Ariel Maiorano

Criptografía

Técnicas de desarrollo para profesionales

Materiales adicionales en la

Alfaomega

CRIPTOGRAFÍA

Técnicas de desarrollo para profesionales

Ariel Maiorano

CRIPTOGRAFÍA

Técnicas de desarrollo para profesionales

Ariel Maiorano

Buenos Aires • Bogotá • México DF • Santiago de Chile

Alfaomega

Maiorano, Ariel Horacio
Criptografía : técnicas de desarrollo para profesionales. - 1a ed. - Buenos Aires : Alfaomega Grupo Editor Argentino, 2009.
292 pp. ; 17x23 cm. - (Para profesionales)

ISBN 978-987-23113-8-4

1. Informática. 2. Criptografía. I. Título
CDD 005.82

Edición: Damián Fernández
Coordinadora: Yamila Trujillo
Corrección: Romina Aza-Silvia Mellino
Diagramación de interiores: Patricia Baggio
Diseño de tapa: Oscar Iturralde
Revisión de armado: Silvia Mellino

Internet: http://www.alfaomega.com.mx

Empresas del grupo:

Argentina: Alfaomega Grupo Editor Argentino, S.A.
Paraguay 1307 P.B. “11”, Buenos Aires, Argentina, C.P. 1057
Tel.: (54-11) 4811-7183 / 8352
E-mail: ventas@alfaomegaeditor.com.ar

México: Alfaomega Grupo Editor, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, México, D.F., México, C.P. 03100
Tel.: (52-55) 5089-7740 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A.
Carrera 15 No. 64 A 29, Bogotá, Colombia
PBX (57-1) 2100122 - Fax: (57-1) 6068648
E-mail: scliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A.
General del Canto 370-Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786
E-mail: agechile@alfaomega.cl

Agradecimientos

Agradezco particularmente a Damián Fernández, editor de Alfaomega Grupo Editor y, de manera general, a su equipo, por su predisposición absoluta, valiosa ayuda e inmerecida paciencia, ciertamente puesta a prueba en más de una ocasión.

Mensaje del editor

Los conocimientos son esenciales en el desempeño profesional, sin ellos es imposible lograr las habilidades para competir laboralmente. La universidad o las instituciones de formación para el trabajo ofrecen la oportunidad de adquirir conocimientos que serán aprovechados más adelante en beneficio propio y de la sociedad. El avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos. Cuando se toma la decisión de embarcarse en una vida profesional, se adquiere un compromiso de por vida: mantenerse al día en los conocimientos del área u oficio que se ha decidido desempeñar.

Alfaomega tiene por misión ofrecerles a estudiantes y profesionales conocimientos actualizados dentro de lineamientos pedagógicos que faciliten su utilización y permitan desarrollar las competencias requeridas por una profesión determinada. Alfaomega espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

Los libros de Alfaomega están diseñados para ser utilizados dentro de los procesos de enseñanza-aprendizaje, y pueden ser usados como textos guía en diversos cursos o como apoyo para reforzar el desarrollo profesional.

Alfaomega espera contribuir así a la formación y el desarrollo de profesionales exitosos para beneficio de la sociedad.

Ariel H. Maiorano

Profesional informático. Desarrolla servicios y aplicaciones para Internet desde hace más de diez años. Actualmente, desde el estudio informático m-sistemas (`http://www.m-sistemas.com.ar`) se ha encargado, junto con su equipo, del análisis, desarrollo y puesta en marcha de proyectos de diversa índole para importantes compañías y organizaciones nacionales e internacionales.

Se destacan, entre las implementaciones criptográficas, aquellas desarrolladas para el comercio electrónico, la firma digital, el almacenamiento seguro de información y la interoperabilidad segura entre sistemas.

Contenido

Prefacio

Podríamos preguntarnos: Si no desarrollamos sistemas de espionaje que manejen secretos de estado o industriales, como vemos en las películas en donde los diálogos hacen referencia a cifrados, encriptación, o rupturas o quiebres de códigos y contraseñas, ¿necesitamos incorporar criptografía fuerte o basada en complejos algoritmos matemáticos en nuestras aplicaciones? La respuesta es afirmativa. A usted, como desarrollador, le será necesario diseñar y desarrollar sistemas que deban identificar a los usuarios que los utilizan, que deban registrar sus acciones, certificar los contenidos que han generado, identificarse ante sistemas de terceros, etc.

La primera alternativa que usted tiene es la de implementar un algoritmo propio, basado en una simple sustitución de letras y números o, incluso, hasta dejar la información sin cifrar, sólo "escondida", suponiendo que nadie la encontrará en donde usted la ha almacenado. La otra opción, si es que desea que la información para resguardar en su sistema no sea comprometida con no más que un poco de suerte o conocimientos elementales de informática y de las técnicas de codificación, es utilizar estándares criptográficos de seguridad, hasta el momento al menos, comprobados.

El texto que tiene en sus manos lo ayudará si ha optado por la segunda alternativa. Ya sea que tenga que desarrollar un sistema que manejará usuarios y contraseñas de manera segura, codificar el acceso a, o consumo de, un Web-service o servicio-Web que utilice información cifrada, o decidir respecto de la implementación de la tecnología criptográfica para el resguardo de información sensible –asumiendo la responsabilidad que esto pudiera generar–. En estas páginas encontrará la guía necesaria que le permitirá elegir, con basamentos teóricos y prácticos, entre las alternativas actualmente disponibles de los algoritmos criptográficos estandarizados en los entornos de desarrollo más popularmente utilizados.

El propósito de este libro es dar cuenta de cómo han de utilizarse las distintas herramientas o funciones criptográficas, disponibles en los lenguajes de programación y motores de base de datos más populares en la actualidad, para el desarrollo de sistemas seguros.

Antes de comenzar a leer:

En este libro se utiliza la tipografía Courier en los casos en que se hace referencia a código o acciones que se han de realizar en la computadora, ya sea en un ejemplo o cuando se refiere a alguna función mencionada en el texto. También se usa para indicar menús de programas, teclas, URLs, grupos de noticias o direcciones de correos electrónicos.

Para comodidad del lector, el código utilizado en el libro se encuentra disponible para ser descargado desde:

`http://www.alfaomega.com.mx/archivosadicionales`

Introducción general

1

Generalidades de la criptografía.

Definir qué es la criptografía puede no ser una tarea sencilla. Si han de contemplarse las diversas aplicaciones que ha tenido a lo largo de la historia, una definición general podría contradecir a otra que sea coherente a cómo es actualmente implementada en los sistemas informáticos. Si en lugar de tratarse de un libro de cuestiones referentes a la aplicación práctica de tecnología informática, éste versara, por ejemplo, sobre matemáticas formales o sobre estrategias militares, hasta el primer acercamiento a las nociones básicas acerca de la criptografía se estructuraría de manera diferente.

Con el ánimo de definir la criptografía, con las aclaraciones precedentes, el primer paso no puede ser otro que referirse al diccionario como punto de partida y detallar las adecuaciones que consideramos necesarias al área que nos compete. Hemos de aclarar, por supuesto, que no ha de esperarse de un diccionario más que una definición general, las más de las veces escueta, de cualquier término técnico o científico. Por supuesto, ése es el objetivo que cumple el diccionario y el error correría por nuestra cuenta si allí pretendiésemos encontrar una acabada descripción de la materia.

El diccionario de la Real Academia Española define a la criptografía como “el arte de escribir con clave secreta o de un modo enigmático” (*kryptos*, del griego –ocultar– y grafía, *graphos* –escribir–).

Tal definición nos da una idea, a grandes rasgos, de que la criptografía se utiliza para escribir de manera tal que el resultado de esa escritura pueda ser interpretado únicamente por quien conozca la clave secreta o el modo “enigmático”. Sin embargo, esta definición es incompleta en la actualidad respecto de las técnicas utilizadas en sistemas informáticos.

Lo primero que notaremos es la categorización de arte. Si bien, para los puristas, en algunos casos se trate de materia discutible, la criptografía actualmente se entiende

tanto como parte o rama de la ciencia matemática, como rama de la ciencia de la computación o informática. Respecto de lo referido como clave secreta, también objetaremos la necesidad de inclusión de lo que llamamos "clave o llave pública" en los sistemas criptográficos asimétricos. Por último, pero no menos importante, debe tenerse en cuenta que las funciones criptográficas de hoy día tienen aplicación más allá del resguardo confidencial de información. Entre ellas, la de certificar la autoría, autenticidad e integridad de una información particular (dentro de, por ejemplo, las implementaciones de firma digital).

Ya en el ámbito de la informática, existe además una diferenciación elemental en cuanto a cómo, a lo largo de su –ya más corta– historia, se ha utilizado la criptografía, que no debería pasarse por alto en un primer acercamiento a la materia. Para la clarificación de esta diferencia diremos que existen –en programas de escritorio, servicios, sitios Web, etc.– aquellas implementaciones criptográficas que sirven para evitar que la información sea leída por un usuario ocasional de estas aplicaciones o servicios y aquellas otras que impiden a entes que dispongan de gran capacidad de procesamiento y el conocimiento acerca de las teorías criptoanalíticas actuales, acceder a, o descifrar, esta información. Escribió Schneier[1], a este respecto, que existe criptografía para evitar que su hermana menor acceda al contenido de sus archivos, y, por otra parte, aquella otra criptografía que impedirá a los gobiernos más desarrollados el acceso a su información.

Esto significa que muchos desarrolladores, a la hora de implementar un resguardo de información, han utilizado técnicas antiguas o rudimentarias. El resultado de esto detendrá a quien no tenga los conocimientos necesarios para adentrarse en la decodificación o desciframiento de tal información, pero no resistirá el "ataque" de un matemático o de un informático con nociones de criptoanálisis. Distinto es el caso, por supuesto, de implementaciones que utilicen funciones criptográficas basadas en algoritmos matemáticos, estandarizados internacionalmente en algunos casos, de robustez comprobada.

Al tener estas consideraciones en mente, la siguiente definición, del especialista español Ramió Aguirre[2] parece más acertada:

> *"Rama inicial de las matemáticas y en la actualidad también de la informática y la telemática, que hace uso de métodos y técnicas con el objeto principal de cifrar, y por tanto proteger, un mensaje o archivo por medio de un algoritmo, usando una o más claves."*

Por último, veamos también la siguiente definición, utilizada por los autores Menezes, Van Oorschot y Vanstone[3], quienes abordan en su libro una descripción matemáticamente formal al describir los protocolos y algoritmos criptográficos tratados allí:

"Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather one set of techniques."

Un intento de traducción sería:

"La criptografía es el estudio de técnicas matemáticas relacionadas con los aspectos de la seguridad de la información tales como la confidencialidad, la integridad de datos, la autenticación de entidad y de origen. La criptografía no comprende sólo a los medios para proveer seguridad de información, sino a un conjunto de técnicas".

Finalizada entonces esta breve introducción, y en cuanto a lo más reciente en la materia, recordemos que hasta hace poco menos de cuarenta años el estudio y el uso de la criptografía eran exclusivos de las fuerzas militares de cada nación. Las más desarrolladas utilizaban gran cantidad de recursos, tanto para el perfeccionamiento de sus técnicas como para el análisis de la información cifrada de sus enemigos. Fue por entonces cuando la investigación –y divulgación– académica tuvo un crecimiento repentino muy importante. Aproximadamente en el mismo período, comenzaba la proliferación de la computación en las empresas privadas y el uso de computadoras personales. Otro hecho importante también, en relación directa con la demanda de implementaciones criptográficas seguras por parte del sector privado y de los usuarios particulares es, sin lugar a dudas, el uso masivo de Internet a partir del año 2000. Hoy es posible comprar casi cualquier bien a través de este medio, pagar servicios, consultar nuestra información bancaria, etc.

Debido a estos factores, en la actualidad disponemos en los diferentes entornos de desarrollo, de variedad de implementaciones de sofisticados algoritmos criptográficos, conocidos hasta el momento como seguros. Éstos son analizados de forma constante por matemáticos y criptógrafos de todo el mundo.

Seguimos sin saber, por supuesto, cuán avanzados están en la materia los esfuerzos privados –secretos– de organismos gubernamentales relacionados con la seguridad o inteligencia de cada país. Pero el avance en el campo académico, y el escrutinio de miles de especialistas, puede asegurarnos hoy día que, valiéndonos de lo públicamente conocido y reconocido como seguro, seguiríamos el camino más prudente a la hora de resguardar nuestra información. Esto es así aunque sepamos, por la experiencia a lo largo de los años, que se han detectado y publicado problemas en algoritmos que se habían considerado seguros, y que, en la materia, lamentablemente poco puede asegurarse con absoluto rigor de manera definitiva.

Historia.

La historia de la criptografía se remonta, en su utilización inicial y limitada, hasta unos cuatro mil años de la mano de los egipcios. El libro de referencia respecto de esta historia –hasta las técnicas matemáticas modernas, no inclusive– se titula *The Codebreakers*, escrito por David Kahn[4]. Referimos al lector interesado en esta porción de la historia de la materia a este interesante libro, ya que aquí se comentarán muy brevemente sólo algunos hitos importantes.

Cuatro mil años atrás, un maestro egipcio escribió, mediante la utilización de jeroglíficos, la historia de su Señor, dando lugar al nacimiento de la criptografía. No se trataba de un sistema criptográfico que pudiese compararse con los actuales, pero utilizaba las técnicas de sustitución y transposición de símbolos de una manera similar a lo que se describirá en el apartado siguiente como base del concepto general de encriptación o cifrado. A través de la utilización de estas técnicas también se realizaron, desde las antiguas civilizaciones egipcias y occidentales, escritos de índole religiosa. Existen diversos estudios científicos que proponen interpretaciones respecto de la posible sustitución (cifrado) de palabras, partes de palabras y símbolos (letras y números) en el viejo testamento. Por aquel entonces también, en China se utilizó un pequeño código de cuarenta símbolos con propósitos militares, que los tenientes y soldados debían aprender para emitir y recibir –cifrando y descifrando, respectivamente– órdenes que pudiesen ser interceptadas por el enemigo.

Ya más avanzada la historia, en el mundo occidental la criptografía fue verdaderamente utilizada a partir del florecimiento de la diplomacia. Se conocen muchas historias respecto de la correspondencia de los reyes europeos de los siglos XV y XVI, como la que cuenta cómo el rey Enrique IV de Francia, en el año 1589, interceptó un mensaje cifrado de Felipe de España a uno de sus oficiales, y con la ayuda de un matemático de la corte, pudo descifrarlo.

El siguiente punto de inflexión en la historia de la criptografía, y último hasta la llegada masiva de las computadoras, se sucede durante la Segunda Guerra Mundial. En ese entonces, aunque no se dejaron de lado los sistemas manuales, se construyeron y se utilizaron máquinas de cifrado mecánicas y electromecánicas. Los alemanes desarrollaron diferentes variantes de su famosa máquina llamada “Enigma”. Se trataba de una máquina de rotores que automatizaba considerablemente los cálculos que era necesario realizar para las operaciones de cifrado y descifrado de mensajes. Su mecanismo estaba constituido fundamentalmente por un teclado, similar al de las máquinas de escribir, que controlaba una serie de interruptores eléctricos, un engranaje mecánico y un panel de luces con las letras del alfabeto. Hubo muchas variantes de estas máquinas, que hasta fueron fabricadas por empresas privadas con fines comerciales.

Existe un museo en Maryland, el *National Cryptologic Museum*, mantenido por la NSA, que exhibe una máquina “Enigma” con la cual los visitantes pueden cifrar y descifrar mensajes ellos mismos. Se muestra una foto a continuación.

Fig. 1-1. Máquina "Enigma" exhibida en el museo de la NSA.

La historia continúa ahora con las teorías de la información de Shannon y el avance matemático en la materia, que se abordarán con cierto detalle más adelante. De cualquier manera, conceptos elementales del cifrado, como los de sustitución y transposición, aunque con otras técnicas y metodologías, aplican hoy día de manera similar o como lo hicieron históricamente. Veremos de qué tratan estas técnicas en el apartado que sigue y cómo continuó la historia de la criptografía luego, ya entendida como criptografía moderna.

Cifrados por sustitución y transposición.

Antes de la llegada masiva de las computadoras, la criptografía consistía en algoritmos basados en caracteres (símbolos: Letras y números) que sustituían caracteres o los transponían entre sí, siempre intercambiando el uno por el otro. De estos algoritmos criptográficos, los más sofisticados hacían ambas cosas sobre secuencias que se repetían.

Por supuesto, hoy día, los sistemas criptográficos son más complejos, pero el principio general se mantiene. La diferencia principal, en el ámbito de la informática, es que los algoritmos trabajan con bits en lugar de caracteres (éstos son ahora los símbolos). Para los puristas, esto se trata sólo de un cambio de tamaño del alfabeto utilizado: De uno de veintiséis elementos en el idioma inglés –veintisiete en el español– a uno de dos –símbolos o caracteres 0 y 1–.

Muchos sistemas criptográficos actuales todavía combinan elementos de estas técnicas de sustitución y transposición. Un cifrado por sustitución es aquel donde cada carácter o símbolo del mensaje original es sustituido por otro texto cifrado. El receptor luego invierte la sustitución, a partir del texto cifrado, para obtener el mensaje original.

En el famoso algoritmo de cifrado “César”, que recibe su nombre en referencia a Julio César (por haber sido utilizado por él para proteger sus mensajes de importancia militar), cada letra del mensaje original es reemplazada por la letra que le sigue tres posiciones adelante, módulo 26. Esto significa que, con nuestro alfabeto, la letra A por ejemplo, es reemplazada o sustituida por la letra D, la letra B por la letra E, etc. Si acaso no quedó claro qué significa “módulo 26”, entiéndase que la letra X sería sustituida por la A, la Y por la B y la Z por la C. Es decir, se vuelve a empezar sobre el alfabeto o espacio de caracteres posibles. Se trata de un algoritmo de sustitución simple, ya que el alfabeto cifrado es una rotación directa (en tres posiciones) del alfabeto original y no una permutación arbitraria.

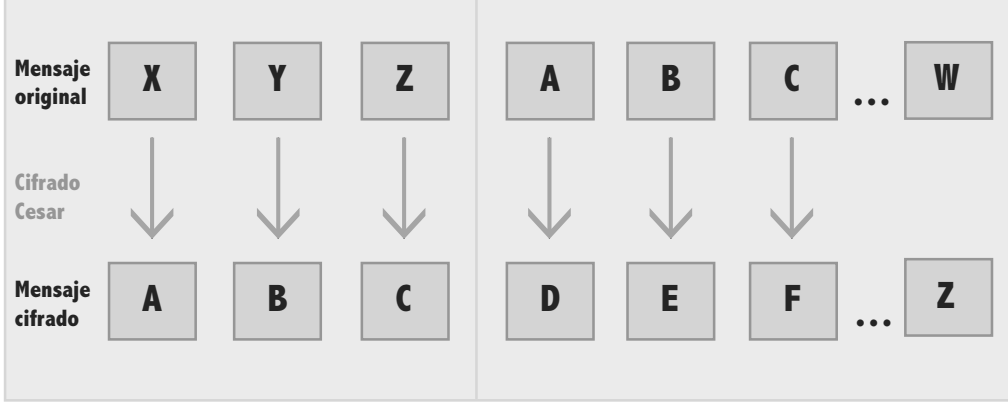

Fig. 1-2. Cifrado “César”.

El programa ROT13, históricamente disponible en los sistemas UNIX, también implementa un algoritmo de cifrado por sustitución simple. En éste, la letra A es reemplazada, en el texto cifrado, por la letra N; la B por la O, etc. (Nótese que la letra Ñ no es parte de la codificación de caracteres ASCII).

Debe aclararse que este programa no se utiliza con fines de resguardar confidencialmente información; o más bien, sería mejor decir que su uso es sólo para ocultar mensajes o archivos de texto cuando se debe permitir el fácil recupero de la información original. Imagínese, por ejemplo, las soluciones de un crucigrama, que en el diario se imprimen cabeza abajo.

Con el fin de generalizar lo anterior, diremos que nunca debe usarse un sistema de este tipo con la intención de proteger información sensible. Podríamos vernos inclinados a pensar que si en lugar de realizar un desplazamiento o sustitución por tres posiciones (cifrado "César") o por trece (cifrado del programa ROT13) implementásemos un algoritmo que sustituya los caracteres del mensaje original de a siete posiciones, nadie podrá descifrar el resultado obtenido. Esto es falso, ya que mediante un análisis sencillo, un analista experimentado podrá solucionar, "romper" o "quebrar" (en inglés, *break*) nuestro algoritmo en pocos minutos. Insistiremos con esto: Debemos fiarnos únicamente de funciones y algoritmos comprobados; por más que la metodología o implementación sean públicas, serán siempre (salvo que ideemos un nuevo algoritmo matemático robusto) más seguras que las implementaciones propias supuestamente secretas.

Los algoritmos de cifrado por transposición trabajan de manera diferente: Los caracteres del mensaje original se mantendrán y sólo cambiarán sus posiciones relativas entre ellos. En un cifrado por transposición de columnas simple, el mensaje original es escrito en columnas, es decir, de manera vertical, sobre papel cuadriculado por ejemplo. El texto cifrado será entonces lo que pueda leerse en este papel de manera tradicional (horizontal). Aunque se trate de un ejemplo simple, notaremos su punto débil más importante. El texto cifrado puede evidenciar, desde un principio, no sólo la técnica utilizada, sino propiedades clave del mensaje original, ya que las letras de un idioma o lenguaje aparecen en determinadas cantidades, es decir, tienen la misma frecuencia estadística en el mensaje original y en el cifrado. Aunque al procesar la información o mensaje cifrado, a través de una segunda transposición, aumentaría considerablemente la seguridad del sistema (y de hecho existen cifrados más complejos que implementan ésta y otras técnicas más sofisticadas), esta metodología no es tan común en implementaciones actuales como lo es el cifrado por sustitución.

Hemos hablado en el apartado anterior de la máquina de cifrado alemana "Enigma". Tenemos allí un ejemplo de estas técnicas, ya que cuando se pulsaba una tecla, por ejemplo la correspondiente a la letra A, a través de su sistema electromagnético y, luego, a partir de los diferentes procesos de sus rotores (el corazón de las sucesivas transposiciones y sustituciones), el símbolo cifrado terminaba siendo, por ejemplo, una letra J.

Esteganografía.

Describiremos en este apartado un término en cierta medida periférico a la noción más purista de la criptografía. La esteganografía trata de cómo ocultar, dentro de un mensaje público, información secreta. En este caso, el hecho mismo de que existe información "extra" a la original es un secreto. Históricamente, un típico ejemplo de estas técnicas fue la utilización de tinta invisible en las cartas. La información o mensaje original es la carta manuscrita. Con tinta invisible, el emisor o remitente escribirá el mensaje secreto. El receptor de la carta (y del mensaje oculto) tendrá a bien calentar el papel o aplicarle la sustancia correspondiente para la revelación del secreto.

El concepto clave aquí es que el cartero ni siquiera sabe que existe un secreto. Si el remitente, o emisor, y el receptor hubiesen utilizado otra técnica criptográfica y el cartero fuese lo suficientemente curioso como para abrir el sobre o forzar el sello, las letras, números y símbolos sin sentido aparente le hubiesen sugerido la existencia de un mensaje que se consideró privado.

Otros ejemplos históricos de la esteganografía se remontan hasta la antigüedad. Los griegos utilizaban tablas de madera recubiertas de cera sobre las cuales escribían con objetos punzantes y luego raspaban para poder escribir de nuevo (recuérdese la expresión "*tabula rasa*"). Se conoce que fueron ocultados mensajes escritos directamente sobre la madera, que posteriormente se recubría con cera. De parte de Herodoto también conocemos la utilización de tatuajes en el cuero cabelludo de un esclavo que, sin evidenciarlo y ya con el cabello crecido, contenían un mensaje o una información secreta (por ejemplo, una advertencia a Grecia respecto de planes de invasión por parte de los Persas).

Por último, para citar una aplicación moderna, comentaremos que estas técnicas se han utilizado en archivos de imágenes, con programas especialmente diseñados para agregar a un archivo JPG, por ejemplo, pequeñas diferencias codificadas que apenas alterarán la imagen en la pantalla. Luego, con el mismo programa, se recupera la información extra agregada.

Como algunas de las consideraciones descriptas en el apartado anterior, debe saberse que esta técnica no se recomienda para implementaciones actuales. Se describe por razones históricas y para tener entender por qué no hay que utilizarla como introducción a los sistemas de criptografía basados en algoritmos matemáticos.

Aquí, además, hemos de destacar especialmente que tenemos un claro ejemplo relativo a los aspectos negativos de la seguridad por ocultación. No es seguro fiarse de que no descubrirán el secreto. Si por algún motivo se evidencia o sospecha que existe un mensaje oculto, éste podrá ser descubierto de manera sencilla.

Criptografía moderna.

Concluimos previamente lo referido a la historia de la criptografía con una alusión a Claude Shannon. Podemos decir que la criptografía moderna se desarrolla recién a partir de sus trabajos en las áreas de la teoría de la información y de las comunicaciones. Es considerado por muchos autores el padre de la criptografía matemática, ya que sus artículos y libros respecto de la teoría matemática de la comunicación y la teoría de los "sistemas secretos" –que escribió alrededor del año 1950, pero que fueron publicados recién a mediados de los años setenta–, han establecido las bases para las implementaciones de los algoritmos criptográficos actuales.

El público general tuvo acceso a gran parte de estos trabajos y recién a partir de mediados de los años setenta a otros posteriores, ya que habían sido captados exclusivamente por agencias de seguridad de los Estados Unidos, como la NSA. De esta época también ha de destacarse la llegada de las computadoras y la publicación del primer borrador del algoritmo de criptografía simétrica DES (*Data Encryption Standard*). El DES fue el primer algoritmo público, basado en técnicas matemáticas y criptográficas modernas, aprobado por la NSA. Aunque en la actualidad se considera inseguro, por la corta longitud de su clave o llave (56 bits), que lo expondría a ataques de fuerza bruta, aún es utilizado en muchas implementaciones (particularmente su versión mejorada 3DES o TripleDES). Más adelante, DES fue reemplazado por el AES (*Advanced Encryption Standard*) en el año 2001, basado en el algoritmo desarrollado por dos criptógrafos belgas.

Al continuar con la historia de la criptografía moderna, un hecho importantísimo y que también determinó, por ejemplo, cómo hoy compramos de manera segura a través de Internet, ha sido la publicación de un artículo revolucionario (de Whitfield Diffie y Martin Hellman) respecto de nuevas direcciones en la criptografía. Se trataba, en resumen, de un nuevo método para distribuir las llaves o claves criptográficas y esto generó lo que más adelante conoceríamos como cifrado asimétrico o de llave pública.

Antes de esto, todos los sistemas criptográficos (de cifrado simétrico) requerían que ambas partes, emisor y receptor, conociesen o acordasen la misma clave criptográfica o contraseña, que, por supuesto, debían mantener en secreto. Con esta nueva metodología, en cambio, se utilizan dos claves o llaves criptográficas (una privada y una pública), relacionadas matemáticamente y que se combinan. De esta manera, el emisor con su par de claves o llaves podrá cifrar mensajes para un receptor particular conociendo sólo su clave pública. Se desarrollará este modelo más adelante, pero se desprende ya de lo anterior la solución a la problemática necesidad del intercambio de claves criptográficas secretas, o contraseñas, del modelo simétrico.

Los otros usos o aplicaciones de las técnicas criptográficas modernas son, además de la confidencialidad y valiéndose principalmente de las dos metodologías de cifrado mencionadas y de las funciones de una vía y/o *hashing*, aquellos que implementan mecanismos para verificar la integridad y/o autoría de una información o documento particular. Consideremos dentro de estos a, por ejemplo, los mecanismos de

firma digital, ya parte de la legislación de muchos países, basados principalmente en las técnicas de cifrado asimétrico.

Para terminar se aclara que hasta aquí han llegado los avances en la materia. En la actualidad, existen diferentes protocolos que describen cómo ha de ser la interacción entre las partes y los algoritmos, que se especifican matemáticamente al establecer de manera concreta las operaciones o pasos que le corresponderá a cada parte para realizar la tarea en cuestión. Por último, existen programas, entornos de desarrollo y librerías para desarrolladores, que implementan estos algoritmos y brindan la posibilidad de utilizar criptografía al usuario o de incorporarla en programas al desarrollador. Si bien podemos confiarnos de los últimos avances –hasta estandarizados en algunos casos–, ya que son puestos a prueba continuamente por expertos pertenecientes a la comunidad académica y a la industria (que intentan "romperlos" o "quebrarlos"), no es posible asegurar que tal cosa nunca suceda.

El estándar actual para criptografía simétrica, el AES, se considera muy seguro, pero ha habido sugerencias de expertos respecto de la necesidad de aumentar la longitud de su clave. Muchas implementaciones han sufrido algunas adaptaciones y mejoras de diseño a lo largo de su historia. En los últimos años, también se han publicado los problemas del esquema de cifrado Wi-Fi o del sistema utilizado para cifrar y controlar el uso de los discos DVD. Por otra parte, no están demostradas formalmente algunas de las teorías matemáticas que fueron base de la criptografía de clave pública.

Al pensar en el hardware, en el cual podrían implementarse estas técnicas y algoritmos modernos y para darnos una idea del avance de la tecnología (después de ver una figura de la máquina "Enigma" en uno de los apartados anteriores), se muestra a continuación una foto de una supercomputadora (Roadrunner, de IBM) que es considerada la más veloz del mundo. Con un costo de construcción de 133 millones de dólares, alcanzó en el año 2008 un pico de $1\,026 \times 10^{15}$ operaciones de punto flotante por segundo.

Usos de la criptografía.

En esta sección hablaremos rápidamente sobre los problemas a los cuales la implementación de técnicas criptográficas puede dar solución. Se mencionaron anteriormente conceptos como la firma digital, basada principalmente en el cifrado asimétrico y contemplada ya en las leyes de muchos países.

También se ha visto que nos podemos valer de la criptografía frente a la necesidad de verificar o corroborar la integridad de una información o documento. Ya tenemos entonces una idea acerca de los posibles usos o aplicaciones de estas técnicas, o al menos deberíamos saber que no se limitan al resguardo de información en secreto entre dos o más partes –lo que se entenderá como confidencialidad–, sino que existe una variedad de situaciones en las cuales pueden aplicarse.

Fig. 1-3. Supercomputadora “Roadrunner” de IBM

Antes de adentrarnos en cómo es que lo hace, veamos brevemente qué es lo que puede hacer la criptografía por nosotros o por los usuarios de nuestros programas.

Confidencialidad.

La confidencialidad o privacidad de la información es el uso o aplicación principal de la criptografía. Acaso el más importante, es el más antiguo y, muchas veces, el que se supone como único aportado por las técnicas criptográficas. Implica básicamente el mantener en secreto una información determinada (un mensaje para ser transmitido en un

canal de comunicación inseguro, un documento almacenado en un medio no confiable, etc.). El objetivo, está claro, es que sólo aquellas personas que estén autorizadas tengan acceso a la información resguardada (esto es, cifrada o encriptada).

De cualquier manera, no tenemos que olvidarnos que existen diferentes modos de lograr la confidencialidad, desde la protección física hasta algoritmos matemáticos que transforman la información original en datos, en apariencia, no inteligibles. Por supuesto, este libro trata de cómo utilizar o incorporar en nuestros programas, prácticamente, estos algoritmos, pero no debe perderse de vista el objetivo principal, ya que si, por ejemplo, en una empresa cada persona anota su contraseña sobre su escritorio, ningún algoritmo podrá ayudarlos.

Autenticación.

Se trata de otro mecanismo, técnica o uso muy utilizado en la actualidad. Hablar de la autenticación o identificación implica hablar de la corroboración de la identidad de una entidad (una persona, una computadora, un sector de una compañía o empresa, etc.).

Otros tipos de autenticación se refieren, o se aplican, a mensajes particularmente. Aquí hablamos de *message authentication* –autenticación de mensajes–, donde lo que ha de corroborarse es la fuente de la información o mensaje. También es conocido como *data origin authentication* o autenticación de origen de datos.

Por último, entre éstos también podría englobarse a las firmas digitales de documentos electrónicos: Se trata de una forma de "autenticar" la autoría de, o la conformidad a, la información contenida en estos documentos con relación a una entidad determinada, firmante de tales contenidos.

Puede entenderse a la autenticación como un uso o aplicación relacionado con el de la identificación. Esta función aplica a ambas partes o entidades participantes en una comunicación y a la información en sí misma. Sépase que dos partes, al comenzar una comunicación, deberán identificarse entre ellas. La información transmitida deberá ser autenticada respecto del origen (fecha del origen, contenidos, fecha del envío o transmisión, etc.) Por estas razones, este aspecto de la criptografía es comúnmente dividido en dos clases principales: Autenticación de entidades y autenticación de origen de datos. Esto último incluye implícitamente la verificación de integridad de datos.

Verificaciones de integridad.

Al hablar de comprobaciones o verificaciones de integridad nos estamos refiriendo a un uso o aplicación de técnicas criptográficas cada vez más utilizado popularmente, en respuesta a las nuevas formas de ataque de *hackers*, virus y troyanos.

Concretamente, estas verificaciones o comprobaciones corresponden al aseguramiento de que una información particular (un archivo de Word, un programa de ins-

talación de otro programa, etc.) no haya sido alterada por personas no autorizadas o por cualquier otro medio desconocido.

Este mecanismo, entonces, ataca al problema de la alteración no autorizada de datos o información. Para asegurar la integridad de un documento por ejemplo, se debe tener la habilidad de detectar la manipulación de esta información, es decir, de sus contenidos, por partes no autorizadas, sabiendo que dentro de lo que se entiende por manipulación debe contemplarse lo que se agrege, elimine o sustituya de la información.

Mecanismos de no repudio.

Este uso o aplicación consta de la implementación de un mecanismo o técnica –mediante funcionalidades criptográficas, por supuesto– para prevenir que una entidad niegue un envío previo de información, un mensaje, una acción, etc.

Si fuese motivo de disputa, llegado el caso de que una parte o entidad negase, por ejemplo, una acción cometida o un mensaje enviado, a través de las técnicas de no repudio –si éstas han sido implementadas con anterioridad– se contará con los medios necesarios para resolver la situación.

Otros usos o aplicaciones.

Existen variados usos, además de los comentados, de técnicas o funcionalidades criptográficas, que proveen soluciones a problemas de diferente índole. Muchos de estos están basados en los anteriores o derivan de ellos.

Estas otras aplicaciones mencionadas incluyen:

- La autorización: Permiso concreto, a una parte o entidad, para el acceso o la realización de una tarea específica o para actuar bajo una identidad determinada.
- La validación: Medio de proveer una autorización puntual para el uso o la manipulación de información y de recursos.
- El control de acceso: Restricción de acceso a la información o a los recursos para las partes o entidades con privilegio suficiente.
- La certificación: Respaldo de información por una parte o entidad de confianza.
- El fechado: Registro de la fecha de creación o de existencia de una información determinada.
- El atestiguamiento: Verificación de la creación o de la existencia de una información determinada por una parte o entidad distinta a la del creador de la información.

- El recibo: Acuse de recibo de una información determinada.
- La confirmación: Acuse de recibo respecto de servicios que han sido prestados.
- De propiedad: Medio para proveer a una entidad el derecho legal para el uso, o transferencia a terceros, de un recurso o una información.
- El anonimato: Ocultamiento de la identidad de una parte o entidad involucrada en un proceso.
- La retracción o revocación: Medio para quitar o retraer –revocar– una certificación o una autorización.

Criptografía aplicada.

En esta sección veremos cómo se implementan, o aplican prácticamente, los diferentes usos que pueden darse a las técnicas criptográficas en la actualidad. Como se adelantó en la sección anterior, mediante la criptografía es posible solucionar, principalmente, problemas de confidencialidad, autenticación e integridad de una información o mensaje.

Recordemos entonces que encriptar o cifrar se refiere a un conjunto de técnicas, que protegerán o verificarán una información particular, valiéndose de un algoritmo criptográfico. Dejando de lado las funciones de una vía y *hashing*, que en realidad no realizan encriptación ni desencriptación de datos, las técnicas criptográficas asumen la utilización de llaves o claves. Quien no conozca o posea la clave específica correspondiente, no podrá descifrar un mensaje para recuperar la información original.

Luego de aclarar algunas cuestiones respecto de la terminología utilizada en el libro, veremos lo referente a las generalidades de los protocolos criptográficos principales, donde conoceremos los lineamientos que se han de seguir en función de cada parte involucrada en los distintos procesos. Seguirán las descripciones generales de algunos conceptos elementales, necesarios para la utilización de los algoritmos criptográficos prácticamente y, por último, detallaremos a grandes rasgos aquellos algoritmos que son tratados en el libro.

Terminología utilizada.

Este apartado cumple, o intentará cumplir, con las aclaraciones de rigor en cuanto a la terminología utilizada a lo largo del libro. Si bien hay una coherencia casi total entre autores de lengua inglesa, respecto de los términos criptográficos en general, en textos de lengua española se evidencia fácilmente que no todos traducimos de la misma forma.

Con relación a los términos ingleses, se aclara que existen algunos que se utilizarán a lo largo del libro sin que se especifique su traducción, como *hashing* por ejemplo. Traduciremos algunos como *message digest* a resumen de mensaje, pero no se sugerirá una palabra en español para términos que son utilizados en inglés en la práctica, aunque éstos no figuren en el diccionario de la lengua española. Estos términos, a veces, no disponen de una traducción literal o cuando la tienen, no corresponde o no se aplica totalmente al significado que expresa en cuestiones relacionadas con la materia que nos ocupa.

Se describirán a continuación algunos términos que, en textos de autores españoles, probablemente el lector encuentre traducidos de manera diferente:

- **Cifrar o encriptar**. A lo largo del libro se usarán indistintamente. El término correcto, en realidad, sería cifrar. Se eligió utilizar encriptar como sinónimo por el uso común que se ha hecho del término, por influencia de *encrypt,* del inglés. Ramió Aguirre, quien defiende la utilización de cifrar, hace una broma al respecto y nota que encriptar podría ser el acto de introducir a alguien dentro de una cripta.

- **Origen y destino**. También emisor y receptor, fuente y destino, remitente y destinatario, etc. Se trata de un par de términos para referenciar a las dos partes o entidades involucradas en alguna comunicación. Los autores de habla inglesa utilizan diferentes sinónimos de estos términos hasta dentro de un mismo libro, a veces dependiendo del tipo de comunicación de la que se trate. En este libro sucede lo mismo.

- **Texto-plano y texto-cifrado**. Texto-plano se traduce también, acaso correctamente, como "texto en claro". A lo largo del libro se utilizará texto-plano, con relación a la traducción directa o literal del término inglés *plaintext,* más popularmente utilizado. Texto-cifrado deviene de *ciphertext*.

- **Llave o clave**. Lo que ha de destacarse en primera instancia es que, aunque en este libro se aclara en cada caso, pueden no ser consideradas la misma cosa. Si bien en textos en español, como en algunos en inglés, se usan indistintamente, siempre debe quedar en claro cuándo, respecto del término clave, se lo está usando como sinónimo de llave o de contraseña, puesto que no es lo mismo.
- **Autenticación o autentificación.** Aquí las diferencias son más sutiles. En este libro se utilizará autenticación como traducción del término inglés *authentication*. El término español corresponde a autorizar o legalizar algo, como dar fe de la verdad de un hecho o documento, de manera que aplica al uso que se le da a la palabra en criptografía. Con el fin de seguir la línea general de utilizar, entre las variantes posibles, los términos más parecidos a su versión correspon-

diente en el idioma inglés, se usará en este libro mayormente la primera alternativa, y no autentificación, como puede encontrarse en otros textos en español.

- **De bloque y en flujo.** Estos términos se refieren a la forma o tipo de operación de algoritmos simétricos. Del inglés *block* y *stream ciphers,* respectivamente. Se usarán en este libro mayoritariamente las expresiones **"en" flujo** para el tipo *stream* –se encuentra traducido en textos en español también como "de" flujo o "por" flujo– y **de bloque** para los cifrados *block*.

- **Otros términos**. Aquellos otros términos, a veces conflictivos y de difícil traducción, se especificarán en el idioma original y se dará cuenta de lo aproximado o arbitrario de la traducción propuesta en cada caso.

Protocolos criptográficos.

Un protocolo, en términos generales, se entenderá en este libro como una serie de tareas o de pasos, que involucran a dos o más partes, desarrollada para lograr un objetivo.

Al acordar el uso de un protocolo, debe conocerse por todas las partes que han de implementarlo. Es importante destacar que el protocolo no debe ser ambiguo y debe ser completo (tareas o pasos definidos para cada situación).

Específicamente, un protocolo criptográfico es un protocolo que utilizará como herramienta, o estará basado sobre, un algoritmo criptográfico. Existen muchos protocolos criptográficos, diseñados con diferentes objetivos, que dan solución a distintos problemas. Sigue a continuación un detalle breve, que se podrá encontrar ampliado en el segundo capítulo del libro.

- **Protocolos de criptografía simétrica.** Es la forma de criptografía más utilizada actualmente. Dos partes son las involucradas: Una, a quien llamaremos "A", pretende enviar un mensaje a la otra, "B". Básicamente, se trata de lo siguiente: Estas partes acordarán una llave o clave de manera secreta. Luego, la parte "A" encriptará el mensaje (usando el algoritmo y la llave acordados) y enviará este resultado (texto-cifrado) a la parte "B" a través del canal que se ha asumido inseguro. En ese momento, la parte "B", valiéndose del mismo algoritmo y la llave acordada previamente (por canal seguro), podrá obtener el mensaje original, desencriptando el texto-cifrado recibido.

