size_t *outlen, const
unsigned char *in, size_t inlen);
```

- Efectúa la operación de encripción/desencripción usando información de `ctx`. La entrada de datos a encriptar es **`in`**, de longitud **`inlen`**.
- Si **`out`** es NULL, el máximo tamaño del buffer de salida se escribe en el parámetro **`outlen`** (sirve para saber cuánta memoria hay que reservar para **`out`**).
- Si **`out`** no es NULL, y tiene reservada memoria suficiente, guardará los datos encriptados/desencriptados.
- Retorna 1 si se efectúa con éxito, ó 0 en caso contrario.

- **Ejemplos:**

Ejemplo 1: Generación de clave privada de 1024 bits y exponente 65537, generando un archivo formato PEM. Equivale a efectuar el comando:

```
genrsa –out claveprivada.txt –f4 1024
```

```
int
main()
{
    EVP_PKEY_CTX    *keyCtx;
    EVP_PKEY    *privKey = NULL;

    /*Inicializo contexto para clave*/
    keyCtx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);

    /*Inicializo un contexto para generación de clave privada*/
    EVP_PKEY_keygen_init(keyCtx);

    /*Inicializo estructura para clave*/
    privKey = EVP_PKEY_new();
    EVP_PKEY_CTX_ctrl_str(keyCtx, "rsa_keygen_bits", "1024");
    /*OJO
    EVP_PKEY_CTX_set_rsa_keygen_bits y
    EVP_PKEY_CTX_set_rsa_keygen_pubexp
    NO ESTAN EN EVP, son de RSA.H*/
    EVP_PKEY_CTX_ctrl_str(keyCtx, "rsa_keygen_pubexp", "65537");

    /*Genero clave privada*/
    EVP_PKEY_keygen(keyCtx, &privKey);
```

[10] Ver archivo rsa_pmeth.c

```
        saveKeyData(privKey, "claveprivada.txt", PRIVATE);

        /*Libero estructura de la clave privada*/
        EVP_PKEY_free(privKey);
        /*Libero contexto para la clave*/
        EVP_PKEY_CTX_free(keyCtx);
        return EXIT_SUCCESS;
    }
```

Donde la función para guardar la clave es:

```
    int saveKeyData(EVP_PKEY *key, unsigned char *where,unsigned char type)
    {
        BIO *out;
        int ret = SUCCESS;
        out = BIO_new_file(where, "w");
        switch(type)
        {
        case PRIVATE: PEM_write_bio_PrivateKey(out, key, NULL, NULL, 0, 0,
NULL);break;
        case PUBLIC: PEM_write_bio_PUBKEY(out, key);break;
        default: ret = FAILURE;
        }
        BIO_free(out);
        return ret;
    }
```

Ejemplo 2: Obtención de clave pública a partir de una clave privada en formato PEM. Equivale a efectuar el comando:

```
rsa –in claveprivada.txt –out clavepublica.txt –pubout
```

```
    int
    main()
    {
        EVP_PKEY_CTX    *keyCtx;
        EVP_PKEY    *privKey = NULL;

        /*Inicializo estructura para clave*/
        privKey = EVP_PKEY_new();
        if (privKey != NULL)
        {
            if (retrieveKeyData(&privKey, "claveprivada.txt", PRIVATE)==FAILURE)
                printf("Error al recuperar clave privada.");
            keyCtx = EVP_PKEY_CTX_new(privKey, NULL);
            saveKeyData(privKey, "clavepublica.txt", PUBLIC);
            /*Libero estructura de la clave privada*/
            EVP_PKEY_free(privKey);

           /*Libero contexto para la clave*/
            EVP_PKEY_CTX_free(keyCtx);
        }
        return EXIT_SUCCESS;
    }
```

Donde la función de recuperación de datos puede ser:

```
    int retrieveKeyData(EVP_PKEY **key, unsigned char *where, unsigned char type)
    {
        BIO *out;
        int ret = SUCCESS;
        out = BIO_new_file(where, "r");
        switch(type)
        {
```

