Criptografía para desarrolladores

Autor: Ariel H. Maiorano

Ariel es profesional informático que desarrolla servicios y aplicaciones para Internet desde hace más de 10 años a través de su empresa m-sistemas. Se destaca sus implementaciones en criptografía para el comercio electrónico y el almacenamiento seguro de datos.

Además, Ariel es el autor del libro "Criptografía – Técnicas de desarrollo para profesionales" publicado en marzo de 2010. ISBN: 9789872311384

http://www.cuspide.com/detalle_libro.php/9872311382

http://www.amazon.com/CRIPTOGRAFIA-Tecnicas-Desarrollo-Profesionales-Spanish/dp/9872311382

Edición y Corrección de este artículo: Lic. Cristian Borghello, CISSP

Fecha Publicación: 22 de marzo de 2010

Publicado en Segu-Info

Introducción

La intención de este artículo es dar cuenta, de manera general y desde el punto de vista de un desarrollador, de las herramientas actualmente disponibles para implementar criptografía fuerte en aplicaciones y servicios. Se comentan en primera instancia las funcionalidades provistas por los entornos de desarrollo más comunes, luego se presentan tres breves ejemplos prácticos incluyendo código fuente, para terminar con algunas consideraciones respecto a la estandarización y aspectos legales aplicables a esta rama de la seguridad informática.

Al margen de la necesidad de conocimientos mínimos en los lenguajes de programación Java y/o PHP si se quisiera seguir el código fuente de los ejemplos, sí será conveniente contar con ciertas nociones al menos acerca de algunos conceptos criptográficos. En Segu-Info puede encontrase lo necesario y más: véase la sección de criptografía, se trata de un excelente compendio de gran parte de los temas de la materia. Principalmente, y en lo ocupa a este artículo, consúltense lo referente a algoritmos de cifrado simétrico, de cifrado asimétrico y autenticación o autentificación (éste incluye información acerca de funciones de hashing). También se recomienda la consulta de la información incluida dentro de la sección "Criptografía de la A-Z", como ser lo referente a Generadores de Números Aleatorios o Protocolos Criptográficos y Estándares; y de un trabajo sobre los "Fundamentos de Criptografía" disponible desde aquí.

Herramientas disponibles

Actualmente los diferentes entornos de desarrollo poseen una buena variedad de funcionalidades criptográficas incorporadas. Para las tareas más comunes entonces no serán necesarias librerías externas. Cuando, en cambio, fuese preciso utilizar algoritmos algo más específicos -o aquellos de los más utilizados, pero en modos, configuraciones o parametrizaciones particulares-, entonces sí nos veríamos en la necesidad de valernos de librerías, módulos o frameworks adicionales.

De manera general, podría entenderse que tanto para hashing como para cifrado simétrico, el entorno nos proveerá lo necesario en tanto decidamos utilizar los algoritmos más populares. Si bien el cifrado asimétrico es menos común, sí encontraremos implementaciones en versiones modernas de los lenguajes Java y .NET. Esto último no aplica a ambientes limitados; por ejemplo, en el desarrollo Java para teléfonos celulares o dispositivos móviles, J2ME (Java 2 Micro Edition), no se dispone por defecto de función criptográfica alguna.

Al hablar de Java para desarrollos que involucren criptografía, se deberá conocer la Java Cryptography Architecture, o JCA. Esta es una especificación del lenguaje que especifica interfaces y clases abstractas que sirven de base para las implementaciones concretas (engines y providers) de algoritmos criptográficos. La JCA es parte del API de seguridad del lenguaje, desde la JDK 1.1, cuando la JCE era una extensión aparte. Esta JCE (Java Cryptography Extension) se consideró como la implementación concreta de algoritmos criptográficos de acuerdo a los lineamientos definidos por la JCA (incorporada a partir de la JDK 1.4), aunque actualmente las siglas JCA y JCE también se usan de manera general para referir a la arquitectura del framework.

Considerando el entorno de desarrollo .NET, el soporte de criptografía lo encontramos en las funcionalidades provistas por el namespace System.Security.Cryptography. A partir de esta raíz común encontramos clases que proveen implementaciones de varios algoritmos criptográficos. El modelo criptográfico del framework .NET implementa un patrón extensible de herencia de clases derivadas: existen clases de tipo de algoritmo, a nivel abstracto; luego, clases de algoritmos, que heredan de una clase de tipo de algoritmo (abstractas también), y bajo estas últimas, las implementaciones concretas de las clases de algoritmos.

