Introducción

OpenSSL

OpenSSL es una implementación *open source* de SSL, que permite realizar tareas criptográficas como calcular hashes criptográficos, cifrar con algoritmos de cifrado simétrico, en flujo o utilizando múltiples modos de operación, generar claves asimétricas, utilizar los algoritmos para firmar, certificar y revocar claves, y manejar formatos de certificados.

OpenSSL ofrece implementaciones de las funciones necesarias para cada algoritmo criptográfico, en headers como `md5.h` o `aes.h`, pero no presentan una interfaz uniforme. Para esto existe la librería `evp.h`, una abstracción por sobre estas: las funciones que comienzan con `EVP_DigestSign` y `EVP_DigestVerify` implementan firmas digitales y MACs, mientras que `EVP_Encrypt` es utilizado para encripción simétrica y `EVP_Digest` para hashes.

También, para manejar cuestiones de entrada y salida, `bio.h` contiene una serie de funciones de abstracción de I/O que permiten manejar conexiones SSL, conexiones de red no cifradas 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`, que son utilizadas para escribir y leer en formato PEM). En la documentación indica que se pueden usar las funciones de PEM utilizando FILE en lugar de BIO, pero suele presentar problemas.

La idea de este documento es presentar la forma 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, pública y firma digital. Aún así, la cantidad de funciones que ofrece OpenSSL se extiende mucho más de lo que este documento puede abarcar, por lo que se recomienda leer la documentación (utilizando `man` o online).

Para usar un algoritmo primero es necesario obtener la implementación a usar, explícitamente o implícitamente (mediante funciones como `EVP_sha256()` o `EVP_aes_128_cbc()`). Para hacerlo explícitamente se pueden hacer uso de las funciones `EVP_*_fetch`, que toma como parámetros el contexto de la librería (`NULL` para utilizar el default), un identificador y un string de query, y devuelve un objeto `EVP_*` que puede ser usado en otras funciones. En caso de usarse de forma repetida, es recomendable usar fetch explícito.

```
1 EVP_MD *md = EVP_MD_fetch(NULL, "SHA2-256", NULL);
2 EVP_MD_free(md)
3
4 EVP_CIPHER *cipher = EVP_CIPHER_fetch(NULL, "AES-128-CBC", NULL);
5 EVP_CIPHER_free(cipher);
```

DES

> [!WARNING] DES DES es considerado obsoleto, por lo que el proveedor default de OpenSSL no los incluye. En cualquier lado de la practica o TP, si ven DES, se refiere a 3DES (“DES-EDE-ECB”, “DES-EDE-CFB8”, “DES-EDE-CFB1”, etc.)

Java Cryptography Architecture

La JCA es parte del lenguaje Java, como parte del API de seguridad, desde la versión JDC 1.1. Es una especificación del lenguaje que especifica interfaces y clases abstractas que sirven de base para las implementaciones concretas de algoritmos criptográficos.

Esta diseñada de acuerdo con dos principios:

- Independencia e interoperabilidad de las implementaciones
- Independencia y extensibilidad de los algoritmos.

La independencia de las implementaciones se consigue empleando una arquitectura basada en proveedores. Un proveedor se refiere a un paquete o conjunto de paquetes que proporciona una implementación concreta de funcionalidades criptográficas de la API de seguridad de Java.

La interoperabilidad de las implementaciones permite que cada una de ellas pueda usarse con las demás.

Algunos ejemplos de **providers**:

- **SunJCE** (Java Cryptography Extension)
- **BC** (Proyecto Bouncy Castle).

La independencia de los algoritmos se consigue definiendo *tipos de servicios criptográficos* y las clases (**engines**) que proporcionen la funcionalidad de estos servicios.

La extensibilidad de los algoritmos quiere decir que nuevos algoritmos que encajan dentro de alguno de los tipos soportados puedan ser añadidos fácilmente.

Algunos ejemplos de **engines**:

- `SecureRandom`
- `MessageDigest`
- `Signature`
- `Cipher`
- `Mac`
- `KeyFactory`
- `SecretKeyFactory`
- `KeyPairGenerator`
- `KeyGenerator`
- `KeyAgreement`

Paquetes

Package java.security Provee las clases e interfaces para el framework de seguridad, acá mostramos algunas:

- **Interfaces**:
  - `AlgorithmConstraints`
  - `Key`
  - `KeyStore.Entry`
  - `KeyStore.LoadStoreParameter`
  - `KeyStore.ProtectionParameter`
  - `Policy.Parameters`
  - `PrivateKey`
  - `PublicKey`
- **Clases**:
  - `KeyFactory`
  - `KeyPair`
  - `KeyPairGenerator`
  - `KeyStore`
  - `MessageDigest`
  - `SecureRandom`
  - `Signature`
  - `Timestamp`

Package javax.crypto Provee las clases e interfaces para operaciones criptográficas.