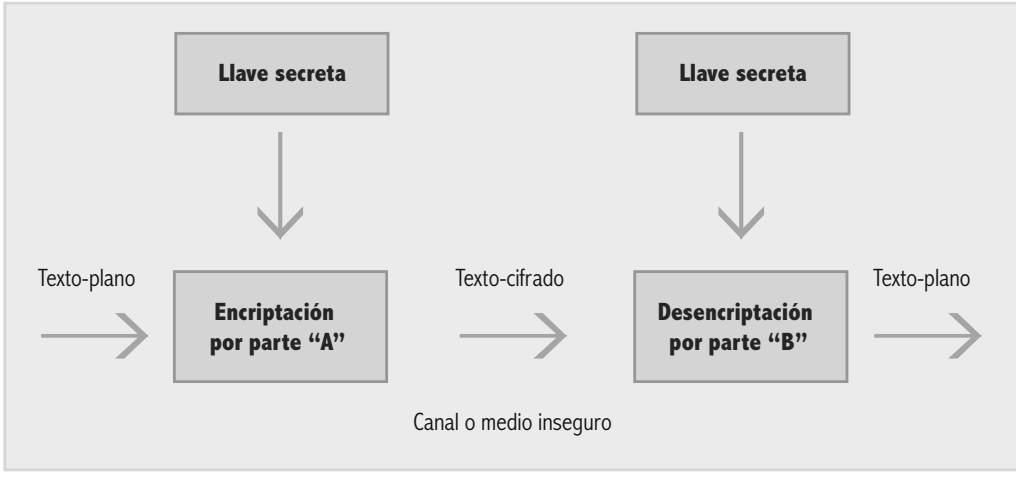

Fig. 1-4. Esquema de protocolo de criptografía simétrica.

- **Funciones de una vía y *hash***. No se trata en este caso de un protocolo criptográfico en sí, pero las funciones de una vía y *hash* forman parte fundamental de otros protocolos y algoritmos criptográficos. A través de ellas, se podrá computar su resultado de manera rápida, pero la obtención de la entrada original, es decir, lo que fue el parámetro de la función a partir del resultado, será inviable. Debe recordarse que no cifran ni descifran información.

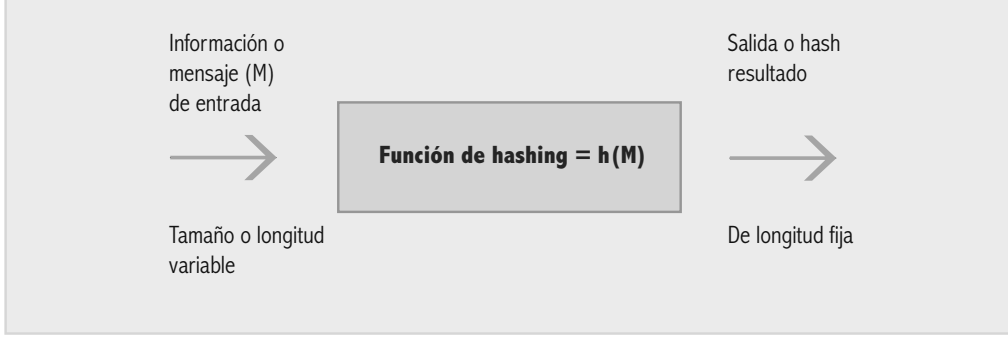

Fig. 1-5. Esquema de funciones de una vía.

- **Protocolos de criptografía asimétrica.** Acordado el uso de criptografía asimétrica entre las partes, la parte "B" (quien debe recibir el mensaje) enviará a la parte "A" (quien debe comunicar la información) su llave pública. La parte "A", entonces, encriptará la información o mensaje utilizando la llave pública recibida de "B". La parte "A" enviará a "B" el resultado de esta encriptación. Entonces, "B" desencriptará el texto-cifrado recibido, valiéndose de su llave privada.

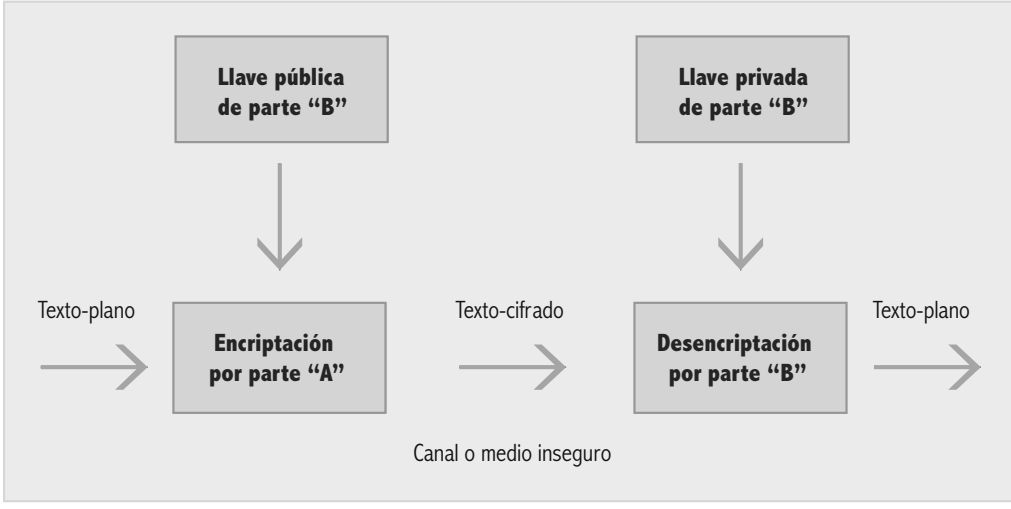

Fig. 1-6. Esquema de protocolo de criptografía asimétrica.

- **Protocolos de firma digital.** La firma digital corresponde a la versión informática de la firma personal manuscrita o firma ológrafa; esto es, se usará para probar la autoría de, o el acuerdo a, la información contenida en un documento electrónico. Existen diferentes protocolos para implementar esta funcionalidad criptográfica. La implementación más utilizada involucra la utilización de funciones *hash* junto con criptografía asimétrica. Básicamente, la parte autora o firmante del documento firmará el *hash* resultante. Esto es: La parte "A" producirá el *hash* del documento, lo encriptará con su llave privada y enviará esto, junto con el documento, a la parte "B". "B" computará por su cuenta el *hash* sobre el documento. Luego, mediante la llave pública de "A", desencriptará el *hash* que "A" ha computado. Entonces, "B" podrá comparar ambos *hashes* y verificar la firma.

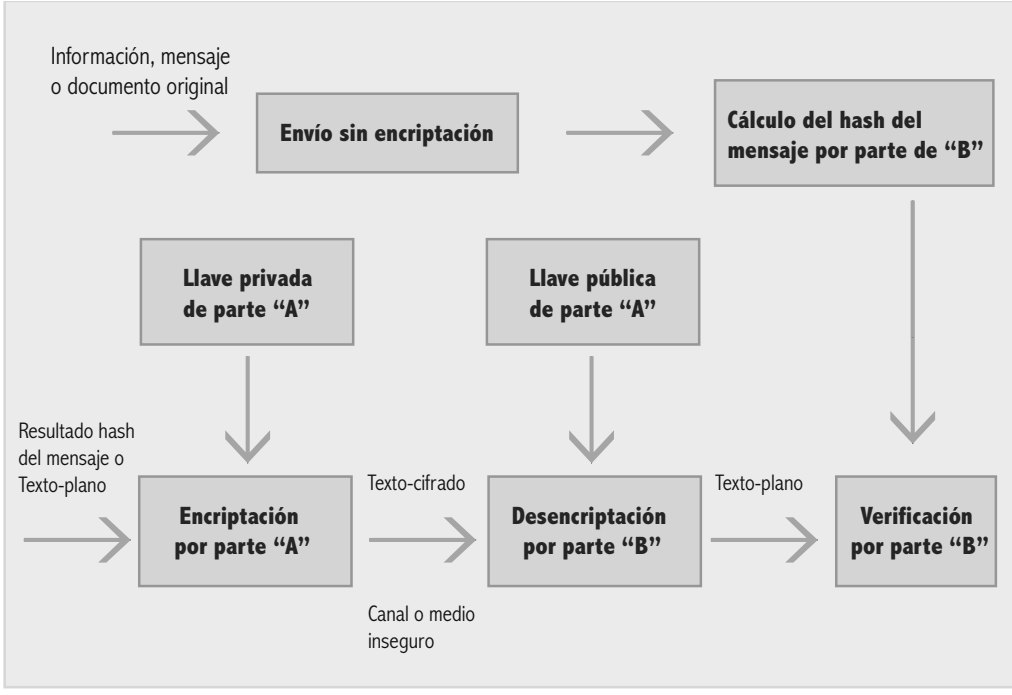

Fig. 1-7. Esquema de protocolo de firma digital (llave pública).

Nociones preliminares al uso de algoritmos criptográficos.

En el apartado anterior hemos visto de manera general los conceptos clave o principales que se han de destacar –centrados en la interrelación entre las partes involucradas– para cada uno de los tipos o formas más importantes de criptografía: Simétrica, asimétrica, funciones de una vía y *hash* y firma digital (ésta ultima entendible también como parte de la criptografía asimétrica).

Veremos ahora cuestiones relativas a cómo se han implementado estos protocolos prácticamente a la aplicación concreta de las tareas de cada parte, es decir, a los algoritmos, para la realización de una comunicación segura o la verificación de una información determinada.

Para esto nos valdremos de un diagrama general y, *a posteriori*, se describirán las nociones necesarias –que no se desprenden de los pasos referidos al tratar los distintos protocolos– para que, a la hora de utilizar alguno de los algoritmos implementados para los entornos de desarrollo tratados en el libro, se comprenda cuál es el rol de los diferentes elementos involucrados. Se advierte al lector que las descripciones son mínimas y se recomienda la lectura del segundo capítulo del libro antes del abordaje de la parte práctica.

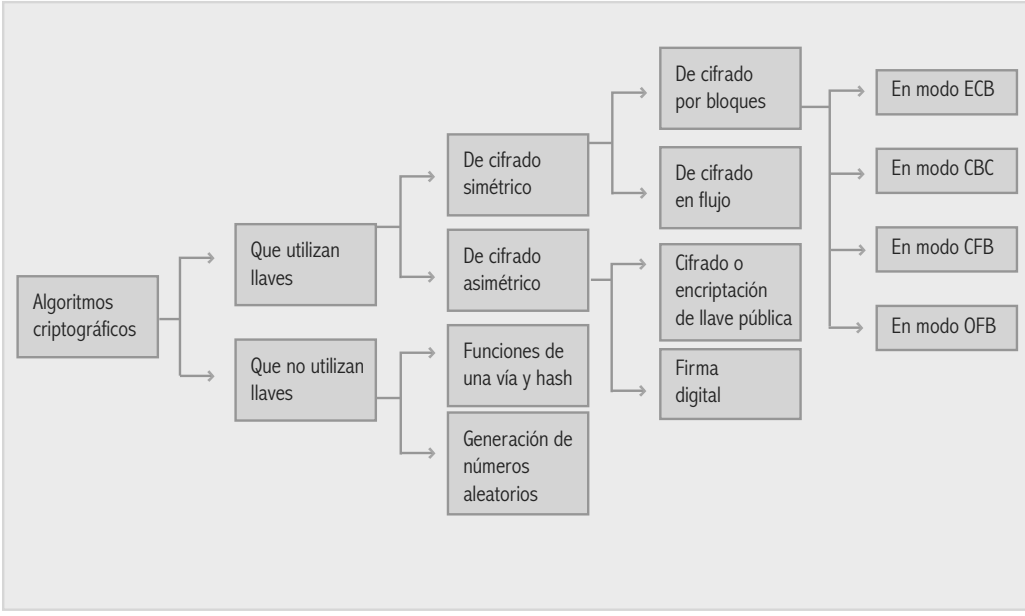

Fig. 1-8. Diagrama general de una categorización posible de algoritmos criptográficos.

Como vemos en el diagrama, los algoritmos de criptografía simétrica pueden dividirse en dos grandes grupos, de acuerdo con cómo trabajan o procesan el texto-plano de entrada. Esto es lo que se entiende por el "tipo" de un algoritmo.

Existen las alternativas que cifran por bloque a la información de entrada, esto es, la tratan de a porciones, o en bloques, de una longitud fija y predeterminada. Por otra parte, existen los algoritmos que operan sobre la entrada en forma de flujo, o *stream*, de a un bit, byte o hasta palabras de 32 bits inclusive, cada vez.

Los diferentes "modos" en que puede operar un algoritmo de criptografía simétrica, mostrados en el diagrama, corresponden a: ECB, en el cual un bloque de entrada o texto-plano se encripta en un bloque de salida o texto-cifrado, independientemente; CBC, donde existe un encadenamiento que agrega una retroalimentación al bloque *n+1* después de haber cifrado el bloque *n*; CFB, también retroalimentado, permitirá utilizar tanto un cifrador por bloques como un cifrador en flujo; y OFB, que se trata de otro modo para utilizar tanto un cifrador por bloques como un cifrador en flujo.

Es importante notar la importancia de un concepto relativo a los cifrados por bloque retroalimentados. El vector de inicialización (*initialization vector* o IV) se utiliza como primera retroalimentación o *feedback* en el proceso de cifrado; por eso es parámetro en estos algoritmos cuando se utilizan en este modo. Por lo general, se utiliza infor-

mación aleatoria. El objetivo es lograr que dos mensajes iguales no generen el mismo texto-cifrado. No es preciso que este bloque sea secreto.

Con respecto a la generación de números aleatorios, se remite al lector a la sección dedicada al tema en el segundo capítulo; pero, por lo pronto, se adelanta la recomendación de no utilizar las funcionalidades provistas por el sistema operativo o por el entorno de desarrollo para la obtención de tales números, sino las provistas por los *frameworks* o librerías criptográficas.

Algoritmos criptográficos.

Veremos en este apartado un breve detalle de los algoritmos criptográficos que actualmente están implementados en las herramientas de desarrollo, lenguajes de programación o librerías y *frameworks* adicionales, que serán tratados en los capítulos prácticos.

Téngase en cuenta que un algoritmo es la especificación concreta y detallada de las operaciones que han de realizarse para lograr un objetivo o resultado o, finalmente, resolver un problema. Los algoritmos criptográficos en particular son especificados mediante funciones matemáticas, que son utilizadas en los procesos de encriptación o desencriptación de datos (que corresponden a los parámetros o entradas de estas funciones).

Este breve detalle pretende servir de guía general a la hora de conocer las características principales de un algoritmo, si se trata de una función de *hashing*, de un algoritmo de criptografía simétrica, asimétrica o de firma digital; tamaño o longitudes de llaves, tamaño de bloque (si aplica), etc. En el segundo capítulo, cada uno de estos algoritmos será descripto con mayor detenimiento.

Dentro de los algoritmos de criptografía simétrica encontramos:

- DES (*Data Encryption Standard*): En 1977 fue estándar federal de los EE.UU. (NIST) y fue reemplazado en 2002 por el algoritmo AES. Es un algoritmo de cifrado por bloques, de 64 bits de longitud. La longitud de la llave es de 56 bits. Se recomienda la utilización de algoritmos más modernos en implementaciones actuales.
- TripleDES o TDES: Basado en el algoritmo DES. Es una versión mejorada de este último. Su forma más simple (ya que hay variantes) consiste en la triple aplicación de DES. La longitud de la llave será de 168 bits, si se opera con tres llaves DES, y de 112 bits si se utilizan dos. El tamaño o longitud del bloque, como DES, es de 64 bits.
- AES (*Advanced Encryption Standard*): Desde el año 2002 es estándar del gobierno federal de los E.E. U.U. (FIPS). También conocido como algoritmo Rijndael (con este nombre fue presentado a concurso). Rijndael y AES corresponden a diferentes nombres de un mismo algoritmo de cifrado por bloques, pero AES

especifica un tamaño de bloque fijo de 128 bits de longitud; para la llave: Tamaños de 128, 192 ó 256 bits; en tanto que Rijndael, mientras sean múltiplos de 32 bits, las longitudes de llave y de bloque podrán variar entre 128 y 256 bits.

- IDEA (*International Data Encryption Algorithm*): Trabaja con bloques de 64 bits y con llaves de 128 bits de longitud. Diseñado en el año 1991, fue utilizado en PGP 2.0. Es uno de los algoritmos de cifrado por bloques opcionales en el estándar OpenPGP.
- Blowfish: Diseñado por Bruce Shneier en el año 1993. Libre de patentes, de dominio público o libre. La longitud de la llave es variable, puede ser de hasta 448 bits. El tamaño o longitud de bloque es de 64 bits. Es un algoritmo de cifrado por bloques.
- Twofish: Algoritmo de cifrado simétrico por bloques también diseñado por Bruce Shneier en el año 1998. A los bloques les corresponde una longitud de 128 bits. La llave, también de longitud variable, será de hasta 256 bits.
- RC4: Algoritmo de cifrado en flujo (*stream cipher*). Fue desarrollado en el año 1984 por Ron Rivest para la compañía RSA Data Security Inc. Públicamente conocido desde el año 1994, es utilizado en protocolos como el SSL y el WEP (redes inalámbricas). Aquí el *keystream*, o flujo de llave, es generado seudoaleatoriamente a partir de permutaciones inicializadas con una llave de entre 40 y 256 bits de longitud.

Algoritmos de *hashing* o de resumen de mensajes y autenticación de mensajes:

- MD5: Versión mejorada de su antecesor, el algoritmo MD4, publicada en el año 1991. Es un algoritmo de *hashing* criptográfico y produce, como corresponde a estas funciones, un *hash* criptográfico, o resumen de mensaje, a partir de una entrada de cualquier tamaño. El *hash* resultante constará de 128 bits de longitud. En lo últimos años se han descubierto algunos problemas en el algoritmo, por lo que se recomienda utilizar un algoritmo alternativo (como SHA-1 o alguno de la familia SHA-2, por ejemplo) en implementaciones actuales.
- SHA-1 (*Secure Hash Algorithm*): Diseñado por la NIST y la NSA. Publicado como estándar en el año 1995. Existe otro estándar, llamado DSS por *Digital Signature Standard*, que utiliza SHA-1 y el algoritmo DSA de firma digital. Generará un *hash* criptográfico de 160 bits de longitud.
- SHA-2 (SHA-224, SHA-256, SHA-384, y SHA-512): Conjunto o familia de algoritmos de *hashing* criptográfico conocido como SHA-2. Las últimas variantes fueron publicadas en el año 2001. El número que sigue a las siglas SHA especifica la longitud en bits del *hashing* criptográfico que genera cada algoritmo.
- HMAC: Se trata, en realidad, de un algoritmo de autenticación de mensajes (MAC) y no de *hashing*. Involucra la utilización de una llave o clave secreta y

cualquier función de *hashing*. Fue publicado en el año 1996 y es también un estándar federal de los EE.UU.

Algoritmos de criptografía asimétrica o de llave pública y firma digital:

- RSA: Algoritmo de cifrado asimétrico más popular en la actualidad. Fue publicado en el año 1977. Aunque se utilizan hoy día llaves de 1 024 bits, se recomienda al menos una longitud de 2 048 para implementaciones actuales. El algoritmo sirve tanto para encriptar y desencriptar información, como para la generación de firmas digitales.
- ElGamal: Algoritmo de cifrado asimétrico que también, como RSA, puede ser utilizado tanto para la encriptación y desencriptación de información como para la firma digital de documentos electrónicos. El algoritmo fue descripto en el año 1984. Está implementado actualmente en aplicaciones criptográficas muy populares, como el software libre GNU Privacy Guard, o GPG, y en versiones recientes de PGP.
- DSA (*Digital Signature Algorithm*): Algoritmo de Firma Digital. Estándar propuesto por la NIST en el año 1991. El algoritmo de llave pública para la firma digital, DSA, es una variante de los algoritmos ElGammal y Schnorr.
- Diffie-Hellman: Primer algoritmo de llave pública, que data del año 1976. No es posible utilizarlo para el cifrado o encriptación de información; sólo permitirá a dos partes, sin conocimiento ni acuerdos previos entre ellas, establecer una llave secreta en conjunto, que podrán compartir luego utilizando criptografía simétrica.

Herramientas de desarrollo disponibles.

Se comentará aquí, brevemente, acerca de las herramientas de las cuales podremos valernos para implementar funcionalidades criptográficas en nuestras aplicaciones, utilizando los lenguajes de programación .NET, Java o PHP. En los próximos capítulos técnicos se desarrollarán, de manera detallada, tanto las funcionalidades incorporadas por defecto en estos lenguajes, como aquellas provistas por librerías o *frameworks* adicionales, desarrolladas por terceros y libremente disponibles en Internet (alternativas *open-source* o de código abierto).

Aún dentro de la parte práctica del libro, luego de los tres capítulos referidos a los lenguajes mencionados, se tratarán las funciones criptográficas provistas por tres motores de base de datos relacionales: Microsoft SQL Server, Oracle y MySql.

Explicados ya algunos conceptos elementales de criptografía, y habiendo conocido qué son a grandes rasgos y para qué pueden ser utilizados los algoritmos más importantes en la actualidad, veremos como adelanto de los capítulos técnicos, qué nos ofrece cada uno de los entornos de desarrollo o programación.

Herramientas de desarrollo criptográficas para entornos Java.

En el lenguaje de programación Java, todo cuanto a criptografía se refiera, gira en torno a la *Java Cryptography Architecture* (Arquitectura Criptográfica de Java o JCA).

Se trata de una especificación del lenguaje Java; la JCA propone interfaces y clases abstractas que sirven de modelo para la implementación de algoritmos criptográficos. Se verá detalladamente más adelante, pero podemos adelantar que esta arquitectura depende de la implementación de *engines* o motores y *providers* o proveedores, que implementarán y proveerán servicios criptográficos de los que podremos valernos en nuestras aplicaciones Java, utilizándolos de una manera *a priori* formalizada, independientemente de la funcionalidad o implementación particular.

La JCA es parte del lenguaje Java, más específicamente, parte del API de seguridad del lenguaje, desde la versión JDK 1.1, cuando JCE era una extensión separada. La JCE, de *Java Cryptography Extension* (Extensión Criptográfica de Java), provee un *framework* e implementaciones de algoritmos para la encriptación, generación e intercambio de llaves criptográficas y autenticación de mensajes. La extensión provee estas implementaciones siguiendo los lineamientos definidos por la JCA, lo que quiere decir que implementa proveedor o *provider* de funciones criptográficas.

La API de la JCE incluye: Encriptación simétrica, con algoritmos como DES, RC2, RC4 e IDEA; encriptación asimétrica, a través del algoritmo RSA (sólo a partir de la versión 5.0 de la JDK); encriptación basada en contraseñas (*Password-Based Encryption* o PBE); manejo de llaves, autenticación de mensajes (MACs o *Message Authentication Codes*) y *hashing*.

Como una alternativa muy interesante, veremos en los capítulos técnicos una librería criptográfica *open-source* o de código abierto, que provee una gran cantidad de implementaciones de diferentes algoritmos. También abordaremos el proyecto de software libre del grupo de *The Legion of the Bouncy Castle*. La API criptográfica de este grupo funciona con todos los entornos y versiones de Java, desde la recientemente liberada J2ME hasta la JDK 1.6, con la infraestructura adicional para adecuar los algoritmos para el *framework* JCE.

Herramientas de desarrollo criptográficas para entornos .NET.

En el entorno de desarrollo .NET, hablar de criptografía implica hablar de las funcionalidades provistas por el *namespace System.Security.Cryptography*. Bajo esta raíz común, encontraremos servicios criptográficos que incluyen la encriptación y desencriptación de datos y otras operaciones como las de la generación de *hashes* criptográficos, la generación de números aleatorios y la autenticación de mensajes, entre otros.

Es así entonces que desde el mismo *framework* .NET encontramos clases que proveen implementaciones de varios algoritmos criptográficos basados en estánda-

res. Las implementaciones de estos algoritmos pueden ser utilizadas de manera sencilla, con valores por defecto seguros. Además, el modelo criptográfico del *framework* .NET de herencia, diseño de *stream* o flujo y configuración son extensibles.

Lo veremos detalladamente en el capítulo técnico referido a este tema, pero adelantaremos que el sistema de seguridad del *framework* .NET implementa un patrón extensible de herencia de clases derivadas, donde la jerarquía es la siguiente: Existirá una clase de tipo de algoritmo, como por ejemplo las clases SymmetricAlgorithm y HashAlgorithm, a nivel abstracto. Luego, encontraremos clases de algoritmos, que heredan de una clase de tipo de algoritmo, por ejemplo RC2 o SHA-1. Este nivel también es abstracto. Por último, veremos que existen las "implementaciones" de estas clases de algoritmos, que heredan de RC2 o SHA-1 según lo anterior. Ejemplos de estas últimas son las clases RC2CryptoServiceProvider y SHA-1Managed. Este nivel, por supuesto, ya no es abstracto, sino implementado completamente.

Herramientas de desarrollo criptográficas para entornos PHP.

Desde la versión 4.0 de PHP, ya contábamos con la función `md5()` incorporada dentro del lenguaje para la generación de *hashes* criptográficos con ese algoritmo. Ésta ha sido ampliamente implementada en aplicaciones Web para cifrar contraseñas, por ejemplo, o hacer pasajes seguros de parámetros. También, desde la versión 4.3 se encuentra disponible una implementación del algoritmo de *hashing* SHA-1.

Sin embargo, ante la necesidad de implementar otros protocolos criptográficos, sean para la generación de *hashes* o de cifrados simétricos o asimétricos, nos veremos obligados a utilizar código (entiéndase librerías o módulos) de terceros. En el capítulo técnico de PHP abordaremos cada alternativa detalladamente, aquí haremos simplemente una breve reseña para adelantar a qué aplica cada una:

- La librería MCrypt para PHP es una interfaz para el acceso a la librería del mismo nombre, que provee soporte para diferentes algoritmos de cifrado en bloque como DES, TripleDES, Blowfish (algoritmo por defecto), 3-WAY, SAFER-SK64, SAFER-SK128, TWOFISH, TEA, RC2 y GOST en los modos de cifrado CBC, OFB, CFB y ECB. Adicionalmente, también soporta RC6 e IDEA, que son considerados "no libres".
- La librería Mhash es una librería libre (bajo la licencia GNU Lesser GPL), que provee una interfaz común o uniforme a un gran número de algoritmos de funciones de una vía o *hash*.
- El paquete Crypt_Blowfish es la implementación en una extensión, paquete o *package* de PEAR del algoritmo desarrollado por Bruce Schneier. Permite la encriptación utilizando el algoritmo Blowfish, sin requerir la extensión PHP de MCrypt.
- La clase Crypt_RSA es una clase PEAR (extensión, paquete o *package*), que provee las funcionalidades necesarias para la generación de llaves, encripta-

ción y desencriptación y verificación de firmas del tipo RSA.

- La clase Crypt_HMAC es también una clase PEAR (extensión, paquete o *package* PHP), que provee la implementación de funciones para calcular HMACs según el estándar RFC 2104.
- La clase Crypt_DiffieHellman implementa el algoritmo criptográfico o protocolo de Diffie-Hellman para el intercambio de llaves. La librería ha sido encapsulada en una clase PEAR (extensión, paquete o *package* PHP) para PHP5.

Herramientas de desarrollo criptográficas en bases de datos.

Como veremos en el capítulo práctico orientado a describir las funcionalidades criptográficas, de las cuales podríamos valernos en el caso de que nuestras aplicaciones utilicen alguno de los motores de bases de datos relacionales, entre los más utilizados actualmente, y a los que se hará referencia, están los que proveen variedad de implementaciones de funciones criptográficas.

Se describirá que, de la misma manera en que cuentan con funciones generales como las del manejo de fechas, operaciones sobre cadenas de caracteres o funciones matemáticas, desde sentencias en lenguaje SQL será posible invocar funciones para la generación de *hashes* criptográficos o descifrar un campo de un registro que ha sido cifrado al almacenarse.

Veremos en ese capítulo las funcionalidades provistas por tres de los más populares motores de base de datos relacionales, actualmente disponibles: Microsoft SQL Server (versiones 2000, 2005, 2008), con sus funciones para la criptografía simétrica, asimétrica –único motor que provee esta funcionalidad– y generación de *hashes* criptográficos Oracle (versiones 9i, 10i y 11i) y sus paquetes o *packages* incorporados de funciones de *hashing* y criptografía simétrica; y, por último, MySQL (versiones 3.23, 4.1, 5.0 y 6.0.1), también con sus funciones para el cálculo de *hashes* y encriptación simétrica.

---

[1] Schneier, Bruce. *Applied Cryptography*. 2a ed. John Wiley & Sons, EE.UU., 1996.

[2] Ramió Aguirre, Jorge. *Libro Electrónico de Seguridad Informática y Criptografía* [en línea]. Versión 4.1. Escuela Universitaria de Informática de la Universidad Politécnica de Madrid, España. Sexta edición de 1 de Marzo de 2006. <http://www.criptored.upm.es/guiateoria/gt_m001a.htm>
[Consulta: Junio 2008].

[3] Menezes, Alfred J.; Van Oorschot, Paul C.; Vanstone, Scott A. *Handbook of Applied Cryptography*. 5ta ed. CRC Press, EE.UU., 2001.

[4] Kahn, David. *The Codebreakers - The Story of Secret Writing*. [Abridged] ed.;Weidenfeld and Nicolson, Inglaterra, 1974.

2

Introducción a la criptografía

En este capítulo veremos los conceptos relativos a los protocolos y algoritmos criptográficos, que nos servirán de base para comprender mejor cómo convendrá aplicar las herramientas y funcionalidades que proveen los distintos lenguajes de programación en nuestros desarrollos. Estas funcionalidades serán descriptas detalladamente en los capítulos prácticos que seguirán a éste, por cada entorno de desarrollo tratado.

Aquí se especificará a grandes rasgos de qué hablamos al hacer referencia a un protocolo. Detallaremos los algoritmos más populares y se tratarán temas relativos, por ejemplo, a cuáles son las características de cada una de las formas de cifrado, a los tipos y modos del cifrado simétrico y a los conceptos referentes a las longitudes de las llaves criptográficas, entre otros.

Si bien el capítulo anterior brindó una rápida introducción general que permitía abordar directamente los capítulos prácticos, al finalizar este capítulo se dispondrá de un conocimiento más acabado respecto de varios aspectos de la criptografía, como así también de las características y funcionamiento de algunos algoritmos, lo que permitirá decidir mejor qué alternativas podrían aplicarse en cada caso.

Así mismo, se recuerda que este libro es básicamente práctico; tiene como objetivo que el desarrollador conozca cuáles son las funciones, clases, librerías, *frameworks*, etc., actualmente disponibles de las que puede valerse, y cómo hacer uso de ellas en sus programas, de acuerdo con el lenguaje o lenguajes que domine.

Por lo tanto, sugeriremos al lector interesado en adentrarse más profundamente en los detalles de los algoritmos, matemáticas involucradas o protocolos y algoritmos no tratados aquí, consultar Schneier[1], un trabajo obligado en la materia, sobre el cual se basa parte de lo que a continuación sigue. Téngase en cuenta que aquí se ha tratado esta teoría de manera resumida (y en tanto aplicaba a los conceptos necesarios para la utilización de las herramientas de desarrollo criptográficas provistas en los entornos de programación tratados en el libro).

Los estándares en donde son especificados cada uno de los algoritmos criptográficos, referenciados hacia el final del libro, y los códigos fuentes de distribución libre u *open-source* que implementan estos algoritmos serán la fuente de la cual podrá valerse el lector si quisiere conocer con mayor detalle cómo funciona alguno de estos algoritmos.

Introducción a los protocolos criptográficos.

En primer lugar, deberemos dejar en claro de qué estamos hablando al hacer referencia a un protocolo. Podremos entenderlo como una serie de tareas o de pasos que involucra a dos o más partes, diseñada para lograr un objetivo.

La serie de tareas o de pasos implica aquí que existe una secuencia ordenada, de principio a fin, en donde cada paso debe ejecutarse a su turno; un paso no será realizado antes de que el anterior en la secuencia finalice.

Acerca de que dos o más partes estan involucradas en un protocolo, notaremos que se trata de la condición principal, ya que involucra las reglas, procedimientos, normas o conductas de una parte o entidad en un medio particular, pero en relación con otra u otras partes.

Por último, el objetivo es la razón de ser del protocolo; debe obtenerse algún resultado, beneficio o cambio de estado a partir de su implementación o uso.

En nuestra vida cotidiana estamos en contacto con diversos protocolos y los implementamos de manera informal. A la hora de usar el teléfono, por ejemplo, realizamos una serie de pasos preestablecidos, conocidos o acordados, explícita o implícitamente entre las dos o más partes involucradas, con un fin u objetivo particular.

Con referencia a ejemplos de protocolos más formales, quienes nos desempeñamos laboralmente dentro del área de la informática o de las tecnologías de la información, en mayor o menor medida, ya tenemos conocimiento también de lo que es un protocolo de red, por ejemplo. Sabemos que comprende una serie de reglas técnicas específicas y sólo en tanto las computadoras en la red utilicen el mismo protocolo será posible la comunicación.

Antes de continuar resumiremos esto con la afirmación de que un protocolo debe conocerse por todas las partes que han de implementarlo y estas partes deben acordar su utilización; no debe ser ambiguo (cada paso ha de estar bien definido); y, por último, debe ser completo, esto es, debe estar definido un paso o tarea por cada situación posible.

Con respecto a los protocolos criptográficos, consideremos que son aquellos que involucran a algún o a algunos algoritmos criptográficos, pero el objetivo final del protocolo va más allá de mantener información en secreto. Las partes que acuerden utilizarlos podrán tenerse o no confianza entre ellas. Éstas querrán, por ejemplo, compartir un secreto, probar sus identidades o firmar digitalmente un contrato.

En reglas generales, el objetivo final de un protocolo criptográfico, o del acuerdo e implementación del mismo, será prevenir y detectar el acceso indebido a, o la alteración de, información sensible, privada, crítica o que ha de ser resguardada, por el motivo que fuese.

Como es convención en textos que tratan estas materias, nos referiremos a las partes involucradas en los protocolos criptográficos como Alice, Bob, Carol y Dave. Nótese el orden alfabético; en algunos textos se hará referencia al usuario o parte "A", usuario o parte "B", etc. Es muy probable que al consultar el material de la bibliografía, o cualquier publicación en línea o impresa que describa un protocolo, fuera ésta escrita en idioma inglés o en español, a la hora de los ejemplos se encontrará con estos personajes. Mantendremos la convención y, como regla general, puede recordarse que los dos primeros personajes, Alice y Bob, o partes "A" y "B", serán los más activos. Con ellos se ejemplificarán los protocolos que involucran a dos partes, siendo Alice quien incie el proceso o el primer paso del protocolo. Dave y Carol participarán cuando sean necesarias una tercera y/o cuarta parte.

Antes de continuar, a manera de resumen recordemos que, como bien advierte Tena Ayuso[2]:

> *"La criptografía cubre hoy en día objetivos distintos, a veces muy alejados, del tradicional y más conocido de la transmisión secreta de información. Este tipo de aplicaciones se engloba dentro de lo que se denomina protocolo criptográfico, que es un protocolo (es decir, un conjunto bien definido de etapas, implicando a dos o más partes y acordado por ellas, designado para realizar una tarea específica) que utiliza como herramienta algún algoritmo criptográfico. Existe una amplia variedad de ellos que dan respuesta a diferentes objetivos. Se trata de un tema muy amplio y en rápido crecimiento".*

Protocolos de criptografía simétrica.

Veremos en primera instancia la forma más conocida, más antigua y más utilizada actualmente de criptografía. Nos estamos refiriendo específicamente a la criptografía simétrica.

Debe quedar en claro que dos partes son las involucradas: Una parte, que llamaremos "A" o Alice, pretende enviar un mensaje a la otra, "B" o Bob. Estas dos partes, como primer paso, acordarán el uso de una forma de criptografía o cripto-sistema (protocolo, algoritmo). Luego acordarán una llave o clave. La parte "A" encriptará el mensaje (texto-plano), valiéndose del algoritmo y de la llave. El resultado de esta última operación será la obtención del texto-cifrado correspondiente a la función del algoritmo, con el texto-plano y la llave como entradas. La parte "A" enviará a "B" el texto-cifrado. La parte "B", entonces, utilizando el mismo algoritmo y la llave acordada previamente, podrá desencriptar el texto-cifrado para obtener el texto-plano, es decir, el mensaje original.

Por supuesto, el medio por donde se enviará el texto-cifrado se asume inseguro. Hemos de seleccionar un algoritmo seguro, estandarizado y lo suficientemente probado por la comunidad académica –y la industria y entes gubernamentales–, para lograr contar con la seguridad de que, sólo a partir del texto-cifrado, una tercera parte no podrá desencriptar el mensaje.

De cualquier manera, nunca deberíamos descuidar aspectos tangenciales a la selección e implementación de los protocolos y/o algoritmos criptográficos, como ser la mala elección de una llave o clave, o el almacenamiento inseguro de la misma. Sepamos que si una tercera persona pudiera obtener fácilmente la llave, independientemente del algoritmo utilizado, la comunicación no sería segura, ya que el texto-cifrado y la llave es todo lo que se necesitará para desencriptar el mensaje.

Lo que la parte "A" y "B" deben acordar en secreto, previamente a comunicarse utilizando criptografía simétrica, es la llave que han de resguardar con el mayor cuidado. No sucede igual con la elección del algoritmo; no debe importarnos que la tercera persona conozca o no el algoritmo utilizado para la encriptación, ya que eso podría acordarse públicamente. Pero, al hablar de criptografía simétrica, utilizando un algoritmo comprobado, la seguridad de las comunicaciones estará basada en mantener en secreto la llave entre las partes que sí deben acceder a la información cifrada.

Esto último representa quizá el problema o aspecto negativo más importante, característico de la criptografía simétrica. Las llaves deben acordarse (o distribuirse) en secreto. Ha de ser tan importante luego su resguardo como la información que se ha de cifrar, ya que, conocida la llave, se podrá obtener el mensaje o información original. Especial interés cobra esta característica al considerar que, al asumir que una tercera persona pudo capturar los textos-cifrados expuestos en el canal o medio inseguro, al obtener la llave podrá desencriptarlos todos, como así también podrá encriptar mensajes nuevos, permitiéndole, dependiendo de las particularidades de la implementación, realizar una impostura de identidad; esto implica que con la llave podrá encriptar cualquier mensaje de la misma manera que la parte "A" lo haría.

Por último, y con referencia a los potenciales problemas para considerar a la hora de implementar esta metodología, particularmente para la comunicación segura entre más partes, será prudente tener en cuenta que por cada par de usuarios, o partes, se necesitará una llave. Si usted formase parte de un equipo de cuatro personas, deberá disponer de tres llaves, una para comunicarse con cada una de las partes, y lo mismo corre para el resto. De esta manera, entre las cuatro partes se manejarán en todo el grupo: Tres llaves (las propias) más otras tres llaves (compartidas entre sí por las otras tres partes), haciendo un total de seis llaves. La fórmula para calcular esta cantidad de llaves es $n(n-1)/2$, siendo n la cantidad de partes. Como se desprende de la fórmula, al incrementarse la cantidad de partes crecerá rápidamente la cantidad de llaves para manejar. Al pensar en un grupo o equipo numeroso, como un conjunto que se ha de administrar, donde las partes o usuarios deben ocuparse del resguardo de tantas llaves, la opción más conveniente para recomendar es la utilización de criptografía asimétrica, que veremos más adelante.

Funciones de una vía y *hash*.

El concepto general de las funciones de una vía es que, a través de ellas, se podrá computar su resultado de manera relativamente rápida, pero, en cambio, la obtención de la entrada (el parámetro de la función) a partir del resultado será prácticamente inviable. Esto quiere decir que siendo *f* la función de una vía, se podrá calcular *f(x)* de manera sencilla, pero obtener *x* demandará años, aunque dispongamos de toda la capacidad de procesamiento que podamos adquirir.

Deberá quedar en claro también que una función de una vía no se trata de un protocolo criptográfico en sí. No se utilizan para encriptar información, sino que forman parte fundamental de muchos algoritmos y técnicas criptográficas. Es relevante la aclaración, ya que no cifran ni descifran información.

Estrictamente, no existen comprobaciones matemáticas de la existencia de funciones con la característica de la imposibilidad de cálculo de su inversa, ni siquiera de la posibilidad de construirlas. Existen sí muchas que lo parecen, que permiten su cálculo en forma rápida y que, hasta el momento, se desconoce una forma sencilla de obtener su inversa. Hecha esta nota de rigor, de aquí en más llamaremos funciones de una vía a aquellas que en "apariencia" lo son.

Con algo más de especificidad encontramos, dentro de las funciones de una vía, las funciones de *hashing* criptográfico. Éstas tienen varios sinónimos en la literatura de la materia: Resúmenes de mensajes (*message digests*), huellas digitales (*fingerprints*), funciones de compresión, funciones de contracción, chequeos de integridad de mensajes (*message integrity check* o MIC), códigos de detección de manipulación (*manipulation detection code* o MDC) y *checksums* criptográficos.

El concepto principal de estas funciones es que tomarán como entrada una información o mensaje de longitud variable (una contraseña o los contenidos de un archivo que empaquete el instalador de un programa), que se denominará pre-imagen, para convertirlo en una información de salida de longitud fija (en el algoritmo MD5, por ejemplo, será de 128 bits de longitud; en SHA-1, de 160 bits). A esta salida se la denominará valor de *hash*.

Repitiendo en parte lo anterior, estas funciones operan en una dirección (una vía); será posible calcular el *hash* a partir de una pre-imagen, pero será inviable generar una pre-imagen cuyo *hash* corresponda a un resultado en particular.

En relación con esto, la cualidad de ser libre de colisiones es otra característica importante de estas funciones. Esto significa que una función de *hashing* debería hacer muy poco probable que dos pre-imágenes distintas generen un mismo *hash* o resultado.