La función de hashing md5() en PHP quizás sea la función criptográfica de un entorno de desarrollo más popularmente utilizada. Estuvo disponible desde la versión 4.0 del lenguaje, y a partir de la versión 4.3 se contaba ya con sha1(). Lamentablemente, no hay mucho más. Para cifrar, firmar, autenticar, etc., en este entorno, serán necesarias librerías o frameworks adicionales.

Con respecto a los motores de base de datos, al menos los más comúnmente implementados hoy en día, nótese que así como cuentan con funciones generales para el manejo de strings, fechas, operaciones matemáticas, etc. (a usarse dentro de sentencias en lenguaje SQL por supuesto), también disponen de funciones para la generación de hashes criptográficos o para el cifrado o descifrado de información (a almacenar o no por el motor).

Resumiendo ahora respecto a qué podríamos hacer con lo disponible por defecto en una instalación estándar de entornos de desarrollo Java, .NET y PHP, y en los principales motores de bases de datos, a continuación se intentará un breve detalle:

- Java: Desde la JDK 1.4 -o J2SE 1.4.2- encontraremos los algoritmos de hashing MD2, MD5, SHA-1, SHA-256, SHA-384 y SHA-512. Se podrá firmar digitalmente mediante el algoritmo DSA, cifrar simétricamente mediante AES, Blowfish, DES, DESede -o TripleDES, o 3DES- y RC2; asimétricamente, a partir de 1.5 -o J2SE 5.0- con RSA.

- .NET: Se dispone de MD5 y SHA1 en todas las versiones del framework .NET. La familia SHA-2 (SHA-256, SHA-385, SHA-512) sólo está disponible a partir de la versión 3.5, igual que el algoritmo de cifrado simétrico AES. En todas las versiones en cambio dispondremos de DES, TripleDES y RC2. También se dispone aquí de los algoritmos asimétricos DSA y RSA.
- PHP: Aquí encontraremos menos opciones, estas son, los algoritmos de hasing MD5 y SHA-1 únicamente.
- Oracle: Funciones de hashing MD5 y cifrado simétrico DES desde versión 8i; TripleDES desde 9i; MD4 y SHA-1, AES y RC4, desde la versión 10g.
- MS-SQL: A partir de la versión 2005 se dispone de MD2, MD4, MD5 y SHA-1; algoritmos de cifrado simétrico disponibles de acuerdo a instalación de sistema Windows, por ejemplo DES y/o AES, y lo mismo para el cifrado asimétrico; por ejemplo, RSA.
- MySql: Función de hashing MD5 disponible desde versión 3.23. SHA-1, DES y AES desde 4.0/4.1. Familia SHA-2 disponible a partir de versiones 6.0/6.0.5.

Esto sería entonces lo que podríamos considerar incluido dentro de cada uno de los lenguajes o entornos más comúnmente utilizados. Téngase en cuenta además que algunos algoritmos o protocolos se definen a partir de otras funciones criptográficas, que muchas veces sí están incorporadas. Por lo tanto, en tales casos, no sería estrictamente necesaria la utilización de librerías extra (pongamos por caso, lo especificado en el estándar PKCS #5 respecto a la generación de llaves a partir de contraseñas; ver ejemplo práctico #1).

De lo anterior se desprende que, si se tuvieran necesidades específicas no contempladas en las versiones a utilizar de la máquina virtual Java o en el ambiente PHP, se vería uno obligado a utilizar librerías o frameworks adicionales. Si bien hay muchas alternativas disponibles (muchas ellas de uso libre y de código abierto), se comentarán algunas opciones a continuación.

En PHP, mediante la librería Mcrypt, obtendremos la posibilidad de utilizar los algoritmos de cifrado simétrico DES, TripleDES, Blowfish, Rijndael (AES), Twofish, IDEA, RC2, RC4, RC6 y Serpent entre otros. La librería MHash proveerá una variedad importante de funciones de hashing, incluyendo por supuesto a la familia SHA-2. Por último, el package PEAR Crypt_RSA servirá para generar llaves, firmar y comprobar firmas, y, por supuesto, cifrar y descifrar asimétricamente. Podrían también consultarse los packages Crypt_HMAC, Crypt_HMAC2, y Crypt_DiffieHellman (estos dos últimos sólo para PHP 5), como sus nombres lo indican, para generar HMACs o códigos de autenticación de mensajes basados en hashing (ver ejemplo práctico #2), y para el intercambio de claves (algoritmo DH), respectivamente.