- **Interfaces**:
  - `SecretKey`
- **Clases**:
  - `Cipher`
  - `EncryptedPrivateKeyInfo`
  - `KeyAgreement`
  - `KeyGenerator`
  - `Mac`
  - `SecretKeyFactory`

Instalación de los proveedores

El proveedor `SunJCE` es el proveedor estándar de la distribución Java, por lo cual no es necesario instalarlo ni registrarlo, y soporta algoritmos para distintas funcionalidades criptográficas, entre otras:

- Cifrado simétrico y asimétrico
- Distribución de claves
- Generación de claves
- MAC y Hash: HMAC-MD5, HMAC-SHA1
- Firma Digital
- Generación de Certificados Digitales

El proveedor Bouncy Castle se obtiene del sitio del proyecto, pero los ejemplos que están en este apunte usan `SunJCE`.

Hash

OpenSSL

Para digests de mensajes se usan dos estructuras, `EVP_MD` y `EVP_MD_CTX`, mediante las siguientes funciones:

- `EVP_MD_CTX *EVP_MD_CTX_new(void)` reserva espacio, inicializa y retorna un contexto para digest.
- `int EVP_MD_CTX_free(EVP_MD_CTX *ctx)` elimina y libera la memoria utilizada por el contexto.
- `int EVP_MD_CTX_reset(EVP_MD_CTX *ctx)` resetea el contexto, permitiendo reutilizarlo. _____
- `int EVP_Digest(const void *data, size_t count, unsigned char *md, unsigned int *size, const EVP_MD *type, ENGINE *impl)` es un wrapper de `DigestInit_ex`, `Update` y `Final_ex`. Hashea *count* bytes de *data* usando el algoritmo *type* del `ENGINE` *impl*. El hash es guardado en *md* y se guarda su longitud en *size* si es diferente de `NULL`, escribiendo como máximo `EVP_MAX_MD_SIZE` bytes. Esto es útil si todo el mensaje puede ser cargado en memoria, procesandolo todo en una llamada.

- `int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl)` setea el `ctx` para usar el algoritmo `type`.
- `int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, size_t cnt)` hashea *cnt* bytes de *d* al contexto `ctx`. Puede llamarse varias veces según sea necesario.
- `int EVP_DigestFinal_ex()` devuelve el hash guardado en *ctx* y lo pone en *md*. Si *s* es diferente de `NULL` se guarda la longitud del hash en bytes, escribiendo como maximo `EVP_MAX_MD_SIZE`, a menos que se haya cambiado el tamaño del hash si la implementación lo permite.

Ejemplos

SHA-256 EVP_Digest Calcula el hash de un array en una sola llamada utilizando SHA-256.

```c
int main(int argc, char* argv[]) {
    char msg[] = "hace mucho calor hoy";
    unsigned char md[EVP_MAX_MD_SIZE];
    unsigned int md_len;

    const EVP_MD* md_type = EVP_MD_fetch(NULL, "SHA256", NULL);

    EVP_Digest(msg, strlen(msg), md, &md_len, md_type, NULL);

    for (unsigned int i = 0; i < md_len; i++) {
        printf("%02x", md[i]);
    }

    return EXIT_SUCCESS;
}
```

SHA-256 EVP_Update Calcula el hash del archivo binario de si mismo utilizando el algoritmo SHA-256.

```c
int main(int argc, char* argv[]) {
    unsigned char md[EVP_MAX_MD_SIZE];
    EVP_MD_CTX* mdctx;
    unsigned int md_len;
    char buffer[BUFSIZ];
    FILE* file = fopen(argv[0], "r");

    const EVP_MD* md_type = EVP_MD_fetch(NULL, "SHA256", NULL);

    mdctx = EVP_MD_CTX_new();

    EVP_DigestInit_ex(mdctx, md_type, NULL);
```

```
13
14      size_t read = 0;
15      while ((read = fread(buffer, 1, BUFSIZ, file))) {
16          EVP_DigestUpdate(mdctx, buffer, read);
17      }
18
19      EVP_DigestFinal_ex(mdctx, md, &md_len);
20
21      for (unsigned int i = 0; i < md_len; i++) {
22          printf("%02x", md[i]);
23      }
24
25      return EXIT_SUCCESS;
26  }
```

JCA

Las clases que se usan sor `MessageDigest` para generar el hash de una secuencia de bytes, y `Mac` para general un hash con clave.