En la actualidad, además de servir de base a otros algoritmos o protocolos criptográficos, los usos más comunes de estas funciones de *hashing* son la del registro de contraseñas en espacios de almacenamientos no confiables y la de realizar verifica-

ciones de integridad, generalmente sobre archivos. La generación de llaves para criptografía simétrica a partir de una contraseña, *passphrase* o frase-clave, es también otra aplicación importante para la cual nos valemos de estas funciones, como se verá en las secciones prácticas más adelante.

Por último, en la sección referente a la criptografía de llave pública, veremos que en los sistemas de cifrado asimétrico se hace uso de funciones de una vía, del tipo particular *trapdoor* (de puerta trasera o tramposa, en tanto dispone de un mecanismo o trampa secreta, que hará posible –o fácil– el cómputo de la inversa de la función).

Protocolos de criptografía asimétrica.

La aparición de la criptografía asimétrica representó un cambio de paradigma muy importante en la materia, en 1976, cuando Whitfield Diffie y Martin Hellman la describieron explicando que implicaba la utilización de dos llaves: Una pública y una privada.

Aquí también hablamos, hasta que se refiera el rol de los certificadores al menos, de una parte deseando comunicarse, de manera segura, con otra parte (dos partes son las involucradas en el protocolo).

Al describir el método paso por paso, usando de ejemplo las partes “A” y “B”, como cuando se describió la criptografía simétrica, podemos resumir el procedimiento de la siguiente manera. El primer paso corresponde al acuerdo entre las partes de la utilización de un protocolo de criptografía asimétrica. En el segundo paso, la parte “B” (quien ha de recibir la información) envía su llave pública a la parte A (la parte que debe comunicar la información). La parte “A”, entonces, encriptará la información o mensaje, utilizando la llave pública recibida de la parte “B”. La parte “A” enviará a la parte “B” el resultado de esta encriptación, es decir, el texto-cifrado. Por último, la parte “B” desencriptará el texto-cifrado (que fue encriptado con su propia llave pública) mediante su llave privada.

Es muy común que en implementaciones multi-usuario, las llaves públicas de todas las partes se encuentren almacenadas en una base de datos de acceso público. En esta modalidad, la parte “B” no necesitaría enviar su llave pública a la parte “A”; ésta la obtendría desde la base de datos. El proceso que sigue y los conceptos generales aplican de la misma manera.

Dado que éste es un libro práctico de criptografía, debe destacarse especialmente que en la práctica el cifrado asimétrico no se utiliza para encriptar mensajes, o lo que sería concretamente la información para comunicar, sino que se implementa para encriptar llaves.

En el caso de la criptografía simétrica, como hemos visto en la sección que describe esa metodología, la necesidad del acuerdo de una llave secreta impone un problema de difícil solución entre las partes, si no se dispone de un medio o canal seguro para hacerlo.

Por otra parte, la criptografía asimétrica es considerablemente más lenta que la simétrica, en un orden aproximado de mil veces. Además, es vulnerable a ataques de *chosen-plaintext,* o texto-plano elegido; esto quiere decir que si sabemos que el mensaje para transmitir será uno de un conjunto de *n* posibles mensajes, al ser la llave del receptor pública, una tercera parte podría encriptar los *n* posibles mensajes, o textos-planos, y comparar el resultado con el texto-cifrado capturado en el medio o canal inseguro y así inferir cuál ha sido el mensaje enviado.

Por lo tanto, y ahora nos resultará evidente, la utilización más común de la criptografía asimétrica es para hacer llegar a la otra parte una llave criptográfica simétrica y realizar así la comunicación del mensaje o información usando criptografía simétrica en lugar de asimétrica.

El rol de los certificadores o árbitros, citado más arriba, se refería a que cuando, por ejemplo, Alice o la parte "A" necesite enviar un mensaje a Bob o parte "B", quizá no tenga manera de comprobar la autenticidad e integridad de la llave pública de Bob. Alice la habrá obtenido de una base de datos, la habrá recibido por correo electrónico o descargado desde un sitio Web, pero si desconfía deberá consultar a una tercera parte, un certificador de confianza asumida, para verificarla antes de realizar la encriptación y el envío del mensaje cifrado.

Veremos más detalles a este respecto en la sección siguiente, donde se describirán protocolos de firma digital que también necesitan de un árbitro o certificador para la verificación de la autenticidad de las llaves públicas. De cualquier manera, se aclara para finalizar, que un certificado de llave pública es la llave pública de una parte ("A" o "B") firmada otra parte ("C"), en la cual se confía de antemano.

Protocolos de firma digital.

La firma digital corresponde a la versión informática de la firma personal manuscrita o firma ológrafa. Se verá que en realidad, los protocolos de firma digital proporcionan, en principio, mayor seguridad que las firmas tradicionales. Estas últimas se han utilizado ampliamente como prueba de autoría o acuerdo de una parte, entre otros usos, siempre en referencia a un documento, o mejor dicho, a los contenidos de un documento. Sin embargo, como hemos mencionado, pueden resultar inseguras en tanto que pueden ser, con relativa facilidad, aplicadas de manera deshonesta. Podrían por ejemplo ser tomadas desde una pieza de papel para pasarlas a otra, o los documentos podrían ser modificados luego de la aplicación de la firma.

La firma digital se trata de una tecnología que pretende resolver estos problemas en documentos digitales. De acuerdo con la legislación de cada país, tendrá tanto carácter o validez jurídica como la firma personal manuscrita y podrá ser admisible como prueba en un juicio. Países como Argentina, Brasil, Chile, México y Perú, entre otros, han adoptado, en mayor o menor medida, la utilización de la firma digital como alternativa a la ológrafa y tienen legislación al respecto que ya apunta al principio de equi-

valencia funcional; de manera que todos los documentos que pueden ser firmados con firma tradicional, pueden hacerlo también con firma digital.

Por ejemplo, podemos ver, en la siguiente dirección del sitio Web del Senado de la República de Chile, una noticia reciente respecto de la ley proyecto que da validez a los documentos electrónicos presentados en los procesos judiciales. El texto, que modifica el Código de Procedimiento Civil y la ley sobre firma electrónica, fue aprobado por unanimidad de la Sala del Senado.

```
http://www.senado.cl/prontus_senado/site/artic/20070904/pags/20070904173541.html
```

Como otro ejemplo de legislaciones propias de cada país, si accedemos a la dirección Web que sigue es posible consultar la ley peruana que regula la utilización de la firma electrónica, otorgándole la misma validez y eficacia jurídica que el uso de una firma manuscrita u otra análoga que conlleve manifestación de voluntad.

```
http://www.congreso.gob.pe/ntley/Imagenes/Leyes/27269.pdf
```

Por último, referimos al lector al sitio Web de Firma Digital de la República Argentina, mantenido por la Secretaría de Gestión Pública, donde se encontrará información acerca de toda la legislación argentina en materia de firma digital, como así también referencias a noticias relacionadas al tema de otros países.

```
http://www.pki.gob.ar/
```

Por supuesto, se recomienda el asesoramiento profesional a la hora de implementar un sistema que, valiéndose de la tecnología de firmas digitales, deba servir de soporte jurídico-legal.

Volviendo ahora a las cuestiones que nos ocupan, respecto del protocolo en sí, deberíamos decir protocolos, porque existen diferentes alternativas que implementan los procesos de firma digital de diversas maneras. Se desarrollarán brevemente a continuación.

La primera posibilidad para comentar será la que corresponde a la firma de documentos electrónicos, mediante criptografía simétrica y un árbitro. Aquí la parte “A”, que pretende firmar un documento y enviarlo a la parte “B”, lo hará con la ayuda de una tercera parte –llamada parte “C”, quien será el árbitro en quien se confía– de la siguiente manera: La parte “A” encripta el mensaje que se ha de enviar a la parte “B” con su llave secreta –recuérdese que hablamos de criptografía simétrica– pero lo envía a la parte “C”. Esta parte “C” lo desencriptará con la llave secreta, compartida con “A”. Al texto-plano obtenido, la parte “C” agregará una nota respecto de que ha recibido el mensaje de “A”, encriptará todo esto con la llave secreta compartida con la parte “B” –diferente de la primera, por supuesto–, quien lo recibirá, y al desencriptarlo, encontrará el mensaje y la certificación –por la parte “C”– de que fue la parte “A” quien lo envió.

Como otra de las posibilidades que describiremos básicamente a continuación, existe aquella que se vale de la criptografía asimétrica (en lugar de la simétrica, como en la posibilidad anterior). Si se utilizara un algoritmo como RSA, que permite la encriptación y desencriptación de datos, tanto con la llave pública como con la privada, la parte "A", al encriptar un mensaje con la llave privada, ya generaría un mensaje firmado. Otras partes podrían descifrarlo con la llave pública de "A" y comprobarlo. En cambio, en otros algoritmos, como DSA (que no puede ser utilizado para encriptar y desencriptar información, ya que es un algoritmo para el intercambio de llaves, como veremos más adelante), se utiliza un algoritmo separado para la firma digital.

En resumen, para el primer caso, los pasos serán que la parte "A" encriptará el mensaje con su llave privada (lo que implica una firma sobre el mensaje) y se lo enviará a "B", quien podrá desencriptarlo únicamente con la llave pública de "A", comprobando que sólo alguien con la llave privada de "A" pudo haberlo encriptado; así, se considera que es autentico, que "A" lo ha firmado.

La última posibilidad que comentaremos involucra la utilización de funciones *hash* junto con criptografía asimétrica. En la práctica, ésta es la implementación más utilizada, por la ineficacia o lentitud generada a la hora de firmar mensajes extensos con las metodologías anteriores. Aquí, en lugar de firmar un mensaje o documento, la parte "A" firmará el *hash* resultante de tal documento. El algoritmo de *hashing* deberá acordarse de antemano. Los pasos del protocolo se resumen de la siguiente manera: La parte "A" producirá el *hash* del mensaje o documento, lo encriptará con su llave privada y enviará el *hash* resultante, junto con el documento, a la parte "B". La parte "B" computará por su cuenta el *hash* sobre el documento. Luego, mediante la llave pública de "A" desencriptará el *hash* que la parte "A" ha computado. Entonces, si "B" compara el *hash* que él ha generado con el que acaba de desencriptar (generado por "A") y coinciden, podrá comprobar o verificar la firma de la parte "A" sobre el documento.

Generación de números aleatorios.

Si bien no es un protocolo en sí mismo –como las funciones de una vía, vistas anteriormente–, debe tenerse en cuenta que la generación segura de números aleatorios tiene una importancia fundamental en criptografía. La operación es más compleja de lo que en principio podría aparentar.

Gracias a Alan Turing, criptógrafo y reconocido matemático, citado habitualmente como el padre de la ciencia de la computación y precursor de la informática moderna, sabemos que las computadoras pueden entenderse como máquinas de estado finito con entrada infinita. Sabrá disculpar el lector esta definición elemental de una máquina de Turing, pero para el tema que tratamos aclararemos, únicamente, que la máquina es capaz de asumir cualquier número fijo y finito de configuraciones o estados posibles y el estado de ésta en un determinado momento, junto con la información de la entrada, determinará unívocamente cómo se comportará en ese momento.

Así entonces, las computadoras dispondrán de entradas, se procesarán de acuerdo con operaciones completamente predecibles y generarán las salidas correspondientes, pero nunca dejarán de ser, en realidad, instrumentos determinísticos. No podrán generar información, estados o salidas verdaderamente aleatorios.

Esto implica que un generador de números aleatorios implementado en una computadora será periódico y necesitará entrada aleatoria, pero una computadora no puede proveer tal cosa.

Existen aplicaciones (programas de SSH, por ejemplo) que al momento de la instalación le solicitan al usuario que mueva el *mouse* durante algunos segundos. El programa necesita generar un par de llaves (la llave pública y la llave privada) para su operación y, al verse en la necesidad de números aleatorios para tal generación, obtiene entrada aleatoria de esta manera. Otras formas incluyen tomar audio por micrófono, específicamente ruido o estática. Acerca de si usted y yo moveríamos el *mouse* de la misma forma, y si tal forma sería verdaderamente azarosa, quizá no pueda establecerse de manera rigurosa. Se supondrá aleatoria, por supuesto, pero considérese que, si simplemente nos preguntaran por un número al azar, sería apresurado pensar que se podría responder cabalmente, esto es, de manera verdaderamente azarosa o aleatoria. Remitimos al lector interesado en conocer la falsa apariencia de aleatoriedad de un número generado no ya por una computadora, sino por una persona, al estudio de la teoría freudiana.

Volviendo al tema que nos ocupa, si bien casi todos los sistemas operativos o compiladores proveen una llamada a sistema o una función, respectivamente, para la obtención de un número aleatorio, veremos a continuación por qué valerse de estas implementaciones no es lo más conveniente para cuestiones relacionadas con la materia que estamos tratando. La mayoría de estos generadores no son lo suficientemente seguros para su uso en criptografía; las más de las veces, ni siquiera proveen un buen margen de aleatoriedad. Es imposible obtener un número verdaderamente aleatorio a partir de una computadora. Por esto es que hablaremos de aquí en más de números seudoaleatorios.

Podría destacarse como el problema principal de los generadores estándares, convencionales o ya incorporados, que no producen una secuencia de números aleatorios. Por supuesto, en juegos por ejemplo, tal falta de aleatoriedad no será notada; pero puede implicar una limitación importante, una inseguridad crítica, si hablamos de su uso en criptografía. Necesitamos, en algunos casos, una gran cantidad de números y si una implementación de un generador empezara a repetirse relativamente pronto (el problema de la periodicidad en la secuencia de números aleatorios generados, que se comentó anteriormente), se vería comprometida la seguridad del sistema. Idealmente, la secuencia debería "parecer" aleatoria, debería ser impredecible y, además, debería no ser reproducible.

Las técnicas para obtención de estos números seudoaleatorios, o las secuencias de ellos, de manera computacional varían: Están basadas en generadores de congruencia lineal o retroalimentación (*feedback)* de registros de desplazamiento, por ejem-

plo. Al margen de las técnicas basadas en información obtenida desde afuera, como el ejemplo del programa SSH comentado previamente muchos de estos generadores necesitarán que pasemos como parámetro o *seed* (semilla). Esta información inicial servirá para basar los procesos que generarán los números aleatorios en datos provistos por el usuario o fuentes externas y así lograr combinar un mecanismo o máquina determinantes con información, en teoría, aleatoria. Será trabajo de la aplicación, cada vez, el obtener el *seed* que se ha de utilizar para alimentar al generador de números aleatorios criptográfico por el que se haya optado.

No se desarrollarán aquí en detalle las técnicas mencionadas. Teniendo en cuenta el objetivo práctico del libro, se recomienda al lector que, en la necesidad de números aleatorios seguros en sus desarrollos, no utilice el generador provisto por el compilador o el sistema operativo, es decir, que no utilice la función *random* que el lenguaje de programación le provee. Las diferentes librerías y *frameworks* criptográficos tratados en los capítulos prácticos disponen de generadores más seguros.

Introducción a los algoritmos criptográficos.

Un algoritmo consiste en un conjunto determinado y finito de pasos o acciones, con el objetivo de solucionar un problema. Los algoritmos criptográficos son funciones matemáticas, que son utilizadas en los procesos de encriptación o desencriptación de datos, que serán la entrada o parámetro de estas funciones. Debe entenderse que si bien se hace referencia a procesos de encriptación y desencriptación, también existen funciones criptográficas como las *hashing*, o para la generación de números aleatorios, que serán consideradas de la misma forma.

A diferencia de cómo hemos descripto los protocolos vistos en secciones anteriores, aquí no necesariamente tratamos de relaciones entre las partes y el nivel de detalle debe ser concreto y no abstracto; las especificaciones de los algoritmos se realizan mediante notación matemática.

Se tratarán en esta sección cuestiones generales relativas a los algoritmos criptográficos y detallaremos las características principales de aquellos que, ya fuese por su popularidad o propiedades distintivas, actualmente son considerados como los más importantes. Estos conceptos serán muy útiles a la hora de optar por uno u otro y ayudarán a comprender mejor cómo funcionan internamente las implementaciones de estos algoritmos encontradas en los entornos de desarrollo, *frameworks* y librerías adicionales, que se detallarán en la parte práctica del libro.

Matemáticas involucradas.

Se advierte al lector que los conceptos matemáticos que trataremos a continuación serán abordados de una manera muy rápida y resumida, con relación a cuestiones criptográficas exclusivamente, y con el solo objetivo de que sea posible la compren-

sión general de las descripciones de algunos algoritmos que siguen más adelante y que hacen referencia a estos conceptos.

La teoría asociada a las funciones criptográficas se basa en una serie de teorías matemáticas, entre ellas, las que trataremos aquí: La teoría de la información, la de los números (a ésta corresponden también las cuestiones relativas a las características de los números primos y factorización) y la de la complejidad de los algoritmos.

Teoría de la información.

Esta teoría consiste en el estudio de la cantidad de información contenida en los mensajes y llaves, así como su entropía. Servirá para mensurar la cantidad de información que contienen los mensajes o información que se ha de cifrar, a través del número medio de bits necesarios para codificar todos los posibles mensajes, utilizando una codificación óptima.

Pensemos en la codificación de los días de la semana; alcanzaría con la utilización de tres bits de información, por ejemplo:

000 podría corresponder al Lunes.

001 podría corresponder al Martes.

010 podría corresponder al Miércoles.

011 podría corresponder al Jueves.

100 podría corresponder al Viernes.

101 podría corresponder al Sábado.

110 podría corresponder al Domingo.

111 no sería utilizado.

Podríamos utilizar cadenas de caracteres para corresponder cada uno de estos posibles valores, pero utilizaríamos más bits sin necesidad, sin más información, es decir, de manera menos óptima.

Formalmente, la cantidad de información de un mensaje X es medida por la entropía de un mensaje y se denota $H(X)$. La entropía de un mensaje que informe el sexo de una persona será de un bit, la de los días de la semana, algo menor a tres bits (nótese que nos ha sobrado un número utilizando tres bits, por eso es que la entropía será algo menor).

Algo más detalladamente a este respecto: Si un fenómeno tiene un grado de indeterminación k y sus estados son equiprobables, la probabilidad p de que se dé uno de esos estados será $1/k$. Entonces:

$$c_i = log_2 (k/1) = log_2 [1/(1/k)] = - log_2 p$$

Si cada uno de estos estados tiene una probabilidad distinta p_i, la entropía H será igual a la suma ponderada de la cantidad de información:

$$H = - p_1 log_2 p_1 - p_2 log_2 p_2 - ... - p_k log_2 p_k$$

$$H = - \sum_{i=1}^{k} p_i log_2 p_i$$

De un mensaje determinado, que lo hemos llamado *X*, su entropía, que se representa por H*(X)*, es el valor medio ponderado de la cantidad de información de los diversos estados del mensaje. Es una medida de la incertidumbre media acerca de una variable aleatoria y el número de bits de información.

Dentro de esta teoría no deberíamos omitir hacer, al menos, una breve mención del concepto de ratio de un lenguaje. Formalmente se expresa:

$$r = H(M)/N$$

donde *N* es la longitud del mensaje. El ratio absoluto de un lenguaje es el número máximo de bits que pueden ser codificados en cada carácter; si hubiese *L* caracteres en el lenguaje, formalmente:

$$R = \log_2 L$$

Por último, la redundancia de un lenguaje, convencionalmente llamada *D*, está dada por la diferencia de estos últimos valores:

$$D = R - r$$

Otro concepto importante dentro de esta teoría es el de la distancia de unicidad. Esta distancia corresponde al bloque de texto-cifrado mínimo necesario para que se pueda intentar, con ciertas expectativas de éxito, un ataque en búsqueda de la clave usada para cifrar.

Terminando con esta breve introducción a la teoría de la información, notaremos que existen dos técnicas para ocultar o disimular las redundancias en el texto-plano llamadas de confusión y de difusión. La primera disimulará las relaciones entre el texto-plano y el texto-cifrado, para frustrar los intentos de estudio de este último en busca

de patrones (lo que es entendido también como sustitución). La segunda técnica disipará la redundancia del texto-plano, cambiando el orden del texto-plano en el texto-cifrado al encriptar (lo que se entiende por transposición).

Teoría de los números.

Esta teoría se basa en el estudio de las matemáticas discretas y cuerpos finitos que permiten las operaciones de cifrado y descifrado. Veremos a continuación una introducción a lo que se entiende por aritmética modular.

El concepto clave aquí es el de la congruencia, que es la base en la que se sustentan las operaciones de encriptación en matemática discreta.

Sean dos números enteros *a* y *b,* se dice que *a* es congruente con *b* en el módulo o cuerpo *n* (Z_n), si y sólo si, existe algún entero *k* que divide de forma exacta la diferencia (*a* - *b*).

Esto último puede expresarse:

$$a - b = k \,.\, n$$

$$a \equiv_n b$$

$$a \equiv b \bmod n$$

Son muy comunes los algoritmos criptográficos que utilizan los cálculos de mod *n*, ya que este cómputo de logaritmos discretos y raíces cuadradas puede resultar un problema de difícil solución.

No dejaremos de lado lo relativo a los números primos, que trataremos, destacando su importancia, a la hora de estudiar los algoritmos de llave pública. Los números primos son aquellos enteros mayores a 1 cuyos únicos factores son el número 1 y el número mismo. Existen infinitos números primos, pero en tanto avanzamos hacia números grandes, más dificultoso es encontrarlos.

Dos números son primos relativos si no comparten otro factor entre sí que el número 1; su máximo común divisor es 1. El algoritmo de Euclides es utilizado para hallar estos divisores entre dos números. El mismo algoritmo sirve para calcular la inversa de *a mod n*. Otras fórmulas o funciones utilizadas en el tratamiento de números primos son el pequeño teorema de Fermat, la función φ de Euler y el teorema del resto chino.

Dentro de esta teoría se encuentra también otra estructura utilizada en criptografía –por ejemplo en el algoritmo AES– conocida como campos de Galois, denotados GF(*p*), donde *p* es un número primo grande o la potencia de un número primo grande.

Como último concepto para tratar en la teoría de los números, nos referiremos brevemente a la factorización. Factorizar un número significa encontrar los números primos que son factores de ese número.

$$10 = 2 \cdot 5$$

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

$$252\,601 = 41 \cdot 61 \cdot 101$$

El problema de la factorización es uno de los más antiguos en la teoría de los números. Es simple pero demanda mucho tiempo. Terminaremos haciendo referencia al algoritmo más rápido conocido, llamado *Number field sieve* (NFS), que se emplea para factorizar números de aproximadamente más de 110 dígitos.

Teoría de la complejidad de algoritmos.

Esta teoría consiste en el estudio de la clasificación de los problemas como, computacionalmente, tratables o intratables. Posibilitará, entre otras cosas, conocer la fortaleza de un algoritmo y tener así una idea de su vulnerabilidad computacional. Compara algoritmos y técnicas determinando su seguridad.

La complejidad de un algoritmo está determinada por el poder computacional necesario para ejecutarlo. Esta complejidad es comúnmente mensurada por dos variables: T (para complejidad de tiempo) y S (para complejidad de espacio o requerimiento de memoria). Ambas son generalmente expresadas en función de n, cuando n es el tamaño de la entrada.

Veamos por ejemplo que, si la complejidad de tiempo de un algoritmo dado es $4n^2 + 7n + 12$, entonces la complejidad computacional está en el orden de n^2, lo que se expresa del siguiente modo: O($n2$).

Esta notación permite ver cómo el tamaño de la entrada afecta al tiempo y al espacio requerido. Por ejemplo, si $T = O(n)$, al duplicar la entrada duplicaremos el tiempo de ejecución del algoritmo. Si en cambio T fuese $O(2^n)$, el agregado de un bit al tamaño de entrada duplicaría el tiempo de ejecución del algoritmo.

En cuanto a clasificaciones, un algoritmo es lineal si su complejidad de tiempo es O(n); también pueden ser cuadráticos, cúbicos, etc. Todos estos algoritmos son polinomiales; su complejidad es de $O(n^m)$, donde m es una constante. Además, se clasificará como un algoritmo que tiene tiempo de ejecución polinomial no determinista (en este caso exponencial) a aquel que dependa exponencialmente del tamaño de la entrada.

En criptografía son de especial interés las funciones $f(x)$ de una vía, o un solo sentido, tales que resulte sencillo calcular $f(x)$ pero muy dificultoso, o inviable, calcular $f^{-1}(x)$, salvo que conozcamos un secreto o trampa. Esto es así porque dan lugar a problemas del tipo NP, o polinomiales no deterministas, que son computacionalmente difíciles de tratar, como, por ejemplo, el problema de la factorización o del logaritmo discreto.

Tipos *Block* y *Stream* de algoritmos criptográficos.

Los algoritmos de criptografía simétrica pueden categorizarse en dos grupos, según cómo operen sobre la entrada o texto-plano y la salida o texto-cifrado. Esto determina lo que llamamos "tipo" del algoritmo. Las dos alternativas corresponden a, primero, aquellos que traten la entrada de a porciones, o en bloques, de una longitud determinada; la segunda, a los algoritmos que operan sobre la entrada en forma de flujo, o *stream*, de a un bit, byte o hasta palabras de 32 bits inclusive, cada vez.

Recapitulando, los algoritmos de criptografía simétrica pueden ser de encriptación o cifrado por bloques (en inglés, *block ciphers*) o de encriptación o cifrado en flujo (*stream ciphers)*.

En los algoritmos cifradores por bloques, el mismo bloque de entrada o texto-plano terminará resultando en el mismo bloque de salida o texto-cifrado, en tanto se utilice la misma llave, y de manera tal que no se involucren resultados del cifrado del bloque anterior (acerca de esto expondremos más adelante, al tratar los modos).

En cambio, en los cifradores de flujo, el bit o byte de texto-plano resultará encriptado en un bit o byte de texto-cifrado, diferente cada vez.

El algoritmo AES de encriptación simétrica por ejemplo, que se describirá más adelante, es un cifrador por bloques, que opera sobre bloques de 128 bits de longitud. El algoritmo RC4 es un *stream cipher*, o cifrador por flujo, que encriptará de a un byte (8 bits) a la vez.

Modos ECB, CBC, CFB y OFB de algoritmos criptográficos.

En cuanto a los modos criptográficos, aplican, como los tipos que se han descripto, a los algoritmos de cifrado simétrico. Un modo comúnmente combina al algoritmo con alguna forma de retroalimentación, o *feedback,* y una serie de operaciones. Estas operaciones serán simples, ya que la seguridad dependerá del algoritmo y no del modo en que es utilizado.

Veremos, a continuación, los diferentes modos que encontraremos como opciones en las funcionalidades provistas por los entornos de programación que trataremos en el libro.

- **ECB**: Del inglés, *electronic codebook*, o libro de códigos electrónico. Este es el modo más simple de usar un cifrador por bloques. Un bloque de entrada o texto-plano es encriptado produciendo un bloque de salida o texto-cifrado. Cada uno de estos bloques es encriptado de forma independiente (ninguna forma de *feedback*). El nombre proviene del hecho de que un bloque de entrada corresponderá siempre al mismo bloque de salida (utilizando la misma clave, por supuesto) y podría construirse un libro de códigos, de correspondencias, entre textos-cifrados y textos-planos.

- **CBC**: *Cipher Block Chaining* o cifrado con encadenamiento de bloque. En este caso, el encadenamiento agrega una retroalimentación, o *feedback,* al cifrador por bloques. El resultado o salida de la encriptación de un bloque será utilizado, se combinará, en el proceso de encriptación del siguiente bloque. Aquí, entonces, cada bloque de texto-cifrado depende no sólo del bloque de texto-plano que se encriptó, sino de todos los bloques encriptados anteriormente. De manera concreta, el proceso implica, al encriptar, la aplicación de la operación XOR sobre el texto-plano y el bloque de texto-cifrado obtenido de la encriptación del bloque anterior. ¿Qué sucede con el primer bloque? No disponemos de texto-cifrado de un bloque anterior. Esto se resuelve con la utilización de información aleatoria como primer bloque de texto-cifrado. De esta manera, se consigue que dos mensajes iguales no generen el mismo texto-cifrado. A este bloque de texto-cifrado se lo conoce como vector de inicialización, *initialization vector* o IV. No es preciso que este bloque sea secreto, como no lo son los otros bloques de texto-cifrado producidos al encriptar el mensaje.
- **CFB**: *Cipher Feedback* o cifrado retroalimentado. Es un modo que permite utilizar cifradores por bloques como cifradores por flujo. De esta forma, la información puede ser encriptada en tamaños menores a la longitud de un bloque. El modo se basa en la utilización de una pila o cola del tamaño del bloque de entrada. Inicialmente, se utilizará un vector de inicialización (como los usados en el modo CBC) y, mediante operaciones XOR y de desplazamiento, luego de la encriptación se obtiene el primer byte (si ésa ha sido la longitud determinada) de texto-cifrado. Una diferencia para notar respecto del modo CBC, en tanto el uso del vector de inicialización, es que en este caso debe ser único; debe utilizarse uno diferente por cada mensaje para cifrar.
- **OFB**: *Ouput feedback* o retroalimentación de salida. Este es otro modo para utilizar un cifrador por bloques como un cifrador por flujo. La diferencia con respecto a CFB estriba en que aquí una cantidad determinada de bits del bloque de salida anterior se traslada hacia las posiciones menos significativas de la pila o cola. Como con CFB, se utiliza también un IV, que debe ser único pero no debe ser secreto.

Relativo a las llaves criptográficas.

Al utilizar criptografía simétrica, la seguridad del sistema dependerá de dos cosas: La fortaleza del algoritmo y la longitud de la llave. Suponiendo que el algoritmo sea lo suficientemente fuerte o seguro como para descartar un ataque sobre él, la única alternativa que restará para comprometer la seguridad del sistema será un ataque por fuerza bruta, esto es, la prueba de todas las llaves posibles.

Si la llave fuese, por ejemplo, de 8 bits de longitud, existirán 2^8 o 256 llaves posibles; 256 pruebas deberíamos realizar para descubrir la llave, teniendo un 50% de probabilidades de encontrar la llave al haber realizado la mitad de las pruebas.

El algoritmo DES, por ejemplo, utiliza una longitud de llave de 56 bits, como veremos más adelante; otros más modernos, como AES o Blowfish, utilizan 128 bits. En el caso del algoritmo DES, entonces, tenemos 2^{56} llaves posibles. Suponiendo que una computadora o sistema distribuido pudiese probar un millón de llaves por segundo, se tardarían 2 285 años en encontrar la llave. Para el caso de llaves de 128 bits de longitud, el número de años sería de 10^{25}. Debe tenerse en cuenta que se han realizado estudios que demostraron que, disponiendo de una suma importante de dinero –hablaríamos de grandes corporaciones o gobiernos–, se podrían construir sistemas que probasen todo el espacio de llaves de DES (56 bits) en algunas horas. Para el caso de los otros algoritmos de longitudes de llave de 128 bits, se estima que aún sería imposible un ataque por fuerza bruta completo que no demorase muchos años.

Por supuesto, todo esto vale en tanto no utilicemos los bits que representarían "1234" o "clave", directamente como nuestra llave secreta. Aunque 128 bits, que corresponden a 16 bytes, nos puede dar la idea de utilizar una cadena de caracteres ASCII (de, a lo sumo, 16 caracteres) como llave, esto no debe hacerse. Por lo general, a través de funciones de *hashing* o algoritmos específicos a tal objetivo (donde, por ejemplo, se aplica *hashing* iterativamente hasta 1 000 o 2 000 veces), se obtendrán los 128 bits necesarios a partir de la cadena de caracteres de la llave o frase-clave (*passphrase*). De esta manera, se obtendrá una fortaleza mayor en llave, teniendo ésta mejor o mayor entropía (mejor aprovechados los 128 bits), y serán evitados los ataques por fuerza bruta del tipo "diccionario", donde no se prueba todo el espacio de llaves sino sólo palabras comúnmente utilizadas como contraseñas.

En lo que a criptografía asimétrica se refiere, se recuerda que los algoritmos más popularmente utilizados se basan en la dificultad de factores de números grandes que sean resultado del producto de dos números primos.

Notaremos que estos algoritmos también serán vulnerables a ataques por fuerza bruta, aunque de otro tipo, a diferencia de los tratados más arriba. Aquí, comprometer la seguridad del sistema no implica la prueba de cada una de las llaves posibles, sino el intento de factorizar el número resultado del producto de los dos números primos grandes.

Para tener una idea de cómo, a través de los años, se han podido factorizar este tipo de números, contra pronósticos mucho más optimistas, recomendamos al lector la consulta de los resultados del "*RSA Factoring Challange*" o "Desafío de Factores RSA". El concurso público fue mantenido por la empresa, o más precisamente por el sector o división "*RSA Laboratories*", hasta el año 2007 y a partir de entonces quedó inactivo.

La página Web oficial se encuentra en la siguiente dirección:

```
http://www.rsa.com/rsalabs/node.asp?id=2093
```

El concurso, básicamente, se trataba de que en los laboratorios RSA generaran un producto a partir de números primos grandes. Publicaban este resultado y aseguraban que los factores fueran descartados. Comenzó en el año 1991, para el número

RSA-100, con un premio de 1 000 dólares que obtuvo, ese mismo año, un matemático holandés.

El número RSA-100 implica una longitud de 100 dígitos decimales en el número producto (los factores encontrados por el matemático fueron los dos números primos, de 50 dígitos decimales cada uno).

Los últimos números RSA factorizados fueron RSA-640 (640 dígitos binarios, o bits, correspondientes a 193 dígitos decimales) y RSA-200 (200 dígitos decimales, o 663 bits), ambos en el año 2005 y por el mismo equipo de personas (Bahr, M. Boehm, J. Franke, T. Kleinjung). Según la información provista por el equipo a los laboratorios RSA, el esfuerzo demandado correspondió al equivalente de 30 años de procesamiento de un CPU de 2.2 GHz Opteron, durante cinco meses calendario.

Visto esto, nos debería quedar en claro la razón por la cual, como se verá más adelante, aunque actualmente sean utilizadas llaves asimétricas de 1 024 bits de longitud, ya se recomienden longitudes de al menos 2 048 bits.

Antes de terminar, aclararemos que al generar las llaves, sean éstas las secretas para criptografía simétrica o los pares pública-privada para el cifrado asimétrico, siempre es recomendable la utilización de información aleatoria para generarlas; pero deben tenerse en cuenta las consideraciones vistas en la sección anterior.

Selección de un algoritmo.

A la hora de incorporar criptografía en nuestros desarrollos, nos veremos en la necesidad de optar por alguno de los algoritmos disponibles en nuestro entorno de programación, *frameworks* o librerías adicionales.

En primera instancia, deberíamos tener en claro que la encriptación asimétrica no reemplaza a la simétrica, que ambos mecanismos proveen soluciones a problemas diferentes y cuándo nos será conveniente la utilización de cada una. Lo mismo en tanto a las funciones de *hashing*: No deberemos confundirnos pensando que pueden ser utilizadas para la encriptación y/o desencriptación de información. Si estos conceptos elementales no estuviesen claros, será conveniente la relectura de las primeras secciones de este capítulo antes de seguir adelante.

Como regla general, además de las cuestiones técnicas que serán vistas en las siguientes secciones, podemos decir que un factor para tener en cuenta debe ser el de la legalidad de la implementación y la utilización de un algoritmo determinado en el país en donde fuera a funcionar o a distribuirse la aplicación.

Si bien convendrá tener presentes los detalles de cada uno de los algoritmos, de manera general es posible asegurar que el algoritmo DES, por ejemplo, es menos seguro que sus alternativas más modernas, como AES o Twofish. Siendo AES el elegido en concurso para el establecimiento de un estándar federal para criptografía simétrica en los E.E. U.U., representa actualmente una opción prudente. Con respecto a los algoritmos de *hashing*, SHA-1 y las diferentes versiones de SHA-2, son recomendadas

antes que MD5, algoritmo sobre el cual se han descubierto diferentes problemas recientemente, que han puesto en duda su robustez. Por último, para el caso de criptografía asimétrica, quizá contemos con menos alternativas. Lo más conservador significará la utilización del algoritmo RSA, minuciosamente analizado a través de los años, ya que en tanto se utilicen llaves de una longitud segura, no representará, hasta donde sabemos, ningún riesgo.

Algoritmos de criptografía simétrica.

Algoritmos DES y TripleDES.

DES, por sus siglas en inglés de *Data Encryption Standard*, es un algoritmo de criptografía simétrica que fue aprobado por el Intituto Nacional Americano de Estándares –de los E.E.U.U– o *American National Standards Institute* (ANSI), en el año 1981. Antes fue aprobado también como estándar federal del mismo país, FIPS (*Federal Information Processing Standard*), en el año 1976 (publicado en 1977 como FIPS PUB 46).

Se trata de un algoritmo de cifrado por bloques, que encriptará información en bloques de 64 bits de longitud. En la encriptación, un bloque de este tamaño será entrada o *input* del algoritmo, el texto-plano y, junto con la llave –ya que se trata de un algoritmo simétrico– se producirá una salida o *output* del mismo tamaño, el texto-cifrado.

La misma llave y el mismo algoritmo se utilizan tanto para el cifrado como para el descifrado de información (salvo por una pequeña diferencia en el manejo de la llave). La longitud de la llave es de 56 bits y es posible utilizar cualquier número que quepa en ese tamaño como llave.

El algoritmo se basa fundamentalmente en una combinación de técnicas de confusión y difusión. En este caso particular, se realiza una sustitución seguida de una permutación del texto de entrada, en función de la llave. Esta operación se conoce como *round* o ronda. El algoritmo DES realiza 16 rondas, lo que quiere decir que aplica este proceso 16 veces sobre el bloque texto-plano (en la encriptación).

TripleDES (también escrito TDES) es un algoritmo formado a partir del algoritmo DES. Básicamente realiza el proceso de DES tres veces y es también del tipo de cifrado por bloques. Cuando los 56 bits de longitud de la llave de DES resultaron escasos, se ideó una alternativa para que, manteniendo el mismo algoritmo, se pudiese incrementar el tamaño de la llave. De esta manera, surge entonces este nuevo algoritmo, que en su versión o variante más simple no es otra cosa que la triple aplicación del algoritmo DES, con una llave que correspondería al conjunto de las tres llaves DES aplicadas.

En cuanto a la especificación técnica del algoritmo DES, y describiéndolo aquí a grandes rasgos, destacaremos que, como hemos dicho, opera en bloques de 64 bits de longitud de la entrada o texto-plano. Luego de una permutación inicial, el bloque es

dividido en dos bloques de 32 bits. Se aplican entonces 16 *rounds*, o rondas, de las mismas operaciones, en las cuales la información es combinada con la llave. Luego de las 16 rondas, las dos mitades citadas vuelven a juntarse para dar lugar a una última permutación.

Entonces, para terminar, si B_i es el resultado de la iteración número *i*, L_i y R_i son las dos mitades de B_i, K_i la llave para la ronda *i* y f la función que aplica sobre la llave (permutaciones, transposiciones, operaciones XOR), una ronda implicará que:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

Algoritmo AES.

El algoritmo AES, siglas de *Advanced Encryption Standard* o Estándar para la Encriptación Avanzada, es, como su predecesor DES, un algoritmo de criptografía simétrica por bloques. Es ya un estándar del gobierno de los E.E. U.U., siendo su objetivo reemplazar al algoritmo DES.

Fue anunciado por el Instituto Nacional de Estándares y Tecnología, o N*ational Institute of Standards and Technology* (NIST), en el año 2001. Es, desde 2002, un estándar del gobierno federal de los E.E. U.U., bajo un *Federal Information Processing Standard* (FIPS): Concretamente, como FIPS PUB 197.

Es también conocido como "Rijndael", sobre la base de los apellidos de sus creadores, Joan Daemen y Vincent Rijmen, quienes presentaron el algoritmo al *Advanced Encryption Standard Contest* y fue seleccionado entre otros quince algoritmos finalistas (entre ellos Twofish, de Bruce Schneier).

Nótese que, ya que el estándar AES ha sido algo más restrictivo, Rijndael permite un mayor rango de tamaños posibles de bloques y de longitudes de llaves. AES especifica un tamaño de bloque fijo de 128 bits de longitud; para la llave tamaños de 128, 192 ó 256 bits. En Rijndael, en tanto sean múltiplos de 32 bits, las longitudes de llave y bloque podrán variar entre 128 y 256 bits.

Con respecto a la especificación, o funcionamiento en sí del algoritmo, se destacará que Rijndael (o AES) utiliza lo que se conoce como campo Galois (apellido de un matemático francés, precursor de esta teoría del álgebra abstracta) para llevar a cabo gran parte de sus operaciones matemáticas. Este campo es una construcción matemática especial, en donde las operaciones de adición, sustracción, multiplicación y división son redefinidas y donde se dispondría de un número limitado de enteros.

Más precisamente, en Rijndael, el campo Galois utilizado sólo permite un número

de 8 bits (un número del 0 al 255) dentro de él. Todas las operaciones matemáticas definidas en este ámbito resultarán en un número de 8 bits.

Por ejemplo, en Rijndael, ambas, la suma y la resta representan operaciones XOR, no hay diferencia entre la adición y la sustracción. Para el caso de la multiplicación, las cosas ya no son tan simples. Veámoslo rápidamente: Teniendo dos números de 8 bits, *a* y *b*, y un producto *p*, también de 8 bits, los pasos que se han de seguir implicarían:

$$p = 0$$

Iterando, con $i = 1$ hasta 8:

Si el bit menos significativo de *b* es 1 entonces:

$$p = p \oplus a$$

Se realiza una copia de resguardo de *a*, luego

$$a = a << 1$$

Si la copia de resguardo de a, en su bit más significativo, poseía un 1:

$$a = a \oplus \text{0x1b (constante, expresada en hexadecimal)}$$

$$b = b >> 1$$

Terminado el ciclo de ocho iteraciones, *p* contendrá el valor resultado del producto entre *a* y *b*.