Si Java fuese nuestro ambiente o entorno de desarrollo, no debería dejar de consultarse la documentación del proyecto open-source de Bouncy Castle. Desde allí puede descargarse una librería de funciones que implementa una cantidad muy importante de algoritmos criptográficos. Podría decirse que implementa la mayoría de los más conocidos y utilizados, tanto para el cifrado simétrico y asimétrico, hashing, generación de claves, códigos de autenticación de mensajes, etc. (entre otros: MD2/4/5, SHA1/2, Tiger, Whirlpool, AES, Blowfish, DES, DESede, GOST, IDEA, RC2/4/5/6, SEED, Serpent, Twofish, HC256, RSA, ElGamal, Diffie-Hellman).

Se distribuye como una implementación de lo que sería un provider JCE, o de manera separada, para ser utilizada por ejemplo en desarrollos J2ME, aunque no exclusivamente por supuesto (ver ejemplo práctico #3).

Antes de terminar este apartado, algunas aclaraciones: si se ha de desarrollar en C/C++ (para sistemas Unix o Windows), quizás la referencia obligada siga siendo, como hace ya varios años, la librería de código abierto OpenSSL. Implementaciones criptográficas en otros lenguajes de programación muchas veces hacen uso de esta librería, aunque será necesario investigar en cada caso cuál sería la mejor alternativa. Al margen entonces del entorno y/o librerías a utilizar, y a este respecto entonces, también cabe aclarar que como sucede en diferentes ramas de la seguridad informática, es menester andar con pie de plomo. Desconfíese de ofertas rimbombantes, por lo general de código cerrado, que acentúan la conveniencia del precio o la impresionante -a veces inverosímil- velocidad del "nuevo" método de cifrado, y poco o nada especifican respecto a los algoritmos o estándares que implementa. En cambio, los algoritmos popularmente reconocidos en la actualidad fueron y son constantemente analizados por sectores de la industria y académicos a gran escala. Por esto es que sería prudente, por lo general, decidirse por algoritmos e implementaciones públicos y abiertos, que no hayan sufrido problemas de seguridad (quiebres en los algoritmos, bugs en las implementaciones) durante algunos años. Véase, a manera de ejemplo a este respecto, el siguiente artículo en este sitio.

Ejemplo práctico #1: Registro seguro de contraseñas en base de datos

Si bien es muy común encontrar implementaciones que registran las contraseñas "cifradas" (no se realiza una encriptación y desencriptación, si no que se aplica una función criptográfica de una vía, tanto para el registro incial como para las posteriores verificaciones) en base de datos mediante MD5, no sería ésta la alternativa más segura. Debido en parte a la popularidad de las rainbow tables, y de sitios como éste , o éste, hoy debe considerarse la utilización salts y de una cantidad importante de iteraciones (comúnmente llamada iteration count, que en el ejemplo a seguir implica la aplicación reiterada de una función de hashing) para evitar problemas en caso de que la base datos fuese comprometida. Nos basaremos en el estándar de PKCS #5 2.0 de RSA, luego devenido en el RFC 2898. Se implementará, en lenguaje PHP y sin utilizar librerías adicionales, una función para lo que en el estándar es llamado "Key Derivation", concretamente, la primera de las alternativas (ver sección 5.1 del RFC), aunque la segunda es la recomendada. Se establece allí además que la cantidad de iteraciones ha de ser de al menos 1000, y que el tamaño del salt deberá ser de 8 bytes como mínimo.

```
	<?php


	// Función para generar/derivar una llave (hash) según
recomendaciones en PCKS#5
	// o RFC 2898 a partir de una contraseña y un salt.
	function hashContrasena($salt, $contrasena)	{
		// se asume que $salt se recibe como string hexa de 32
caracteres (16 bytes).
		$salt_bytes = pack('H32', $salt);
		// se utilizará SHA-1, primera iteración
```

```
            $llave_bytes = pack('H40', sha1($contrasena . $salt_bytes));
            // 998 iteraciones...
            for ($i=0; $i<998; $i++)        {
                    $llave_bytes = pack('H40', sha1($llave_bytes));
            }
            // se retorna última (#1000) iteración
            return sha1($llave_bytes);
      }

      // Imaginando que llegamos hasta aquí cuando el usuario completó su $contrasena por
      // primera vez. Se deberá generar un salt e invocar a la función para derivar la
      // llave para luego guardar ambos valores (llave o hash y salt) en base de datos.

      $salt = md5(rand());
      $contrasena = "mi frase clave de prueba.";

      $hash = hashContrasena($salt, $contrasena);

      // En lugar de imprimirlo para ejemplificar el caso, como se ha dicho, se registraría
      // en base de datos...
      echo ' Salt: ';
      echo $salt;
      echo ' Hash: ';
      echo $hash;

      ?>
```