```
        case PRIVATE: PEM_read_bio_PrivateKey(out, key, NULL,NULL);break;
        case PUBLIC: PEM_read_bio_PUBKEY(out, key, NULL, NULL);break;
        default: ret = FAILURE;
        }
        BIO_free(out);
        return ret;
    }
```

Ejemplo 3: Encripción con clave pública almacenada en `clavepublica.txt` del texto *"Inteligencia, dame el nombre exacto de las cosas... Que mi palabra sea la cosa misma, creada por mi alma nuevamente."*

Equivale al comando:

```
rsautl –in in.txt –out outRsa –inkey clavepublica.txt –pubin –encrypt –pkcs
```

y se puede desencriptar con el comando:

```
rsautl –in outRsa –inkey claveprivada.txt –decrypt -pkcs
```

```
int
main()
{
    EVP_PKEY_CTX    *ctx;
    EVP_PKEY    *pubKey = NULL;
    unsigned char *in = "Inteligencia, dame el nombre exacto de las cosas... Que
mi palabra sea la cosa misma, creada por mi alma nuevamente.";
    unsigned char *out;
    int outlen;

    /*Inicializo estructura para clave*/
    pubKey = EVP_PKEY_new();

    /*Obtengo la clave desde un archivo*/
    retrieveKeyData(&pubKey, "clavepublica.txt", PUBLIC);

    /*Inicializo un contexto para encriptar con clave privada*/
    ctx = EVP_PKEY_CTX_new(pubKey, NULL);

    EVP_PKEY_encrypt_init(ctx);

    /* Setear padding*/
    EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING);
    /* Establecer tamaño para buffer*/
    EVP_PKEY_encrypt(ctx, NULL, &outlen, in, strlen(in));
    out = malloc (outlen);
    EVP_PKEY_encrypt(ctx, out, &outlen, in, strlen(in));
    saveEncryptedDataBin(out, outlen, "outRsa");

    /*Libero estructura de la clave privada*/
    EVP_PKEY_free(pubKey);
    /*Libero contexto para la clave*/
    EVP_PKEY_CTX_free(ctx);

    free(out);
    return EXIT_SUCCESS;
}
```

8. Firma Digital.

8.1. firma y verificación con funciones de rsa.h y md5.h

- **Funciones:**

```
int RSA_private_encrypt(int flen, unsigned char *from, unsigned char *to, RSA
*rsa, int padding);
```

- Maneja firmas RSA en un bajo nivel.
- Firma **flen** bytes de **from** (en general un hash) usando la clave privada **rsa**, guardando la firma en **to**.
- La variable **to** debe apuntar a RSA_size(rsa) bytes de memory.
- El padding puede ser: RSA_PKCS1_PADDING o RSA_NO_PADDING.
- Retorna el total de bytes encriptados

```
int RSA_sign(int type, unsigned char *m, unsigned int m_len, unsigned char
*sigret, unsigned int *siglen, RSA *rsa);
```

- Firma el hash de un mensaje, de tamaño **m_len** usando la clave privada **rsa**. Guarda la firma en **sigret** y su tamaño en **siglen**.
- La variable **sigret** debe apuntar a RSA_size(**rsa**) bytes de memoria.
- El tipo **type** denota el algoritmo de digesto utilizado para generar **m** (**NID_sha1**, **NID_ripemd160** o **NID_md5**)
- Retorna 1 en caso de éxito, 0 en caso contrario.

```
int RSA_public_decrypt(int flen, unsigned char *from, unsigned char *to, RSA
*rsa,int padding);
```

- Maneja firmas RSA en un bajo nivel.
- Recupera el digesto del mensaje a partir de la firma **from** de **flen** bytes de longitud usando la clave pública **rsa**.
- La variable **to** debe apuntar a una sección de memoria suficientemente grande como para guardar el digesto del mensaje que es menor que **RSA_size(rsa) - 11**.
- El **padding** es el utilizado al firmar.
- Retorna el total de bytes obtenidos

```
int RSA_verify(int type, unsigned char *m, unsigned int m_len, unsigned char
*sigbuf, unsigned int siglen, RSA *rsa);
```

- Verifica que la firma **sigbuf** de tamaño **siglen** se corresponde con un digesto **m** dado, de longitud **m_len.**
- El tipo **type** denota el algoritmo de digesto utilizado para generar **m** (**NID_sha1**, **NID_ripemd160** o **NID_md5**)
- Retorna 1 en caso de éxito, 0 en caso contrario.

8.2. firma y verificación con funciones EVP_PKEY_... de evp.h

- **Funciones:**

```
int EVP_PKEY_sign_init(EVP_PKEY_CTX *ctx);
```

- Inicializa el contexto de algoritmo de clave pública para una operación de firma.