Ejemplos

SHA-256 Utilizando el método `update` de `MessageDigest` se agregan bytes al hash, mientras que con el método `digest` obtenemos el hash resultante.

```
1  public class ejemploHash {
2      public static void main(String[] args) throws
           NoSuchAlgorithmException {
3          // Definimos texto plano al que se aplica el hash
4          byte[] texto_plano = "hace mucho calor hoy".getBytes();
5          // Generacion del hash con algoritmo MD5
6          MessageDigest sha = MessageDigest.getInstance("SHA-256");
7          sha.update(texto_plano);
8          byte[] resultado = sha.digest();
9          System.out.println("Hash: " + HexFormat.of().formatHex(
               resultado));
10     }
11 }
```

HMAC SHA-256 Ahora hay que tener en cuenta que primero debe generarse la clave, para luego efectuar el MAC. Las clases que se usan son `KeyGenerator` para generar claves secretas para un algoritmo específico, `SecretKey` es la clave criptográfica secreta, y `Mac` para generar un hash con clave.

```
1  public class ejemplo_hmac {
2      public static void main(String[] args) throws Exception {
```

```
3           //Se genera una clave
4           KeyGenerator kg = KeyGenerator.getInstance("HmacSha256");
5           SecretKey sk = kg.generateKey();
6           //Se efectuará MAC con algoritmo HMAC MD5
7           Mac mac = Mac.getInstance("HmacMD5");
8           //Se inicializa con la clave:
9           mac.init(sk);
10          //Se efectúa el HMAC
11          byte[] texto_plano = "hace mucho calor hoy".getBytes();
12          byte[] resultado = mac.doFinal(texto_plano);
13          System.out.println("SHA-256: " + HexFormat.of().formatHex(
                resultado));
14      }
15  }
```

Cifrado Simétrico

OpenSSL

Funciones

Para instanciar un contexto:

- `EVP_CIPHER_CTX *EVP_CIPHER_CTX_new()` inicializa un contexto para cifrado.
- `void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *ctx)` limpia toda la información del contexto y libera toda la memoria asociada, incluyendo *ctx*. Debería ser llamada cuando se terminan las operaciones para evitar que información sensible siga en memoria.

Para cifrar:

- `int EVP_EncryptInit_ex2(EVP_CIPHER_CTX *ctx, const EVP_CIPHER * type, const unsigned char *key, const unsigned char *iv, const OSSL_PARAM params[])` configura el *ctx* para usar el algoritmo *type*. *key* es la clave simétrica e *iv* es el IV a usar. La longitud de la key e iv dependen del algoritmo a utilizar, y pueden ser obtenidos utilizando `EVP_CIPHER_key_length(type)` y `EVP_CIPHER_iv_length(type)`, en caso de usar ECB, IV es NULL. Si no se especifica un IV para `EVP_CIPH_GCM_MODE`,

este sera generado internamente. Se puede invocar `EncryptInit_ex2` con `NULL` en todos los campos menos en *type* y luego actualizarlos, en caso que los parámetros default del algoritmo no sean apropiados.

- `int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, const unsigned char *in, int inl)` cifra *inl* bytes de *in* y los escribe a *out*. Si *in* y *out* apuntan a la misma ubicación la encripción se hará en el lugar, en caso contrario se debe asegurar que los dos buffers no se pisen. Esta función debe ser llamada multiples veces para cifrar bloques sucesivos de data
- `int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl)` se encarga de finalizar la encripción: Si el padding esta habilitado, cifra lo que quedo en un block parcial, usando PKCS como padding, y la guarda en *out*, que debe tener espacio suficiente para al menos un bloque, y guarda los bytes escritos en *outl*.

Para descifrar:

- `int EVP_DecryptInit_ex2(EVP_CIPHER_CTX *ctx, const EVP_CIPHER * type, const unsigned char *key, const unsigned char *iv, const OSSL_PARAM params[])`
- `int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, const unsigned char *in, int inl)`
- `int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl)`

Para cifrar o descifrar, dependiendo del valor de `enc` (1 para cifrar, 0 para descifrar, -1 si ya se seteo el valor en una llamada anterior):

- `int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl, const unsigned char *key, const unsigned char *iv, int enc)`
- `int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, const unsigned char *in, int inl)`

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

- `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, que es descartado cuando se descifrar. 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.

Ejemplos

Estas son algunas de las macros útiles para obtener información del algoritmo en base al `EVP_CIPHER` y `EVP_CIPHER_CTX`.

```
#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)
```

AES-128-CBC Cifra un mensaje usando AES-128-CBC, con una key e IV fijos.

```
int main(int argc, char* argv[]) {
    unsigned char* in = "Lorem ipsum dolor sit amet, consectetur
        adipiscing elit.";
    int inl = strlen(in);
    unsigned char out[BUFSIZ];
    int outl, templ;
    unsigned char* k = "0123456789012345"; // 128 bits = 16 bytes
    unsigned char iv[] = "5432109876543210";

    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    const EVP_CIPHER* cipher = EVP_CIPHER_fetch(NULL, "AES-128-CBC",
        NULL);
```

```
11
12      EVP_EncryptInit_ex2(ctx, cipher, k, iv, NULL);
13
14      EVP_EncryptUpdate(ctx, out, &outl, in, inl);
15
16      printf("Encripta primero %d bytes\n", outl);
17
18      EVP_EncryptFinal(ctx, out + outl, &templ);
19      printf("Finalmente se encriptan %d bytes\n", templ);
20
21      for (int i = 0; i < outl + templ; i++) {
22          printf("%02x", out[i]);
23      }
24
25      return EXIT_SUCCESS;
26  }
```