Aclaraciones o notas importantes:

1. En este ejemplo, de una forma poco conveniente, o, al menos, no recomendada para la generalidad de los casos, se obtiene un salt utilizando el generador aleatorio provisto por el lenguaje PHP y luego se realiza MD5 de una forma acaso arbitraria sólo para obtener rápidamente un valor de 16 bytes "relativamente" aleatorio. En favor de poder resumir para explicar el tema que se está tratando, se resignó la codificación que utilizaría un generador de números pseudo-aleatorios seguro (entiéndase, criptográfico). Sin embargo, hemos de recordar de cualquier manera que respecto a la necesidad planteada en este ejemplo, el salt no debe ser secreto y lo importante es que sea único. Mejor dicho (?) "lo más único posible", y no sólo para el sistema en donde va implementarse, porque supóngase que se convirtiese en práctica común el utilizar el nombre usuario como salt (considerando que éste sería único en el sistema por supuesto), una entidad que disponga de gran poder de procesamiento y almacenamiento

podría generar tablas rainbow utilizando salts como "admin", "administrador", "jperez", etc. Un generador criptográfico será lo apropiado. El estándar usado como referencia explicita "a large and sufficiently random salt". Su misión principal sigue siendo la de evitar o disminuir la eficacia de dictionary attacks. Consúltese también el concepto criptográfico de "nonce".

2. No se ha completado el ejemplo, pero se entenderá que a la hora de verificar la contraseña de un usuario, el proceso será el mismo, salvo que el salt no será generado, sino recuperado desde la base datos, junto con el hash, para luego de computar el hash a partir de este salt y la contraseña completada por el usuario en el proceso de login, comparar el hash almacenado contra el resultado de este cómputo.

Ejemplo práctico #2: Uso típico de códigos de autenticación de mensajes

Se presenta un caso típico de la verificación o validación de parámetros sensibles. Se considera como ejemplo el envío de un newsletter vía e-mail, que deberá incluir un link hacia un servidor web del remitente para que el destinatario pueda cancelar su suscripción (sin autenticarse por supuesto).

Existen diversos mecanismos para evitar lo que se conoce como parameter tampering. La solución propuesta es uno de estos mecanismos, y puede utilizarse tanto para la necesidad específica de este ejemplo como para tantas otras situaciones en las cuales controles de este tipo serían necesarios (por lo general, aquellas cuando no se quiere molestar al usuario pidiéndole que se identifique en el sistema). La alternativa de solución propuesta la implementaremos mediante el uso de HMAC (Hash-based Message Authentication Code).

La versión más simple de esto podría implementarse sencillamente obteniendo un hash criptográfico del parámetro concatenado a la llave secreta. Sin embargo, de acuerdo a posibles vulnerabilidades de esta técnica (ver trabajo al respecto), se seguirá lo establecido por la RFC 2104 (nótese que se trata del mismo autor del trabajo anterior). En esta variante, en lugar de H(llave | parámetro) o H(parámetro | llave), implementa H(llave1 | H(llave2 | parámetro)), siendo llave1 y llave2 derivadas de la llave original. Se mostrará entonces el código significativo utilizando el lenguaje PHP, valiéndonos del package PEAR Crypt_HMAC, que implementa este estándar según indica su documentación.

Para dejar en claro cuál sería el problema, si no realizáramos control alguno, el e-mail enviado contendría un link similar al siguiente:

```
http://www.newsletter.com/unsuscribe.php?IDSUSCRIBER=44
```

De esta forma entonces, queda en evidencia que el número 44 corresponde al identificador en base de datos del registro de la cuenta. Si un usuario cambiase ese número en la barra de direcciones, podría borrar suscripciones de otros usuarios.