```
int EVP_PKEY_sign(EVP_PKEY_CTX *ctx, unsigned char *sig, size_t *siglen, const unsigned
char *tbs, size_t tbslen);
```

- Efectúa la firma de los datos **ya han sido procesados por una función de digesto `tbs`**, de longitud **`tbslen`**.
- Si `sig` es NULL, se obtiene el máximo tamaño de buffer de salida en la variable `siglen`.
- Si `sig` no es NULL, almacenará la firma correspondiente a **`tbs`**.

```
int EVP_PKEY_verify_init(EVP_PKEY_CTX *ctx);
```

- Inicializa el contexto de algoritmo de clave pública para una operación de verificación de firma. Después de la inicialización pueden setearse los parámetros adecuados para la misma (padding, algoritmo de hash, etc)

```
int EVP_PKEY_verify(EVP_PKEY_CTX *ctx, unsigned char *sig, size_t siglen, const unsigned
char *md, size_t mdlen);
```

- Hace una operación de verificación usando **`ctx`**. La firma se especifica en el argumento **`sig`**, mientras que los datos originales contra los que se verifica la firma están en el argumento **`md`**. Nótese que la verificación se hace sobre los datos originales **sometidos a digesto.**
- Retorna 1 si la verificación fue exitosa y 0 en caso contrario.

- **Ejemplos:**

  Ejemplo 1: Firma de archivo “picture.jpg”, usando como algoritmo de hash md5.

  Equivale a

```
OpenSSL> dgst –md5 –out pictureH picture.jpg
OpenSSL> rsautl –in pictureH –inkey claveprivada.txt –out firma –sign
```

  Y se puede verificar con:

```
OpenSSL> dgst –md5 –verify clavepublica.txt –signature firma picture.jpg
```

```
int
main()
{
    EVP_PKEY *privK;
    EVP_PKEY_CTX    *ctx;
    unsigned char *data = NULL;
    unsigned char *sig;
    size_t datalen;
    size_t siglen;

    /*Recupero la clave privada para firmar con ella*/
    privK = EVP_PKEY_new();
    retrieveKeyData(&privK, "claveprivada.txt", PRIVATE);
    ctx = EVP_PKEY_CTX_new(privK, NULL);
    /*Inicialización de estructura para firma*/
    EVP_PKEY_sign_init(ctx);
    /*Se obtienen los datos del archivo*/
    retrieveData(&data, &datalen, "picture.jpg");
    /*Se obtiene el digesto de los datos*/
    hash(&data, &datalen);
    /*Se setea padding y algoritmo de hash en el contexto*/
    EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING);
    EVP_PKEY_CTX_set_signature_md(ctx, EVP_md5());
    /*Se determina longitud del buffer*/
    EVP_PKEY_sign(ctx, NULL, &siglen, data, datalen);
    sig = OPENSSL_malloc(siglen);
    if (EVP_PKEY_sign(ctx, sig, &siglen, data, datalen)<=0)
        printf("Error al firmar...");
    else
        saveData(sig, siglen, "firma");
    EVP_PKEY_free(privK);
    EVP_PKEY_CTX_free(ctx);
    OPENSSL_free(sig);
    return EXIT_SUCCESS;

}
```

8.3. firma con EVP_Sign... y verificación con EVP_Verify... de evp.h

- **Estructuras:**

  Se utilizan EVP_MD_CTX y EVP_PKEY.

- **Funciones:**

```
int EVP_SignInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
```

- Inicializa el contexto de un algoritmo de firma. El contexto ya debe estar inicializado con `EVP_MD_CTX_init()`.

- Retorna 1 si termina con éxito, ó 0 en caso contrario.

```
int EVP_SignUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
```

- Efectúa el hash de **cnt** bytes de datos en **d** y los ubica en el contexto **ctx**.
- Retorna 1 si termina con éxito, ó 0 en caso contrario.