Passwords

Para generar la clave y el vector de inicialización a partir de una password, vamos a hacer uso de las siguientes dos funciones de OpenSSL:

- `int PKCS5_PBKDF2_HMAC(const char *pass, int passlen, const unsigned char *salt, int saltlen, int iter, const EVP_MD *digest, int keylen, unsigned char *out)`
- `int PKCS5_PBKDF2_HMAC_SHA1(const char *pass, int passlen, const unsigned *salt, int saltlen, int iter, int keylen, unsigned char *out)`

El parámetro *pass* es la password a usarse, y si *passlen* es -1 se calcula la longitud usando `strlen()`. `salt` es el salt a usar, y si es `NULL` entonces *saltlen* tiene que ser 0. *iter* es la cantidad de iteraciones a realizarse y se recomienda un mínimo de 1000. `digest` indica la función de hash a utilizarse, `PKCS5_PBKDF2_HMAC_SHA1` es equivalente a llamar a `PKCS5_PBKDF2_HMAC` usando SHA1. La key derivada va a ser escrita en *out*, y el tamaño del buffer es indicado mediante *keylen*.

Una vez obtenido el *out*, para utilizarlo para encripción simétrica hay que dividirlo correctamente en key e iv.

Ejemplo El siguiente ejemplo es equivalente a utilizar el comando:

```
openssl enc -aes-128-cbc -k "password" -pbkdf2 -S "00000000"
```

```
int main(int argc, char* argv[]) {
    EVP_CIPHER* cipher = EVP_CIPHER_fetch(NULL, "AES-128-CBC", NULL);

    const int keylen = EVP_CIPHER_key_length(cipher);
    const int ivlen = EVP_CIPHER_iv_length(cipher);

    unsigned char* key_iv_pair = malloc(keylen + ivlen);

    const unsigned char salt[8] = {0};

    PKCS5_PBKDF2_HMAC("password",
                      -1, salt, sizeof(salt),
                      10000, EVP_sha256(), keylen + ivlen, key_iv_pair)
                          ;

    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex2(ctx, cipher, key_iv_pair, key_iv_pair + keylen,
        NULL);

    unsigned char buffer[BUFSIZ];
    size_t read = 0;
    int outl = 0;
    while ((read = fread(buffer, 1, BUFSIZ, stdin))) {
        EVP_EncryptUpdate(ctx, buffer, &outl, buffer, read);
        fwrite(buffer, 1, outl, stdout);
    }

    EVP_EncryptFinal_ex(ctx, buffer, &outl);
    fwrite(buffer, 1, outl, stdout);

    free(key_iv_pair);
    EVP_CIPHER_CTX_free(ctx);
    EVP_CIPHER_free(cipher);

    return EXIT_SUCCESS;
}
```

JCA

Las clases (**engines**) que se usan son:

- `SecureRandom`: Para generar números pseudoaleatorios
- `Cipher`: Luego de ser inicializada con la clave correspondiente, se usa para el cifrado/descifrado de la información
- `KeyFactory`: Para convertir claves criptográficas opacas tipo `Key` en especificaciones de clave

y viceversa

- `SecretKeyFactory`: Para convertir claves criptográficas tipo `SecretKey` en especificaciones de clave y viceversa. Solo para claves simétricas.
- `KeyGenerator`: Para generar claves secretas para un algoritmo específico.

El primer paso es generar una clave. Para ello, se genera una instancia de un generador de claves para el algoritmo en particular que se quiera usar (DES, AES, etc.).

Luego debe crearse una instancia del "cifrador" `Cipher`. Al hacerlo, se establece el nombre del algoritmo, el modo de cifrado y el modo de padding. Los valores defaults son "AES", "ECB" y "PKCS5Padding".

Finalmente se efectúa el cifrado (o descifrado). Debe previamente inicializarse el "cifrador" con los parámetros `ENCRYPT_MODE` o `DECRYPT_MODE`. ### Ejemplos