Como se adelantó más arriba, la solución vendrá de la mano de la utilización de un HMAC. Se generará esta suerte de checksum seguro, criptográficamente seguro, para que un destinatario sólo pueda borrar su suscripción (ni el nombre de la variable CGI ni su valor -el número 44- son secretos). De esta forma, el link entonces podría ser:

```
http://www.newsletter.com/unsuscribe.php?IDSUSCRIBER=44&HMAC=XXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXX
```

Aquí vemos que se agrega una variable, y el valor de esta variable será el resultado de la función HMAC, cuya entrada será el número 44 y una clave secreta que sólo quien envía el newsletter conoce (utilizará esta clave al momento de realizar los envíos y también, por supuesto, al momento de confirmar el valor del parámetro o variable IDSUSCRIBER cuando el usuario quiere eliminar su suscripción).

Resta ahora únicamente codificar la función que a partir del número identificador del usuario suscripto y una llave secreta, genere el HMAC. Lo que se agrega también, es la implementación de una función sin usar librerías externas, e invocaciones para comparar los resultados. Esto último hecho en base al código de HMAC.php, groseramente resumido para ejemplificar el caso.

```
<?php

require_once 'HMAC.php';

function generarHMAC($contrasena, $parametros)  {
  // se utilizará MD5 (opcionamente puede utilizarse SHA-1)
  $crypt = new Crypt_HMAC($contrasena, 'md5');
  return $crypt->hash($parametros);
}

function generarHMAC_local($contrasena, $parametros)   {
  // supondremos contraseña < 64 caracteres, debe 'paddearse' a
  // ese tamaño, que es el del bloque MD5, es decir, 512 bits.
  $contrasena = str_pad($contrasena, 64, chr(0));
  // números para padding especialmente seleccionados, según RFC
  $llave1 = (substr($contrasena, 0, 64) ^ str_repeat(chr(0x5C), 64));
  $llave2 = (substr($contrasena, 0, 64) ^ str_repeat(chr(0x36), 64));
  // H(llave1 | H(llave2 | parámetro))
  return md5($llave1 . pack('H32', md5($llave2 . $parametros)));
}
echo ' Utilizando Crypt_HMAC: ';
echo generarHMAC("Secret0", "44");
echo ' Utilizando versión local: ';
echo generarHMAC_local("Secret0", "44");

?>
```

Tanto para la generación del link, como para su eventual verificación posterior, la función a invocar será la misma. De esta manera lo que se logra entonces es que si un usuario suscripto desconoce la llave secreta, no podrá eliminar suscripciones de otros usuarios.

Dos notas o aclaraciones obligadas: 1. No se verifica cuándo el usuario realizaría esta acción, ni sobre qué suscripción (en caso de que hubiese más de una). Frente a tales necesidades, considérese que, por ejemplo, un timestamp y un identificador de suscripción pueden ser nuevas variables o parámetros. De esta forma, las variables CGI serán independientes, pero al armar el HMAC, usaríamos el número original (identificador de usuario suscripto) concatenado a las nuevos parámetros. 2. Sería conveniente generar la llave (si ha de hacerse a partir de una contraseña), por ejemplo, de acuerdo a lo especificado en PKCS #5 (ver ejemplo #1); sólo para demostrar el tema que nos ocupa en este ejemplo, y no confundir los conceptos, se utiliza la contraseña como llave directamente.

Ejemplo práctico #3: Criptografía en teléfonos celulares

Aquí sólo se mostrarán las porciones clave del código de un proyecto propio de código abierto que incluye tres implementaciones básicas de cifrado simétrico para teléfonos celulares que soporten Java. La página web desde la cual es posible descargar los binarios y los fuentes completos, aquí.

Como allí se indica, se trata de una implementación de prueba que se basa en la librería de código abierto del proyecto Bouncy Castle. Esto permitió que pueda configurarse muy fácilmente el algoritmo de criptografía simétrica a utilizar (AES -default-, Twofish o Serpent) y el tamaño de llave (128 o 256 -default- bits). Se utiliza el modo de cifrado CBC, salts e IVs de 16 bytes, generados aleatoriamente. Lo ejemplos, o porciones de código que se pegan en lo que sigue corresponden a la clase sct.Crypto, donde se agruparon las necesidades criptográficas de los programas.