El símbolo $\oplus$ representa a la operación XOR, y $<<$ al desplazamiento a izquierda, $>>$ a derecha, de bits, aplicado en el operando de la izquierda, en la cantidad que especifique el operando de la derecha.

Se comentó acerca de este proceso de multiplicación porque su inversa se utiliza en la generación de la S-box, *Sustitution Box* o caja de sustitución del algoritmo. A grandes rasgos, el múltiplo inverso de un número de entrada se almacenará en dos variables temporales, sobre las cuales se realizarán operaciones de rotación, desplazamientos y XOR, para la obtención, finalmente, del valor transformado.

Lamentablemente, tratándose éste de un algoritmo relativamente complejo, más allá de las generalidades comentadas para conocer las nociones más importantes respecto de la especificación del algoritmo y qué estructuras matemáticas implementa, la descripción completa del mismo quedaría fuera del alcance de este libro. Se remite al lector a los estándares comentados, para obtener una descripción detallada y acabada respecto de la especificación formal y completa del algoritmo.

Algoritmo IDEA.

El algoritmo IDEA, de sus siglas en inglés *International Data Encryption Algorithm,* o Algoritmo Internacional de Encriptación de Datos, fue diseñado por Xuejia Lai y James L. Massey en el año 1991.

Fue un algoritmo propuesto como reemplazo del algoritmo DES. IDEA corresponde a una versión mejorada de PES (*Proposed Encryption Standard* o Estándar de Encriptación Propuesto), primer nombre del algoritmo, que luego, y antes de IDEA, también fue llamado IPES (*Improved PES* o PES Mejorado).

Este algoritmo trabaja con bloques de 64 bits de longitud, con una llave de 128 bits. El proceso para cifrar y descifrar es muy similar.

Parte de la popularidad de este algoritmo se debe a que fue utilizado en PGP 2.0 y es uno de los algoritmos opcionales en el estándar OpenPGP.

El concepto principal en cuanto al diseño del algoritmo radica en que se trata de una mezcla de operaciones de diferentes grupos algebraicos. Tres grupos forman parte de esta mezcla: XOR, adición en modulo 2^{16} y multiplicación en modulo $2^{16} + 1$.

En relación con cuestiones específicas del funcionamiento del algoritmo, tendremos en cuenta que trabaja con un bloque de entrada de 64 bits de longitud. Este bloque será dividido luego en cuatro sub-bloques de 16 bits cada uno: X_1, X_2, X_3 y X_4. Esto corresponderá a la entrada de la primera ronda del algoritmo. El algoritmo consta de ocho rondas en total. En cada una de estas rondas se realizan operaciones XOR, adiciones y multiplicaciones sobre los cuatro sub-bloques, entre sí y con las seis sub-llaves de 16 bits. Entre rondas, el segundo y tercer sub-bloque son intercambiados. Por último, los cuatro sub-bloques son combinados con cuatro sub-llaves en una transformación de salida.

El proceso de la llave quizá necesite una explicación más acabada: En primera instancia, los 128 bits serán divididos en ocho sub-llaves de 16 bits cada una y, a partir de esto, se obtienen las seis sub-llaves que se usarán para la primera ronda y las dos primeras sub-llaves para utilizar en la segunda ronda. Luego, la llave será rotada hacia la izquierda, en 25 bits, y se la volverá a dividir en ocho sub-llaves. Las primeras cuatro serán usadas en la segunda ronda (recuérdese que ya teníamos las dos sobrantes de la llave original para completar las seis necesarias o utilizadas en una ronda) y las cuatro sub-llaves restantes serán usadas en la tercera ronda. Luego, la llave es rotada nuevamente, de la misma manera que antes, y el proceso continua de esta forma hasta terminar el algoritmo.

Se detallan a continuación los pasos involucrados en cada ronda:

1. Multiplicar X_1 y la primera sub-llave.
2. Sumar X_2 y la segunda sub-llave.
3. Sumar X_3 y la tercera sub-llave.

4. Multiplicar X_4 y la cuarta sub-llave.
5. Realizar XOR entre los resultados de los pasos 1 y 3.
6. Realizar XOR entre los resultados de los pasos 2 y 4.
7. Multiplicar el resultado del paso 5 con la quinta sub-llave.
8. Sumar los resultados de los pasos 6 y 7.
9. Multiplicar el resultado del paso 8 con la sexta sub-llave.
10. Sumar los resultados de los pasos 7 y 8.
11. Realizar XOR entre los resultados de los pasos 1 y 9.
12. Realizar XOR entre los resultados de los pasos 3 y 9.
13. Realizar XOR entre los resultados de los pasos 2 y 10.
14. Realizar XOR entre los resultados de los pasos 4 y 10.

La salida de la ronda corresponde a los cuatro sub-bloques que son resultados de los pasos 11 al 14. Se intercambiarán luego, como hemos dicho, el segundo y el tercero para obtener lo que será la entrada para la siguiente ronda.

Terminadas las ocho rondas, el proceso de transformación final de salida citado implicaría realizar nuevamente los cuatro primeros pasos de una ronda:

1. Multiplicar X_1 y la primera sub-llave.
2. Sumar X_2 y la segunda sub-llave.
3. Sumar X_3 y la tercera sub-llave.
4. Multiplicar X_4 y la cuarta sub-llave.

Esto para que, por último, los cuatro sub-bloques sean agrupados nuevamente produciendo la salida, es decir, el texto-cifrado.

Algoritmos Blowfish y Twofish.

El algoritmo Blowfish ha sido desarrollado o diseñado por Bruce Shneier en el año 1993. Fue ideado con la intención de reemplazar al algoritmo DES, proponiéndolo como una alternativa sin problemas de patentamiento y libre de uso o de dominio libre. Se trata también, por supuesto, de un algoritmo criptográfico de llave simétrica y cifrado por bloques.

El algoritmo ha gozado de popularidad entre la comunidad y muchos *frameworks* y librerías criptográficas para desarrolladores, como así también aplicaciones cripto-

gráficas, lo implementan. Sin embargo, según opinión de algunos expertos, para implementaciones actuales, quizá la opción del algoritmo AES sea la conveniente.

Ha resistido un intenso cripto-análisis a través de los años, no habiéndose detectado problemas importantes en el algoritmo. Con respecto a la especificación de este algoritmo, destacaremos que implementa un cifrado por bloques, con una llave (simétrica) de longitud variable.

El algoritmo consta de dos partes: La expansión de la llave y la encriptación de la información de entrada (que se procesará en bloques de 64 bits de longitud). La primera de estas partes se refiere a la conversión de la llave (de hasta 448 bits de longitud) en varios *arrays* o arreglos de sub-llaves. La segunda parte, respecto de la encriptación, consiste en una función que se aplica 16 veces. Cada una de estas 16 iteraciones, o rondas, consistirá de una permutación dependiente de la llave y de una sustitución dependiente, tanto de la llave como de la información de entrada. Casi todas las operaciones son adiciones y operaciones XOR en variables de 32 bits de longitud.

Veamos en símbolos, de manera resumida, cómo trabajaría el algoritmo para la encriptación, teniendo en cuenta que se trata de una red Feistel consistente en 16 rondas. La entrada x es un bloque de 64 bits de longitud. El *array P* contiene a las 18 sub-llaves de 32 bits. El símbolo $\oplus$ representa a la operación XOR.

Se divide x en dos mitades de 32-bit: x_L, x_R

Iterando, con i = 1 hasta 16:

$$x_L = x_L \oplus P_i$$

$$x_R = F(x_L) \oplus x_R$$

Se intercambian x_L y x_R

Se intercambian x_L y x_R

$$x_R = x_R \oplus P_{17}$$

$$x_L = x_L \oplus P_{18}$$

Se combinan x_L y x_R

La función F() se especifica del siguiente modo, dividiendo en primera instancia a x_L en cuatro cuartos de 8 bits cada uno: *a, b, c*, y *d*. Se define que:

$$F(x_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \oplus S_{3,c}) + S_{4,d} \bmod 2^{32}$$

Teniendo en cuenta que *S* representa a las S-boxes, *Sustitution Boxes* o cajas de sustitución, de las cuales se vale el algoritmo; en este caso, cuatro de 32 bits, teniendo 256 items cada una:

$$S_{1,0}, S_{1,1},..., S_{1,255}$$
$$S_{2,0}, S_{2,1},..., S_{2,255}$$
$$S_{3,0}, S_{3,1},..., S_{3,255}$$
$$S_{4,0}, S_{4,1},..., S_{4,255}$$

Comentaremos brevemente acerca del algoritmo Twofish, que fue creado por el mismo autor de Blowfish y, si bien está relacionado con este último, es significativamente más complejo. El algoritmo vió la luz en el año 1998 y se trata también de un cifrador por bloques (ahora de 128 bits de longitud), que acepta una llave de longitud variable de hasta 256 bits.

El algoritmo fue finalista entre los algoritmos propuestos para el AES en el *Advanced Encryption Standard Contest,* el concurso en que se estableció ganador al algoritmo Rijndael.

Las características distintivas del algoritmo podrían resumirse en que las S-boxes utilizadas serán dependientes de la llave y de un manejo de esta última relativamente complejo (una parte de la llave afectará al algoritmo de encriptación).

Notaremos de manera breve que, técnicamente, el algoritmo consta de una red Feistel, realiza 16 rondas, aplicando una función biyectiva basada en cuatro S-boxes o cajas de sustitución dependientes de la llave.

Con respecto a sus cuestiones técnicas en detalle, o a su especificación formal completa, se refiere al lector al sitio Web del autor del algoritmo, referenciado al final del libro. Es probable que se necesiten algunos conceptos criptográficos y matemáticos para describirlo, que escaparían al alcance de este libro. Se comentaron las generalidades al respecto para que el lector conozca de qué se trata, entienda qué estructura general implementa y pueda comparar esta información con la de otros algoritmos.

Algoritmo RC4.

El algoritmo de criptografía simétrica RC4 es, a diferencia de los anteriores, un cifrador en flujo o *stream cipher*. DES, TripleDES, IDEA, AES y Blowfish son también algoritmos de criptografía simétrica, pero operan sobre bloques de entrada. Un cifrador en flujo operaría –esto no es así estrictamente en todos los casos, piénsese así de momento para abordar el concepto– sobre cada byte de entrada. RC4 es el algoritmo más popularmente utilizado entre los algoritmos de este tipo.

El algoritmo fue desarrollado, en el año 1984, por Ron Rivest para la compañía RSA Data Security Inc. Por varios años la compañía mantuvo en secreto los detalles del algoritmo, compartiéndolos sólo después de la firma de un acuerdo de no divulgación. En 1994 un anónimo lo hizo público en una lista de correo y, desde entonces, los detalles o especificaciones del algoritmo dejaron de ser secretos. Debe tenerse en cuenta que RC4 es una marca registrada de la compañía. Implementar el algoritmo de manera no oficial es legal, pero no lo sería utilizar el nombre RC4. Es por esta razón que también se encontrará este algoritmo con nombres como ARC4 (ARCFOUR) o Alleged-RC4.

En cuanto a los detalles del algoritmo, RC4 trabaja en modo OFB (ver apartado de modos y tipos en esta misma sección): El flujo de llave es independiente de la entrada o texto-plano. Consta de una S-box (*Sustitution Box* o Caja de Sustitución) de 8 x 8, que referenciaremos más adelante con los símbolos: S_0, S_1,..., S_{255}. Los elementos corresponden a una permutación de los números 0 a 255 y es una función –la permutación– de la llave de longitud variable. Contempla dos contadores, que llamaremos *i* y *j*, inicializados en cero.

Para la generación de un byte aleatorio, los pasos son:

$$i = (i + 1) \bmod 256$$

$$j = (j + S_i) \bmod 256$$

Se intercambian S_i y S_j

$$t = (S_i + S_j) \bmod 256$$

$$K = S_t$$

Entonces, sobre el byte que contiene *K* y sobre el texto-plano se le aplica la operación XOR para producir el texto-cifrado; lo mismo si, en cambio, se hiciera sobre el texto-cifrado para producir el texto-plano.

La inicialización de la S-box se detalla de la siguiente manera. En primera instancia, se completará linealmente: $S_0 = 0$, $S_1 = 1$,..., $S_{255} = 255$. Luego, se completará otro *array* o arreglo de 256 bytes con la llave, repitiéndola tantas veces como sea necesario para llenarlo completamente: K_0, K_1,..., K_{255}. Se establece el índice o contador *j* nuevamente a cero, para luego:

Iterando con $i = 0$ hasta 255:

$$j = (j + S_i + K_i) \bmod 256$$

Se intercambian S_i and S_j

De esta manera finaliza el algoritmo. Una característica importante para tener en cuenta de este algoritmo es que es rápido; el proceso de encriptación es, en un orden de aproximadamente 10 veces, más rápido que el del algoritmo DES.

Algoritmos de funciones de *hash*.

Algoritmo MD5.

El algoritmo de *hashing* criptográfico MD5 es una versión mejorada de su antecesor, MD4. Las siglas MD corresponden a *Message Digest* o resumen de mensaje. Fueron diseñados por Ron Rivest (la "R" en RSA), quien luego de los resultados del cripto-análisis sobre MD4 por otros criptógrafos, parcialmente críticos, lo extendió; a esta versión extendida y más compleja la llamó MD5. El algoritmo fue publicado en el año 1991.

El algoritmo produce, a partir de una entrada no limitada en cuanto a su tamaño, un *hash* criptográfico, o resumen de mensaje, de 128 bits de longitud. Se lo emplea actualmente en diferentes aplicaciones de seguridad, una de las más populares es la comprobación de integridad de archivos. Desde muchos sitios Web que ofrecen descargas de archivos, se ofrece también el resultado del *hashing* MD5 obtenido a partir del archivo, a manera de *checksum* o suma de comprobación, para poder confirmar, luego de descargarlo u obtenerlo de una fuente no confiable, que el *hash* criptográfico corresponde o no al publicado en el sitio Web. Estas comprobaciones suelen representarse en una cadena de 32 caracteres alfanuméricos, que corresponderán a los 16 bytes en formato hexadecimal.

En el año 1996, se descubrió un problema en el diseño de MD5. No se trataba de un problema o falla crítica, que volviese inseguras las implementaciones actuales, pero distintos expertos y criptógrafos propusieron la utilización de otros algoritmos de *hashing* criptográfico. Durante el año 2004, otros problemas más importantes fueron descubiertos y la seguridad del protocolo fue puesta en tela de juicio. Por último, un grupo de investigadores describió, en el año 2007, cómo es posible la confección de dos archivos distintos que generan el mismo *hashing* MD5.

Ninguno de estos problemas citados implica que las implementaciones actuales de *checksums* o de registros de contraseñas mediante *hashes* (las dos aplicaciones más populares del algoritmo actualmente) se hayan vuelto vulnerables. Al menos por ahora, no se ha descubierto una técnica que permita obtener la entrada original a partir de *hash* o alterar a discreción un archivo (salvo en las pruebas citadas, sobre archivos de contenido particularmente seleccionados) y mantener su *checksum* o *hash* MD5 resultante. La recomendación, de cualquier manera, es la de la implementación de otro algoritmo, como SHA-1, que veremos en el siguiente apartado.

Ya adentrándonos en la especificación concreta del algoritmo, se detalla que luego del preprocesamiento inicial, la entrada se procesa en bloques de 512 bits de longitud, divididos en 16 sub-bloques de 32 bits. La salida del algoritmo será una serie de cua-

tro bloques de 32 bits, que concatenados representarán el *hash* criptográfico de 128 bits de longitud, como hemos visto.

El mensaje o información de entrada es "paddeado", o completado utilizando un *padding,* agregando un bit de valor 1 seguido de tantos bits en 0 como sean necesarios, de manera tal que la longitud resultante sea 64 bits menor de un múltiplo de 512. Luego, se concatenará una representación del tamaño o longitud original de la entrada utilizando 64 bits. Esto implica que, en esta instancia, el mensaje tendrá una longitud en bits múltiplo de 512.

El paso siguiente es la inicialización de cuatro variables de 32 bits, llamadas *chaining variables*, o variables de encadenamiento, con valores prefijados:

A = 0x01234567

B = 0x89abcdef

C = 0xfedcba98

D = 0x76543210

Por cada bloque de 512 bits del mensaje, se producirá una iteración del ciclo principal del algoritmo. Dentro de éste, las cuatro variables son copiadas en otras cuatro variables temporarias:

$a = A$

$b = B$

$c = C$

$d = D$

Dentro de esta iteración principal, entonces, se realizarán cuatro *rounds* o rondas. Cada ronda implementa una serie de operaciones, y la aplicación de una función distinta, 16 veces. Cada una de estas funciones corresponde a una no lineal, que se parametrizará sobre tres de estas últimas cuatro variables (*a, b, c,* y *d)*; cada vez se adicionarán a este resultado la cuarta variable –la que no fue parámetro–, junto con un sub-bloque de la entrada y una constante. Sobre este último resultado se realizará una rotación o desplazamiento hacia la derecha en una cantidad variable de bits y se sumará el resultado a *a, b, c,* o *d.* Finalmente, el resultado reemplazará el contenido de alguna de estas cuatro variables (*a, b, c,* o *d*).

Cuatro son las funciones no lineales referidas; sólo una de ellas será utilizada en cada una de las cuatro rondas, en 16 ocasiones como hemos visto, donde en cada ocasión se combinarán de manera diferente los tres parámetros o variables entre *a, b, c* y *d*. El símbolo $\oplus$ corresponde a la operación XOR, $\wedge$ a AND, $\vee$ a OR y $\neg$ a NOT.

$F(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$

$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$

$H(X,Y,Z) = X \oplus Y \oplus Z$

$I(X,Y,Z) = Y \oplus (X \vee (\neg Z))$

Por último, luego de las cuatro rondas, de 16 pasos cada una, *a, b, c* y *d* son sumadas a *A, B, C* y *D,* y el algoritmo continuará con el siguiente bloque de entrada. Terminado el proceso del último bloque, el resultado final corresponderá a la concatenación de *A, B, C* y *D.*

Algoritmo SHA.

El Instituto Nacional de Estándares y Tecnología, o *National Institute of Standards and Technology* (NIST), en conjunto con la Agencia Nacional de Seguridad o *National Security Agency* (NSA) diseñaron el algoritmo SHA; éste fue nombrado a partir de las siglas del inglés *Secure Hash Algorithm* o Algoritmo de *Hash* Seguro.

Fue diseñado por estos organismos para ser utilizado en otro estándar, el llamado DSS por *Digital Signature Standard*, que implementa el algoritmo DSA de firma digital, visto anteriormente.

Existen, en realidad, cinco versiones o variantes del algoritmo: SHA-1, que generará un *hashing* criptográfico de 160 bits, y las variantes que le siguieron, que se llaman en conjunto SHA-2 y corresponden al conjunto de los algoritmos SHA-224, SHA-256, SHA-384 y SHA-512. Dentro de este último conjunto, el número que sigue a las siglas SHA especifica la longitud en bits del *hashing* criptográfico que genera el algoritmo.

El algoritmo, en mayor medida SHA-1, se implementa actualmente en diversas aplicaciones criptográficas. Como con MD5, quizá las más populares sean las del registro de *hashes* de contraseñas y la generación *checksums* o comprobaciones de integridad de archivos (además, por supuesto, del objetivo para el que fue creado: Su utilización junto al algoritmo DSA para la firma digital).

Con respecto a la especificación del algoritmo, el sistema de *padding* es muy similar al del algoritmo MD5, descripto en el apartado anterior. Se recomienda revisar este apartado si no se ha hecho, ya que las similitudes son varias. Luego, en lugar de cuatro variables de 32 bits con valores prefijados, SHA utiliza cinco –recuérdese que el algoritmo produce un *hash* criptográfico de 160 bits de longitud–, que se inicializarán de la siguiente manera:

A = 0x67452301

B = 0xefcdab89

C = 0x98badcfe

D = 0x10325476

E = 0xc3d2e1f0

También, como en el algoritmo MD5, el proceso principal consta de un ciclo dentro del cual, en cada iteración, se procesarán 512 bits del mensaje o información de entrada. Una diferencia para notar es que cada iteración aplica una función no lineal, como MD5, pero 20 veces en lugar de 16. Las adiciones y el corrimiento o desplazamiento son similares a los de MD5.

Veamos cuáles son las funciones no lineales utilizadas en SHA, que aplicarán ahora sobre cinco variables temporarias (en lugar de cuatro) que serán los parámetros *a, b, c, d* y *e*. Como con lo anterior, el símbolo $\oplus$ corresponde a la operación XOR, $\wedge$ a AND, $\vee$ a OR y $\neg$ a NOT.

$f_t(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$, para $t = 0$ to 19.

$f_t(X,Y,Z) = X \oplus Y \oplus Z$, para $t = 20$ to 39.

$f_t(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$, para $t = 40$ to 59.

$ft(X,Y,Z) = X \oplus Y \oplus Z$, para $t = 60$ to 79.

El número *t* representa el número de operación, que estará entre 0 y 79. Recuérdese que se trata de cuatro rondas, de 20 operaciones que involucran a una de estas funciones, en cada uno de los pasos u operaciones.

De manera similar a MD5, pero con cinco variables en lugar de cuatro, el resultado final estará compuesto por la concatenación de *A, B, C, D* y *E.*

Códigos de autenticación de mensaje (MAC) y algoritmo HMAC.

Hablaremos, brevemente, sobre esta muy utilizada aplicación de las funciones de una vía o *hashing*. Los códigos de autenticación de mensajes, o MACs, por sus siglas del inglés *Message Authentication Codes*, implican el uso de funciones de una vía de *hashing* criptográfico dependientes de una llave. Los MACs tienen las mismas propiedades que las funciones de *hashing* vistas, pero incluyen una llave. Únicamente quien conozca la llave podrá verificar el *hash* o resumen de mensaje. Este mecanismo es utilizado para probar la autenticidad de una información –un mensaje o un archivo– entre quienes conozcan la llave secreta.

Es posible aplicar esta técnica para autenticar archivos, frente a otros usuarios o, para un mismo usuario, para comprobar que el archivo no haya sido alterado, de ma-

nera que puede ser utilizado también para pruebas de integridad (además de para pruebas de autenticidad entre usuarios, como hemos visto).

A este respecto, pensemos en el siguiente ejemplo, que ayudará a comprender el objetivo de este mecanismo. Imaginemos que guardamos en nuestra computadora una serie de archivos, para cada uno de los cuales hemos calculado el *hash* correspondiente, mediante MD5 por ejemplo, y registramos ese resultado en un archivo Excel. Quien pretendiera modificar alguno de nuestros archivos, sabiendo que mantenemos los *hashes* de cada uno, podrá modificar este valor en el archivo Excel, luego de calcularlo él mismo sobre el archivo que modificó. De esta manera, no nos enteraríamos de esta alteración. Si, en cambio, en lugar de una función de *hashing* hubiésemos utilizado un MAC, esto no hubiera sido posible, ya que quien modifique uno de nuestro archivos desconocerá la llave secreta.

Una forma sencilla de convertir una función de *hashing* en un MAC es encriptar el resultado, el *hash* obtenido, con un algoritmo de encriptación simétrica. Sin embargo son más comunes las implementaciones que combinan la llave secreta con el mensaje, involucrando una función de *hashing* criptográfico.

Un algoritmo que implementa este protocolo es el HMAC. Fue publicado en el año 1996 por Mihir Bellare, Ran Canetti y Hugo Krawczyk. Fue estandarizado en EE.UU. bajo el FIPS (*Federal Information Processing Standard)* como PUB 198.

HMAC puede ser utilizado con cualquier función de *hashing*, siendo las más comúnmente utilizadas MD5 y SHA-1. Cuando se utilice la primera, se especifica HMAC-MD5; en cambio, HMAC-SHA-1 implica que se usa SHA-1. HMAC-SHA-1 y HMAC-MD5 forman parte de la especificación de los protocolos IPsec y TLS.

La definición formal del algoritmo es:

$$\text{HMAC}_k(m) = h\,(\,(K \oplus opad) \,\|\, h\,(\,(K \oplus \text{ipad}\,) \,\|\, m\,)$$

donde *h* es la función de *hashing* criptográfico, *K* la llave secreta, “paddeada” o completada hacia la derecha con ceros al tamaño de bloque de la función de *hashing* (512 bits cuando se utiliza MD5 o SHA-1), *m* el mensaje para ser comprobado, || corresponde a la operación de concatenación, $\oplus$ a la operación XOR, el *padding* externo *opad* repite el byte 0x5c, y el interno *ipad* lo propio con el valor 0x36, por el tamaño de un bloque.

Algoritmos de criptografía asimétrica.

Algoritmo RSA.

RSA es el algoritmo de cifrado asimétrico más popular en la actualidad. Creado por Ron Rivest, Adi Shamir y Leonard Adleman –nótese que son las iniciales de los apellidos las

que forman el nombre del algoritmo– fue publicado en el año 1977. Desde entonces ha resistido un extensivo cripto-análisis a través de los años, pero, en realidad, no se ha comprobado matemáticamente que sea seguro; tampoco ha sido comprobado lo contrario, pero esto mismo sugiere ya un nivel de confianza del algoritmo muy importante.

Actualmente, entonces, el algoritmo es considerado seguro, en tanto sean utilizadas llaves de longitud suficientemente seguras (se siguen utilizando llaves de 1 024 bits, pero ya se recomienda al menos una longitud de 2 048) e implementaciones actuales del algoritmo, donde se utilicen esquemas de *padding* seguros, como veremos más adelante.

El algoritmo sirve tanto para encriptar –y desencriptar, por supuesto– como para la generación de firmas digitales. Es, en la actualidad, ampliamente utilizado en protocolos de comercio electrónico, entre otras aplicaciones.

No hay que olvidar que debe usarse un *padding* criptográfico, tal como el definido en el estándar PKCS#1, para completar o "paddear" el mensaje previamente a la encriptación. Así mismo, para la firma digital también debe tenerse en cuenta la utilización de un *padding* seguro, como el del esquema RSA-PSS, por ejemplo, y la misma llave no deberá utilizarse para ambas operaciones (la de encriptación y la de firma digital).

La seguridad de RSA está basada en la dificultad de realizar el factoreo de números grandes. La llave privada y la pública son generadas o calculadas en función de un par de números primos, del orden de los 200 dígitos o más grandes aún (en el *RSA Factoring Challange*, el número RSA-200, por ejemplo, correspondía a un número decimal de 200 dígitos, o 663 bits de llave, y fue factorizado en dos números decimales de 100 dígitos en el año 2005).

Al describir ya concretamente al algoritmo, se establece que para la generación del par de llaves (llave pública y llave privada) se deberán seleccionar dos números primos grandes aleatorios, p y q, y se calculará n como su producto:

$$n = pq$$

La llave de encriptación, e, será elegida también de manera aleatoria, tal que e y $(p - 1)(q - 1)$ sean primos relativos.

La llave de desencriptación d será obtenida despejando la ecuación:

$$ed \equiv 1 \bmod (p - 1)(q - 1)$$

En otras palabras, o mejor dicho, en otros símbolos:

$$d = e^{-1} \bmod ((p - 1)(q - 1))$$

Los números *e* y *n* componen la llave privada; el número *d* corresponde a la llave privada; *p* y *q* serán descartados pero no revelados.

A la hora de encriptar un mensaje *m*, éste deberá ser dividido en bloques más pequeños que *n* y cada parte del texto-cifrado, *c*, será obtenida mediante:

$$c_i = m_i^e \bmod n$$

Para la desencriptación, cada parte o bloque del texto-cifrado se tomará para calcular:

$$m_i = c_i^d \bmod n$$

Algoritmo ElGamal.

El esquema de ElGamal también puede ser utilizado tanto para la encriptación y desencriptación de información (criptografía asimétrica), como para la firma digital de documentos electrónicos.

El algoritmo fue descripto por Taher Elgamal en el año 1984. Está basado en otro algoritmo, el del acuerdo de llaves de Diffie-Hellman.

Este esquema está implementado actualmente en aplicaciones criptográficas muy populares como el software libre GNU Privacy Guard, o GPG, y en versiones recientes de PGP.

A diferencia del algoritmo RSA, visto en el apartado anterior, este algoritmo obtiene su seguridad a partir de la dificultad de calcular logaritmos discretos en un campo finito.

Resumiremos al algoritmo describiendo de manera rápida sus componentes principales, comenzando por la generación de las llaves pública y privada. Esto implicará la selección de un número primo *p* y dos números aleatorios, *g* y *x*, tales que *g* y *x* sean menores a *p*. A continuación se calculará:

$$y = g^x \bmod p$$

La llave pública estará compuesta por *y*, *g* y *p*; mientras que *x* corresponderá a la llave privada.

Para la firma de un mensaje *M*, se elegirá un número aleatorio *k,* tal que éste sea un primo relativo a $p - 1$, para calcular luego:

$$a = g^k \bmod p$$

Lo que sigue será la obtención de *b* a partir de la ecuación:

$$M = (xa + kb) \bmod (p - 1)$$

La firma será entonces *a* y *b* y el valor *k* se mantendrá secreto. Por último, la verificación de una firma deberá confirmar que:

$$y^a a^b \bmod p = g^M \bmod p$$

La utilización de ElGamal para la encriptación es muy similar al algoritmo de intercambio de llaves de Diffie-Hellman, salvo por un par de ligeras diferencias. Para encriptar un mensaje *M*, se deberá obtener un *k* de la misma manera que para la firma, pero luego se calculará:

$a = g^k \bmod p$

y

$b = y^k M \bmod p$

Entonces *a* y *b* representarán el texto-cifrado.

Algoritmo DSA.

DSA, por sus siglas del inglés *Digital Signature Algorithm*, o Algoritmo de Firma Digital, es un estándar del gobierno federal de los E.E. U.U., dentro de un *Federal Information Processing Standard* (FIPS).

Fue propuesto por el NIST, en el año 1991. La patente del algoritmo fue atribuida a David Kravitz, ex-empleado de la Agencia Nacional de Seguridad de los E.E. U.U. (NSA). El estándar fue llamado DSS (*Digital Signature Standard*).

El algoritmo de llave pública para la firma digital DSA es una variante de los algoritmos ElGammal y Schnorr.

La especificación del algoritmo se resumirá a continuación:

- *p* corresponderá a un número primo, de una longitud de 512 a 1 024 bits (actualmente, el estándar ha sido modificado, recomendando la utilización de 2 048 bits), en tanto que esta longitud sea múltiplo de 64.
- *q* corresponderá a un número de 160 bits de longitud, factor primo de $p - 1$.
- *g* será igual a $h^{(p-1)/q} \bmod p$, donde h será cualquier número menor a $p - 1$ tal que $h^{(p-1)/q} \bmod p$ sea mayor que 1.
- *x* corresponderá a un número menor que *q*.

- y será igual a g^x mod p.

De esta forma, el número x representará la llave privada y la llave pública será y.

Para la firma de un mensaje m, se deberá obtener un número aleatorio k, tal que éste sea menor a q, y calcular:

$$r = (g^k \bmod p) \bmod q$$

y

$$s = (k^{-1} (H(m) + xr)) \bmod q$$

Nótese que $H(m)$ es una función de *hashing*; el estándar especifica la implementación del algoritmo de *hashing* criptográfico SHA-1. Los números calculados r y s representarán la firma.

La verificación de la misma se realizaría calculando:

$$w = s^{-1} \bmod q$$
$$u_1 = (H(m) * w) \bmod q$$
$$u_2 = (rw) \bmod q$$
$$v = ((g^{u1} * y^{u2}) \bmod p) \bmod q$$

Entonces, si v resultase igual a r, la firma será verificada.

Algoritmo Diffie-Hellman.

Hablaremos aquí del algoritmo Diffie-Hellman, que si bien no puede ser utilizado para la encriptación y desencriptación de información, permitirá a dos partes, sin conocimiento previo la una de la otra, establecer conjuntamente una llave secreta compartida sobre un canal o medio inseguro. Típicamente, esta llave secreta será luego utilizada para encriptar las comunicaciones sobre el canal inseguro, mediante el uso de criptografía simétrica. Se debe recordar que el algoritmo se utiliza para el intercambio o distribución de llaves: Dos partes podrán utilizarlo para la generación de una llave secreta compartida, pero no para encriptar o desencriptar mensajes.

Fue el primer algoritmo de llave pública conocido, diseñado por Whitfield Diffie y Martin Hellman en el año 1976. La seguridad del algoritmo está basada en la dificultad de calcular logaritmos discretos en un campo finito, en comparación con la facilidad de calcular la exponenciación en el mismo campo finito.

En cuanto a la especificación matemática del algoritmo, las dos partes acordarán un número primo grande, *n,* y un número *g,* tal que este último sea primitivo base mod *n*. Estos dos enteros no son secretos, lo que quiere decir que las dos partes pueden acordarlos sobre el canal inseguro. El protocolo continúa de la siguiente manera: La parte "A" elige un número aleatorio grande, *x*, y envía a la parte "B" lo que sigue:

$$X = g^x \bmod n$$

La parte "B" también elegirá un número aleatorio grande, *y*, y enviará a la parte "A":

$$Y = g^y \bmod n$$

La parte "A" calculará:

$$k = Y^x \bmod n$$

Lo propio hará la parte "B", calculando:

$$k' = X^y \bmod n$$

Entonces tanto *k* como *k´* serán iguales a $g^{xy} \bmod n$. Ninguna persona –o computadora– que haya monitoreado el canal inseguro puede calcular ese valor, ya que sólo conocería *n, g, X* e *Y.* Dada la dificultad del cálculo del logaritmo discreto (*n* debe ser un número grande, en esto radica en parte la seguridad del sistema) para recuperar *x* o *y*, la llave secreta generada, *k*, puede ser compartida por las partes de manera segura.

[1] Schneier, Bruce. *Applied Cryptography*. 2a ed. John Wiley & Sons, EE.UU., 1996.

[2] Tena Ayuso, Juan. *Protocolos Criptográficos y seguridad en redes*. Servicio de publicaciones Universidad de Cantabria, España, 2003.

Criptografía en entornos Java

3

El lenguaje de programación Java fue creado en el año 1991 por James Gosling, aunque en ese entonces era llamado Oak. Fue desarrollado, en un principio, para codificadores o conversores para sistemas de televisión, pero ya con la filosofía del soporte multiplataforma como consigna.

Originalmente desarrollado para la compañía Sun Microsystems, fue liberado por ésta en el año 1995 bajo el nombre de Java Platform. Su sintaxis deriva de los lenguajes C y C++ y es un lenguaje exclusivamente orientado a objetos. En su utilización más común, el código fuente es compilado generando un *bytecode* o pseudo-código máquina, que puede ser ejecutado en una máquina virtual Java (*Java Virtual Machine* o JVM). Lo destacable respecto de esto último es que el programa compilado podrá correr o ejecutarse en diferentes plataformas de hardware, con diferentes sistemas operativos, en tanto el sistema disponga de una JVM instalada.

Con respecto a las comunicaciones e Internet, aspecto de especial interés para lo referente a la criptografía, la primera aparición popular de la tecnología se dio de la mano de los *applets*. Se trata de pequeños programas que se ejecutan dentro de una página Web. Los navegadores, en aquel entonces, fueron distribuidos con una JVM incorporada. Al detectar en una página la referencia a uno de estos componentes, el navegador lo descargaba y ejecutaba automáticamente. Cabe aclarar que esta ejecución tiene lugar en la capa "cliente", es decir, en el sistema que visualiza la página a través del navegador.

Hoy, y desde Java 2, el lenguaje es ampliamente utilizado en la capa "servidor" en su plataforma J2EE (*Java 2 Enterprise Edition* o edición empresarial). La diferencia principal respecto de la versión estándar, es decir, la J2SE (*Java 2 Standard Edition*) es el agregado, en J2EE, de una serie de librerías que proveen funcionalidades para implementar sistemas multicapa, distribuidos y tolerantes a fallas sobre un servidor de aplicaciones mediante una arquitectura modular.

Por último, pero no menos importante, respecto de la utilización de esta tecnología no omitiremos la mención, al menos, de la plataforma J2ME (*Java 2 Micro Edition*).

Últimamente, debido al auge en el desarrollo de sistemas para dispositivos móviles, se ha popularizado esta versión o plataforma del lenguaje, que se trata de un sub-conjunto de las especificaciones originales, ideado para la implementación de aplicaciones en sistemas o micro-sistemas de recursos limitados, como pueden ser teléfonos celulares o PDAs (*Personal Digital Assistant* o Asistente Digital Personal).

En el año 2006, la versión 2 de Java o, mejor dicho, cada una de sus plataformas (J2SE o Standard Edition, J2EE o Enterprice Edition y J2ME o Micro Edition) fueron renombradas por Java EE, Java SE y Java ME, respectivamente.

Hacia mediados del año 2007 Sun Microsystems liberó completamente el código fuente de su distribución –salvo una porción de la cual no mantenía el *copyright* o derecho de copia–, como proyecto *open-source* bajo la licencia GPL (*General Public License*) de GNU.

Implementaciones incorporadas.

JCA o Java Cryptography Architecture.

Lo referente a la JCA (*Java Cryptography Architecture* o arquitectura criptográfica de Java) se introdujo en la distribución estándar dentro, o a partir de, la API de seguridad de Java, uno de los componentes básicos del lenguaje. En sus primeras versiones se encontraba encapsulada dentro del paquete o *package* `java.security` exclusivamente. A partir de la *Java Developer Kit* (JDK) 1.1 se incorporó allí –en la API de seguridad– la definición del *framework* de la JCA. Se trata de un *framework* para el acceso y el desarrollo de funcionalidades criptográficas. En la primera versión citada, incorporaba APIs para la firma digital y generación de *hashes* criptográficos.

No es difícil confundir los conceptos de aplicación, alcance y funcionalidades compartidas entre la *Java Cryptography Extensión* o JCE, que se describirá más adelante, y la JCA. Para intentar aclarar estas diferencias diremos primero que la JCA propone interfaces, clases abstractas, modelos para la implementación de algoritmos criptográficos –ya veremos los detalles respecto de la terminología utilizada: *Engines* o motores y *providers* o proveedores–, que luego puedan ser utilizados de una manera *a priori* formalizada. Es decir, salvo por la funcionalidad para generar *hashes* y firmar digitalmente, la JCA no provee otras implementaciones de funcionalidades criptográficas concretas. Por lo que, si necesitásemos, por ejemplo, cifrar de manera simétrica un documento, nos veremos obligados a utilizar las implementaciones provistas por la JCE que, siguiendo los lineamientos de la JCA, provee implementaciones de diferentes algoritmos criptográficos.

La JCA es parte del lenguaje Java, más específicamente, parte del API de seguridad del lenguaje, desde la versión JDK 1.1, cuando JCE era una extensión separada. Esto fue así ya que, en aquel entonces, la JCE no podía distribuirse libremente por impedimentos de las reglas de exportación vigentes en los EE.UU., aplicables a algoritmos de criptografía categorizada como “fuerte” (*strong cryptography*).

La arquitectura JCA ha sido diseñada de acuerdo con dos principios: La independencia e interoperabilidad de las implementaciones y la independencia y extensibilidad de los algoritmos.

La independencia de las implementaciones se consigue empleando una arquitectura basada en proveedores. El término proveedor de servicios criptográficos se refiere a un paquete, o conjunto de paquetes, que proporcionan una implementación concreta de las funcionalidades criptográficas de la API de seguridad de Java. Diferentes proveedores deben poder ser utilizados de manera transparente por la aplicación.

La interoperabilidad de las implementaciones destaca que cada una de ellas puede utilizarse con las demás; por ejemplo, usar las llaves generadas por otra implementación o verificarlas.

La independencia de los algoritmos se consigue definiendo tipos de servicios criptográficos y las clases que proporcionen la funcionalidad de estos servicios. A estas clases se las denomina *engine* o motor. Dentro de la JCA, algunas de estas clases son, por ejemplo, `MessageDigest`, `Signature` y `KeyFactory`.

La extensibilidad de los algoritmos establece que los nuevos algoritmos que se incorporen dentro de alguno de los tipos soportados puedan ser añadidos fácilmente.

Volviendo a lo referido acerca del modelo basado en *engines* o motores, detallaremos brevemente las clases principales que abordaremos a lo largo del capítulo, para tener en claro para qué utilizaríamos cada una:

- `SecureRandom`: La usaremos para generar números aleatorios o seudo-aleatorios.
- `MessageDigest`: Utilizada para generar o calcular el *message digest (hash)* de una información específica.
- `Signature`: Luego de inicializada con las llaves correspondientes, se usará para firmar digitalmente cierta información (y para luego verificarla).
- `Cipher`: También, luego de ser inicializada con la/s llave/s correspondiente/s, se utilizará para la encriptación y desencriptación de información. Se dispone de diferentes tipos de algoritmos: Simétricos de tipo *bulk* (por ej., AES, DES, DESede, Blowfish, IDEA), de tipo *stream* (por ej., RC4), asimétricos (por ej., RSA) y de *password-based encryption* (PBE).
- *Message Authentication Codes* (`MAC`): Como en el caso de `MessageDigest`, también se usan para generar *hashes*, pero son inicializadas con las llaves correspondientes *a priori* para proteger la integridad de los mensajes.
- `KeyFactory`: Clase usada para convertir llaves criptográficas de tipo `Key` en especificaciones de llave (representaciones transparentes del material de llave) y viceversa.
- `SecretKeyFactory`: Utilizada para convertir llaves criptográficas opacas de tipo `SecretKey` en especificaciones de llave (representaciones transparen-

tes del material de llave) y viceversa. Las `SecretKeyFactorys` son `KeyFactorys` especializadas, que crean llaves secretas (simétricas) únicamente.

- `KeyPairGenerator`: Generaremos con esta clase un par de llaves (pública y privada) aptas para su uso con un algoritmo especificado.
- `KeyGenerator`: La usaremos para generar llaves secretas para su utilización junto con un algoritmo especificado.
- `KeyAgreement`: Utilizada por dos o más partes para acordar y establecer una llave específica, que será usada por una operación criptográfica particular.