AES Para calcular el cifrado de un mensaje utilizando AES, primero tenemos que tener una clave (en este caso generada random en el momento), y dependiendo del modo de encadenamiento que seleccionamos se requiere un IV (por ejemplo, ECB no lo necesita).

```
1  public class ejemplo_aes {
2      public static void main(String[] args) throws Exception {
3          // Se genera la clave para AES
4          KeyGenerator keygen = KeyGenerator.getInstance("AES");
5          SecretKey aesKey = keygen.generateKey();
6          // Se genera el vector de inicialización
7          byte[] ivBytes = new byte[16];
8          new SecureRandom().nextBytes(ivBytes);
9          IvParameterSpec iv = new IvParameterSpec(ivBytes);
10
11         // Se genera instancia de Cipher
12         Cipher aesCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
13         // Se inicializa el cifrador para poder cifrar con la clave
14         aesCipher.init(Cipher.ENCRYPT_MODE, aesKey, iv);
15         // Texto a cifrar.
16         byte[] cleartext = "Contenido de prueba".getBytes();
17         // Se cifra
18         byte[] ciphertext = aesCipher.doFinal(cleartext);
19         System.out.println("El cifrado es: " + Base64.getEncoder().
               encodeToString(ciphertext));
20         System.out.println("Ahora descifra...");
21         // Se descifra
22         aesCipher.init(Cipher.DECRYPT_MODE, aesKey, iv);
23         byte[] cleartext_out = aesCipher.doFinal(ciphertext);
24         System.out.println("El descifrado es: " + new String(
               cleartext_out, StandardCharsets.UTF_8));
25     }
26 }
```

AES con PBKDF2 Este es un ejemplo de como usar PBKDF2 para obtener una key e iv en base a una password. El algoritmo devuelve un buffer de la longitud solicitada con bytes "random", y luego hay que dividirlo correctamente en dos partes, key e IV.

```java
public class ejemplo_aes_pbkdf2 {
    public static void main(String[] args) throws Exception {
        String password = "password";
        final int keyLength = 256;
        final int ivLength = 128;
        final int iterations = 10000;
        byte[] salt = new byte[]{0};

        // Generamos una secuencia de bytes a partir de la clave
        SecretKeyFactory skf = SecretKeyFactory.getInstance("
            PBKDF2WithHmacSHA256");

        PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
            iterations, keyLength + ivLength);
        byte[] randBytes = skf.generateSecret(spec).getEncoded();
        // La dividimos en key e iv
        byte[] keyBytes = copyOf(randBytes, keyLength / 8);
        byte[] ivBytes = copyOfRange(randBytes, keyLength / 8,
            randBytes.length);
        // Instanciamos la key e iv
        Key key = new SecretKeySpec(keyBytes, "AES");
        IvParameterSpec iv = new IvParameterSpec(ivBytes);

        // Se genera instancia de Cipher
        Cipher aesCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        // Se inicializa el cifrador para poder descifrar con la clave
        aesCipher.init(Cipher.ENCRYPT_MODE, key, iv);
        // Texto a cifrar.
        byte[] cleartext = "Contenido de prueba".getBytes();
        // Se cifra
        byte[] ciphertext = aesCipher.doFinal(cleartext);
        System.out.println("El cifrado es: " + Base64.getEncoder().
            encodeToString(ciphertext));
        System.out.println("Ahora descifra...");
        // Se descifra
        aesCipher.init(Cipher.DECRYPT_MODE, key, iv);
        byte[] cleartext_out = aesCipher.doFinal(ciphertext);
        System.out.println("El descifrado es: " + new String(
            cleartext_out, StandardCharsets.UTF_8));
    }
}
```

Cifrado Asimétrico

OpenSSL

De vuelta vamos a hacer uso de las funciones `EVP`, esta vez con la estructura `EVP_PKEY`, que es una estructura genérica que puede contener varios tipos de claves asimétricas, sea la clave publica, la privada o las dos, y según las claves presentes puede ser usada para firmar, verificar firmas, derivar keys, etc.

- `EVP_PKEY_CTX *EVP_PKEY_CTX_new(EVP_PKEY *pkey, ENGINE *e)`
- `EVP_PKEY_CTX *EVP_PKEY_CTX_new_id(int id, ENGINE *e)`: Normalmente usado no hay una `EVP_PKEY` asociada a las operaciones, como en la generación de clave. Los IDs están definidos como `EVP_PKEY_RSA`, `EVO_PKEY_DSA`, `EVP_PKEY_DH`, ...
- `void EVP_PKEY_CTX_free(EVP_PKEY_CTX *ctx)`

- `EVP_PKEY *EVP_PKEY_new(void)`: Devuelve una estructura nueva.
- `void EVP_PKEY_free(EVP_PKEY *key)`: Libera la memoria utilizada.

- `int EVP_PKEY_keygen_init(EVP_PKEY_CTX *ctx)`: Inicializa el contexto para la generación de claves/
- `int EVP_PKEY_CTX_ctrl(EVP_PKEY_CTX *ctx, int keytype, int optype, int cmd, int p1, void *p2)`: Permite aplicar alguna opción específica al contexto, expresada en forma de texto, tal como seria por linea de comando. Por ejemplo, si *type* es `"rsa_keygen_bits"` y *value* es `"1024"`, aplica al contexto la opción de que al generar la clave esta sea de 1024 bits. También pueden usarse las funciones `EVP_PKEY_set_*`.
- `int EVP_PKEY_generate(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey)`: genera la clave, y los parametros de key o key son guardados en *ppkey*. Si `*ppkey` es `NULL`, se asigna memoria y luego debe ser liberada.