Para poder seguir el código fuente de la generación de la llave y del cifrado, convendrá tener presentes los nombres y tipos de las variables miembro de la clase:

```
        private static final int SALT_LEN = 16; // bytes
        private static BlockCipher engine = null;
        private static byte[] salt = null;
        private static byte[] key = null;
        private static byte[] IV = null;
        private static byte[] _plainText = null;
        private static int keyLen = 0;      // bits
        private static int blockLen = 0;    // bits
        private static boolean debug = false;
```

Lo que sigue corresponde a la función general de inicialización (también parametrización) de esta clase:

```
        /**
         * General initialization; sets algorithm, keylen, debug mode -
not to use in production
         */
        static void initEngine(String algorithm, int keyLen, boolean
debug) throws IllegalArgumentException   {
            Crypto.debug = debug;
```

```
            if (keyLen != 128 && keyLen != 256)
                throw new IllegalArgumentException("Key size must be 128
or 256 bits.");
            Crypto.keyLen = keyLen;
            if (algorithm.equalsIgnoreCase("aes")) {
                engine = new AESEngine();
            } else if (algorithm.equalsIgnoreCase("serpent")) {
                engine = new SerpentEngine();
            } else if (algorithm.equalsIgnoreCase("twofish")) {
                engine = new TwofishEngine();
            } else  {
                throw new IllegalArgumentException("Algorithm must be
AES, Serpent or TwoFish.");
            }
            Crypto.blockLen = engine.getBlockSize() * 8; // bytes -> bits
            Crypto.salt = null;
            Crypto.key = null;
            Crypto.IV = null;
            if (debug)   {
                System.out.println("initEngine(): debug: " +
Crypto.debug);
                System.out.println("initEngine(): keyLen: " +
Crypto.keyLen);
                System.out.println("initEngine(): blockLen: " +
Crypto.blockLen);
                System.out.println("initEngine(): engine algorithm: " +
Crypto.engine.getAlgorithmName());
            }
            return;
        }
```

Siguiendo con lo que sería una ejecución típica del programa para cifrar, lo que sigue correspondería a generar la llave a partir de una contraseña o frase-clave (pass-phrase):

```
        /**
         * Generates and set new key, salt and iv - note pass-phrase
parameter is cleared
         */
        static void setNewKeyMaterial(char[] passPhrase)  {
            if (engine == null || keyLen == 0 || blockLen == 0)
                throw new IllegalStateException("Engine not
initialized.");
            byte[] passPhraseBytes =
PKCS5S2ParametersGenerator.PKCS5PasswordToBytes(passPhrase);
            clearCharArray(passPhrase);
```

```
            PKCS5S2ParametersGenerator pg = new
PKCS5S2ParametersGenerator();
            byte[] tmpRandom = new
ThreadedSeedGenerator().generateSeed(20, true); // xxx: 20 random bytes to
feed/seed SecureRandom, fast mode
            SecureRandom sr = SecureRandom.getInstance("SHA256PRNG"); //
xxx: fixed algorithm
            sr.setSeed(tmpRandom);
            salt = new byte[SALT_LEN];
            sr.nextBytes(salt);
            pg.init(passPhraseBytes, salt, 1000);   // 1000 iterations -
RSA's PKCS #5, IETF RFC 2898
            ParametersWithIV paramIV = (ParametersWithIV)
pg.generateDerivedParameters(keyLen, blockLen);
            IV = paramIV.getIV();
            KeyParameter paramKey = (KeyParameter)
paramIV.getParameters();
            key = paramKey.getKey();
            clearByteArray(passPhraseBytes);
            if (debug)   {
                System.out.println("setNewKeyMaterial(): random: " + new
String(Hex.encode(tmpRandom)));
                System.out.println("setNewKeyMaterial(): salt: " + new
String(Hex.encode(salt)));
                System.out.println("setNewKeyMaterial(): iv: " + new
String(Hex.encode(IV)));
                System.out.println("setNewKeyMaterial(): key: " + new
String(Hex.encode(key)));
            }
            return;
        }
```