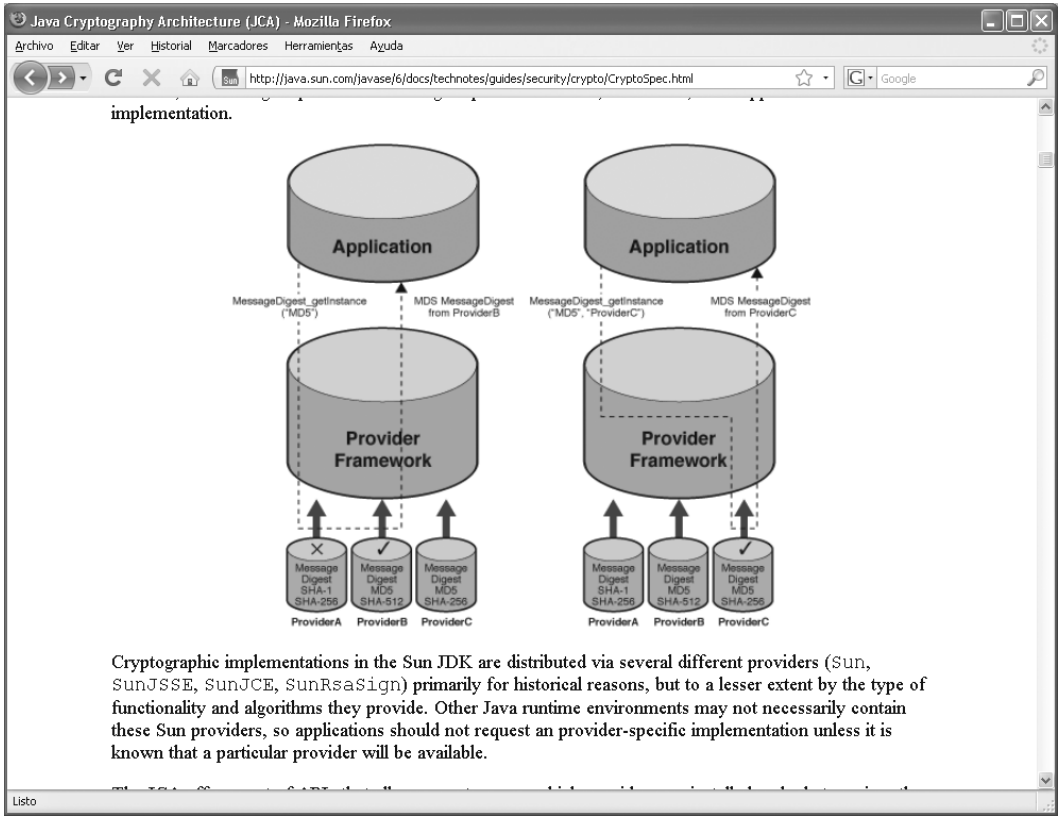

Fig. 3-1. Diagrama ilustrativo de la estructura de la JCA (*Java Cryptography Architecture* o arquitectura criptográfica de Java).

Librerías y *frameworks* adicionales.

JCE o Java Cryptography Extension.

La extensión criptográfica de Java o JCE –de sus siglas del inglés *Java Cryptography Extension*–, provee un *framework* e implementaciones de algoritmos para la encriptación, generación e intercambio de llaves criptográficas y autenticación de mensajes. Como se ha abordado en el apartado anterior, provee estas implementaciones siguiendo los lineamientos definidos por la JCA; esto es, implementando un "proveedor" o *provider* de funciones (implementaciones) para realizar operaciones criptográficas.

El diseño de la JCE está basado en los mismos principios que adopta la JCA: Independencia de la implementación y, hasta donde es posible, independencia de los algoritmos. La JCE usa la misma aquitectura de *providers* o proveedores. Estos pueden ser incorporados al *framework* JCE, agregando nuevos algoritmos de manera sencilla.

La API de la JCE incluye: Encriptación simétrica, con algoritmos como DES, RC2, RC4 e IDEA; encriptación asimétrica, a través del algoritmo RSA (sólo a partir de la versión 5.0 de la JDK o 1.5, es decir, la versión siguiente a la 1.4); encriptación basada en contraseñas (*Password-Based Encryption* o PBE); manejo de llaves, autenticación de mensajes, *hashing* y *Message Authentication Codes* (MAC).

La distribución de Java 2 SDK, versión 1.4, brinda un proveedor JCE estándar llamado `SunJCE`, preinstalado y registrado. Este proveedor soporta los siguientes servicios o funcionalidades criptográficas:

- Implementaciones de los algoritmos criptográficos DES (FIPS PUB 46-1), Triple DES y Blowfish, en los modos *Electronic Code Book* (ECB), *Cipher Block Chaining* (CBC), *Cipher Feedback* (CFB), *Output Feedback* (OFB) y *Propagating Cipher Block Chaining* (PCBC).
- Generadores de las llaves adecuadas para los algoritmos DES, Triple DES, Blowfish, HMAC-MD5, y HMAC-SHA1.
- Una implementación de encriptación MD5 con DES-CBC, basada en contraseña (PBE) definida en PKCS #5.
- *Secret-key factories* que proveen conversiones bi-direccionales entre objetos de llaves de DES, Triple DES y PBE.
- Una implementación del algoritmo para el acuerdo de llaves entre dos o más partes de Diffie-Hellman.
- Un generador del par de llaves –pública y privada– para el algoritmo Diffie-Hellman.
- Un generador de parámetros para el algoritmo de Diffie-Hellman.
- Una *key factory,* que provee conversiones bi-direccionales entre objetos de llaves de Diffie-Hellman.

- Manejadores de parámetros para el algoritmo de Diffie-Hellman, DES, Triple DES, Blowfish, y PBE.
- Implementaciones de los algoritmos de *hashing* o *keyed-hashing* HMAC-MD5 y HMAC-SHA1, definidos en la RFC 2 104.
- Una implementación del esquema de *padding* descripto en PKCS #5.
- Una implementación de una *keystore* para el tipo propietario llamado "JCEKS".

Más adelante en este capítulo, en las secciones "Librerías y frameworks adicionales", "Codificación de encriptación simétrica" y "Codificación de encriptación asimétrica" veremos código de ejemplo para implementar diferentes funcionalidades de criptografía; adelantaremos únicamente aquí que al especificar los nombres de los algoritmos no estamos obligados a respetar las diferencias entre mayúsculas y minúsculas, es decir que se tratan de manera *case-insensitive* por lo tanto, las siguientes expresiones son equivalentes:

```
MessageDigest.getInstance("SHA-1")
MessageDigest.getInstance("sha-1")
MessageDigest.getInstance("sHa-1")
```

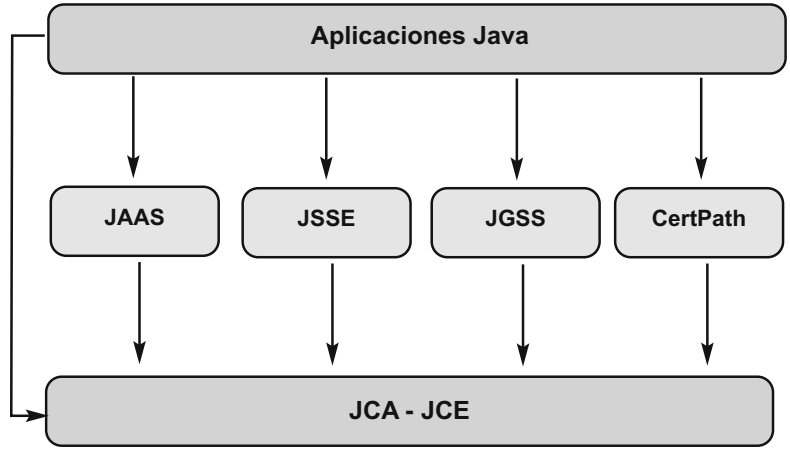

Fig. 3-2. Relaciones entre la JCA – JCE, otras APIs de seguridad y las aplicaciones.

Bouncy Castle Crypto APIs para Java.

Si bien hemos considerado en el apartado anterior la extensión criptográfica de Java o JCE como una librería o *framework* adicional por razones históricas, a partir de la

distribución de Java 2 SDK, versión 1.4, se incluye un proveedor JCE estándar llamado `SunJCE`, preinstalado y registrado. En este apartado trataremos acerca de la alternativa más popular a esta implementación. Se trata del proyecto de software libre que lleva adelante el grupo *The Legion of the Bouncy Castle*, literalmente "La Legión del Castillo que Rebota", refiriéndose a los castillos inflables en los que juegan los chicos.

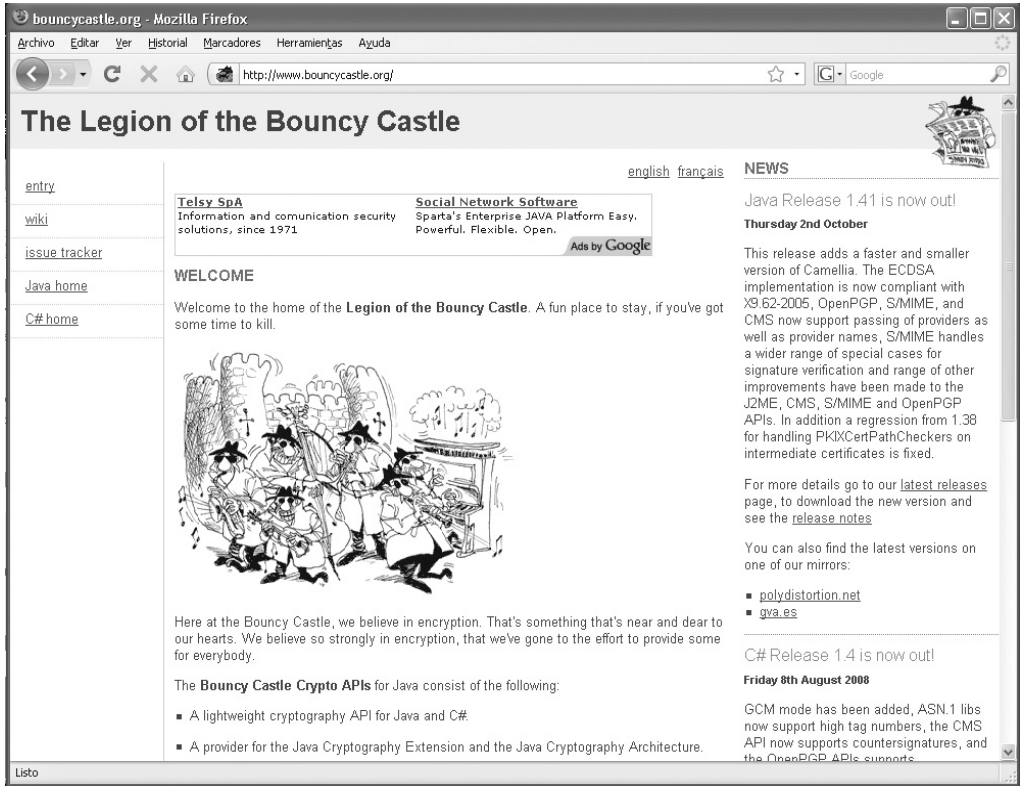

Fig. 3-3. Página Principal del sitio de The Legion of the Bouncy Castle.

La dirección del sitio Web del proyecto es `www.bouncycastle.org` y la sección específica para Java se encuentra en `www.bouncycastle.org/java.html`. (El proyecto también mantiene una API de servicios o implementaciones criptográficas para C#: `www.bouncycastle.org/csharp.html`).

De acuerdo con la documentación disponible en el sitio, las Bouncy Castle Crypto APIs para Java consisten en lo siguiente:

- Una API criptográfica “liviana” (el término se aplica aquí en el sentido de ligera, rápida, reducida, etc.; el proyecto pretende mantener el soporte para las máquinas virtuales corriendo J2ME).
- Un proveedor para la *Java Cryptography Extension* (JCE) y la *Java Cryptography Architecture* (JCA).
- Una implementación “limpia” de la JCE 1.2.1.
- Una librería para la lectura de objetos codificados en ASN.1.
- Una API de cliente TLS “liviana”.
- Generadores para certificados X.509 versión 1 y 3, CRLs versión 2 y archivos PKCS12.
- Generadores para certificados de atributos X.509 versión 2.
- Generadores / procesadores para S/MIME y CMS (PKCS7/RFC 3 852).
- Generadores / procesadores para OCSP (RFC 2 560).
- Generadores / procesadores para TSP (RFC 3 161).
- Generadores / procesadores para OpenPGP (RFC 2 440).
- Una versión firmada de un jar para ser utilizado con la JDK 1.4 a 1.6 y la JCE de Sun.

La API criptográfica de Bouncy Castle funciona con todos los entornos y versiones de Java, desde la recientemente liberada J2ME hasta la JDK 1.6, con la infraestructura adicional para adecuar los algoritmos para el *framework* JCE.

Salvo en los casos en los cuales se especifique lo contrario, el software producido por el proyecto está cubierto por la licencia de *MIT X Consortium*. La librería de OpenPGP también incluye una versión modificada de BZIP2, licenciada bajo la *Apache Software License* Versión 1.1.

El proyecto distribuye las clases JCE únicamente para las versiones o distribuciones JCE de Bouncy Castle para las JDK 1.1, JDK 1.2 y JDK 1.3. La distribución para las JDK 1.4 a 1.6 contienen sólo el proveedor y la API “liviana”. Las distribuciones para la JDK 1.0, J2ME, la JDK 1.1 y posteriores “livianas” sólo incluyen la API criptográfica de Bouncy Castle.

Esto significa: Si va a utilizar la JDK 1.4 o posterior, deberá descargar el *provider* o proveedor únicamente, que incluirá la API criptográfica, pero no precisará las clases para la implementación JCE. Si ha de desarrollar para la J2ME en cambio, la única alternativa es utilizar directamente la API “liviana” de Bouncy Castle.

Una nota adicional para tener en cuenta en caso de utilizar este *framework* con la JDK 1.4 o posterior: Se deberá utilizar el jar firmado (*provider*) y deberá descargar las *policy files* para el JCE de Sun para el correcto funcionamiento del proveedor. Las

policy files pueden ser descargadas desde donde están disponibles en la JDK. Si no se instalan estos archivos, el mensaje de error o excepción que recibirá será el siguiente:

```
java.lang.SecurityException: Unsupported keysize or algorithm
parameters
         at javax.crypto.Cipher.init(DashoA6275)
```

A manera de adelanto de cómo se utilizaría esta librería, y para reconocer (en los listados de código fuente que se incluirán luego en este capítulo) cuáles hacen uso de la extensión de Bouncy Castle, veremos cuál es el paquete principal que se ha de importar y cómo se especifica o determina el uso de este *provider* en lugar de `SunJCE` (preinstalado y registrado a partir de la JDK 1.4).

Para importar el *package* principal utilizaremos:

```
import org.bouncycastle.jce.provider.BouncyCastleProvider;
```

Para determinar el uso del provider de Bouncy Castle codificaremos:

```
Security.addProvider(new BouncyCastleProvider());
```

La forma de identificar al *provider* de Bouncy Castle es a través de la abreviatura `"BC"` y deberá indicarse cuando se quiera utilizar alguna de las implementaciones de algoritmos que soporta el *provider* Bouncy Castle.

Veamos dos ejemplos al respecto: El primero obtendrá una instancia de la clase `Cipher` (cifrador) y el segundo de una clase `KeyGenerator` (generador de llave).

```
Cipher cifrador =
 Cipher.getInstance("RSA/ECB/PKCS1Padding", "BC");


KeyGenerator keyGen =
 KeyGenerator.getInstance("DES", "BC");
```

Recuérdese, por último, que esta librería no forma parte de las distribuciones estándares de Java y al compilar y ejecutar, por lo tanto, se deberá incluir el jar obtenido desde la sección de descargas del sitio del proyecto.

Librerías Cryptix.

Comentaremos brevemente en este apartado acerca de las librerías Cryptix. Se trata en este caso, como *The Legion of the Bouncy Castle*, de un proyecto *open-source* o de código libre o abierto. No nos detendremos a desarrollar en mayor detalle esta librería, ya que el proyecto, aunque no comunicó el cierre o abandono oficialmente, parece no estar activo desde el año 2005.

La documentación refiere que las librerías pueden ser utilizadas con las JDK versiones 1.2, 1.3 y 1.4; pero nada especifican por supuesto, respecto de las versiones 5 o 6, ya que la última versión de la librería disponible para descarga en el sitio del proyecto es un *snap* del año 2005.

Además, las librerías Cryptix no se distribuyen en un jar firmado para utilizar dentro de contextos de seguridad, como sí lo hace el proyecto Bouncy Castle (téngase en cuenta que la implementación JCE incluida por defecto a partir de la JDK1 versión 1.4, SunJCE, está preinstalada y firmada). Por esto, se recomienda que, en caso de utilizar una implementación alternativa de proveedores JCE a la provista por Sun Microsystems por defecto, se utilice la del proyecto de Bouncy Castle y no ésta.

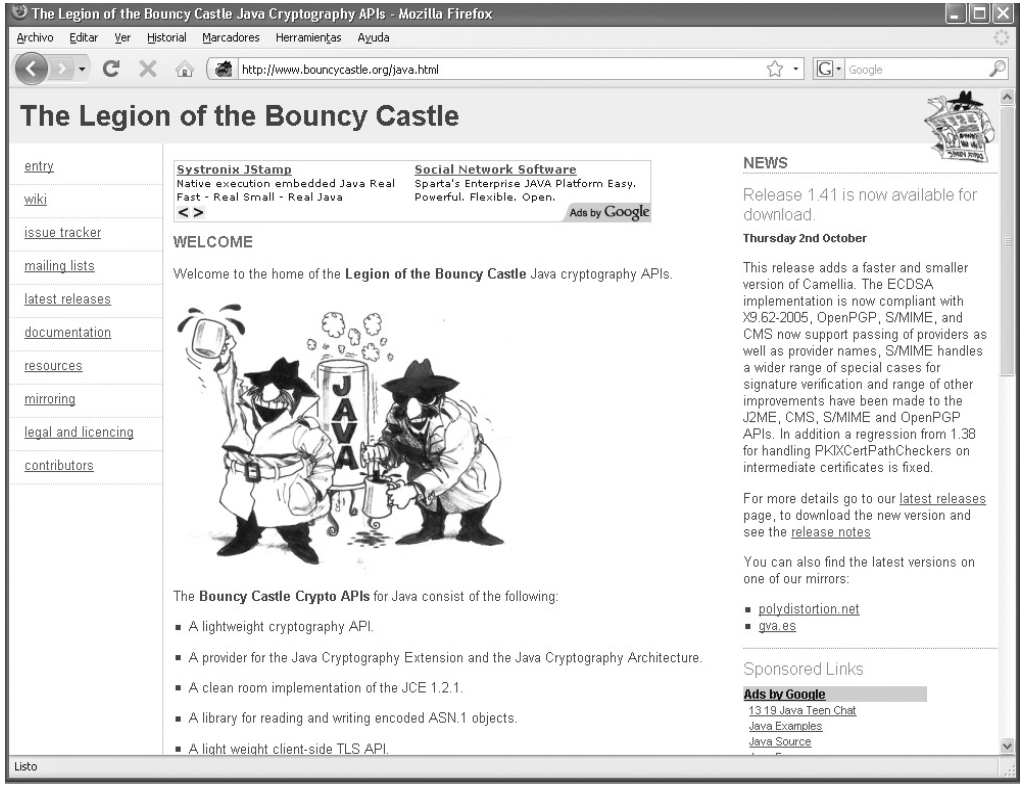

Fig. 3-4. Sección Java de la página de The Legion of the Bouncy Castle.

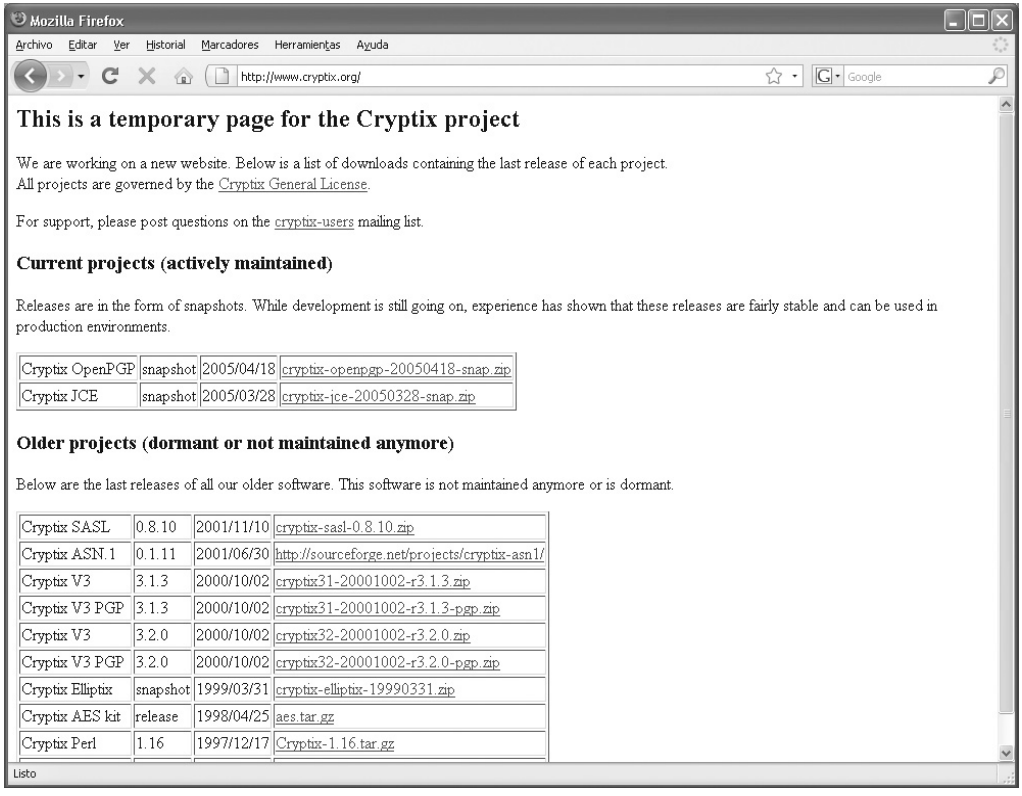

Fig. 3-5. Página principal (temporaria) del proyecto Cryptix - `http://www.cryptix.org`.

El *provider* JCE de Cryptix es definido en su documentación como un *plugin* criptográfico para el *framework* JCE de Sun Microsystems. Hablamos, por supuesto, como en el caso del SunJCE o el *provider* JCE de Bouncy Castle, de la implementación de los algoritmos criptográficos de acuerdo con los lineamientos de la JCA/JCE.

Con respecto al licenciamiento, la librería queda bajo su propia *Cryptix General Licence*, definida por *The Cryptix Foundation Limited*. Con *Copyright (C) 1995, 1996, 1997, 1998, 1999, 2000*, la licencia básicamente establece que las librerías pueden ser utilizadas (y distribuidas tanto en forma de código fuente como binaria) en tanto se mantenga la cita al derecho de copia de la *Cryptix Foundation*.

La última versión de la implementación JCE Cryptix soporta los algoritmos que se detallan a continuación.

Para firmar digitalmente:

- RSASSA-PSS (con opción de MD2, MD4, MD5, RIPEMD-128, RIPEMD-160, SHA-1, SHA-256, SHA-384, SHA-512 o Tiger como algoritmo de *hashing*).
- RSASSA-PKCS1 (con MD4, MD5, RIPEMD-128, RIPEMD-160 o SHA-1).

Funciones de *hashing*:

- MD2
- MD4
- MD5
- RIPEMD-128
- RIPEMD-160
- SHA-0
- SHA-1
- SHA-256
- SHA-384
- SHA-512
- Tiger

Criptografía de llave pública:

- RSASSA-OAEP (con opción de MD2, MD4, MD5, RIPEMD-128, RIPEMD-160, SHA-1, SHA-256, SHA-384, SHA-512 o Tiger como algoritmo de *hashing*).
- RSASSA-PKCS1.

Criptografía simétrica, en modos CBC, CFB, ECB, OFB y OpenPGP CFB, con opción de *no padding* o *padding* PKCS #5:

- AES
- Blowfish
- CAST5
- DES
- IDEA
- MARS
- RC2
- RC4
- RC6
- Rijndael
- SKIPJACK
- Serpent
- Square
- TripleDES
- Twofish

Ya que no va a encontrarse código, que utilice esta librería más adelante, en este capítulo, veremos aquí una clase de ejemplo de encriptación simétrica incluida en la distribución oficial de las librerías. Se aclaró que esta clase es parte de la distribución oficial (*package examples*) porque parece necesitar una confirmación en el caso de la longitud de la llave para TripleDES (está comentado en el código); lo supuesto allí es correcto.

```
/* $Id: SymmetricCipher.java,v 1.7 2000/07/28 20:09:20 gelderen Exp $
 *
 * Copyright (C) 1995-2000 The Cryptix Foundation Limited.
 * All rights reserved.
 *
```

```
 * Use, modification, copying and distribution of this software is

 * subject the terms and conditions of the Cryptix General Licence.
   You should have
 * received a copy of the Cryptix General Licence along with this
   library;
 * if not, you can download a copy from http://www.cryptix.org/ .
 */
package cryptix.jce.examples;


import sun.misc.BASE64Encoder;
import java.security.Security;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.ShortBufferException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.BadPaddingException;

import javax.crypto.spec.IvParameterSpec;


/**
 * Sample app for creation of ciphers, using them on a few bytes.
 *
 * @author: Josef Hartmann (jhartmann@bigfoot.com)
 * @version: $Revision: 1.7 $
 */
public final class SymmetricCipher
{
   /** cipher object */
   private Cipher cipher=null;

   /** keygenerator object */
   private KeyGenerator kg=null;

    /**
     * Starts the application. Create keys, iv´s, run cipher on a few
       bytes
```

```
     * and pass values for using the FileDEncryption class.
     *
     * @param algorithm Algorithm.
     * @param mode      Cipher mode.
     * @param padding   Padding scheme the cipher should use.
     * @param provider  Name of provider as string.
     * @param filename  Name of the file to [de|en]crypt.
     */
    public void run(String algorithm, String mode, String padding,
                    String provider, String filename)
    {
        try
        {
            // create a cipher object: ("algorithm/mode/padding",
            // provider)
            // currently cryptix only supports padding types:
            // NoPadding,None
            cipher = Cipher.getInstance(
                         algorithm+"/"+mode+"/"+padding,provider);

            // get a key generator for the algorithm.
            kg = KeyGenerator.getInstance(algorithm,provider);

            int strength = 0; // strength of key
            IvParameterSpec spec = null; // initialization vector
            byte[]          iv   = null; // initialization vector as
                                         //  byte[]

            // Check which algorithm is used and define key size.
            if (algorithm=="Blowfish")
            {
                // valid values are: starting from 40bit up to
                // 448 in 8-bit increments
                strength=448;
            }
            else if (algorithm=="CAST5")
            {
                // valid values are: starting from 40bit up to 128bit
                using
                // 8bit steps.
                strength=128;
            }
            else if (algorithm=="DES")
            {
                strength=56;
            }
            else if (algorithm=="TripleDES"||algorithm=="DESede")
```

```
{
    // FIXME: is that correct ? Waiting for cryptix'
    // suppport
    // if 3DES.
    strength=3*64;
}
else if (algorithm=="Rijndael")
{
    strength=256; //valid values are: 128, 192, 256
}
else if (algorithm=="SKIPJACK")
{
    // fixed size: 80 bits
    strength=80;
}
else if (algorithm=="Square")
{
    strength=128;
}
else
{
    throw new RuntimeException();
}

System.out.println("Using keylength: "+strength+" bits.");
System.out.println("Blocksize: "+cipher.getBlockSize()*8+
" bits.");
System.out.println();

// init key generator with the key size and some random data.
// Use SecureRandom only if use trust it. It is not a really
// verified PRNG, yet.
kg.init(strength, new SecureRandom());

// create a secret key from the keygenerator.
SecretKey key = kg.generateKey();

// init cipher for encryption

// ECB does not need an initialization vector others do.
if (mode=="ECB")
{
    cipher.init(Cipher.ENCRYPT_MODE, key);
}
else
{
// These modes need an iv with a valid block size in
```

```
        // order to be used.
        SecureRandom sr = new SecureRandom();
        // allocate memory for iv.
        iv = new byte[ cipher.getBlockSize()];
        // Get next bytes from the PRNG.
        sr.nextBytes(iv);
        // create the IV class.
        spec = new IvParameterSpec(iv);
    }

    if (filename!=null)
    {
        FileDEncryption fe = new FileDEncryption(filename, key,
        iv, algorithm,mode,padding,provider);

        System.out.println("*** BEGIN file encryption! ***");
        System.out.println();

        // encrypt a file
        fe.go();
        System.out.println("*** END file encryption! ***");
        System.out.println();


        System.out.println("*** BEGIN file decryption! ***");
        System.out.println();

        // decrypt a file
        fe.reTurn();
        System.out.println("*** END file decryption! ***");
        System.out.println();
    }

    // Build a few bytes for encryption.
    byte[ ] text1 = ("text for encryption. You will not recognize
    it " +
                    "after encryption.").getBytes();
    byte[ ] text2 = ("!holdrio! more bytes for encryption. " +
                    "aaaaaaaaaaaaaaaaaaEND").getBytes();

    // Usage of [ de|en] cryption for a few bytes.
    try
    {
        // use encryption mode using specified key and IV.
        cipher.init(Cipher.ENCRYPT_MODE, key, spec);
    }
```

```
                catch(InvalidAlgorithmParameterException iape)
                {
                    System.out.println(
                       "cipher.init: InvalidAlgorithmParameterException.");
                    iape.printStackTrace();
                }

                // Ask the cipher how big the output will be.
                // Depending on the padding there will be more
                // output bytes than the sum of input bytes.
                int outLength = cipher.getOutputSize(text1.length+
                text2.length);
                System.out.println("Output bytes: "+outLength);
                System.out.println();

                byte [] encr1 = cipher.update(text1);

                byte [] encr2 = cipher.doFinal(text2);

                System.out.println("cipher: "+new String(encr1)+new
                String(encr2));
                System.out.println();

                /*
                BASE64Encoder encoder=new BASE64Encoder();
                String result=encoder.encode(encr1+encr2);
                System.out.println("BASE64: "+result+"\n");
                */

                try
                {
                    // decryption
                    Cipher decipher = Cipher.getInstance(

algorithm+"/"+mode+"/"+padding,provider);
                    decipher.init(Cipher.DECRYPT_MODE, key, spec);
                    byte[] deciph1 = decipher.update(encr1);

                    byte[] deciph2 = decipher.doFinal(encr2);
                    System.out.println(algorithm+"/"+mode+"/"+padding+
                    " decrypted: "+new String(deciph1)+new String
                    (deciph2)+  " "+"
                    "+decipher.getOutputSize(encr1.length+encr2.length));

                }
                catch(InvalidAlgorithmParameterException iape)
                {
```

```
                    System.out.println(
                       "cipher.init: InvalidAlgorithmParameterException.");
                    iape.printStackTrace();
                }
            }
            catch (NoSuchAlgorithmException nsae)
            {
                System.out.println("No such algorithm!\n");
                nsae.printStackTrace();
            }
            catch (NoSuchPaddingException nspe)
            {
                System.out.println("No such padding!\n");
                nspe.printStackTrace();
            }
            catch (NoSuchProviderException nspre)
            {
                System.out.println("No such provider found!\n");
                nspre.printStackTrace();
            }
            catch (InvalidKeyException ike)
            {
                System.out.println("Invalidkey Exception!\n");
                ike.printStackTrace();
            }
            /*
            catch (ShortBufferException sbe)
            {
                System.out.println("Short buffer exception!\n");
                sbe.printStackTrace();
            }
            */
            catch (IllegalBlockSizeException ibse)
            {
                System.out.println("Illegal block size exception!\n");
                ibse.printStackTrace();
            }
            catch (BadPaddingException bpe){
                System.out.println("Bad padding exception!\n");
                bpe.printStackTrace();
            }
        }
    }
```

Listado 3-1. Clase de ejemplo tomada desde la distribución de las librerías Cryptix para el cifrado simétrico.

Codificación de encriptación simétrica.

Ejemplificaremos en esta sección los pasos necesarios del proceso de generación de una llave, la creación e inicialización de un objeto `Cipher` o cifrador, la realización de una encriptación y de una desencriptación, mediante las implementaciones de los algoritmos de cifrado simétrico disponibles en los *providers* JCE `SunJCE` (preinstalado y registrado en la JDK a partir de la versión 1.4) y `BC` (del proyecto Bouncy Castle).

Se recomienda la lectura de los primeros apartados del capítulo, ya que los conceptos de la arquitectura JCA (*Java Cryptography Architecture*) y de la extensión JCE (*Java Cryptography Extension*) serán necesarios para comprender aspectos clave de la forma de utilizar estas clases.

Refiriendo lo visto en el apartado introductorio de la JCA, veamos de cuáles clases, *engines* o motores, nos valdremos para la codificación de funcionalidades de criptografía simétrica:

- `SecureRandom`: Usaremos esta clase para generar números aleatorios o seudoaleatorios.
- `Cipher`: Clase que, luego de ser inicializada con la/s llave/s correspondiente/s, se utilizará para la encriptación y desencriptación de información. Se dispone de diferentes tipos de algoritmos: Simétricos de tipo *bulk* (por ejemplo, AES, DES, DESede, Blowfish, IDEA), de tipo *stream* (por ejemplo, RC4), asimétricos (por ejemplo, RSA) y de *password-based encryption* (PBE).
- `KeyFactory`: Clase usada para convertir llaves criptográficas de tipo `Key` en especificaciones de llave (representaciones transparentes del material de llave) y viceversa.
- `SecretKeyFactory`: Utilizada para convertir llaves criptográficas opacas de tipo `SecretKey` en especificaciones de llave (representaciones transparentes del material de llave) y viceversa. Las `SecretKeyFactorys` son `KeyFactorys` especializadas, que crean llaves secretas (simétricas) únicamente.
- `KeyGenerator`: La utilizaremos para generar llaves secretas para su uso junto a un algoritmo especificado.

Como primer ejemplo veremos cómo implementaríamos el proceso de cifrado o encriptación, haciendo uso del algoritmo *Advanced Encryption Standard* o AES.

El primer procedimiento será la generación de la llave. Para crear una llave AES necesitaremos generar una instancia de un generador de llaves para este algoritmo en particular. No se especificará un proveedor específico porque no tenemos la necesidad de utilizar alguna implementación en especial. No lo inicializaremos con parámetros fuera de los tomados por defecto, lo que hará que la implementación provea una fuente aleatoria y una longitud de llave para la generación de la llave. El código es el siguiente:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecretKey aesKey = keygen.generateKey();
```

Crearemos ahora una instancia del objeto principal en los procesos de encriptación y desencriptación: El cifrador. Para tal fin, usaremos uno de los métodos *factory* de la clase `Cipher`. Estos métodos se parametrizan mediante el nombre del algoritmo, el modo opcional y el modo de *padding* (también opcional).

Crearemos en este ejemplo un cifrador del algoritmo AES en modo de cifrado *Electronic Codebook* o ECB, con *padding* de tipo PKCS5. Como con la generación de llaves, no especificaremos un proveedor en particular.

El nombre estándar para el algoritmo AES es "AES", para el modo de encriptación *Electronic Codebook* es "ECB" y, por último, el tipo de *padding* PKCS5 tiene como nombre estándar "PKCS5Padding". Veamos el código que instancia el objeto cifrador `Cipher` y lo inicializa para encriptar, con la llave obtenida con el código anterior:

```
Cipher aesCipher;
// Crear la instancia del cifrador
aesCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

// Inicializar el cifrador para encriptar
aesCipher.init(Cipher.ENCRYPT_MODE, aesKey);
```

Y ahora veamos cómo se realizaría la encriptación:

```
// El texto de ejemplo a encriptar
byte[] cleartext = "Contenido de prueba".getBytes();

// Encriptar el texto
byte[] ciphertext = aesCipher.doFinal(cleartext);
```

Por último, veamos cómo codificaríamos lo necesario para descifrar lo anterior:

```
// Inicializar el mismo cifrador, pero
// ahora para desencriptar
aesCipher.init(Cipher.DECRYPT_MODE, aesKey);

// Desencriptar el texto cifrado
byte[] cleartext_out = aesCipher.doFinal(ciphertext);
```

Se entiende, por supuesto, que luego de estos procedimientos, los valores de las variables `cleartext` y `cleartext_out` serán idénticos.

Por supuesto, este se trata de un ejemplo elemental, en el cual utilizamos la misma instancia de la llave secreta (`aesKey`), obtenida a partir de un `KeyGenerator` para el algoritmo AES. Esto significa que la llave se generará aleatoriamente y su longitud será la correspondiente al algoritmo (no es el caso de AES, pero si el algoritmo soportara diferentes longitudes de llave, se utilizará aquella definida por defecto en la implementación del *provider*).

En un caso más real, el proceso de desencriptación no tendría acceso a la instancia del objeto de la llave, por lo que tendríamos que recuperarla, por ejemplo, desde un archivo, utilizando `SecretKeySpec`. Otra alternativa es servirse de las implementaciones de los algoritmos de *Password-Based Encryption* (PBE) del *provider* instalado o en uso, para obtener la llave a partir de una contraseña o una *passphrase* (una frase-clave en lugar de una sola palabra-clave o *password*).

Veremos más adelante en este capítulo, el código de ejemplo para la primera alternativa comentada, es decir, para recuperar la llave con la cual alimentaremos al cifrador. Lo que sigue es un ejemplo de cómo se generaría una llave a partir de una contraseña o *passphrase* (segunda alternativa comentada):

```
PBEKeySpec pbeKeySpec;
PBEParameterSpec pbeParamSpec;
SecretKeyFactory keyFac;

// Definición del salt (es posible, y se recomienda,
// hacerlo aleatoriamente)
byte[] salt = {
(byte)0xc7, (byte)0x73, (byte)0x21, (byte)0x8c,
(byte)0x7e, (byte)0xc8, (byte)0xee, (byte)0x99
};

// Iteration count: cantidad de veces que se 'hasheará' la
// contraseña/passphrase
int count = 20;

// Instanciar el objeto de parámetros PBE
pbeParamSpec = new PBEParameterSpec(salt, count);

// Para obtener nuestra instancia de SecretKey...
pbeKeySpec = new PBEKeySpec("contraseña".toCharArray(););
keyFac = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey pbeKey = keyFac.generateSecret(pbeKeySpec);

// Instanciar el cifrador PBE
Cipher pbeCipher = Cipher.getInstance("PBEWithMD5AndDES");

// Inicializar el cifrador PBE con llave y parámetros
```

```
pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);

// Nuestro texto-plano
byte[] texto_plano = "Contenido de prueba".getBytes();

// Encriptación...
byte[] cifrado = pbeCipher.doFinal(texto_plano);
```

En los ejemplos se utilizan o registran las *passphrases* y/o contraseñas en instancias del objeto `String`, pero se recomienda utilizar *arrays* de bytes (como lo hacen todas las interfaces de la JCE) o instancias de `StringBuffer`, que pueden ser "limpiadas" de memoria. Esto como medida de seguridad contra la posible exposición de la información secreta en la memoria del sistema (no se asegura cuándo actuará el *garbage collector* para los `Strings`).

Dependiendo del *provider* utilizado, disponemos de diferentes alternativas de algoritmos de PBE. Listamos algunos a manera de ejemplo (téngase presente que el primer algoritmo en el nombre corresponde a la función de *hashing,* que será utilizada para generar la llave a partir de la contraseña o *passphrase*, y el segundo será el utilizado para el proceso de cifrado, en algunos casos también el modo):

- PBEWithMD5AndDES
- PBEWithSHAAndBlowfish
- PBEWithSHAAnd128BitRC4
- PBEWithSHAAndIDEA-CBC
- PBEWithSHAAnd3-KeyTripleDES-CBC

De acuerdo con el *provider* utilizado, recomendamos usar alguna otra alternativa a `PBEWithMD5AndDES`, que dispondrá de una mayor longitud de llave. Con respecto a la generación de números aleatorios para el *salt*, es conveniente utilizar, también dependiendo del *provider*, una implementación de `SecureRandom`. Por ejemplo, para obtener 8 bytes aleatorios, codificaríamos:

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[8];
random.nextBytes(bytes);
```

A continuación, basándonos en el código de ejemplo que utiliza el algoritmo Blowfish provisto en la documentación de la JCE de Sun Microsystems, traduciremos y comentaremos detalladamente los pasos involucrados en el proceso de encriptación para luego referir el programa completo.

Asimismo, se recomienda al lector referirse al apartado "Almacenamiento de información cifrada", dentro de la sección dedicada a la codificación de casos prácticos. Allí encontrará un ejemplo, comentado paso a paso, de la utilización de estas herramientas para la generación de una llave secreta, la encriptación y la desencriptación de archivos utilizando el algoritmo Blowfish.