- `int EVP_PKEY_decrypt_init_ex(EVP_PKEY_CTX *ctx, const OSSL_PARAM params[])`: Inicia el contexto para descifrar.

- `int EVP_PKEY_decrypt(EVP_PKEY_CTX *ctx, unsigned char *out, size_t *outlen, const unsigned char *in, size_t inlen)`: Performa la operación de descifrar usando *ctx*. Si *out* es `NULL` el tamaño mínimo requerido para el output buffer es escrito en `*outlen`, caso contrario se guarda en `*outlen` la cantidad de data descifrar que se escribió.

- `int EVP_PKEY_encrypt_init_ex(EVP_PKEY_CTX *ctx, const OSSL_PARAM params[])`
- `int EVP_PKEY_encrypt(EVP_PKEY_CTX *ctx, unsigned char *out, size_t *outlen, const unsigned char *in, size_t inlen)`: Si *out* es `NULL` entonces el tamaño máximo del output buffer se guarda en *outlen*.

Ejemplos

Generación de Clave RSA El siguiente programa es equivalente al comando:

```
openssl genrsa -out private_key.txt -f4 1024
```

```
#define SUCCESS 0
#define FAILURE 1

enum { PRIVATE, PUBLIC };

int saveKeyData(EVP_PKEY* key, const char* file, unsigned char type) {
    BIO* out;
    int ret = SUCCESS;
    out = BIO_new_file(file, "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;
}

int main(int argc, char* argv[]) {
```

```
25      EVP_PKEY_CTX* ctx;
26      EVP_PKEY* pkey = NULL;
27
28      ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
29
30      EVP_PKEY_keygen_init(ctx);
31
32      EVP_PKEY_CTX_ctrl_str(ctx, "rsa_keygen_bits", "1024");
33
34      EVP_PKEY_CTX_ctrl_str(ctx, "rsa_keygen_pubexp", "65537");
35
36      EVP_PKEY_keygen(ctx, &pkey);
37
38      saveKeyData(pkey, "private_key.txt", PRIVATE);
39
40      EVP_PKEY_free(pkey);
41
42      EVP_PKEY_CTX_free(ctx);
43
44      return EXIT_SUCCESS;
45  }
```

Obtener la Clave Pública Este programa obtiene la clave pública a partir de una clave privada en formato PEM, equivalente a:

```
1  openssl rsa -in private_key.txt -out public_key.txt -pubout
```

```
1  int retrieveKeyData(EVP_PKEY** key, const char* file, unsigned char
       type) {
2      BIO* out;
3      int ret = SUCCESS;
4      out = BIO_new_file(file, "r");
5      switch (type) {
6      case PRIVATE:
7          PEM_read_bio_PrivateKey(out, key, NULL, NULL);
8          break;
9      case PUBLIC:
10         PEM_read_bio_PUBKEY(out, key, NULL, NULL);
11         break;
12     default: ret = FAILURE;
13     }
14     BIO_free(out);
15     return ret;
16 }
17
18 int main(int argc, char* argv[]) {
19     EVP_PKEY_CTX* ctx;
20     EVP_PKEY* privKey = NULL;
21
22     privKey = EVP_PKEY_new();
23
```

```
24      retrieveKeyData(&privKey, "private_key.txt", PRIVATE);
25      ctx = EVP_PKEY_CTX_new(privKey, NULL);
26      saveKeyData(privKey, "public_key.txt", PUBLIC);
27
28      EVP_PKEY_free(privKey);
29      EVP_PKEY_CTX_free(ctx);
30
31      return EXIT_SUCCESS;
32  }
```

Cifrar con RSA Esto es el equivalente a cifrar con el comando:

```
1  openssl pkeyutl -in in.txt -out outRsa2 -inkey public_key.txt -pubin -
       encrypt -pkeyopt rsa_padding_mode:pkcs1
```

Y descifrar con el comando:

```
1  openssl pkeyutl -in outRsa -inkey private_key.txt -decrypt -pkeyopt
       rsa_padding_mode:pkcs1
```

```
1  void saveEncryptedDataBin(const unsigned char* out, size_t outlen,
       const char* file) {
2      FILE* f = fopen(file, "w");
3      fwrite(out, 1, outlen, f);
4      fclose(f);
5  }
6
7  int main(int argc, char* argv[]) {
8      EVP_PKEY_CTX
9          * ctx;
10     EVP_PKEY
11         * pubKey = NULL;
12     unsigned char* in =
13         "Inteligencia, dame el nombre exacto de las cosas... Que mi
               palabra sea la cosa misma, creada por mi alma nuevamente.";
14     unsigned char* out;
15     size_t outlen;
16
17     /*Inicializo estructura para clave*/
18     pubKey = EVP_PKEY_new();
19     /*Obtengo la clave desde un archivo*/
20     retrieveKeyData(&pubKey, "public_key.txt", PUBLIC);
21     /*Inicializo un contexto para cifrar con clave privada*/
22     ctx = EVP_PKEY_CTX_new(pubKey, NULL);
23     EVP_PKEY_encrypt_init_ex(ctx, NULL);
24     /* Setear padding*/
25     EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING);
26     /* Establecer tamaño para buffer*/
27     EVP_PKEY_encrypt(ctx, NULL, &outlen, in, strlen(in));
28     out = malloc(outlen);
29     EVP_PKEY_encrypt(ctx, out, &outlen, in, strlen(in));
```

```
30      saveEncryptedDataBin(out, outlen, "outRsa");
31      /*Libero estructura de la clave privada*/
32      EVP_PKEY_free(pubKey);
33      /*Libero contexto para la clave*/
34      EVP_PKEY_CTX_free(ctx);
35      free(out);
36      return EXIT_SUCCESS;
37  }
```