Veamos ahora el código de la función (o método, para los puristas) de encriptación o cifrado principal:

```
        /**
         * Takes plain text input bytes and returns cipher text
(encrypted) bytes - note plain-text parameter is cleared
         */
        static byte[] Encrypt(byte[] plainText)  {

            if (engine == null || keyLen == 0 || blockLen == 0)
                throw new IllegalStateException("Engine not
initialized.");
            if (IV == null)
                throw new IllegalStateException("IV not initialized.");
            if (key == null)
```

```
                throw new IllegalStateException("Key not initialized.");


            KeyParameter paramKey = new KeyParameter(key);
            ParametersWithIV paramIV = new ParametersWithIV(paramKey,
IV);


            BufferedBlockCipher cipher = new
PaddedBufferedBlockCipher(new CBCBlockCipher(engine));
            cipher.init(true, paramIV);


            byte[] cipherText = new
byte[cipher.getOutputSize(plainText.length)];


            int outputLen = cipher.processBytes(plainText, 0,
plainText.length, cipherText, 0);
            try {
                cipher.doFinal(cipherText, outputLen);
            } catch (DataLengthException ex) {
                if (debug)
                    ex.printStackTrace();
                throw new IllegalArgumentException(ex.getMessage());
            } catch (IllegalStateException ex) {
                if (debug)
                    ex.printStackTrace();
                throw new IllegalArgumentException(ex.getMessage());
            } catch (InvalidCipherTextException ex) {
                if (debug)
                    ex.printStackTrace();
                throw new IllegalArgumentException(ex.getMessage());
            }


            if (debug)   {
                System.out.println("Encrypt(): salt: " + new
String(Hex.encode(salt)));
                System.out.println("Encrypt(): iv: " + new
String(Hex.encode(IV)));
                System.out.println("Encrypt(): key: " + new
String(Hex.encode(key)));
                System.out.println("Encrypt(): plain text: " + new
String(Hex.encode(plainText)));
                System.out.println("Encrypt(): cipher text: " + new
String(Hex.encode(cipherText)));
            }
```

```
            return cipherText;
        }
```

Estandarización

Podría resumirse que, en lo que a criptografía se refiere y en relación a la estandarización, son tres los organismos más activos. La famosa ISO (International Organization for Standardization), que publica estándares ampliamente implementados a nivel mundial. La ANSI (American National Standards Institute) y la NIST (National Institute of Standards and Technology) siendo entonces las otras dos, que desempeñan el papel de principales entidades de registro y estandarización de los EE.UU. en materia de tecnología.

De los estándares publicados por las tres organizaciones, los FIPS (Federal Information Processing Standards) de la NIST corresponden a las publicaciones de estándares criptográficos más popularmente referenciados de manera general.

El objetivo de tales publicaciones es normalizar sus recomendaciones, a ser consideradas y/o adoptadas por entidades gubernamentales no militares de los EE.UU. y sus contratistas. Sin embargo, al ser públicos y de validez reconocida, por supuesto que también son referencia de otros sectores involucrados de alguna forma con la criptografía. En referencia a esto, nótese que, por ejemplo, fue la NIST la organización que estandarizó primero al algoritmo DES como el estándar de encripción de los EE.UU. También la encargada de sustituirlo después por el algoritmo AES.

Al margen de estas tres organizaciones, corresponde citar también a la IETF (Internet Engineering Task Force), un grupo abierto de técnicos y compañías interesados en la operación armónica de Internet. Estándares como el x.509, TLS v1, y OpenPGP fueron desarrollados por esta organización, pero es reconocida principalmente por manejar los RFC (Request For Comments).

Han de contemplarse también las situaciones en las cuales no existe estándar formal publicado por alguna de estas organizaciones, o cuando la industria o el sector académico han generado lo que se llamaría un estándar de facto, probablemente a convertirse en un estándar formal eventualmente. En el ejemplo práctico # 2 vimos justamente que el "estándar" sobre el que nos basamos (PKCS #5) fue definido por una compañía privada (RSA). Luego, esas mismas especificaciones, conformaron un RFC. Otro ejemplo: como se comentó más arriba, el estándar TLS v1 le corresponde a la IETF, pero las diferencias de éste respecto al protocolo SSL 3.0 o v3 desarrollado y publicado por Netscape son menores.

Aspectos legales

Para hacer una rápida referencia al menos a las cuestiones legales relativas a la criptografía, locales e internacionales, comenzaremos por notar que Argentina es uno de los países participantes del acuerdo de Wassenaar. Básicamente, los lineamientos del acuerdo buscan limitar la exportación de bienes de "doble-uso" (¿uso-doble?), es decir, aquellos que además de su uso civil, podrían ser utilizados con fines de desarrollo militar.