El primer paso será entonces obtener la llave –secreta– que se ha de utilizar para el algoritmo de cifrado simétrico Blowfish. Esta llave se obtiene a partir de un generador de llaves; el generador se instancia de la siguiente forma:

```
KeyGenerator genLlave =
 KeyGenerator.getInstance("Blowfish");
```

Ahora, a partir de la instancia del generador de llaves, obtenemos una llave aleatoria:

```
SecretKey llaveSecreta = genLlave.generateKey();
```

Recuperamos el *array* de bytes de la llave generada, que nos será necesario a la hora de parametrizar el cifrador o instancia del objeto `Cipher`, de esta forma:

```
byte[] raw = llaveSecreta.getEncoded();
```

Continuando con lo anterior, instanciamos un `SecretKeySpec` con la información de la llave:

```
SecretKeySpec llaveSecretaSpec =
 new SecretKeySpec(raw, "Blowfish");
```

Ahora, utilizamos la clase `Cipher` o cifrador con el algoritmo que hemos seleccionado para implementar, en este caso, Blowfish:

```
Cipher cipher = Cipher.getInstance("Blowfish");
```

Inicializamos nuestra instancia del cifrador para cifrar o encriptar utilizando la llave generada, encapsulada en el objeto parámetro:

```
cipher.init(Cipher.ENCRYPT_MODE, llaveSecretaSpec);
```

Se realiza la encriptación. Aquí obtenemos el resultado cifrado, en el *array* de bytes, utilizando el método que encriptará en un sólo paso la cadena de caracteres de prueba que tomamos para el ejemplo:

```
byte[] infoCifrada =
 cipher.doFinal("Contenido de prueba".getBytes());
```

Se refiere al lector al ejemplo que utiliza el algoritmo Blowfish de la documentación de la JCE de Sun Microsystems.

Por último, incluiremos el código completo de una clase para ejemplificar la encriptación y desencriptación simétrica utilizando *padding.* Nos mostrará cómo puede variar la longitud de la salida del cifrador, es decir, del *ciphertext* o texto-cifrado, si la longitud de la entrada es una cantidad de bytes no múltiplo del *block-size* o tamaño de bloque del algoritmo utilizado. Usaremos en este ejemplo al *provider* de Bouncy Castle específicamente (ver parámetro `"BC"` en la generación de la instancia del objeto cifrador).

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;


public class EjemploLongitudSalida
{
    public static void main(String[] args) throws Exception
    {

        // obtenemos un llave aleatoria

        KeyGenerator keygen = KeyGenerator.getInstance("AES");

        SecretKey sKey = keygen.generateKey();

        SecretKeySpec key = new SecretKeySpec(sKey.getEncoded(),"AES");

        // creamos la instancia del cifrador

        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");

        // defininimos nuestra entrada (23 bytes).

        byte[] entrada = "Contenido de prueba".getBytes();

        // imprimimos cuánto mide la entrada

        System.out.println("Longitud de entrada: "
        + entrada.length + " bytes.");

        // encriptamos nuestra entrada teniendo en cuenta
        // lo retornado por update() y doFinal(), es decir
        // la cantidad de bytes procesados. Se utiliza el
        // método cipher.getOutputSize() para determinar
        // cuál será el tamaño del array en el cual se
```

```
            // registrará la información cifrada.

            cipher.init(Cipher.ENCRYPT_MODE, key);

            byte[ ] cifrado = new byte[ cipher.getOutputSize(entrada.length)];

            int cifrado_len = cipher.update(entrada, 0, entrada.length,
            cifrado, 0);

            cifrado_len += cipher.doFinal(cifrado_len, cifrado_len);

            // imprimimos cuánto mide la salida

            System.out.println("Longitud de texto-cifrado: "
            + cifrado_len + " bytes.");

            // ahora desencriptamos lo anterior

            cipher.init(Cipher.DECRYPT_MODE, key);

            byte[ ]  texto_plano = new byte[ cipher.getOutputSize(cifrado_len)];

            int texto_plano_len = cipher.update(cifrado, 0, cifrado_len,
             texto_plano, 0);

            texto_plano_len += cipher.doFinal(texto_plano, texto_plano_len);

            // imprimimos cuánto mide el texto-plano descifrado

            System.out.println("Longitud de texto-plano: " +
             texto_plano_len + " bytes.");

        }
    }
```

Listado 3-2. Ejemplo que evidencia cómo puede variar la longitud de la salida de un cifrador con una entrada de tamaño no múltiplo del *block-size* del algoritmo por el *padding*.

Codificación de encriptación asimétrica.

Veremos en esta sección diferentes alternativas para codificar o implementar criptografía asimétrica, utilizando el modelo de la JCA/JCE para los *providers* `SunJCE` (por defecto a partir de la JDK 1.4) y `BC` (del proyecto Bouncy Castle).

Se recomienda al lector que, antes de adentrarse en esta sección, lea la anterior si no lo ha hecho y, también, los dos primeros apartados del capítulo. Si bien aquí se

desarrollarán explicaciones y ejemplos acerca de algunas funcionalidades (clases) no tratadas aún, la clase más importante, `Cipher`, se utilizará con algoritmos de criptografía asimétrica, pero de igual manera que en los casos anteriores.

Repasemos cómo hemos de usar las clases *engine* o motor, que aplican a lo que encriptación asimétrica se refiere:

- `SecureRandom`: La usaremos para generar números aleatorios o seudoaleatorios.
- `Signature`: Luego de inicializada con las llaves correspondientes, se usará para firmar digitalmente cierta información (y para luego verificarla).
- `Cipher`: También, luego de ser inicializada con la/s llave/s correspondiente/s, se utilizará para la encriptación y desencriptación de información. Se dispone de diferentes tipos de algoritmos: Simétricos de tipo *bulk* (por ejemplo, AES, DES, DESede, Blowfish, IDEA), de tipo *stream* (por ejemplo, RC4), asimétricos (por ejemplo, RSA) y de *password-based encryption* (PBE).
- `KeyPairGenerator`: Generaremos con esta clase un par de llaves (pública y privada) aptas para su uso con un algoritmo especificado.
- `KeyAgreement`: Utilizada por dos o más partes para acordar y establecer una llave específica, que será usada por una operación criptográfica particular.

Veamos, por ejemplo, cómo obtenemos un par de llaves (pública y privada), a través de una instancia del generador `KeyPairGenerator` para el algoritmo DSA (*Digital Signature Algorithm*). Codificamos:

```
KeyPairGenerator keyGen =
 KeyPairGenerator.getInstance("DSA");
```

Todos los generadores de pares de llaves comparten los conceptos de longitud de llave y fuente de números aleatorios. Los métodos de inicialización de la clase `KeyPairGenerator` requieren, como mínimo, la longitud de la llave. Si la fuente de números aleatorios no es especificada, una implementación de `SecureRandom` (del *provider* con mayor prioridad) es utilizada. Entonces, siguiendo con nuestro ejemplo, para generar llaves de 1 024 bits de longitud:

```
SecureRandom random =
 SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
```

Por último, obtenemos o generamos el par de llaves. Lo codificamos de la siguiente manera:

```
KeyPair pair = keyGen.generateKeyPair();
```

Aprovecharemos ahora para, utilizando el par de llaves obtenido en el ejemplo anterior, describir cómo firmar y luego verificar información.

Primero obtenemos una instancia del objeto `Signature`, de esta forma:

```
Signature dsa = Signature.getInstance("SHA1withDSA");
```

Ahora, usando el par de llaves generado anteriormente, inicializamos nuestra instancia `dsa` con la llave privada:

```
PrivateKey priv = pair.getPrivate();
dsa.initSign(priv);
```

Firmamos la información contenida en el *array* de bytes `data`:

```
dsa.update(data);
byte[] sig = dsa.sign();
```

Veamos ahora cómo verificamos la firma obtenida; obtenemos la llave privada e inicializamos –para verificar– nuestra instancia `dsa`:

```
PublicKey pub = pair.getPublic();
dsa.initVerify(pub);
```

Finalmente, verificamos concretamente la firma, guardando en nuestra variable booleana `verifica` el resultado de la operación:

```
dsa.update(data);
boolean verifica = dsa.verify(sig);
```

Como se ha adelantado más arriba, a la hora de encriptar y desencriptar información, también será la clase `Cipher` la que se utilizará para hacerlo con algoritmos de criptografía asimétrica. Conociendo ya esta clase, y habiendo visto en los últimos ejemplos cómo manejar clases como `KeyPairGenerator`, `KeyPair`, `PublicKey` y `PrivateKey`, estamos en condiciones de abordar códigos fuente que implementan cifrados de llave pública o criptografía asimétrica.

Veamos un ejemplo completo basado en una clase del libro *Beginning Cryptography with Java*[1] para la encriptación y desencriptación, utilizando el algoritmo RSA y a través del *provider* del proyecto Bouncy Castle.

```
import java.math.BigInteger;
import java.security.KeyFactory;
```

```
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;

import javax.crypto.Cipher;


public class EjemploRSA
{
    public static void main(String[] args) throws Exception
    {
        byte[]         input = new byte[] {  (byte)0xbe, (byte)0xef };
        Cipher        cipher =
                       Cipher.getInstance("RSA/None/NoPadding", "BC");
        KeyFactory     keyFactory = KeyFactory.getInstance("RSA", "BC");

        // creamos las llaves

        RSAPublicKeySpec pubKeySpec = new RSAPublicKeySpec(
               new BigInteger("d46f473a2d746537de2056ae3092c451", 16),
               new BigInteger("11", 16));

        RSAPrivateKeySpec privKeySpec = new RSAPrivateKeySpec(
               new BigInteger("d46f473a2d746537de2056ae3092c451", 16),
               new BigInteger("57791d5430d593164082036ad8b29fb1", 16));

        RSAPublicKey pubKey =
        (RSAPublicKey)keyFactory.generatePublic(pubKeySpec);

        RSAPrivateKey privKey =
        (RSAPrivateKey)keyFactory.generatePrivate(privKeySpec);

        // proceso de encriptación

        cipher.init(Cipher.ENCRYPT_MODE, pubKey);

        byte[] cipherText = cipher.doFinal(input);

        // proceso de desencriptación

        cipher.init(Cipher.DECRYPT_MODE, privKey);

        byte[] plainText = cipher.doFinal(cipherText);

    }
}
```

Listado 3-3. Clase de ejemplo de encriptación y desencriptación utilizando el algoritmo RSA del provider `BC`.

Como alternativa para la generación de llaves de manera aleatoria, veamos que para esta etapa del proceso, salvando la distancia de obtener dos llaves, en el caso de cifrar asimétricamente no existen mayores diferencias respecto del mismo proceso para el caso simétrico:

```
// Obtenemos el generador de llaves para el algoritmo
// especificado
KeyPairGenerator generator =
 KeyPairGenerator.getInstance("RSA", "BC");

// Inicializamos el generador (el parámetro determina la
// longitud de llaves deseada)
generator.initialize(256);

// Obtenemos el par de llaves
KeyPair parLlaves = generator.generateKeyPair();

// Para recuperar la llave pública
Key llavePublica = parLlaves.getPublic();

// Para la recuperación de la llave privada
Key llavePrivada = parLlaves.getPrivate();
```

Veamos otro ejemplo completo con referencia a lo anterior:

```
import java.security.*;
import javax.crypto.*;

public class EjemploRSA2 {

          public static void main (String[] args) throws Exception {

          // Definir entrada o texto-plano

          byte[] texto_plano = "Contenido de prueba".getBytes();

          // Generar llaves RSA de 1024 bits de longitud

          KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");

          keyGen.initialize(1024);

          KeyPair key = keyGen.generateKeyPair();
```

```
            // Obtener instancia del cifrador

            Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");

            // Inicializar nuestra instancia del cifrador

            cipher.init(Cipher.ENCRYPT_MODE, key.getPublic());

            // Encriptar nuestro texto-plano

            byte[] cifrado = cipher.doFinal(texto_plano);

            // Imprimir la salida como prueba

            System.out.println( new String(cifrado) );

            // Inicializar para desencriptación

            cipher.init(Cipher.DECRYPT_MODE, key.getPrivate());

            // Desencriptar
            byte[] nuevo_texto_plano = cipher.doFinal(cifrado);

            // Imprimir la salida como prueba

            System.out.println( new String(nuevo_texto_plano) );

        }

    }
```

Listado 3-4. Segundo ejemplo para encriptación y desencriptación utilizando el algoritmo RSA; llaves generadas aleatoriamente.

Se refiere al lector al ejemplo o prueba de concepto, utilizado como referencia, que implementa el algoritmo Diffie Hellman para el intercambio de llaves entre dos partes de la sección de programas de ejemplo, dentro de la documentación de la JCE provista por Sun Microsystems.

Codificación de funciones de una vía y *hash.*

Abordaremos en esta sección el cómo hacer uso de las implementaciones de funciones de una vía o *hashing*, utilizando el modelo de la JCA/JCE, para los *providers*

`SunJCE` (preinstalado y registrado a partir de la JDK 1.4) y `BC` (del proyecto de *The Legion of The Bouncy Castle*).

Como en las secciones referidas a la codificación de criptografía simétrica y asimétrica, se recomienda al lector que, antes de continuar con esta sección, lea los dos primeros apartados del capítulo (respecto de la JCA y la JCE) si no lo ha hecho aún.

Repasando entonces las clases *engine* o motor, que trataremos en esta sección particularmente, recordaremos:

- `MessageDigest`: Utilizada para generar o calcular el *message digest* (*hash*) de una información específica.
- *Message Authentication Codes* (`MAC`): Como en el caso de `MessageDigest`, también se usan para generar *hashes*, pero son inicializadas con las llaves correspondientes *a priori* para proteger la integridad de los mensajes.

Hechas las aclaraciones preliminares, se desarrollará ahora cómo utilizar estas clases valiéndonos de algunos ejemplos concretos, que usarán algoritmos de *hashing* como SHA-1 o MD5.

En primer término, generamos una instancia de un objeto de la clase `MessageDigest` (resumen de mensaje o *hashing* que, como se ha establecido anteriormente, los concebimos como sinónimos):

```
MessageDigest sha = MessageDigest.getInstance("SHA-1");
```

Con esto se asigna un objeto de `MessageDigest` correspondientemente inicializado, a la variable `sha`. La implementación soporta el algoritmo *Secure Hash Algorithm* (SHA-1), como es definido por la NIST (*National Institute for Standards and Technology*) en el documento FIPS 180-2.

Supongamos ahora que tenemos tres *arrays* de bytes, i1, i2 e i3, que en conjunto forman la entrada o el mensaje a partir del cual queremos obtener su *hashing* o resumen de mensaje. Esto podría lograrse a partir de lo siguiente:

```
sha.update(i1);
sha.update(i2);
sha.update(i3);
byte[] hash = sha.digest();
```

Otra alternativa equivalente podría ser:

```
sha.update(i1);
sha.update(i2);
byte[] hash = sha.digest(i3);
```

Una vez que el *hash* ha sido calculado, la instancia del objeto `MessageDigest` es reinicializada automáticamente, quedando lista para recibir una nueva entrada y calcular nuevamente el *hashing*. La información suministrada anteriormente (parámetros del método `update`) se pierde.

Veamos ahora el mismo uso de la clase `MessageDigest` pero dentro de un programa que imprimirá el resultado de la aplicación de diferentes algoritmos a nuestra cadena de caracteres de prueba. Se usará para este ejemplo el *provider* `BC` (de *Bouncy Castle*).

```
import java.security.*;
import javax.crypto.*;

public class EjemploHashes {

   public static void main (String[] args) throws Exception {

          // definimos la entrada o texto-plano

          byte[] texto_plano = "Contenido de prueba".getBytes();

          // generamos el hash con el algoritmo MD2 y lo imprimimos

           MessageDigest mdMD2 = MessageDigest.getInstance("MD2", "BC");

          mdMD2.update(texto_plano);

          System.out.println( "MD2: " +
          hexStringFromBytes(mdMD2.digest()));

          // ahora con el algoritmo MD4

           MessageDigest mdMD4 = MessageDigest.getInstance("MD4", "BC");

          mdMD4.update(texto_plano);

          System.out.println( "MD4: " +
          hexStringFromBytes(mdMD4.digest()));

          // algoritmo MD5

           MessageDigest mdMD5 = MessageDigest.getInstance("MD5", "BC");

          mdMD5.update(texto_plano);

          System.out.println( "MD5: " +
          hexStringFromBytes(mdMD5.digest()));
```

```
// algoritmo SHA-1

MessageDigest mdSHA1 = MessageDigest.getInstance("SHA-1", "BC");

mdSHA1.update(texto_plano);

System.out.println( "SHA-1: " +
hexStringFromBytes(mdSHA1.digest()));

// algoritmo SHA-256

MessageDigest mdSHA256 = MessageDigest.getInstance
("SHA-256","BC");

mdSHA256.update(texto_plano);

System.out.println( "SHA-256: " +
// hexStringFromBytes(mdSHA256.digest()));

// algoritmo SHA-384

 MessageDigest mdSHA384 = MessageDigest.getInstance("SHA-384",
"BC");

mdSHA384.update(texto_plano);

System.out.println( "SHA-384: " +
hexStringFromBytes(mdSHA384.digest()));

// algoritmo SHA-512

 MessageDigest mdSHA512 = MessageDigest.getInstance("SHA-512",
"BC");

mdSHA512.update(texto_plano);

System.out.println( "SHA-512: " +
hexStringFromBytes(mdSHA512.digest()));

// algoritmo Tiger

MessageDigest mdTiger = MessageDigest.getInstance("Tiger",
"BC");

mdTiger.update(texto_plano);

System.out.println( "Tiger: " +
hexStringFromBytes(mdTiger.digest()));
```

```
    }


    public String hexStringFromBytes(byte[] b)
    {
           static final char[] hexChars = {'0','1','2','3','4',
           '5','6','7','8','9','A','B','C','D','E','F'};

           String hex = "";

           int msb = 0;
           int lsb = 0;
           int i;

           for (i = 0; i < b.length; i++)
           {
                  msb = ((int)b[i] & 0x000000FF) / 16;
                  lsb = ((int)b[i] & 0x000000FF) % 16;
                  hex = hex + hexChars[msb] + hexChars[lsb];
           }
           return(hex);
    }

}
```

Listado 3-5. Generación de *hashes* utilizando diferentes algoritmos del *provider* de *Bouncy Castle*.

Recordemos ahora que un código de autentificación de mensaje provee una manera de verificar la integridad de la información transmitida sobre un canal inseguro, pero que incluye el uso de una llave privada en el proceso. Sólo quien disponga de la llave privada podrá verificar el mensaje.

La generación de códigos de autentificación de mensajes (MAC, de *Message Authentication Co*de) está estrechamente relacionada con los algoritmos de *hashing*; sin embargo, la implementación JCE propone una serie de funciones alternativas a las utilizadas para el *hashing* únicamente, que encapsulan lo particular de esta forma de uso. A continuación, se muestra como trabaja con HMAC MD5.

```
KeyGenerator kg = KeyGenerator.getInstance("HmacMD5");
SecretKey sk = kg.generateKey();

Mac mac = Mac.getInstance("HmacMD5");
mac.init(sk);
byte[]resultado = mac.doFinal("Contenido de prueba".getBytes());
```

Codificaciones de casos prácticos.

Almacenamiento de información cifrada.

Para la primera resolución práctica de un problema hemos elegido, a manera introductoria, la más típica de las necesidades que podemos satisfacer mediante el uso de la criptografía. Imaginemos que nuestro sistema requiera almacenar información en una fuente no confiable (ejemplos de estas fuentes pueden ser: Archivos dentro de un directorio NFS en sistemas UNIX, archivos de un recurso compartido en entornos Windows o un campo en una base de datos compartida con otras aplicaciones o sistemas). Daremos por hecho que la ejecución del programa es segura. Esto quiere decir que se confía en que nuestra aplicación, al ejecutarse, comprobará las credenciales de cada usuario y permitirá el acceso a la información (para descifrar por el mismo programa) correspondientemente.

Se trata entonces de un problema o necesidad idóneo para la aplicación de criptografía simétrica. Debemos cifrar cierta información que ha de circular –o almacenar en nuestro caso– en canales –o fuentes– no confiables, y que sólo quienes conozcan la llave privada puedan descifrarlo. Respecto de nuestro problema particular, será nuestra aplicación la que cifrará y descifrará la información, ésta conocerá la clave privada que utilizará para ambas operaciones. Como se ha afirmado, se asume que la ejecución del programa dispondrá de las validaciones necesarias para no permitir la ejecución a usuarios que no deban acceder a, o descifrar, la información resguardada.

Lo implementaremos utilizando el algoritmo Blowfish de Bruce Schneier, cuya ejecución se encuentra disponible en el *provider* JCE `SunJCE` (incorporada a la JCA, es decir, como parte de la distribución estándar), a partir de la Java 2 SDK versión 1.4.

Antes de adentrarnos en el código de ejemplo del caso tratado, repasaremos los conceptos básicos de la JCA/JCE para la utilización de algoritmos de cifrado asimétrico.

1. Se trata de un algoritmo que utiliza llaves privadas. Necesitamos una para cifrar la información que se ha de almacenar. Utilizaremos un generador de llaves para el algoritmo Blowfish, de esta forma:

```
KeyGenerator genLlave =
 KeyGenerator.getInstance("Blowfish");
```

2. Obtenemos una llave aleatoria a partir de la instancia del generador de llaves:

```
SecretKey llaveSecreta = genLlave.generateKey();
```

3. Nuestro programa deberá guardar, en una fuente de almacenamiento que se

considere segura, esta llave para las posteriores operaciones de cifrado y descifrado. Obtenemos el *array* de bytes de la siguiente manera:

```
byte[] raw = llaveSecreta.getEncoded();
```

4. Nos será necesaria una instancia de `SecretKeySpec` para utilizar como parámetro a la hora de inicializar nuestra instancia de `Cipher`; básicamente, encapsula la información de la llave:

```
SecretKeySpec llaveSecretaSpec = new SecretKeySpec(raw,
 "Blowfish");
```

5. Creamos la instancia de `Cipher`, o cifrador, utilizando el algoritmo seleccionado:

```
Cipher cipher = Cipher.getInstance("Blowfish");
```

6. Inicializamos nuestra instancia del cifrador, en este caso, para cifrar o encriptar utilizando nuestra llave (en el modo y *padding* default):

```
cipher.init(Cipher.ENCRYPT_MODE, llaveSecretaSpec);
```

7. Por último, se obtiene el *array* de bytes con el *cipher-text* o texto cifrado, utilizando el método que encriptará, en un paso, la información de ejemplo:

```
byte[] infoCifrada =
 cipher.doFinal("Contenido de prueba".getBytes());
```

Ya destacados los aspectos más relevantes acerca de la utilización de la JCA/JCE para el cifrado simétrico utilizando Blowfish, a continuación se encuentran los programas completos para: 1. Restablecer el archivo de llave; 2. Cifrar un archivo de cualquier formato utilizando tal llave; 3. Descifrar lo anterior.

```
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.util.*;

public class GeracionLlaveBF {
```

```
  // información de configuración general
  final static String archivoLlave   = "llave.bin";


  /**
   * Método para generar la llave.
   */
  public bool generarLlave() {

bool resultado = flase;

      KeyGenerator genLlave = KeyGenerator.getInstance("Blowfish");

      try {

      // generar llave para Blowfish
      KeyGenerator genLlave = KeyGenerator.getInstance("Blowfish");

      // obtener instancia de SecretKet
      SecretKey llaveSecreta = genLlave.generateKey();

      // array binario de llave
      byte[ ]  raw = llaveSecreta.getEncoded();

      // guardar información binaria de llave en archivo
      File file = New File(archivoLlave);
      FileOutputStream file_output = new FileOutputStream (file);
      DataOutputStream data_out = new
      DataOutputStream (file_output);
      for (int i=0; i < raw.length; i++) {
           data_out.writeByte(raw[ i] );
      }
      file_output.close ();

      return true;

      // excepción general
      } catch (Exception e) {

      e.printStackTrace();

      }

      return resultado;

      }
```

```
         /**
         * Punto de entrada al programa (función main).
         */
         public static void main(String[] args)
            throws java.lang.InterruptedException {

        bool resultado = false;

        // instanciación del objeto
        GeracionLlaveBF gll = new GeracionLlaveBF();

        // generación de llave
        resultado = gll.generarLlave();

        // mensaje por consola
        if (resultado){
            System.out.println("Llave generada exitosamente");
        } else {
            System.out.println("Error al generar llave");
        }

    }

}
```

Listado 3-6. Generación de un archivo de llave para el algoritmo Blowfish.

```
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.util.*;

public class EncriptacionBF {

    // información de configuración general
    final static String archivoLlave   = "llave.bin";


    /**
     * Método para encriptar un archivo.
     */
    public bool encriptarArchivo(String archivo) {
```

```
bool resultado = flase;
      byte[ 56]  raw;
      byte[ 1024000]  tmp;
      int tmp_len;

      try {

          // leer archivo de llave
          File file_ll = New File(archivoLlave);
          FileInputStream file_input = new FileInputStream
          (file_ll); DataInputStream data_in  =
          new DataInputStream (file_ll_input );
          data_in.read(raw);
          data_in.close();

          // leer archivo a cifrar
          File file_t = New File(archivo);
          FileInputStream file_t_input =
          new FileInputStream (file_t);
          DataInputStream data_t_in    =
          new DataInputStream (file_t_input );
          tmp_len = data_t_in.read(tmp);
          data_t_in.close();

          // cifrar archivo
          SecretKeySpec llaveSecretaSpec =
          new SecretKeySpec(raw, "Blowfish");
          Cipher cipher = Cipher.getInstance("Blowfish");
          cipher.init(Cipher.ENCRYPT_MODE, llaveSecretaSpec);
          byte[ ]  infoCifrada =  cipher.doFinal(tmp);

          // guardar información binaria de archivo cifrado
          File file = New File(archivo + ".crypt");
          FileOutputStream file_output = new FileOutputStream (file);
          DataOutputStream data_out =
          new DataOutputStream (file_output);
          for (int i=0; i < infoCifrada.length; i++) {
              data_out.writeByte(infoCifrada[ i] );
          }
          file_output.close ();

          return true;

      // excepción general
      } catch (Exception e) {

          e.printStackTrace();
```

```
        }

        return resultado;

    }


    /**
     * Punto de entrada al programa (función main).
     */
    public static void main(String[] args)
            throws java.lang.InterruptedException {

  bool resultado = false;

        // instanciación del objeto
        EncriptacionBF ebf = new EncriptacionBF();

        // encriptación de archivo de prueba
        resultado = ebf.encriptarArchivo("texto.txt");

        // mensaje por consola
        if (resultado){
            System.out.println("Archivo encriptado exitosamente");
        } else {
            System.out.println("Error al encriptar archivo");
        }

    }

}
```

Listado 3-7. Encriptación de archivos utilizando el algoritmo Blowfish.

```
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.util.*;

public class DesencriptaciónBF {

    // información de configuración general
    final static String archivoLlave   = "llave.bin";
```

```
  /**
   * Método para encriptar un archivo.
   */
  public bool desencriptarArchivo(String archivo) {

bool resultado = flase;
      byte[ 56]  raw;
      byte[ 1024000]  tmp;
      int tmp_len;

      try {

          // leer archivo de llave
          File file_ll = New File(archivoLlave);
          FileInputStream file_input = new FileInputStream (file_ll);
          DataInputStream data_in    =
          new DataInputStream (file_ll_input );
          data_in.read(raw);
          data_in.close ();

          // leer archivo a descifrar
          File file_t = New File(archivo);
          FileInputStream file_t_input = new FileInputStream (file_t);
          DataInputStream data_t_in    =
          new DataInputStream (file_t_input);
          tmp_len = data_t_in.read(tmp);
          data_t_in.close();

          // descifrar archivo
          SecretKeySpec llaveSecretaSpec =
          new SecretKeySpec(raw, "Blowfish");
          Cipher cipher = Cipher.getInstance("Blowfish");
          cipher.init(Cipher.DECRYPT_MODE, llaveSecretaSpec);
          byte[ ]  infoDeCifrada =  cipher.doFinal(tmp);

          // guardar información de archivo descifrado
          File file = New File(archivo + ".decrypt");
          FileOutputStream file_output = new FileOutputStream (file);
          DataOutputStream data_out =
          new DataOutputStream (file_output);
          for (int i=0; i < infoDeCifrada.length; i++) {
              data_out.writeByte(infoDeCifrada[ i] );
          }
          file_output.close ();
```

```
                return true;

            // excepción general
            } catch (Exception e) {

                e.printStackTrace();

            }

            return resultado;

        }


        /**
         * Punto de entrada al programa (función main).
         */
        public static void main(String[] args)
                throws java.lang.InterruptedException {

      bool resultado = false;

            // instanciación del objeto
            DesencriptaciónBF dbf = new DesencriptaciónBF();

            // desencriptación de archivo de prueba
            resultado = dbf.desencriptarArchivo("texto.txt.crypt");

            // mensaje por consola
            if (resultado){
                System.out.println("Archivo desencriptado exitosamente");
            } else {
                System.out.println("Error al desencriptar archivo");
            }

        }

    }
```

Listado 3-8. Desencriptación de archivos utilizando el algoritmo Blowfish.

Registro de contraseñas cifradas en bases de datos.

Trataremos ahora el abordaje de otra problemática típica en los sistemas o aplicaciones multiusuario. Hablamos de una aplicación contable o financiera, un sistema de

control de producción o un sitio Web de comercio electrónico; todos ellos son utilizados por diferentes personas, afectando de diferentes maneras la información del sistema como un todo. La metodología más común para la implementación de estas verificaciones es a través de un *login* y *password,* o control de acceso por nombre de usuario y contraseña.

El inconveniente que esto plantea es respecto del almacenamiento de las claves de los usuarios. La herramienta criptográfica que nos ayudará son las funciones de una vía o *hashing*. El modelo de implementación se remonta, conceptualmente al menos, hasta los primeros sistemas UNIX, que guardaban cifradas (con una variación del algoritmo DES) las contraseñas de cada usuario. Esto posibilitaba el libre acceso al archivo que almacenaba las contraseñas. Cuando un usuario ingresaba al sistema, introducía su contraseña, ésta se cifraba y se comparaba con la versión del archivo para ese usuario. Si coincidían, se permitía el acceso. Es importante notar cómo se utiliza aquí la variación del algoritmo DES desde la función del sistema UNIX "`crypt(3)`" como función de *hashing* y no para descifrar simétricamente.

El ejemplo que se desarrollará a continuación utiliza el algoritmo de *hashing* SHA-1 para validar la contraseña obtenida a través de la interfaz contra el resultado de la función registrado en base de datos al momento de establecer su contraseña.

```
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;


public class ValidaciónAcceso {

    // información de conexión a la base de datos
    final static String driverClass    =
     "oracle.jdbc.driver.OracleDriver";
    final static String connectionURL  =
     "jdbc:oracle:thin:@localhost:1521:SCH1";
    final static String db_username    = "usuario_db";
    final static String db_password    = "contraseña_db";

    // objeto de conexión
    Connection   con                   = null;

    /**
```

```
     * Construcción del objeto. Desde aquí se tomará la conexión
     * a la base de datos
     * Oracle.
     */
    public ValidacionAcceso() {

        try {

            // cargar driver JDBC
            Class.forName(driverClass).newInstance();

            // conectar a base de datos
            this.con = DriverManager.getConnection(connectionURL,
            db_username, db_password);

        // posbiles excepciones
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }


    /**
     * Método para realizar el control de acceso.
     */
    public bool validarAcceso(String usuario, String contrasena) {

        Statement stmt      = null;
        ResultSet rset      = null;
        String queryString  = null;
        bool resultado      = false;

        try {

            // generar hash SHA-1
            MessageDigest sha = MessageDigest.getInstance("SHA-1");

            // obtener nuestro hash resultado de la contraseña
            sha.update(contrasena);
            byte[] cifrado = sha.digest();
```

```
            // definición consulta sql
            queryString = "SELECT id FROM usuarios WHERE usuario =
            "+ usuario +" AND contrasena = '"+ cifrado.toString(); +"'";

            // instanciar objeto statement para ejecutar consulta
            stmt = con.createStatement ();

            // obtener objeto de resulset realizando la consulta
            rset = stmt.executeQuery(queryString);

            // coinciden los hashes?
            resultado = rset.next();

            // cerrar ResultSet
            rset.close();

            // cerrar statement
            stmt.close();

        // excepción general
        } catch (Exception e) {

            e.printStackTrace();

        }

        return resultado;

    }

    /**
     * Cerrar la conexión a la base de datos.
     */
    public void cerrarConexion() {

        try {
            // cerrar conexión
            con.close();

        } catch (SQLException e) {

            e.printStackTrace();

        }

    }
```

```
    /**
     * Punto de entrada al programa (función main).
     */
    public static void main(String[] args)
            throws java.lang.InterruptedException {

  bool resultado = false;

        // instanciación del objeto
        ValidaciónAcceso va = new ValidacionAcceso();

        // validación de prueba: usuario1/contrsena1
        resultado = va.validarAcceso("usuario1", "contrasena1");

        // mensaje por consola
        if (resultado){
            System.out.println("Contraseña correcta");
        } else {
            System.out.println("Usuario y/o constraseñas incorrectos");
        }

        // cerrar conexión a base de datos
        va.cerrarConexion();