JCA

Las clase (**engines**) que se usan son:

- `Secure Random`: Para generar números pseudoaleatorios.
- `Cipher`: Luego de ser inicializada con la clave correspondiente, se usa para el cifrado/descifrado de la información.
- `KeyPairGenerator`: Para generar un par de claves (pública y privada) para un algoritmo específico.
- `KeyAgreement`: Para acordar y establecer una clave específica que será usada por una operación criptográfica particular.

El primer paso es generar un par de claves. Para ello, se genera una instancia del generador `KeyPairGenerator`. Los métodos de inicialización de la clase requieren como mínimo la longitud de la clave. Luego con el método `generateKeyPair` se termina de generar un objeto `KeyPair`.

Luego debe crease una instancia de `Cipher`. Al inicializarlo, si se usa `ENCRYPT_MODE` se debe pasar la clave pública o la clave privada si se usa `DECRYPT_MODE`.

Ejemplos

Cifrar y Descifrar con RSA-2048 Similar a Cifrado Simétrico, pero ahora hay que generar un par de claves y después utilizar la correcta para cifrar y descifrar.

```
1  public class ejemploCifradoRSA {
2      public static void main(String[] args) throws Exception {
3          // Se genera el par de claves
4          KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
5          keyGen.initialize(2048);
6          KeyPair key = keyGen.generateKeyPair();
7          Key pubKey = key.getPublic();
8          Key privKey = key.getPrivate();
9          // Se genera instancia de Cipher
10         Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
11         cipher.init(Cipher.ENCRYPT_MODE, pubKey);
12         // Se cifran los datos
13         byte[] data = "Contenido de Prueba".getBytes();
```

```
14          byte[] cipherText = cipher.doFinal(data);
15          // Vemos el cifrado
16          System.out.println("El cifrado es: " + Base64.getEncoder().
                encodeToString(cipherText));
17          // Se descifran los datos
18          cipher.init(Cipher.DECRYPT_MODE, privKey);
19          byte[] descipherText = cipher.doFinal(cipherText);
20          // Vemos el resultado
21          System.out.println("El descifrado es: " + new String(
                descipherText,
22                  StandardCharsets.UTF_8));
23      }
24  }
```

Firma Digital

OpenSSL

Estas son las funciones para realizar una firma y verificarla utilizando claves publicas con `EVP_PKEY`:

- **`int`** `EVP_PKEY_sign_init_ex(EVP_PKEY_CTX *ctx, const OSSL_PARAM params[])`: Inicializa *ctx* para firmar.
- **`int`** `EVP_PKEY_sign(EVP_PKEY_CTX *ctx, unsigned char *sig, size_t *siglen, const unsigned char *tbs, size_t tbslen)`: Si *sig* es `NULL` se guarda el tamaño máximo del buffer de output en *siglen*, si no *siglen* debe tener la longitud del buffer *sig*, y la longitud de la firma resultante sera guardada. No hashea antes de firmar, y entonces generalmente es usada para firmar digests, para firmar mensajes de longitud arbitraria hay que usar `EVP_DigestSignInit`.

- **`int`** `EVP_PKEY_verify_init_ex(EVP_PKEY_CTX *ctx, const OSSL_PARAM params[])`
- **`int`** `EVP_PKEY_verify(EVP_PKEY_CTX *ctx, const unsigned char *sig, size_t siglen, const unsigned char *tbs, size_t tbslen)`: Devuelve 1 si la verificación es correcta, 0 si fallo, y valores negativos en caso de error.

Luego, usando `EVP_Sign` y verificando con `EVP_Verify`:

- `int EVP_SignInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl)`: *ctx* debe ser creado, y *type* deben ser obtenido, tal como se explica en Hash.
- `int EVP_SignUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt)`: Hashea *cnt* bytes de *d* en el *ctx*, y puede ser llamada varias veces.
- `int EVP_SignFinal_ex(EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s, EVP_PKEY *pkey, OSSL_LIB_CTX *libctx, const char *propq)`: Firma el hash en *ctx* usando la key privada en *pkey* y guarda la firma en *sig*, que debe tener al menos `EVP_PKEY_get_size(pkey)` bytes de espacio. La cantidad de bytes escritos en *sig* se guarda en *s*.