```
int EVP_SignFinal(EVP_MD_CTX *ctx,unsigned char *sig,unsigned int *s, EVP_PKEY *pkey);
```

- Firma los datos en **ctx** usando la clave privada **pkey** y ubica la firma en **sig**. El número de datos escritos se escribe como entero en **s**, y es como máximo `EVP_PKEY_size(pkey)`
- Retorna 1 si termina con éxito, ó 0 en caso contrario.

```
int EVP_PKEY_size(EVP_PKEY *pkey);
```

- Retorna el máximo tamaño de una firma en bytes. El tamaño real podría ser menor.

```
int EVP_VerifyInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
```

- Inicializa el contexto de verificación para usar un tipo de algoritmo de digesto. El contexto ya debe estar inicializado con `EVP_MD_CTX_init()`.
- Retorna 1 si termina con éxito, ó 0 en caso contrario.

```
int EVP_VerifyUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
```

- Efectúa el hash de **cnt** bytes de datos en **d** y los ubica en el contexto **ctx**.
- Retorna 1 si termina con éxito, ó 0 en caso contrario.

```
int EVP_VerifyFinal(EVP_MD_CTX *ctx,unsigned char *sigbuf, unsigned int siglen,EVP_PKEY
*pkey);
```

- Verifica los datos en **ctx** usando la clave pública **pkey** contra **sigbuf**.
- Retorna 1 si termina con éxito, ó 0 en caso contrario.

- **Ejemplos:**

Ejemplo 1: Firma de archivo “picture.jpg”, usando como algoritmo de hash md5.

Equivale a

```
OpenSSL> dgst –md5 –out pictureH picture.jpg
OpenSSL> rsautl –in pictureH –inkey claveprivada.txt –out firma –sign
```

Y se puede verificar con:

```
OpenSSL> dgst –md5 –verify clavepublica.txt –signature firma picture.jpg
```

```
int
main()
{
    EVP_MD_CTX *ctx;
    EVP_PKEY    *privKey;
    unsigned char *data;
    unsigned char *sig;
    size_t  datalen, siglen;

    /*Obtener datos a firmar*/
    retrieveData(&data, &datalen, "picture.jpg");

    /*Obtener clave privada*/
    privKey = EVP_PKEY_new();
    retrieveKeyData(&privKey, "claveprivada.txt", PRIVATE);

    /*Crea contexto*/
    ctx = EVP_MD_CTX_create();

    /*Preparar para Sign*/
```

```
    EVP_SignInit_ex(ctx, EVP_md5(), NULL);

    /*Firmar*/
    EVP_SignUpdate(ctx, data, datalen);
    siglen = EVP_PKEY_size(privKey);
    sig = malloc(siglen);
    EVP_SignFinal(ctx, sig, &siglen, privKey);
    saveData(sig, siglen, "firma");
    free(sig);
    EVP_PKEY_free(privKey);
    EVP_MD_CTX_cleanup(ctx);
    return EXIT_SUCCESS;
}
```

Ejemplo 2: Verificación de firma "firma" respecto de archivo "picture.jpg", usando como algoritmo de hash md5.

```
int
main()
{
    EVP_MD_CTX *ctx;
    EVP_PKEY    *pubKey;
    unsigned char *data;
    unsigned char *sig;
    size_t  datalen, siglen;

    /*Obtener datos a verificar*/
    retrieveData(&data, &datalen, "picture.jpg");
    retrieveData(&sig, &siglen, "firma");

    /*Obtener clave publica*/
    pubKey = EVP_PKEY_new();
    retrieveKeyData(&pubKey, "clavepublica.txt", PUBLIC);

    /*Crear contexto*/
    ctx = EVP_MD_CTX_create();
    /*Preparar para Verificar*/
    EVP_VerifyInit_ex(ctx, EVP_md5(), NULL);

    /*Verificar*/
    EVP_VerifyUpdate(ctx, data, datalen);
    if (EVP_VerifyFinal(ctx, sig, siglen, pubKey)==SUCCESS)
        printf("Verify OK...");
    else
        printf("Verify ERROR...");
    EVP_PKEY_free(pubKey);
    EVP_MD_CTX_cleanup(ctx);
    return EXIT_SUCCESS;
}
```

9. Sitios consultados:

- **OpenSSL:** www.openssl.org
- **Tutorial OpenSSL/EVP (Juan Segarra Montesinos):** http://spi1.nisu.org/recop/al01/segarra/index.html