Se entiende que el ánimo del acuerdo -y de los países que suscriben por supuesto- es evitar una nueva carrera armamentista. No sería posible desarrollar aquí el tema, pero algunas implementaciones criptográficas, principalmente

aquellas sobre hardware y software comercial (o más bien todo aquello que no quepa dentro de lo llamado commodity cryptography) podría considerarse un bien de doble-uso.

Puede consultarse más información en el sitio del acuerdo, y en la GILC. Desde aquí se accede a los listados de lo que se considera como bien de doble-uso (ver/descargar Category 5 - Part 2). Véase también una guía de cómo implementar -en lo que a trámites se refiere- criprografía en iphone, que da una cabal idea de cómo sería el proceso de clasificación como commodity. Aquí algo similiar, de parte de la Apache Software Fundation, para los miembros de su Project Management Committee (nótese el disclaimer obligado en algunos casos referenciando al sitio web del acuerdo de Wassenaar). Respecto a las reglamentaciones en EE.UU. podría consultarse este fact sheet, y este blog general de leyes de exportación. Por último, tanto este interesante artículo respecto al acuerdo en particular, como la información provista por el sitio de la crypto law survey (véanse sus mapas), son una referencia obligada en la materia y bien pueden servir como punto de partida al abordar estos temas.

Específicamente, en lo que a Argentina se refiere, se recomienda la consulta del sitio de firma digital de la República Argentina, dependiente de la ONTI (Oficina Nacional de Tecnologías de Información) donde se encontrará, como establece su página principal: *[...] información acerca de toda la Legislación Argentina en materia de Firma Digital, el licenciamiento de certificadores en el marco de la Ley Nº 25.506 de Firma Digital, los proyectos implementados por la Administración Pública Nacional, los servicios disponibles y el estado de situación nacional e internacional en el tema.*

Como otro de los aspectos legales a tener en cuenta a la hora de pretender importar, usar, desarrollar y/o exportar criptografía, recuérdese que algunos algoritmos están patentados estableciendo algunas limitaciones en su utilización (por ejemplo, el algoritmo de cifrado simétrico IDEA).

Si bien la mayoría de los algoritmos mencionados en este artículo pertenecen al dominio público, corresponderá asegurarse en cada caso particularmente no hacer uso indebido de algún algoritmo registrado. Puede consultarse este sitio para acceder a un listado detallado de patentes, o a éste resumen de parte de RSA.

Para terminar

Repitiendo entonces en parte lo escrito anteriormente, terminando, un consejo general podría ser el de considerar y dedicar el tiempo extra que demandará implementar criptografía en cualquier tipo de programa o código informático. Convendrá estarse completamente seguro del cómo y del por qué se están programando de tal o cual manera los procesos de cifrado, descifrado, hashing, etc.

Investigar lo relativo a la alternativa que se haya decidido, e intentar mantenerse informado. Se deberá verificar qué hacen en verdad funciones del orden de "Encrypt()" o "Decrypt()" en librerías y/o frameworks de uso general, desconfiar de implementaciones cerradas que no especifican cuáles estándares adoptan, considerar cómo se almacena temporariamente la información sensible en memoria en tiempo de ejecución (y cerciorarse de su limpieza), y en este aspecto, en general, cuidar esta información a priori y a posteriori de los procesos criptográficos; de poco serviría utilizar criptografía fuerte si luego se guardarán en

disco o se transmitirán abiertamente contraseñas, pass-phrases, llaves secretas o los datos a resguardar.

A este respecto, Bruce Schneier, en el prólogo para el libro "Security Engineering" vuelve sobre su concepto de "programar para la computadora del diablo"...

[...]
Programming a computer is straightforward: keep hammering away at the problem until the computer does what it's supposed to do. Large application programs and operating systems are a lot more complicated, but the methodology is basically the same. Writing a reliable computer program is much harder, because the program needs to work even in the face of random errors and mistakes: Murphy's computer, if you will. Significant research has gone into reliable software design, and there are many mission-critical software applications that are designed to withstand Murphy.
Writing a secure computer program is another matter entirely. Security involves making sure things work, not in the presence of random faults, but in the face of an intelligent and malicious adversary trying to ensure that things fail in the worst possible way at the worst possible time...again and again. It truly is programming Satan's computer.
[...]