- `int EVP_VerifyInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl)`: *ctx* debe ser creado tal como se explica en Hash.
- `int EVP_VerifyUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt)`: Hashea *cnt* bytes de *d* y los guarda en *ctx*, y puede ser llamada varias veces para incluir más data.
- `int EVP_VerifyFinal_ex(EVP_MD_CTX *ctx, const unsigned char *sigbuf, unsigned int siglen, EVP_PKEY *pkey, OSSL_LIB_CTX *libctx, const char *propq)`: Verifica la data guardada en *ctx* usando la key publica *pkey* y *siglen* bytes de *sigbuf*. Devuelve 1 si la firma es correcta, 0 si falla, y un valor negativo si ocurrió algún error.

Ejemplos

Firmar un hash usando SHA-256 El siguiente programa es el equivalente a correr:

```
openssl pkeyutl -in picture.jpg -inkey private_key.txt -out
    picture_sign -digest sha256 -rawin
```

Y se puede verificar con:

```
openssl dgst -sha256 -verify public_key.txt -signature picture_sign
    picture.jpg
```

```
void retrieveData(unsigned char** data, size_t* datalen, const char*
    file) {
    FILE* f = fopen(file, "r");
    fseek(f, 0, SEEK_END);
```

```
    *datalen = ftell(f);
    fseek(f, 0, SEEK_SET);
    *data = malloc(*datalen);
    fread(*data, 1, *datalen, f);
    fclose(f);
}

void saveData(unsigned char* data, unsigned int len, const char* file) {
    FILE* f = fopen(file, "w");
    fwrite(data, 1, len, f);
    fclose(f);
}

int main(int argc, char* argv[]) {
    const EVP_MD* md_type = EVP_MD_fetch(NULL, "SHA256", NULL);
    EVP_MD_CTX* ctx;
    EVP_PKEY* privKey;
    unsigned char* data;
    unsigned char* sig;
    size_t datalen;
    unsigned int siglen;

    retrieveData(&data, &datalen, "picture.jpg");

    privKey = EVP_PKEY_new();
    retrieveKeyData(&privKey, "private_key.txt", PRIVATE);

    ctx = EVP_MD_CTX_new();

    EVP_SignInit_ex(ctx, md_type, NULL);

    EVP_SignUpdate(ctx, data, datalen);

    siglen = EVP_PKEY_size(privKey);
    sig = malloc(siglen);

    EVP_SignFinal_ex(ctx, sig, &siglen, privKey, NULL, NULL);
    saveData(sig, siglen, "picture_sign");

    free(sig);
    EVP_PKEY_free(privKey);
    EVP_MD_CTX_free(ctx);
    return EXIT_SUCCESS;
}
```

JCA

Las clases (**engines**) que se usan son:

- `SecureRandom`: Para generar números pseudoaleatorios

- `Signature`: Para firmar digitalmente.
- `KeyPairGenerator`: Para generar un par de claves (pública y privada) para un algoritmo específico.
- `KeyAgreement`: Para acordar y establecer una clave específica que será usada por una operación criptográfica particular

Como en cifrado asimétrico, se genera un par de claves. Una vez obtenido el par de claves, se obtiene una instancia del objeto `Signature`. Por ejemplo, DSA. Con la clave privada, se inicializa la instancia de la firma DSA. Se efectúa la firma de la información contenida en un array de bytes. Para verificar la firma se usa la clave pública.

Ejemplos

Firma DSA Calcula la firma DSA de un mensaje utilizando una clave generada en el momento.

```java
public class ejemploFirmaDigital {
    public static void main(String[] args) throws Exception {
        // Se genera el par de claves
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN
            ");
        keyGen.initialize(1024, random);
        KeyPair pair = keyGen.generateKeyPair();
        // Se genera instancia de Firma
        Signature dsa = Signature.getInstance("SHA1withDSA");
        // Se obtiene la clave privada del par
        PrivateKey priv = pair.getPrivate();
        dsa.initSign(priv);
        // Se firman los datos
        byte[] data = "Estos datos hay que firmar".getBytes();
        dsa.update(data);
        byte[] sign = dsa.sign();
        // Vemos la firma
        System.out.println("La firma es: " + HexFormat.of().formatHex(
            data));
        // Se obtiene la clave publica del par
        PublicKey pub = pair.getPublic();
        dsa.initVerify(pub);
        // Se verifican los datos
        dsa.update(data);
        boolean verifica = dsa.verify(sign);
        // Vemos el resultado de la firma
        if (verifica) System.out.println("Firma Validada");
        else System.out.println("Firma NO Validada");
    }
}
```

Fuentes

- Manual para programar OpenSSL
- OpenSSL Libcrypto manual
- Referencia JCA Java 17