    }

}
```

Listado 3-9. Comprobación de credenciales (contraseñas) utilizando SHA-1.

[1] Hook, David. *Beginning Cryptography with Java*. Wrox, EE.UU., 2005.

4

Criptografía en entornos .NET

El desarrollo del *framework* .NET se comenzó hacia finales de los noventa, originariamente bajo el nombre de *Next Generation Windows Services* (NGWS), que podría traducirse como Servicios Windows de Siguiente Generación. El *framework* .NET versión 1.0 (primera versión) fue liberado en febrero de 2002 para los sistemas Windows 98, NT 4.0, 2000 y XP. La versión 1.1 del *framework* fue publicada en abril de 2003. Se incluyó como parte del sistema operativo Windows Server 2003 y, también, fue parte del segundo *release* de Microsoft Visual Studio .NET (Visual Studio .NET 2003). A principios del año 2006, la versión 2.0 del *framework* .NET estuvo disponible para descarga desde el sitio Web de Microsoft. También se incorporó en Visual Studio 2005 Microsoft SQL Server 2005 y BizTalk 2006. Hacia fines del mismo año la versión 3.0 vio la luz, ya siendo parte integral del nuevo Windows Vista y Windows Server 2008, aunque también disponible para los sistemas operativos Windows XP SP2 y Windows Server 2003. La última versión a la fecha es la 3.5, liberada en noviembre de 2007.

Microsoft desarrolla esta nueva tecnología como alternativa al entorno de programación Java –en continuo avance– de Sun Microsystems. El *framework* .NET tiene varias similitudes con la plataforma de Java, por tal motivo quien conozca el lenguaje de programación Java podría adentrase en .NET, específicamente con el lenguaje C#, y sentirse rápidamente familiarizado con el entorno, metodología y sintaxis de la opción de Microsoft.

El *framework* .NET de Microsoft podría definirse como una tecnología de software, que incluye una librería o biblioteca de funciones (concretamente, clases) pre-codificadas para problemas o necesidades comunes de programación y una máquina virtual que maneja la ejecución de aplicaciones escritas particularmente para el *framework*. Estas aplicaciones se ejecutan en un ambiente o entorno que maneja los requerimientos del programa en *runtime* o tiempo de ejecución. Este entorno de ejecución –componente del *framework* .NET– es conocido como *Common Language Runtime* (CLR) o Lenguaje Común en Tiempo de ejecución, que veremos detalladamente más adelante.

Los componentes principales del *framework* se describirán sólo para comprender las generalidades al respecto de la siguiente manera:

- El .NET *Framework*: Es el ambiente o entorno de ejecución de la plataforma .NET.
- Los lenguajes .NET: Entre ellos, los más popularmente utilizados son C# y VB.NET.
- El *Common Runtime Language* (CRL), que es el motor de ejecución o máquina virtual común a todos los lenguajes .NET.
- El MSIL, *Microsoft Intermedial Language* o Lenguaje Intermedio de Microsoft, al que compilan las aplicaciones (*asemmblies*) .NET. Este es el lenguaje intermedio que es interpretado por el CRL al momento de ejecución.
- El CLS, *Common Language Specification* o Lenguaje de Especificación Común, que engloba los lineamientos que debe cumplir un lenguaje .NET.
- ADO.NET, la nueva interfaz para bases de datos.
- ASP.NET, nueva tecnología para el desarrollo de páginas Web dinámicas, integrada dentro del entorno .NET.
- La librería o biblioteca de clases .NET, que es el conjunto de clases que componen el .NET *framework*.

El *Common Language Runtime* (CLR) es el componente encargado de la ejecución de las aplicaciones .NET; conceptualmente, muy similar a lo que en Java se denomina máquina virtual, este motor se encarga de ejecutar todo el código .NET compilado de tal manera. El CLR es el encargado de convertir este lenguaje intermedio en lenguaje máquina del procesador, esto normalmente se hace en tiempo real por un compilador JIT (*Just-In-Time* o justo a tiempo) que lleva incorporado el CLR.

Se desprende de lo anterior, entonces, que el CLR actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma .NET. Cualquier plataforma para la que exista una versión del CLR podrá ejecutar una aplicación .NET. Microsoft ha desarrollado versiones del CLR para la mayoría de las versiones de Windows, incluidos Windows 95, Windows 98, Windows ME, Windows NT 4.0, Windows 2000, Windows XP y Windows CE (usado en CPUs no pertenecientes a la familia x86). Además, Microsoft ha firmado un acuerdo con Corel para portar el CLR a Linux y también existen grupos de programadores desarrollando de manera independiente versiones *open-source* del CLR para Linux. Asimismo, dado que la arquitectura del CLR está totalmente abierta, es posible que en el futuro se diseñen versiones del mismo para otros sistemas operativos. A colación, también conviene volver a aclarar que desde cualquier lenguaje para el que exista un compilador que genere código para la plataforma .NET es posible utilizar el código generado desde otros lenguajes. Microsoft ha desarrollado un compilador de C#, así como versiones de sus compiladores de Visual Basic (Visual Basic.NET) y C++ (C++ con extensiones

manejadas o gestionadas) y una versión del intérprete de JScript (JScript.NET). La integración de lenguajes es tal que es posible escribir una clase en C# que herede de otra escrita en Visual Basic.NET que, a su vez, herede de otra escrita en C++ con extensiones manejadas o gestionadas.

Componentes del *framework* .NET que hemos de destacar separadamente son ASP.NET, *los Web-services* o servicios Web y Remoting. ASP.NET es el componente dedicado al desarrollo de aplicaciones Web. A través del servidor Web *Internet Information Server* (IIS) de Microsoft, las aplicaciones ASP.NET se podrán ejecutar dentro del CLR y será posible utilizar el conjunto de clases del *framework* .NET para su desarrollo, lo que implica la obtención de una versatilidad y una potencia muy superiores a la alternativa anterior de páginas ASP o ASP 3.0. Con respecto a los servicios Web, sepamos que se trata de una herramienta o metodología de popularidad creciente, para comunicar información a través de Internet entre diferentes computadoras, incluso entre distintos sistemas. Por último, .NET Remoting permite el acceso a objetos de máquinas remotas.

Antes de terminar, para aclarar los conceptos de *managed* y *unmanaged code*, ya que más adelante veremos que las implementaciones criptográficas son en su mayoría del segundo tipo, entendamos que el CLR maneja o gestiona la ejecución de las aplicaciones generadas para la plataforma .NET. Al código de estas aplicaciones se le suele llamar *managed code* o código manejado gestionado, y al código no escrito para ser ejecutado directamente en la plataforma .NET sino por el CPU del sistema, se le suele llamar *unmanaged code*, es decir, código no manejado o no gestionado.

Implementaciones incorporadas.

"System.Security.Cryptography namespace".

El *namespace* `System.Security.Cryptography` provee servicios criptográficos que incluyen la encriptación y desencriptación de datos y otras operaciones como las de la generación de *hashes* criptográficos, la generación de números aleatorios y la autenticación de mensajes.

El *framework* .NET provee implementaciones de varios algoritmos criptográficos basados en estándares. Las implementaciones de estos algoritmos pueden ser utilizadas de manera sencilla, con valores por defecto seguros. Además, el modelo criptográfico del *framework* .NET de herencia, diseño de *stream* o flujo y configuración son extensibles.

El sistema de seguridad del *framework* .NET implementa un patrón extensible de herencia de clases derivadas. La jerarquía es la siguiente:

- Clase de tipo de algoritmo, como por ejemplo las clases `SymmetricAlgorithm` y `HashAlgorithm`. Este nivel es abstracto.

- Clase de algoritmo que hereda de una clase de tipo de algoritmo, por ejemplo `RC2` o `SHA1`. Este nivel también es abstracto.
- Implementación de una clase de algoritmo que hereda también de la clase de algoritmo. Ejemplos son las clases `RC2CryptoServiceProvider` y `SHA1Managed`. Este nivel es implementado completamente.

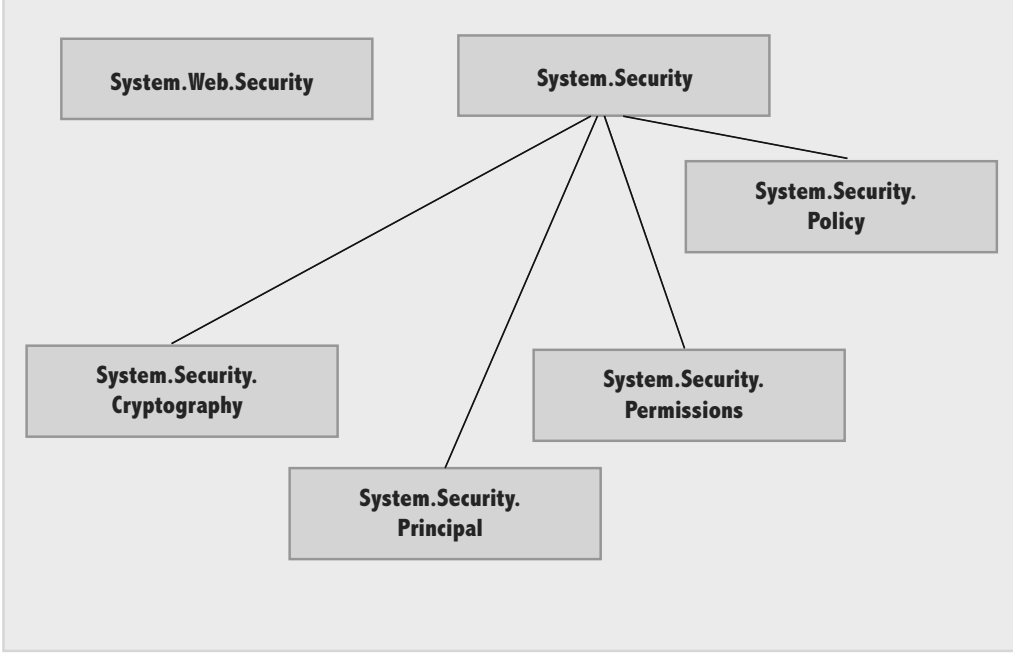

Fig. 4-1. *Namespace* `System.Security.Cryptography` y otras funcionalidades de seguridad.

Las clases incluidas en el *namespace* a partir de Microsoft Visual Studio 2003/.NET *Framework* 1.1 son las siguientes:

- `AsymmetricAlgorithm`: Clase abstracta, para utilizar como base, a partir de la cual todas las implementaciones de algoritmos asimétricos deben heredar.
- `AsymmetricKeyExchangeDeformatter`: Clase base a partir de la cual derivan las clases `Deformatter` de intercambio de llaves asimétricas.
- `AsymmetricKeyExchangeFormatter`: Clase base a partir de la cual derivan las clases `Formatter` de intercambio de llaves asimétricas.

- `AsymmetricSignatureDeformatter`: Clase base a partir de la cual derivan las clases `Deformatter` de firma digital para llaves asimétricas.
- `AsymmetricSignatureFormatter`: Clase base a partir de la cual derivan las clases `Formatter` de firma digital para llaves asimétricas.
- `CryptoAPITransform`: Realiza una transformación criptográfica de los datos.
- `CryptoConfig`: Accede a la información de configuración criptográfica.
- `CryptographicException`: Excepción que se genera al sucederse un error en una operación criptográfica.
- `CryptographicUnexpectedOperationException`: Excepción que se genera cuando se sucede una operación inesperada.
- `CryptoStream`: Define el *stream* o flujo de datos que enlaza a éste con transformaciones criptográficas.
- `CspParameters`: Contiene los parámetros que son pasados al proveedor de servicios criptográficos (CSP) que realiza las computaciones criptográficas. Esta clase no puede ser heredada.
- `DeriveBytes`: Clase base a partir de la cual heredan aquellas que derivan de secuencias de bytes de una longitud determinada.
- `DES`: Clase base para el algoritmo *Data Encryption Standard* (DES) de la cual derivan implementaciones DES.
- `DESCryptoServiceProvider`: Define el objeto *wrapper* para el acceso al CSP del algoritmo DES. No puede heredarse de esta clase.
- `DSA`: Clase base abstracta de la cual las implementaciones del algoritmo *Digital Signature Algorithm* (DSA) deben heredar.
- `DSACryptoServiceProvider`: Define el objeto *wrapper* para el acceso al CSP del algoritmo DSA.
- `DSASignatureDeformatter`: Verifica la firma digital PKCS#1 v1.5 (DSA).
- `DSASignatureFormatter`: Crea una firma digital PKCS#1 v1.5 (DSA).
- `FromBase64Transform`: Convierte un *CryptoStream* desde base 64.
- `HashAlgorithm`: Clase base desde la cual derivan las implementaciones de *hashing* criptográficos.
- `HMACSHA1`: Computa un *Hash-based Message Authentication Code* (HMAC) usando la función SHA-1.
- `KeyedHashAlgorithm`: Clase abstracta desde la cual deben derivar las implementaciones de algoritmos *keyed hash* o de *hashing* con llave criptográfica.

- `KeySizes`: Determina el conjunto de longitudes de llaves válidas para algoritmos criptográficos simétricos.
- `MACTripleDES`: Computa el *Message Authentication Code* (MAC) usando TripleDES para la entrada `CryptoStream`.
- `MaskGenerationMethod`: Clase abstracta a partir de la cual deben derivar las implementaciones de generadores de *mask* o máscara.
- `MD5`: Clase abstracta a partir de la cual derivan las implementaciones del algoritmo MD5.
- `MD5CryptoServiceProvider`: Computa el *hash* MD5 usando la implementación provista por el CSP.
- `PasswordDeriveBytes`: Deriva una llave desde una contraseña.
- `PKCS1MaskGenerationMethod`: Computa una máscara según el estándar PKCS #1, para ser usada por algoritmos de intercambio de llaves.
- `RandomNumberGenerator`: Clase abstracta a partir de la cual derivan implementaciones de números aleatorios.
- `RC2`: Clase base desde la cual derivan las implementaciones del algoritmo RC2.
- `RC2CryptoServiceProvider`: Define el objeto *wrapper* para el acceso a la implementación CSP del algoritmo RC2. La clase no puede ser heredada.
- `Rijndael`: Clase base desde la que derivarán las implementaciones del algoritmo Rijndael.
- `RijndaelManaged`: Clase *managed* o manejada para acceder al algoritmo Rijndael. La clase no puede ser heredada.
- `RNGCryptoServiceProvider`: Implementa un generador de números aleatorios criptográfico usando la implementación provista por el CSP.
- `RSA`: Clase base desde la cual heredan las implementaciones del algoritmo RSA.
- `RSACryptoServiceProvider`: Realiza la encriptación y desencriptación asimétrica usando la implementación del algoritmo RSA provista por el CSP. La clase no puede ser heredada.
- `RSAOAEPKeyExchangeDeformatter`: Desencripta información de intercambio de llave *Optimal Asymmetric Encryption Padding* (OAEP).
- `RSAOAEPKeyExchangeFormatter`: Crea información de intercambio de llave *Optimal Asymmetric Encryption Padding* (OAEP) usando RSA.
- `RSAPKCS1KeyExchangeDeformatter`: Desencripta información de intercambio de llave PKCS #1.

- `RSAPKCS1KeyExchangeFormatter`: Crea información de intercambio de llave PKCS#1 usando RSA.
- `RSAPKCS1SignatureDeformatter`: Verifica una firma RSA PKCS #1 versión 1.5.
- `RSAPKCS1SignatureFormatter`: Crea una firma RSA PKCS #1 versión 1.5.
- `SHA1`: Computa el *hash* SHA-1 para los datos de entrada.
- `SHA1CryptoServiceProvider`: Computa el *hash* SHA-1 para los datos de entrada usando la implementación provista por el CSP. Esta clase no puede ser heredada.
- `SHA1Managed`: Computa el *hash* SHA-1 usando la librería *managed* o manejada.
- `SHA256`: Computa el *hash* SHA-256 para los datos de entrada.
- `SHA256Managed`: Computa el *hash* SHA-256 usando la librería *managed* o manejada.
- `SHA384`: Computa el *hash* SHA-384 para los datos de entrada.
- `SHA384Managed`: Computa el *hash* SHA-384 usando la librería *managed* o manejada.
- `SHA512`: Computa el *hash* SHA-512 para los datos de entrada.
- `SHA512Managed`: Computa el *hash* SHA-512 usando la librería *managed* o manejada.
- `SignatureDescription`: Contiene información acerca de las propiedades de una firma digital.
- `SymmetricAlgorithm`: Clase abstracta a partir de la cual heredan las implementaciones de algoritmos simétricos.
- `ToBase64Transform`: Convierte un CryptoStream a base 64.
- `TripleDES`: Clase base para algoritmos *Triple Data Encryption Standard* desde la cual todas las implementaciones de TripleDES deben derivar.
- `TripleDESCryptoServiceProvider`: Define el objeto *wrapper* para el acceso a la versión CSP de algoritmos TripleDES. Esta clase no puede ser heredada.

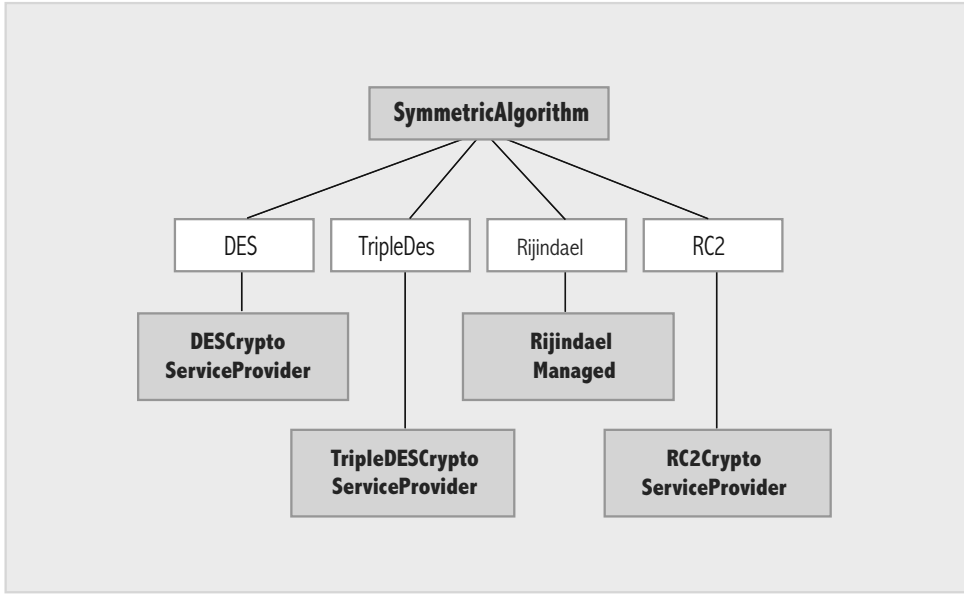

Fig. 4-2. Diagrama ilustrativo del patrón extensible de herencia para algoritmos simétricos.

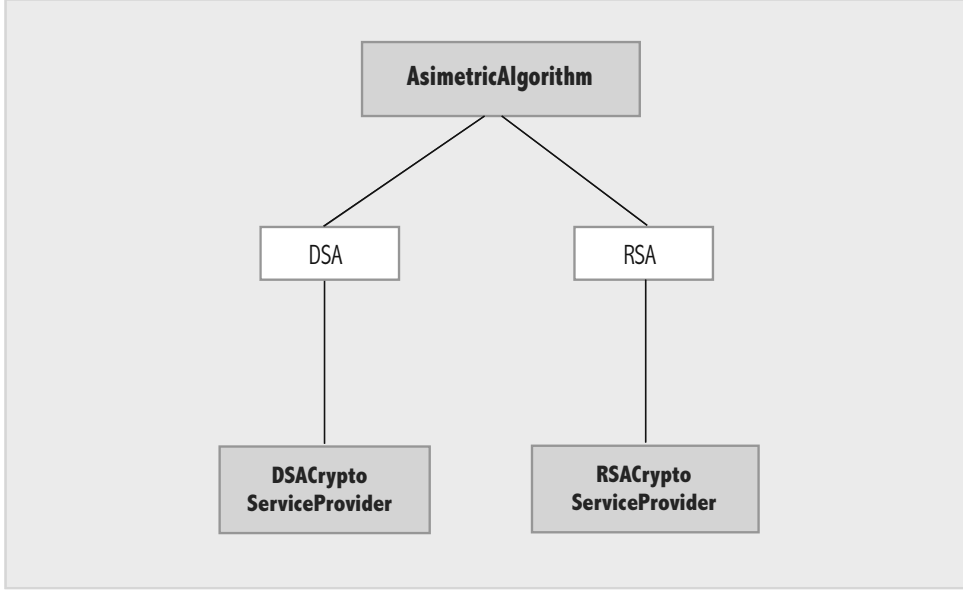

Fig. 4-3. Diagrama ilustrativo del patrón extensible de herencia para algoritmos asimétricos.

Librerías y *frameworks* adicionales.

No comentaremos en detalle ni recomendaremos la utilización de librerías de funciones o *frameworks* criptográficos extra. La primera razón de esto es que el *framework* .NET ya provee un conjunto muy rico de implementaciones de funcionalidades criptográficas sin la necesidad de ningún agregado externo. Otra razón es que si bien existen disponibles en el mercado diferentes alternativas comerciales, salvo por particularidades propias de cada caso, ninguna representa un mayor beneficio, en general, frente a la utilización de las funcionalidades incorporadas en el *namespace* `System.Security.Cryptography` descripto en el apartado anterior.

Por lo tanto, acaso lo conveniente sea utilizar alguna de estas alternativas en el caso de tener una necesidad específica no cubierta por la implementación criptográfica incorporada (por ejemplo, el requerimiento de utilizar un algoritmo no contemplado o la necesidad de hacerlo de modo *managed* o manejado para algún algoritmo no disponible en este modo en el *framework* .NET).

Ya que se trata de una alternativa muy popular en Java, y es un proyecto *open-source,* o de código abierto o libre (de manera que no precisaremos incurrir en gastos onerosos para probarlas), veremos rápidamente de qué tratan las librerías criptográficas de *Bouncy Castle* para C#.

Como se ha explicado en el apartado respecto de la versión para el entorno Java, el proyecto de software libre es llevado adelante por el grupo *The Legion of the Bouncy Castle*. La dirección del sitio Web del proyecto es `http://www.bouncycastle.org`, y la sección específica para el API de servicios o implementaciones criptográficos para C# es `http://www.bouncycastle.org/csharp.html`.

Si bien dentro de la misma documentación encontramos una disculpa respecto a lo limitado de la documentación que se encuentra disponible al momento, prometen mejorarla en un futuro cercano.

Destacaremos los algoritmos implementados de los que podemos hacer uso instalando esta librería:

- Generación y “parseo” de archivos PKCS-12.
- En cuanto al estándar X.509, generadores y “parseadores” de certificados V1 y V3, CRLs V2 y certificados de atributos.
- Algoritmos PBE soportados por PBEUtil: PBEwithMD2andDES-CBC, PBEwithMD2andRC2-CBC, PBEwithMD5andDES-CBC, PBEwithMD5andRC2-CBC, PBEwithSHA1andDES-CBC, PBEwithSHA1andRC2-CBC, PBEwithSHA-1and128bitRC4, PBEwithSHA-1and40bitRC4, PBEwithSHA-1and3-keyDESEDE-CBC, PBEwithSHA-1and2-keyDESEDE-CBC, PBEwithSHA-1and128bitRC2-CBC, PBEwithSHA-1and40bitRC2-CBC, PBEwithHmacSHA-1, PBEwithHmacSHA-224, PBEwithHmacSHA-256, PBEwithHmacRIPEMD128, PBEwithHmacRIPEMD160, y PBEwithHmacRIPEMD256.

- Algoritmos de firma digital soportados por SignerUtilities: MD2withRSA, MD4withRSA, MD5withRSA, RIPEMD128withRSA, RIPEMD160withECDSA, RIPEMD160withRSA, RIPEMD256withRSA, SHA-1withRSA, SHA-224withRSA, SHA-256withRSAandMGF1, SHA-384withRSAandMGF1, SHA-512withRSAandMGF1, SHA-1withDSA, y SHA-1withECDSA.
- Algoritmos de encriptación simétrica: AES, Blowfish, Camellia, CAST5, CAST6, DESede, DES, GOST28147, HC-128, HC-256, IDEA, ISAAC, NaccacheStern, Noekeon, RC2, RC4, RC5-32, RC5-64, RC6, Rijndael, Salsa20, SEED, Serpent, Skipjack, TEA/XTEA, Twofish y VMPC.
- Modos de encriptación simétrica: CBC, CFB, CTS, GOFB, OFB, OpenPGPCFB, y SIC (o CTR).
- Mecanismos de *padding* para encriptación simétrica: ISO10126d2, ISO7816d4, PKCS-5/7, TBC, X.923, y Zero Byte.
- Algoritmos de encriptación asimétrica: RSA (con *blinding*), ElGamal, DSA, y ECDSA.
- Mecanismos de *paddings*/*encodings* para encriptación asimétrica: ISO9796d1, OAEP, y PKCS-1.
- Modos de cifrado de bloque AEAD: CCM, EAX, y GCM.
- Funciones de *hashing* o *digests*: GOST3411, MD2, MD4, MD5, RIPEMD128, RIPEMD160, RIPEMD256, RIPEMD320, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, Tiger, y Whirlpool.
- Mecanismos de firma digital: DSA, ECDSA, ECGOST3410, GOST3410, ISO9796d2, PSS, RSA.
- Intercambio de llaves: Diffie-Hellman y EC-DH.
- Autenticación de mensajes (MACS): CBCBlockCipher, CFBBlockCipher, GOST28147, HMac, ISO9797 Alg. 3, y VMPCMAC.
- Generadores PBE: PKCS-12, y PKCS-5, esquemas 1 y 2.
- OpenPGP (RFC 2440).
- Cryptographic Message Syntax (CMS, RFC 3852), incluyendo API de *streaming*.
- Online Certificate Status Protocol (OCSP, RFC 2560).
- Time Stamp Protocol (TSP, RFC 3161).
- *Elliptic Curve Cryptography* o criptografía de curva elíptica (soporte para curvas F2m y Fp).
- Lectura y escritura de archivos PEM, incluyendo llaves RSA y DSA, con diferentes encriptaciones.

Codificación de encriptación simétrica.

Las clases que proveen funcionalidades criptográficas están incluidas en el *namespace* `System.Security.Cryptography`. Debajo de esta raíz común se encuentran clases y otros *namespaces* especializados para la encriptación simétrica (entre otros). El *framework* de .NET descansa sobre la implementación de la *Microsoft CryptoAPI*, de código del tipo *unmanaged* o no manejado, aunque veremos a continuación que existen dentro de `System.Security.Cryptography` opciones de código *managed*, es decir, ejecutado y controlado por la CLR (*Common Language Runtime*), que entenderemos como la máquina virtual del *framework* .NET y no directamente por la CPU.

Siguiendo con el repaso de lo visto en los primeros apartados del capítulo, sabemos que una aplicación hará uso de estas funcionalidades a través de los *Cryptographic Service Providers* (CSP), o proveedores de servicios criptográficos, implementados para el algoritmo particularmente.

Para la encriptación simétrica de información, las clases disponibles –que heredan de SymmetricAlgorithm– son las siguientes:

- `DESCryptoServiceProvider`: Para el acceso al algoritmo *Data Encryption Standard* (DES).
- `RC2CryptoServiceProvider`: Para el acceso al algoritmo RC2.
- `RijndaelManaged`: Para el acceso al algoritmo Rijndael. Nótese que esta opción es de código *managed*.
- `TripleDESCryptoServiceProvider`: Para el acceso al algoritmo Triple DES.

Veamos algunos ejemplos basados en la documentación oficial. Lo primero será el código que implementa una función que cifrará archivos utilizando el algoritmo Rijndael (versiones para VB.NET y C#).

```
Private Shared Sub EncryptData(inName As String, outName As String, _
rijnKey() As Byte, rijnIV() As Byte)

    ' Crear las instancias de FileStream para manejear los archivos
    Dim fin As New FileStream(inName, FileMode.Open, FileAccess.Read)
    Dim fout As New FileStream(outName, FileMode.OpenOrCreate, _
       FileAccess.Write)
    fout.SetLength(0)

    ' Crear las variables que ayudarán en la lectura y escritura
    Dim bin(100) As Byte ' Almacenamiento para la encriptación
```

```
        Dim rdlen As Long = 0 ' Número total de bytes escritos
        Dim totlen As Long = fin.Length ' Longitud de archivo de entrada
        Dim len As Integer ' Número de bytes a escribir cada vez

        ' Crea la implementación por defecto, esto es RijndaelManaged
        Dim rijn As SymmetricAlgorithm = SymmetricAlgorithm.Create()
        Dim encStream As New CryptoStream(fout, _
           rijn.CreateEncryptor(rijnKey, rijnIV), CryptoStreamMode.Write)

        Console.WriteLine("Encriptando...")

        ' Leer archivo, encriptar y escribir al archivo de salida
        While rdlen < totlen
            len = fin.Read(bin, 0, 100)
            encStream.Write(bin, 0, len)
            rdlen = Convert.ToInt32(rdlen + len)
            Console.WriteLine("{0} bytes procesados", rdlen)
        End While

        encStream.Close()
    fout.Close()
    fin.Close()
    End Sub
```

Listado 4-1. Ejemplo de implementación de una función que utiliza la clase `RijndaelManaged` para cifrar un archivo. Versión VB.NET

```
    private static void EncryptData(String inName, String outName, byte[]
rijnKey, byte[] rijnIV)
     {
          // Crear las instancias de FileStream para manejear los archivos
          FileStream fin = new FileStream(inName, FileMode.Open,
          // FileAccess.Read);
          FileStream fout = new FileStream(outName, FileMode.OpenOrCreate,
          FileAccess.Write);
          fout.SetLength(0);

          // Crear las variables que ayudarán en la lectura y escritura
          byte[] bin = new byte[100];// Almacenamiento para la encriptación
          long rdlen = 0;              // Número total de bytes escritos
          long totlen = fin.Length;    // Longitud de archivo de entrada
          int len;                    // Número de bytes a escribir cada vez

          SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
          // Crea la implementación por defecto, esto es RijndaelManaged
          CryptoStream encStream = new CryptoStream(fout,
```

```
        rijn.CreateEncryptor(rijnKey, rijnIV), CryptoStreamMode.Write);

        Console.WriteLine("Encriptando...");

        // Leer archivo, encriptar y escribir al archivo de salida
        while(rdlen < totlen)
        {
            len = fin.Read(bin, 0, 100);
            encStream.Write(bin, 0, len);
            rdlen = rdlen + len;
            Console.WriteLine("{0} bytes procesados", rdlen);
        }

        encStream.Close();
        fout.Close();

    fin.Close();
}
```

Listado 4-2. Ejemplo de implementación de una función que utiliza la clase `RijndaelManaged` para cifrar un archivo. Versión C#.

En el caso de desear implementar la anterior, pero utilizando otro algoritmo (por ejemplo, Triple-DES), veremos a continuación cómo podríamos hacerlo pero valiéndonos ahora de la clase base `TripleDESCryptoServiceProvider`.

Para el caso de la implementación en VB.NET, las modificaciones necesarias sólo serían el reemplazo de las líneas:

```
    Dim rijn As SymmetricAlgorithm = SymmetricAlgorithm.Create()
    Dim encStream As New CryptoStream(fout, _
       rijn.CreateEncryptor(rijnKey, rijnIV), CryptoStreamMode.Write)
```

Que cambiarían por:

```
    Dim tdes As New TripleDESCryptoServiceProvider()
    Dim encStream As New CryptoStream(fout, _
       tdes.CreateEncryptor(rijnKey, rijnIV), CryptoStreamMode.Write)
```

Para la implementación en C#, ajustaríamos:

```
    SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
    CryptoStream encStream = new CryptoStream(fout,
    rijn.CreateEncryptor(rijnKey, rijnIV), CryptoStreamMode.Write);
```

Cambiándolo por lo siguiente:

```
TripleDESCryptoServiceProvider tdes =
new TripleDESCryptoServiceProvider();
CryptoStream encStream = new CryptoStream(fout,
tdes.CreateEncryptor(rijnKey, rijnIV), CryptoStreamMode.Write);
```

Nótese que se han mantenido los nombres de las variables `rijnKey` (llave) y `rijnIV` (vector de inicialización) para dejar en claro que se utilizan los mismos parámetros que en el ejemplo original, pero aquí contendrán la llave y vector de inicialización para el algoritmo Triple-DES.

Desde el soporte en línea de Microsoft, encontramos dentro de un *how to* o "Cómo hacerlo", el desarrollo para cifrar y descifrar un archivo mediante Visual Basic .NET. Se cita aquí porque se trata de un ejemplo para la implementación del algoritmo DES, aunque deberemos notar que la forma de seleccionar la llave y el vector de inicialización convendría hacerse de alguna de las maneras que fueron descriptas en los capítulos introductorios (con un *hashing* o el proceso de un algoritmo de PBE para la llave y un *timestamp* para el *IV* o vector de inicialización).

La dirección es `http://support.microsoft.com/Kb/301070/es` para la versión en Visual Basic.NET y `http://support.microsoft.com/Kb/307010/es` para la versión C#.

Codificación de encriptación asimétrica.

Nos concentraremos en esta sección en los servicios provistos por el *namespace* `System.Security.Cryptography`, que engloba todo lo que a criptografía se refiere. Esto es, cifrados simétricos, funciones de *hashing*, generación de números aleatorios, etc., incluyendo lo que trataremos a continuación: El cifrado asimétrico de información.

De la misma manera que para la encriptación simétrica, según lo visto en la sección anterior, tendremos acceso a funcionalidades de cifrado asimétrico a través de los *Cryptographic Service Providers* (CSP), o proveedores de servicios criptográficos, implementados para cada algoritmo.

Para la encriptación asimétrica de datos, las clases disponibles –que heredan de `AsymmetricAlgorithm`– son las siguientes:

- `DSACryptoServiceProvider`: Para el acceso al CSP estándar para el algoritmo *Digital Signature Algorithm* (DSA).
- `RSACryptoServiceProvider`: Para el acceso al CSP estándar para el algoritmo RSA.

Siguiendo este orden, primero se planteará lo concerniente a la firma de datos con el algoritmo DSA, mediante la clase `DSACryptoServiceProvider`.

Como se ha explicado en las secciones teóricas del libro, para firmar digitalmente datos, información o un mensaje, utilizando criptografía de llave pública, en primer término el emisor aplicará una función de *hashing* a la información, lo que generará el *message digest* o resumen de mensaje. Luego, encriptará este resultado (el *hash*) con su llave privada para crear su firma personal. El receptor, habiendo recibido la información y la firma, desencriptará la firma utilizando la llave pública del emisor para recuperar el *message digest* y obtendrá, separadamente, el resultado del *hash* de la información mediante el mismo algoritmo que usó el emisor. Si los resultados del *hashing* o *message digest*, concuerdan –esto es: Si el valor desencriptado es el mismo que el obtenido al aplicar el *hashing* sobre la información–, el receptor podrá estar seguro de que la información o el mensaje no fue alterado. Nótese que una firma puede ser verificada por cualquier persona, porque la llave pública del emisor no es información secreta.

Dentro del *framework* .NET, bajo el *namespace* `System.Security.Cryptography`, tenemos a nuestra disposición la clase `DSACryptoServiceProvider`. Nos valdremos de esta clase, entonces, al implementar la funcionalidad criptográfica para proteger la integridad de la información mediante la utilización de una firma digital.

El algoritmo, como el nombre de la clase lo evidencia, es el DSA (`Digital Signature Algorithm`) y soporta llaves con longitudes desde 512 hasta 1024 bits, en incrementos de 64 bits.

Veremos a continuación un código de ejemplo para crear –y luego verificar– una firma digital de un valor de *hash* usando la clase `DSACryptoServiceProvider`.

```
Imports System
Imports System.Security.Cryptography
Imports System.Text


Class EjemploDSA

    Shared Sub Main()

        Try

            ' Crear una nueva instancia de DSACryptoServiceProvider para
            ' la generación de un nuevo par de llaves
            Dim DSA As New DSACryptoServiceProvider()

            ' Crear la instancia de SHA1CryptoServiceProvider para
            'generar el hash SHA-1
            Dim sha As New SHA1CryptoServiceProvider()
```

```
                ' Obtener el resultado del hash de nuestro ejemplo, que luego
                ' firmaremos
                Dim HashValue As Byte() = sha.ComputeHash
                (Encoding.Default.GetBytes("Contenido de prueba")

                ' Almacenar el valor firmado
                    Dim SignedHashValue As Byte() =
                DSASignHash(HashValue,
                DSA.ExportParameters(True), "SHA1")

                ' Verificar el hash y mostrar los resutados
                If DSAVerifyHash(HashValue, SignedHashValue,
                DSA.ExportParameters(False), "SHA1") Then
                Console.WriteLine("El resultado fue verificado.")
                Else
                Console.WriteLine("El resultado NO fue verificado.")
                End If


            Catch e As ArgumentNullException
                Console.WriteLine(e.Message)
            End Try
        End Sub


        Public Shared Function DSASignHash(ByVal HashToSign() As Byte, ByVal
DSAKeyInfo As DSAParameters, ByVal HashAlg As String) As Byte()
            Try

                ' Crear una nueva instancia de DSACryptoServiceProvider
                Dim DSA As New DSACryptoServiceProvider()

                ' Importar la información de la llave
                DSA.ImportParameters(DSAKeyInfo)

                ' Crear una instancia de DSASignatureFormatter y pasar la
                ' DSACryptoServiceProvider para transferir la llave privada
                Dim DSAFormatter As New DSASignatureFormatter(DSA)

                ' Establecer el algoritmo de hashing según parámetro
                DSAFormatter.SetHashAlgorithm(HashAlg)

                ' Crear la firma para HashValue y devolverlo
                Return DSAFormatter.CreateSignature(HashToSign)

            Catch e As CryptographicException
                Console.WriteLine(e.Message)
```

```
                Return Nothing
            End Try
        End Function


       Public Shared Function DSAVerifyHash(ByVal HashValue() As Byte, ByVal
SignedHashValue() As Byte, ByVal DSAKeyInfo As DSAParameters, ByVal HashAlg
As String) As Boolean
            Try

                ' Crear una nueva instancia de DSACryptoServiceProvider
                Dim DSA As New DSACryptoServiceProvider()

                ' Importar información de la llave
                DSA.ImportParameters(DSAKeyInfo)

                ' Crear una instancia de DSASignatureFormatter y pasar la
                ' DSACryptoServiceProvider para transferir la llave privada
                Dim DSADeformatter As New DSASignatureDeformatter(DSA)

                ' Establecer el algoritmo de hashing según parámetro
                DSADeformatter.SetHashAlgorithm(HashAlg)

                ' Verificar la firma y devolver el resultado
                Return DSADeformatter.VerifySignature(HashValue,
                SignedHashValue)

            Catch e As CryptographicException
                Console.WriteLine(e.Message)

                Return False
            End Try
        End Function
    End Class
```

Listado 4-3. Ejemplo que implementa la firma digital utilizando el algoritmo DSA. Versión VB.NET.

```
using System;
using System.Security.Cryptography;
using System.Text;

class EjemploDSA
{
```

```
        static void Main()
        {
            try
            {
               // Crear una nueva instancia de DSACryptoServiceProvider para
               // la generación de un nuevo par de llaves
               DSACryptoServiceProvider DSA =
               new DSACryptoServiceProvider();

               // Crear la instancia de SHA1CryptoServiceProvider para
               // generar el hash SHA-1
               SHA1 sha = new SHA1CryptoServiceProvider();

               // Obtener el resultado del hash de nuestro ejemplo,
               // que luego firmaremos
                byte[] HashValue =
               sha.ComputeHash(Encoding.Default.GetBytes
               ("Contenido de prueba"));

               // Almacenar el valor firmado
               byte[] SignedHashValue = DSASignHash(HashValue,
               DSA.ExportParameters(true), "SHA1");

               // Verificar el hash y mostrar los resutados
               if(DSAVerifyHash(HashValue, SignedHashValue,
               DSA.ExportParameters(false), "SHA1"))
                {
                    Console.WriteLine("El resultado fue verificado.");
                }
                else
                {
                    Console.WriteLine("El resultado NO fue verificado.");
                }
            }

            catch(ArgumentNullException e)
            {
                Console.WriteLine(e.Message);
            }
        }

         public static byte[] DSASignHash(byte[] HashToSign, DSAParameters
DSAKeyInfo, string HashAlg)
        {
            try
            {
```

```
                // Crear una nueva instancia de DSACryptoServiceProvider
                DSACryptoServiceProvider DSA =
                new DSACryptoServiceProvider();

                // Importar la información de la llave
                DSA.ImportParameters(DSAKeyInfo);

                // Crear una instancia de DSASignatureFormatter y pasar la
                // DSACryptoServiceProvider para transferir la llave privada
                DSASignatureFormatter DSAFormatter =
                new DSASignatureFormatter(DSA);

                // Establecer el algoritmo de hashing según parámetro
                DSAFormatter.SetHashAlgorithm(HashAlg);

                // Crear la firma para HashValue y devolverlo
                return DSAFormatter.CreateSignature(HashToSign);
            }

            catch(CryptographicException e)
            {
                Console.WriteLine(e.Message);

                return null;
            }

        }

        public static bool DSAVerifyHash(byte[] HashValue,
byte[] SignedHashValue, DSAParameters DSAKeyInfo, string HashAlg)
        {
            try
            {
                // Crear una nueva instancia de DSACryptoServiceProvider
                DSACryptoServiceProvider DSA =
                new DSACryptoServiceProvider();

                // Importar información de la llave
                DSA.ImportParameters(DSAKeyInfo);

                // Crear una instancia de DSASignatureFormatter y pasar la
                // DSACryptoServiceProvider para transferir la llave privada
                DSASignatureDeformatter DSADeformatter =
                new DSASignatureDeformatter(DSA);

                // Establecer el algoritmo de hashing según parámetro
                DSADeformatter.SetHashAlgorithm(HashAlg);
```

```
                // Verificar la firma y devolver el resultado
                    return DSADeformatter.VerifySignature(HashValue,
SignedHashValue);
            }

            catch(CryptographicException e)
            {
                Console.WriteLine(e.Message);

                return false;
            }

        }

    }
```

Listado 4-4. Ejemplo que implementa la firma digital utilizando el algoritmo DSA. Versión C#.

Continuando con el desarrollo de este capítulo, es el turno ahora de describir la forma de utilizar la clase `RSACryptoServiceProvider`.

Basándonos también en la información provista por la documentación de referencia oficial de la clase, veremos a continuación el código fuente que ejemplificará la implementación de criptografía de llave pública, utilizando el algoritmo RSA mediante los servicios provistos por esta clase (implementación por defecto del algoritmo).

Recordemos que esta clase realiza la encriptación y desencriptación asimétrica, usando la implementación del algoritmo RSA provisto por el *Cryptographic Service Provider* (CSP), o proveedor de servicios criptográficos, y que no puede ser heredada.

La clase `RSACryptoServiceProvider` soporta longitudes de llave desde 384 hasta 16 384 bits, en incrementos de 8 bits, en el caso de que se cuente con el *Microsoft Enhanced Cryptographic Provider* instalado. Si, en cambio, se dispone del *Microsoft Base Cryptographic Provider,* soportará longitudes de llave desde 384 hasta 512 bits, también en incrementos de 8 bits.

Lo que sigue son los ejemplos de la utilización de la clase `RSACryptoServiceProvider` para encriptar un *array* de bytes, obtenido a partir de una cadena de caracteres, para posteriormente desencriptarlo otra vez en una cadena de caracteres.

```
    Imports System
    Imports System.Security.Cryptography
    Imports System.Text
```

```
Class EjemploRSA


    Shared Sub Main()
        Try
            ' Crear instancia de UnicodeEncoding para convertir cadena
            Dim ByteConverter As New UnicodeEncoding()

            ' Crear array de bytes para contener la cadena original, y
            ' los resultados de la encriptación y desencriptación
            Dim dataToEncrypt As Byte() =
            ByteConverter.GetBytes("Contenido de prueba")
            Dim encryptedData() As Byte
            Dim decryptedData() As Byte

           ' Crear una nueva instancia de RSACryptoServiceProvider para
            ' generar las llaves pública y privada
            Dim RSA As New RSACryptoServiceProvider()

            ' Pasaje de datos a encriptar, la información de la llave
           ' pública -RSACryptoServiceProvider.ExportParameters(false)-
            ' y un booleano que especificará que no habrá padding OAEP
            encryptedData = RSAEncrypt(dataToEncrypt,
            RSA.ExportParameters(False), False)

           ' Pasaje de datos a desencriptar, la información de la llave
            ' privada -RSACryptoServiceProvider.ExportParameters(true)-
            ' y un booleano que especificará que no habrá padding OAEP
            decryptedData = RSADecrypt(encryptedData,
            RSA.ExportParameters(True), False)

            ' Mostrar el texto desencriptado por consola
            Console.WriteLine("Texto-plano desencriptado: {0}",
            ByteConverter.GetString(decryptedData))
        Catch e As ArgumentNullException
            ' Tomar la excepción en el caso de que ocurriese un
            ' problema en la encriptación
            Console.WriteLine("El proceso de encriptación falló.")
        End Try
    End Sub


     Public Shared Function RSAEncrypt(ByVal DataToEncrypt() As Byte,
ByVal RSAKeyInfo As RSAParameters, ByVal DoOAEPPadding As Boolean) As Byte()
        Try
            ' Crear una nueva instancia de RSACryptoServiceProvider
```

```
                Dim RSA As New RSACryptoServiceProvider()

                ' Importar la información de la llave RSA. Aquí sólo
                ' necesitaremos la información de la llave pública.
                RSA.ImportParameters(RSAKeyInfo)

                ' Encriptar el array de bytes parámetro, y especificar
                ' el padding OAEP (Windows XP o posterior).
                Return RSA.Encrypt(DataToEncrypt, DoOAEPPadding)
                ' Capturar y mostrar la excepción CryptographicException
            Catch e As CryptographicException
                Console.WriteLine(e.Message)

                Return Nothing
            End Try
        End Function


         Public Shared Function RSADecrypt(ByVal DataToDecrypt() As Byte,
ByVal RSAKeyInfo As RSAParameters, ByVal DoOAEPPadding As Boolean) As Byte()
            Try
                ' Crear una nueva instancia de RSACryptoServiceProvider
                Dim RSA As New RSACryptoServiceProvider()

                ' Importar la información de la llave RSA.
               ' Aquí necesitaremos incluir la información de la llave privada.
                RSA.ImportParameters(RSAKeyInfo)

                ' Desencriptar el array de bytes parámetro, y especificar
                ' el padding OAEP (Windows XP o posterior).
                Return RSA.Decrypt(DataToDecrypt, DoOAEPPadding)
                ' Capturar y mostrar la excepción CryptographicException
            Catch e As CryptographicException
                Console.WriteLine(e.ToString())

                Return Nothing
            End Try
        End Function
    End Class
```

Listado 4-5. Encriptación y desencriptación de una cadena de caracteres mediante el algoritmo RSA. Versión VB.NET.

```
    using System;
    using System.Security.Cryptography;
    using System.Text;
```

```
class EjemploRSA
{

    static void Main()
    {
        try
        {
            // Crear instancia de UnicodeEncoding para convertir cadena
            UnicodeEncoding ByteConverter = new UnicodeEncoding();

            // Crear array de bytes para contener la cadena original, y
            // los resultados de la encriptación y desencriptación
            byte[] dataToEncrypt =
            ByteConverter.GetBytes("Contenido de prueba");
            byte[] encryptedData;
            byte[] decryptedData;

            // Crear una nueva instancia de RSACryptoServiceProvider para
            // generar las llaves pública y privada
            RSACryptoServiceProvider RSA =
            new RSACryptoServiceProvider();

            // Pasaje de datos a encriptar, la información de la llave
            // pública -RSACryptoServiceProvider.ExportParameters(false)-
            // y un booleano que especificará que no habrá padding OAEP
            encryptedData = RSAEncrypt(dataToEncrypt,
            RSA.ExportParameters(false), false);

            // Pasaje de datos a desencriptar, la información de la llave
            // privada -RSACryptoServiceProvider.ExportParameters(true)-
            // y un booleano que especificará que no habrá padding OAEP
            decryptedData = RSADecrypt(encryptedData,
            RSA.ExportParameters(true), false);

            // Mostrar el texto desencriptado por consola
            Console.WriteLine("Texto-plano desencriptado: {0}",
            ByteConverter.GetString(decryptedData));
        }
        catch(ArgumentNullException)
        {
            // Tomar la excepción en el caso de que ocurriese un
            // problema en la encriptación
            Console.WriteLine("El proceso de encriptación falló.");

        }
    }
```

```
        static public byte[] RSAEncrypt(byte[] DataToEncrypt, RSAParameters
RSAKeyInfo, bool DoOAEPPadding)
        {
            try
            {
                // Crear una nueva instancia de RSACryptoServiceProvider
                RSACryptoServiceProvider RSA =
                new RSACryptoServiceProvider();

                // Importar la información de la llave RSA. Aquí sólo
                // necesitaremos la información de la llave pública.
                RSA.ImportParameters(RSAKeyInfo);

                // Encriptar el array de bytes parámetro, y especificar
                // el padding OAEP (Windows XP o posterior).
                return RSA.Encrypt(DataToEncrypt, DoOAEPPadding);
            }
            // Capturar y mostrar la excepción CryptographicException
            catch(CryptographicException e)
            {
                Console.WriteLine(e.Message);

                return null;
            }

        }

        static public byte[] RSADecrypt(byte[] DataToDecrypt, RSAParameters
RSAKeyInfo,bool DoOAEPPadding)
        {
            try
            {
                // Crear una nueva instancia de RSACryptoServiceProvider
                RSACryptoServiceProvider RSA =
                new RSACryptoServiceProvider();

                // Importar la información de la llave RSA. Aquí
               // necesitaremos incluir la información de la llave privada.
                RSA.ImportParameters(RSAKeyInfo);

                // Desencriptar el array de bytes parámetro, y especificar
                // el padding OAEP (Windows XP o posterior).
                return RSA.Decrypt(DataToDecrypt, DoOAEPPadding);
            }
            // Capturar y mostrar la excepción CryptographicException
            catch(CryptographicException e)
```

```
            {
                Console.WriteLine(e.ToString());

                return null;
            }

        }
    }
```

Listado 4-6. Encriptación y desencriptación de una cadena de caracteres mediante el algoritmo RSA. Versión C#.

Veamos, por último, el código de ejemplo que exportaría la información de las llaves criptográficas utilizadas por `RSACryptoServiceProvider` dentro de un objeto (o estructura) llamado `RSAParameters`. Debe recordarse que esta clase no es "serializable".

De esta manera se lo haríamos con VB.NET:

```
        Try

            ' Crear una nueva instancia de RSACryptoServiceProvider
            Dim RSA As New RSACryptoServiceProvider()

            ' Exportar la información de llave a un objeto RSAParameters.
            ' El parámetro "False" implica que se exportará la información
            ' de la llave pública. Deberá utilizarse "True" para exportar
            ' la información de la llave pública y de la llave privada.
            Dim RSAParams As RSAParameters = RSA.ExportParameters(False)


        Catch e As CryptographicException
            ' Capturar error en el proceso
            Console.WriteLine(e.Message)
        End Try
```

Aquí el ejemplo para C#:

```
        try
        {
            // Crear una nueva instancia de RSACryptoServiceProvider
            RSACryptoServiceProvider RSA =
            new RSACryptoServiceProvider();
```

```
            // Exportar la información de llave a un objeto RSAParameters.
            // El parámetro "False" implica que se exportará la información
            // de la llave pública. Deberá utilizarse "True" para exportar
            // la información de las llave pública y de la llave privada.
            RSAParameters RSAParams = RSA.ExportParameters(false);


        }
        catch(CryptographicException e)
        {
            // Capturar error en el proceso
            Console.WriteLine(e.Message);

        }
```

Sin embargo, para la exportación de información de llaves la clase RSA provee métodos para recuperar o establecer estas llaves en formato XML. Veamos un ejemplo para la generación de estos archivos:

```
private static void GenerarLlaves(string archPublica,
 string archPrivada)
{
    // Crear una nueva llave RSA y guardarla
    // en el container
    RSA rsaKey = RSA.Create();

    // Escribir la información de las llaves en archivos
    GuardarArchivo(archPublica, rsaKey.ToXmlString(false));
    GuardarArchivo (archPrivada, rsaKey.ToXmlString(true));
}

private static void GuardarArchivo (string archivo,
 string datos)
{
    // Escribir los datos al archivo
    StreamWriter streamWriter =
     System.IO.File.CreateText(archivo);
    streamWriter.Write(datos);
    streamWriter.Close();
}
```

Codificación de funciones de una vía y *hash.*

De acuerdo con lo recomendado al comienzo del capítulo, y como se ha hecho para el desarrollo de la criptografía simétrica y asimétrica, nos concentraremos, para el abordaje del uso de funciones de *hashing* en el *framework* .NET, en el *namespace* `System.Security.Cryptography`.

Así como para las encriptaciones simétrica y asimétrica, vistas en las secciones anteriores, podremos acceder a las diferentes implementaciones de funciones de *hashing* a través de los *Cryptographic Service Providers* (CSP), o proveedores de servicios criptográficos, implementados para cada algoritmo en particular.

El *namespace* `System.Security.Cryptography` provee las siguientes clases para la generación de *hashes* criptográficos:

- `MD5CryptoServiceProvider`: Computa el *hash* MD5 de los datos de entrada, usando la implementación del CSP.
- `SHA1CryptoServiceProvider` y `SHA1Managed`: Estas clases computan el *hash* SHA-1 de la información de entrada, utilizando la implementación CSP. La primera usa código *unmanaged* o no manejado y la segunda *managed* o manejado.
- `SHA256Managed`, `SHA256Managed` y `SHA256Managed`: Computan el *hash* SHA-256, SHA-384 y SHA-512, respectivamente, para los datos de entrada, usando código *managed*.

Comenzando por lo primero, se desarrollarán algunos ejemplos de cómo implementaríamos una función, que se valdrá de la clase `MD5CryptoServiceProvider`, para obtener la implementación del algoritmo y devolver el resultado del *hash* de un *array*, o arreglo, de bytes como parámetro de entrada.

En primera instancia, veamos el ejemplo para VB.NET; la función será llamada "`generarMD5()`":

```
Function generarMD5 (datos() As Byte) As Byte()
```

Utilizamos a continuación la implementación de la clase abstracta MD5, a través del CSP, clase `MD5CryptoServiceProvider`:

```
    Dim md5 As New MD5CryptoServiceProvider()
```

Computamos el resultado; esto es, el *hash* MD5:

```
    Dim resultado As Byte() = md5.ComputeHash(datos)
```

Devolvemos el resultado y terminamos la función:

```
    Return resultado

End Function
```

Ahora para C#, implementamos la misma funcionalidad:

```
byte[] generarMD5 (byte[] datos
 {
```

Utilizamos a continuación la implementamos de la clase abstracta MD5, a través del CSP, clase `MD5CryptoServiceProvider`:

```
    MD5 md5 = new MD5CryptoServiceProvider();
```

Computamos el resultado; esto es, el *hash* MD5:

```
    byte[] resultado = md5.ComputeHash(datos);
```

Devolvemos el resultado y terminamos la función:

```
    return resultado;
 }
```

Para continuar ejemplificando con código que utilice clases para la generación de *hashes* criptográficos pertenecientes al *namespace* `System.Security.Cryptography`, se agrega un ejemplo que usará la clase abstractra `SHA1` a través del proveedor `SHA-1CryptoServiceProvider`. Se comprobará que, de acuerdo con el diseño o modelo planteado por Microsoft, efectivamente es muy similar a hacerlo para, por ejemplo, el algoritmo MD5.

Cómputo de *hashing* criptográfico SHA-1, en VB.NET:

```
Dim datos As Byte() =
 Encoding.Default.GetBytes("Contenido de prueba")

Dim resultado() As Byte

Dim sha As New SHA1CryptoServiceProvider()

resultado = sha.ComputeHash(datos)
```

Ahora el mismo ejemplo, pero codificado en C#:

```
byte[] datos =
 Encoding.Default.GetBytes("Contenido de prueba");

byte[] resultado;

SHA1 sha = new SHA1CryptoServiceProvider();

resultado = sha.ComputeHash(datos);
```

Se verá, en lo que sigue, ejemplos completos (versiones para VB.NET y C#) para el cómputo y verificación de *hashes* MD5, basados en los provistos por la documentación de referencia oficial de la clase `MD5CryptoServiceProvider`.

Comencemos por la versión para VB.NET:

```
Imports System
Imports System.Security.Cryptography
Imports System.Text

Module EjemploMD5

    ' Obtener el hash de una cadena y devolverlo también
    ' como una cadena, de 32 caracteres (hexadecimal)
    Function getMd5Hash(ByVal input As String) As String

        ' Crear una nueva instancia del objeto MD5CryptoServiceProvider
        Dim md5Hasher As New MD5CryptoServiceProvider()

        ' Convertir la cadena de entrada a un array de bytes y
        ' computar el hash
         Dim data As Byte() =
         md5Hasher.ComputeHash(Encoding.Default.GetBytes(input))

        ' Crear un nuevo Stringbuilder para armar la cadena resultado
        Dim sBuilder As New StringBuilder()

        ' Iterar sobre cada byte del resultado del hashing y convertirlo
        ' a una cadena de caracteres (hexadecimal)
        Dim i As Integer
        For i = 0 To data.Length - 1
            sBuilder.Append(data(i).ToString("x2"))
        Next i

        ' Devolver la cadena de caracteres resultado
```

```
            Return sBuilder.ToString()
        End Function


        ' Verificar el hash de un string
        Function verifyMd5Hash(ByVal input As String, ByVal hash As String)
As Boolean
            ' Obtener el hash de la entrada
            Dim hashOfInput As String = getMd5Hash(input)

            ' Crear un StringComparer y comparar ambos resultados
           Dim comparer As StringComparer = StringComparer.OrdinalIgnoreCase

            If 0 = comparer.Compare(hashOfInput, hash) Then
                Return True
            Else
                Return False
            End If

        End Function


        Sub Main()
            ' Cadena de entrada de prueba
            Dim source As String = "Contenido de prueba"

            ' Obtener el resultado del hash
            Dim hash As String = getMd5Hash(source)

            ' Imprimir resultado
            Console.WriteLine("El resultado del hash MD5 de " + source +
            " es: " + hash + ".")

            Console.WriteLine("Verificando el resultado del hash...")

            ' Imprimir resultado de comparación
            If verifyMd5Hash(source, hash) Then
                Console.WriteLine("Ambos resultados coinciden.")
            Else
                Console.WriteLine("Los resultados no coinciden.")
            End If

        End Sub
    End Module
```

Listado 4-7. Ejemplo para el cómputo y verificación del *hashing* MD5 de una cadena de caracteres. Versión VB.NET.

Veamos ahora la versión de lo anterior para C#:

```
    using System;
    using System.Security.Cryptography;
    using System.Text;

    class EjemploMD5
    {
        // Obtener el hash de una cadena y devolverlo también
        // como una cadena, de 32 caracteres (hexadecimal)
        static string getMd5Hash(string input)
        {
            // Crear una nueva instancia del objeto MD5CryptoServiceProvider
           MD5CryptoServiceProvider md5Hasher = new MD5CryptoServiceProvider();

            // Convertir la cadena de entrada a un array de bytes y
computar el hash
            byte[] data =
             md5Hasher.ComputeHash(Encoding.Default.GetBytes(input));

            // Crear un nuevo Stringbuilder para armar la cadena resultado
            StringBuilder sBuilder = new StringBuilder();

            // Iterar sobre cada byte del resultado del hashing y
            // convertirlo
            // a una cadena de caracteres (hexadecimal)
            for (int i = 0; i < data.Length; i++)
            {
                sBuilder.Append(data[ i] .ToString("x2"));
            }

            // Devolver la cadena de caracteres resultado
            return sBuilder.ToString();
        }

        // Verificar el hash de un string
        static bool verifyMd5Hash(string input, string hash)
        {
            // Obtener el hash de la entrada
            string hashOfInput = getMd5Hash(input);

            // Crear un StringComparer y comparar ambos resultados
            StringComparer comparer = StringComparer.OrdinalIgnoreCase;

            if (0 == comparer.Compare(hashOfInput, hash))
            {
                return true;
            }
            else
            {
```

```
                return false;
            }
        }


        static void Main()
        {

            // Cadena de entrada de prueba
            string source = "Contenido de prueba";

            // Obtener el resultado del hash
            string hash = getMd5Hash(source);

            // Imprimir resultado
             Console.WriteLine("El resultado del hash MD5 de " + source +
" es: " + hash + ".");

            Console.WriteLine("Verificando el resultado del hash...");

            // Imprimir resultado de comparación
            if (verifyMd5Hash(source, hash))
            {
                Console.WriteLine("Ambos resultados coinciden.");
            }
            else
            {
                Console.WriteLine("Los resultados no coinciden.");
            }

        }
    }
```

Listado 4-8. Ejemplo para el cómputo y verificación del *hashing* MD5 de una cadena de caracteres. Versión C#.

Codificaciones de casos prácticos.

Cifrado simétrico de información.

Veremos ahora una alternativa para la implementación práctica de un problema típico, con el que ejemplificaremos el uso de criptografía simétrica en .NET. Como lo adelanta el título, se trata de cifrar o encriptar información de manera simétrica.

Veremos, además, cómo se utiliza la clase `PasswordDeriveBytes` para la ge-neración de una llave segura a partir de una contraseña o frase-clave, seleccionado el algoritmo de *hashing* en que se basará y la cantidad de iteraciones que se han de re-

alizar; esto es, la cantidad de veces que aplicará la función de *hash*. La implementación de esta clase se basa en el estándar PKCS#5 v2.0.

Dentro de una aplicación o servicio podríamos utilizar este código para la transferencia o el registro de la información cifrada; así, entonces, tendremos la seguridad de que nadie –en tanto no conozca la contraseña– podrá obtener los contenidos originales del mensaje o del archivo que hayamos encriptado y podremos transmitirlo a través de un canal inseguro, o almacenarlo en el caso de un archivo, sin mayores cuidados en un disco rígido local, en un recurso compartido en entornos Windows o en un repositorio público en Internet.

```
using System;
using System.IO;
using System.Text;
using System.Security.Cryptography;


public class CifradoSimetrico
{
    // Este método encriptará  el texto-plano utilizando Rijndael y
    // retornará el resultado encodeado en base-64. El parámetro de
    // frase-clave será usado para generar la llave. El salt se usará
    // de manera convencial. El algoritmo de hashing a utilizar
    // también será parámetro, como la cantidad de interacciones (para
    // la contraseña), el vector de inicialización y la longitud de
    // la llave.
    public static string Encriptar(string   textoPlano,
                                   string   fraseClave,
                                   string   salt,
                                   string   algoritmoHash,
                                   int      iteracionesPBE,
                                   string   initVector,
                                   int      longitudLlave)
    {
        // Convertir las cadenas de caracteres (ASCII) a arrays de bytes
        byte[] initVectorBytes = Encoding.ASCII.GetBytes(initVector);
        byte[] saltBytes  = Encoding.ASCII.GetBytes(salt);

        // Convertir el texto-plano (UTF-8) a un array de bytes
        byte[] textoPlanoBytes  = Encoding.UTF8.GetBytes(textoPlano);

        // Generación de la llave (password); estándar PKCS#5 v2.0
        PasswordDeriveBytes password = new PasswordDeriveBytes(
                                                        fraseClave,
                                                        saltBytes,
                                                        algoritmoHash,
                                                        iteracionesPBE);
```

```
        // Tomar los bytes de la llave
        byte[] llaveBytes = password.GetBytes(longitudLlave / 8);

        // Crear la instancia del objeto Rijndael para la encriptación
        RijndaelManaged cifrado = new RijndaelManaged();

        // Establecer el modo de cifrado a CBC
        cifrado.Mode = CipherMode.CBC;

        // Generar la instancia de ICryptoTransform según la llave y el
        // vector de inicialización
        ICryptoTransform encryptor = cifrado.CreateEncryptor(
                                                    llaveBytes,
                                                    initVectorBytes);

        // Memoria en donde almacenaremos la información encriptada
        MemoryStream memoryStream = new MemoryStream();

        // Instancia el CryptoStream a utilizar
        CryptoStream cryptoStream = new CryptoStream(memoryStream,
                                             encryptor,
                                            CryptoStreamMode.Write);
        // Comenzar la encriptación
        cryptoStream.Write(textoPlanoBytes, 0, textoPlanoBytes.Length);

        // Finalizar la encriptación
        cryptoStream.FlushFinalBlock();

        // Convertir la información cifrada a un array de bytes
        byte[] textoCifradoBytes = memoryStream.ToArray();

        // Cerrar los "streams"
        memoryStream.Close();
        cryptoStream.Close();

        // Convertir la información cifrada a una cadena encodeada en
        // base-64
        string textoCifrado = Convert.ToBase64String(textoCifradoBytes);

        // Retorno de la cadena cifrada o encriptada
        return textoCifrado;
    }


    // Este método desencriptará el texto-cifrado usando la implementación
    // del algoritmo Rijndael; de manera similar a cómo lo encripta los
```

```
        // datos el método anterior. Nótese que los parámetros del algoritmo
        // de hashing, longitud de llave, vector de inicialización, salt,
número
        // de iteraciones y la frase-clave, por supuesto, deberán ser los
mismos
        // que aquellos utilizados en la encriptación.
        public static string Desencriptar(string   textoCifrado,
                                  string   fraseClave,
                                  string   salt,
                                  string   algoritmoHash,
                                  int      iteracionesPBE,
                                  string   initVector,
                                  int      longitudLlave)
        {
            // Convertir cadenas de caracteres (ASCII) a arrays de bytes
            byte[] initVectorBytes = Encoding.ASCII.GetBytes(initVector);
            byte[] saltBytes  = Encoding.ASCII.GetBytes(salt);

            // Convertir el texto-cifrado a un array de bytes
            byte[] textoCifradoBytes =
            Convert.FromBase64String(textoCifrado);

            // De manera similar a la del proceso de encriptación,
            // generamos
            // la llave (password)
            PasswordDeriveBytes password = new PasswordDeriveBytes(
                                                          fraseClave,
                                                          saltBytes,
                                                          algoritmoHash,
                                                        iteracionesPBE);

            // Recuperamos los bytes de la llave
            byte[] llaveBytes = password.GetBytes(longitudLlave / 8);

            // Instancia del objeto Rijndael
            RijndaelManaged    cifrado = new RijndaelManaged();

            // Como para la encriptación, se establece el modo CBC
            cifrado.Mode = CipherMode.CBC;

            // Generar la instancia de ICryptoTransform según la llave y el
            // vector de inicialización; aquí, para la desencriptación
            ICryptoTransform decryptor = cifrado.CreateDecryptor(
                                                        llaveBytes,
                                                        initVectorBytes);

            // Memoria que almacena la información encriptada
```

```
            MemoryStream  memoryStream = new MemoryStream(textoCifradoBytes);

            // Instancia de nuestro CryptoStream
            CryptoStream  cryptoStream = new CryptoStream(memoryStream,
                                                          decryptor,
                                                  CryptoStreamMode.Read);

            // Buffer para la información "desencriptada"
            byte[ ]  textoPlanoBytes = new byte[ textoCifradoBytes.Length] ;

            // Comienzo de desencriptación
            int desencriptaByteCount = cryptoStream.Read(textoPlanoBytes,
                                                         0,
                                                 textoPlanoBytes.Length);

            // Cerrar "streams"
            memoryStream.Close();
            cryptoStream.Close();

              // Convertir la información desencriptada a una cadena de
caracteres (UTF-8)
            string textoPlano = Encoding.UTF8.GetString(textoPlanoBytes,
                                                        0,
                                                 desencriptaByteCount);

            // Retorno del texto-plano; cadena de caracteres desencriptada
            return textoPlano;
        }
    }

    public class PruebaCifradoSimetrico
    {

        static void Main(string[ ]  args)
        {
            // Nuestro texto-plano original
            string   textoPlano         = "Contenido de prueba";

            // Nuestra frase-clave
            string   fraseClave         = "Frase secreta";

            // Un valor de salt de ejemplo
            string   salt               = "12345678";

            // Algoritmo de hash a utilizar para generar la
            // llave a partir de la frase-clave ("SHA1" o "MD5")
            string   algoritmoHash      = "SHA1";
```

```
            // Cantidad de iteraciones para el algoritmo PBE
            int      iteracionesPBE = 1000;

            // Vector de incialización de ejemplo (16 bytes)
            string   initVector         = "1234567890123456";

            // Longitud de la llave (128, 192 o 256)
            int      longitudLlave            = 128;

            // Imprimir a consola el texto-plano original
              Console.WriteLine(String.Format("Texto-plano original: {0}",
textoPlano));

            // Realizamos la encriptación
            string  textoCifrado = CifradoSimetrico.Encriptar(textoPlano,
                                                    fraseClave,
                                                    salt,
                                                    algoritmoHash,
                                                    iteracionesPBE,
                                                    initVector,
                                                    longitudLlave);

            // Imprimir a consola el texto-cifrado
                   Console.WriteLine(String.Format("Texto-cifrado:  {0}",
textoCifrado));

            // Realizamos la desencriptación
            textoPlano  = CifradoSimetrico.Desencriptar(textoCifrado,
                                                    fraseClave,
                                                    salt,
                                                    algoritmoHash,
                                                    iteracionesPBE,
                                                    initVector,
                                                    longitudLlave);

            // Imprimir a consola el texto-plano obtenido
              Console.WriteLine(String.Format("Texto-plano obtenido: {0}",
textoPlano));
        }
    }
```

Listado 4-9. Ejemplo práctico para el cifrado de información de manera simétrica. Algoritmo Rijndael.

Funciones que implementan criptografía de llave pública.

Se desarrollará a continuación un ejemplo de código fuente que implementa funciones para la encriptación y desencriptación de información, utilizando el algoritmo asimétrico o de llave pública RSA. La información de las llaves será almacenada en archivos XML.

Al crear una instancia de la clase `RSACryptoServiceProvider`, utilizando el constructor por defecto, éste automáticamente creará un nuevo par de llaves –pública y privada–, listo para usarse. Sin embargo, si queremos hacer uso de llaves creadas con anterioridad, deberemos inicializar la clase con una instancia del objeto `CspParameters`, inicializado o parametrizado con los valores correspondientes. Esto queda ejemplificado en la función de inicialización de la instancia de clase proveedora del servicio criptográfico RSA.

Como hemos explicitado más arriba, en la función que genera o restablece un nuevo par de llaves es en donde, mediante una instancia de la clase `CspParameters` (`cspParams`), almacenaremos la información de las llaves en archivos XML: Uno para la información de la llave privada y otro para la información de la llave pública. Este método deberá ser invocado sólo una vez –en tanto se desee mantener las llaves generadas–, antes de invocar los procesos de encriptación o desencriptación. Por supuesto, si se deseara volver a obtener otro par de llaves y reescribir los archivos XML, se deberá invocar el método nuevamente.

```
using System;
using System.IO;
using System.Security;
using System.Security.Cryptography;


public class ImplementacionRSA
{
  public static RSACryptoServiceProvider rsa;

  // En este método inicializaremos los parámetros correspondientes
  // para la encriptación y desencriptación utilizando nuestras
  // llaves en archivos XML.
  public static void InicializarRSA()
  {

   const int PROVIDER_RSA_FULL = 1;
   const string CONTAINER_NAME = "EjemploImplementacionRSA";
   CspParameters cspParams;
   cspParams = new CspParameters(PROVIDER_RSA_FULL);
   cspParams.Flags = CspProviderFlags.UseMachineKeyStore;
   cspParams.KeyContainerName = CONTAINER_NAME;
   cspParams.ProviderName = "Microsoft Strong Cryptographic Provider";
```

```
 rsa = new RSACryptoServiceProvider(cspParams);
}

// Método que implementa el proceso de encriptación asimétrica
// utilizando la información de llaves en archivos XML
public static string Encriptar(string textoPlano)
{
 InicializarRSA();
 StreamReader reader = new   StreamReader(@"C:\llave_publica.xml");
 string publicOnlyKeyXML = reader.ReadToEnd();
 rsa.FromXmlString(publicOnlyKeyXML);
 reader.Close();

 // Cifrar texto-plano
 byte[] textoPlanoBytes =
 System.Text.Encoding.UTF8.GetBytes(textoPlano);
 byte[] textoCifradoBytes = rsa.Encrypt(textoPlanoBytes,false);
 return Convert.ToBase64String(textoCifradoBytes);
}

// Este método servirá para generar y almacenar un nuevo par de llaves
// para la encriptación y desencriptación asimétrica (llaves pública y
// privada)
public static void GenerarLlaves()
{
 InicializarRSA();

 // Proveer parámetros RSA (públicos y privados)
 StreamWriter writer = new   StreamWriter(@"C:\llave_privada.xml");
 string publicPrivateKeyXML = rsa.ToXmlString(true);
 writer.Write(publicPrivateKeyXML);
 writer.Close();

 // Proveer únicamente parámetros públicos
 writer = new StreamWriter(@"C:\llave_publica.xml");
 string publicOnlyKeyXML = rsa.ToXmlString(false);
 writer.Write(publicOnlyKeyXML);
 writer.Close();

}

// Método que implementa el proceso de desencriptación asimétrica
// utilizando la información de llaves en archivos XML
public static string Desencriptar(string textoCifrado)
{
 InicializarRSA();
```

```
    byte[] textoCifradoBytes = Convert.FromBase64String(textoCifrado);

    StreamReader reader = new StreamReader(@"C:\llave_privada.xml");
    string publicPrivateKeyXML = reader.ReadToEnd();
    rsa.FromXmlString(publicPrivateKeyXML);
    reader.Close();

    // Desencriptar el texto-cifrado
    byte[] textoPlanoBytes = rsa.Decrypt(textoCifradoBytes,false);
    return System.Text.Encoding.UTF8.GetString(textoPlanoBytes);

   }

}
```

Listado 4-10. Clase para la implementación de criptografía asimétrica mediante el algoritmo RSA y archivos de llaves XML.

Criptografía en entornos PHP

5

Nacido en el año 1994 y liberado en 1995, PHP[1] es un lenguaje de programación o, para los puristas, de *scripting* o interpretado. Salvo en configuraciones especiales, el programa PHP se interpreta, no se compila. Esto quiere decir que el programa fuente (por convención en archivos con extensión ".php") contiene las instrucciones o códigos que se interpretarán y ejecutarán cada vez, en cada corrida. No se compilan esos archivos fuente para la generación de un binario ejecutable, como sucede en los otros lenguajes como Visual Basic y C/C++, o para la generación de un código intermedio para ser ejecutado por una máquina virtual, como en el caso de Java o .Net.

El lenguaje fue ideado especialmente para el desarrollo Web y es, quizá el más popular en este tipo de aplicaciones. Dentro de sus características más importantes, y acaso las causas de su popularidad, se destacan su facilidad de uso y el paradigma de ser codificado de manera embebida en código HTML, es decir que, dentro de la misma página Web, se codifica la programación mediante delimitadores o *tags* específicos.

Su sintaxis es parecida a la del lenguaje Perl, otro interpretado o de *scripting*, en la línea del lenguaje de programación C (en lo que a sintaxis general y a programación estructurada se refiere).

Actualmente, su utilización es muy popular en la implementación de soluciones Web con herramientas de código abierto o software libre. Así, uno de los entornos o combinaciones más comunes de herramientas o software es el LAMP (iniciales de Linux, Apache, MySQL y PHP).

Si bien su uso más común, como se ha dicho, es embebido dentro de una página Web, donde el servidor procesa el código encontrado entre los delimitadores, también puede ser usado desde la línea de comandos y hasta como programas independientes o *stand-alone* con interfaz gráfica.

Actualmente, el organismo encargado de mantener las versiones oficiales del intérprete trabaja bajo el nombre de "*The PHP Group*", grupo que se encarga de man-

tener las actualizaciones disponibles, desarrollar módulos y extensiones, administrar la documentación y demás aspectos administrativos que incluyen la definición de facto del lenguaje, ya que no está estandarizado de manera formal.

Implementaciones incorporadas.

Desde las primeras versiones de uso masivo (versión 4.0 en adelante), el lenguaje ya contaba con funciones incorporadas para la generación de *hashes*. El ejemplo por excelencia es la función `md5()`. Ésta es una referencia obligada del lenguaje, acaso no sólo por la popularización del algoritmo MD5, sino por su gran utilización en aplicaciones Web para cifrar contraseñas por ejemplo, o hacer pasajes seguros de parámetros.

Su uso se especifica de la siguiente manera:

```
string md5 ( string $cadena [ , bool $modo_binario ] )
```

Así entonces, por ejemplo, para la generación del *hash* criptográfico MD5 de la cadena de caracteres “Contenido de prueba”, asignarlo a la variable `$hash` e imprimirlo, codificaríamos:

```
<?php

$hash = md5("Contenido de prueba");
echo $hash;

?>
```

Código que imprimirá:

```
17603a7adfa88737d21c1aded1753fd3
```

De esta manera, la función devuelve un *string* o cadena de 32 caracteres de longitud, representando los 16 bytes del resultado del *hash* en formato hexadecimal. El último parámetro de la función es opcional. Se incorporó en la versión 5.0 y su valor por defecto es FALSE. Si el valor del parámetro fuese TRUE, el *hash* md5 sería devuelto en formato binario, no como una cadena de caracteres.

Por otra parte, desde la versión 4.3 también se encuentra disponible una implementación del algoritmo de *hashing* SHA-1. Su utilización es muy similar a lo visto para la implementación de MD5:

```
string sha1 ( string $cadena [ , bool $modo_binario ] )
```

Como con la función `md5()`, la cadena representa el contenido a partir del cual se generará el *hash* o resumen criptográfico. El segundo parámetro también es opcional y por defecto su valor es FALSE. Si se definiese a TRUE, entonces el *hash* es retornado en un formato binario puro, con una longitud de 20 bytes; de lo contrario, el valor devuelto es una cadena de 40 caracteres, que representa el número en notación hexadecimal.

A continuación se observa el ejemplo para obtener el resultado en una cadena de caracteres e imprimirlo utilizando `sha1()`:

```
<?php

$hash = sha1("Contenido de prueba");
echo $hash;

?>
```

lo que imprimirá:

```
f5d240c860be1a48f01348b01cb55c7577c8e46a
```

Existen además funciones complementarias, implementando ambos algoritmos para la generación de *hashes* a partir de los contenidos de archivos. Esto suele utilizarse para confirmar la integridad de archivos descargados de Internet (se utiliza la palabra *checksum* como sinónimo, aunque referimos al lector a los capítulos introductorios para conocer las diferentes nociones que aplican a cada caso) o controlar potenciales diferencias en archivos especiales auditados.

La función que utiliza el algoritmo MD5 se define de la siguiente manera:

```
string md5_file ( string $nombre_archivo [ , bool $modo_binario ] )
```

La alternativa que implementa la misma funcionalidad, pero utilizando el algoritmo SHA-1, es:

```
string sha1_file ( string $nombre_archivo [ , bool $modo_binario ] )
```

Por último, haremos referencia a otra funcionalidad criptográfica incorporada en el lenguaje desde la versión 4.0. Se trata de la función `crypt()`. Esta función genera también un *hash*, pero se implementó especialmente para el registro de contraseñas

en medios inseguros. Tal es la razón de ser de su último argumento opcional, como veremos a continuación:

```
string crypt ( string $cadena [ , string $salt ] )
```

La función devolverá una cadena de caracteres utilizando el algoritmo estándar basado en DES, para la encriptación de contraseñas en sistemas UNIX, o mediante algoritmos alternativos que estén disponibles en el sistema.

Algunos sistemas operativos soportan más de un tipo de algoritmo de encriptación. Es común que el método basado en DES se encuentre reemplazado por una implementación basada en MD5 en sistemas modernos. En esta función PHP, el algoritmo o tipo de algoritmo que se ha de utilizar se determina por el segundo parámetro opcional, `$salt`. Como hemos visto, el *salt* funcionará como parámetro modificador en la generación del resultado, para que dos contraseñas o entradas iguales no produzcan el mismo resultado. Al momento de la instalación del intérprete, PHP determinará las opciones disponibles en sistema para establecer los diferentes tipos de *salts* que la función de `crypt()` soportará. Si en una invocación a la función no se especifica este argumento, PHP generará un *salt* estándar de dos caracteres, salvo que el tipo por defecto sea uno basado en MD5, en cuyo caso se generará uno compatible aleatoriamente.

Recordemos, en principio, que se trata de una función de encriptación, como su nombre lo evidencia, pero a través de una función de una vía. No existe una función `decrypt()` para desencriptar o descifrar la salida de la función `crypt()`.

Como vemos en la definición de la función, la variable `$cadena` es el texto o información para encriptar.

Librerías y *frameworks* adicionales.

MCrypt.

La librería MCrypt para PHP se trata de una interfaz para el acceso a la librería del mismo nombre, que provee soporte para diferentes algoritmos de cifrado en bloque como DES, TripleDES, Blowfish (algoritmo por defecto), 3-WAY, SAFER-SK64, SAFER-SK128, TWOFISH, TEA, RC2 y GOST en los modos de cifrado CBC, OFB, CFB y ECB. Adicionalmente, también soporta RC6 e IDEA, que son considerados “no libres”.

Con el ánimo de brindar una primera aproximación, una idea general respecto del alcance de las implementaciones provistas por la librería, incluiremos aquí una lista de las funciones o prototipos:

```
    resource mcrypt_module_open ( string algorithm, string
algorithm_directory, string mode, string modedirectory)
```

```
    string mcrypt_create_iv ( int size, int source)
```

```
    int mcrypt_get_iv_size ( resource td)
```

```
    int mcrypt_get_key_size ( resource td)
```

```
    int mcrypt_generic_init ( resource td, string key, string iv)
```

```
    string mcrypt_generic ( resource td, string data)
```

```
    bool mcrypt_generic_deinit ( resource td)
```

```
    bool mcrypt_module_close ( resource td)
```

Veremos a continuación, a manera de ejemplo, el código PHP para encriptar y desencriptar una cadena de caracteres:

```
    <?php

    // Cadena de caracteres o string a encriptar
    $cadena = "Contenido de prueba";

    // Llave de encriptación y desencriptación
    $llave = "Esta es nuestra llave secreta";

    // Algoritmo de encriptación
    $cipher_alg = MCRYPT_RIJNDAEL_128;

    // Creación del vector de inicialización
    $iv = mcrypt_create_iv(mcrypt_get_iv_size($cipher_alg,
     MCRYPT_MODE_ECB), MCRYPT_RAND);

    // Imprimir el string original
    print "Cadena original: $cadena <p>";

    // Encriptar $string
    $cadena_encriptada = mcrypt_encrypt($cipher_alg, $llave,
     $cadena, MCRYPT_MODE_CBC, $iv);

    // Convertir a hexadecimal e imprimirlo
```

```
print "Cadena encriptada: " .
 bin2hex($cadena_encriptada)."<p>";

// Desencriptar
$cadena_desencriptada = mcrypt_decrypt($cipher_alg, $llave,
 $cadena_encriptada, MCRYPT_MODE_CBC, $iv);

// Imprimir string desencriptado
print "Cadena desencriptada: $ cadena_desencriptada";

?>
```

Referimos al lector a la sección dedicada a la codificación de criptografía simétrica, dentro de este mismo capítulo, para conocer más acerca del modo de utilización de esta librería.

Veremos, a continuación, cómo podemos confirmar la instalación de la librería y extensión PHP de Mcrypt junto con la disponibilidad de los algoritmos soportados, a partir de la salida de la función `phpinfo()`:

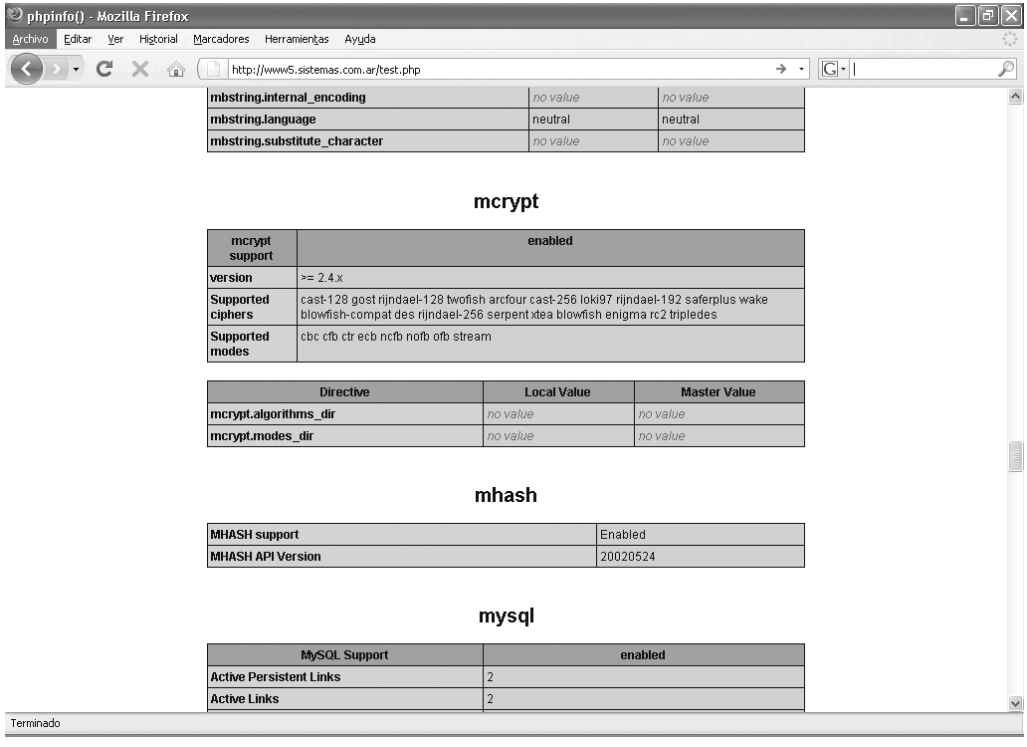

Fig. 5-1. Información respecto del soporte de Mcrypt generada por la función `phpinfo()`.

Mhash

La librería Mhash es una librería libre (bajo la licencia GNU Lesser GPL) que provee una interfaz común o uniforme a un gran número de algoritmos de funciones de una vía o *hashing*. Como hemos visto en apartados anteriores, estos algoritmos pueden ser utilizados para computar verificaciones o *checksums*, resúmenes –verificaciones también– de mensajes o *message digests* y otros tipos de firmas.

El soporte para HMAC implementa lo básico para autenticación de mensajes, siguiendo el estándar RFC 2104. En las últimas versiones, algunos algoritmos generadores de llaves (que utilizan algoritmos de *hashing*) fueron agregados.

Al momento, la librería soporta los siguientes algoritmos: SHA1, SHA160, SHA192, SHA224, SHA384, SHA512, HAVAL128, HAVAL160, HAVAL192, HAVAL224, HAVAL256, RIPEMD128, RIPEMD256, RIPEMD320, MD4, MD5, TIGER, TIGER128, TIGER160, ALDER32, CRC32, CRC32b, WHIRLPOOL, GOST, SNEFRU128 y SNEFRU256.

El código presentado a continuación servirá de ejemplo para entender cómo utilizar la librería.

```
$archivo = 'archivo.txt';
$contenidos = file_get_contents($archivo);

if($contenidos){

  // mhash() aplicará una función de hash especificada por
  // MHASH_MD5 a $contenidos
  $hash_contenido = mhash(MHASH_MD5, $contenidos);

  // Obtener momento actual (Unix timestamp)
  $current = time();

  $salt = $current;
  $password = "Contenido de prueba";

  // mhash_keygen_s2k generará la llave corresponiente a la
  // función de hashing dada y el password o contraseña
  $hash = mhash_keygen_s2k(MHASH_GOST,$password,$salt,20);

  // Se concatena $salt y $hash
  $key = $salt . "|" . bin2hex($hash);

      }

?>
```

Listado 5-1. Ejemplo de utilización de librería Mhash. *Hashing* de un archivo de texto.

Dentro del repositorio público de software *open-source* o de código abierto conocido como *Sourceforge,* se encuentra el sitio que distribuye oficialmente la librería Mhash; en su *homepage* (`http://mhash.sourceforge.net`) vemos una tabla con los algoritmos y sus correspondientes longitudes de salida (expresadas en bits).

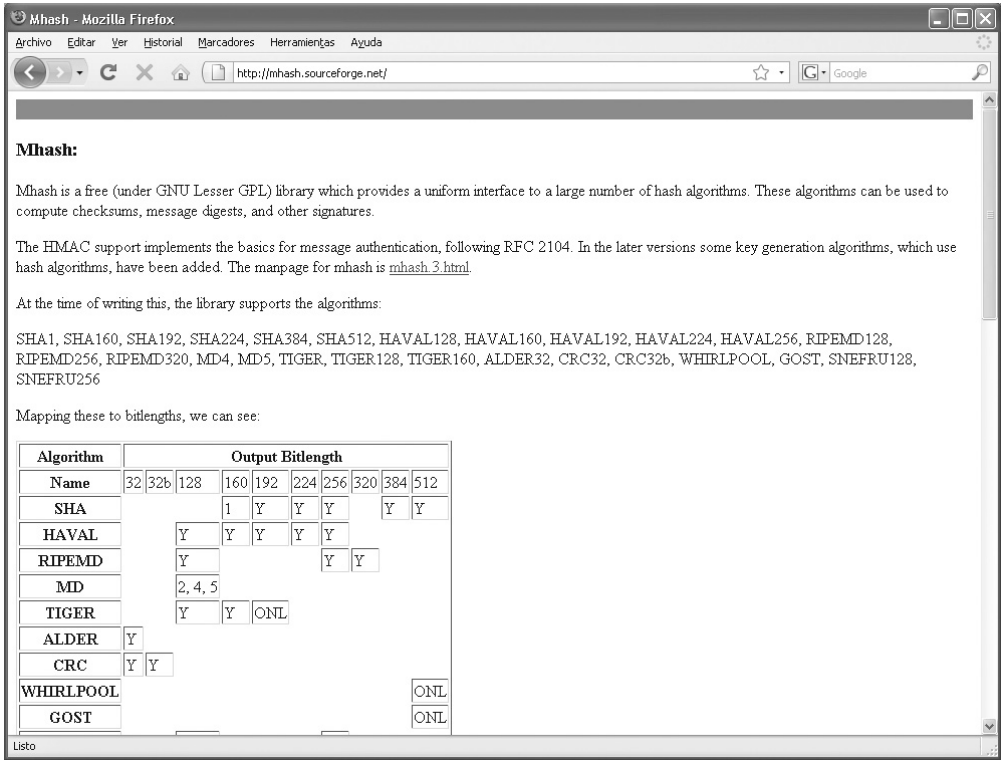

Fig. 5-2. Distribución oficial de la librería Mhash, tabla de algoritmos y longitudes de salida soportadas. `http://mhash.sourceforge.net`.

Referimos al lector a la sección dedicada a la codificación de funciones de una vía o *hashing,* dentro de este mismo capítulo, para conocer más acerca del modo de utilización de esta librería.

Crypt_Blowfish.

Se trata en este caso de la implementación en una extensión, paquete o *package* de PEAR del algoritmo desarrollado por Bruce Schneier. Permite la encriptación utilizando el algoritmo Blowfish, sin requerir la extensión PHP de MCrypt.

Código que ejemplifica el uso de la librería o extensión Crypt_Blowfish:

```
require_once 'Crypt/Blowfish.php';

$archivo = 'archivo.txt';
$contenidos = file_get_contents($archivo);

if($contenidos) {

  $bf = new Crypt_Blowfish('Contenido de prueba');

  // Encriptar los contenidos
  $encriptado = $bf->encrypt($contenidos);
  $archivo_encriptado = @fopen('encriptado.txt', 'w');
  $encriptacion_ok = @fwrite($archivo_encriptado,
   $encriptado);
  if($encriptacion_ok) {
    echo 'Código encriptado en encriptado.txt';
  } else {
    echo 'Error al escribir archivo');
  }
  @fclose($archivo_encriptado);

  // Desencripta los contenidos
  $texto_plano = $bf->decrypt($encriptado);
  $nuevo_archivo = @fopen('nuevo_archivo.txt', 'w');
  $desencriptacion_ok = @fwrite($nuevo_archivo,
   $texto_plano);
  if($desencriptacion_ok) {
    echo 'Código desencriptado en nuevo_archivo.txt';
  } else {
    echo 'Error al escribir archivo');
  }
  @fclose($nuevo_archivo);

}

?>
```

Listado 5-2. Ejemplo de utilización de librería o extensión Crypt_Blowfish. Encriptación y desencriptación de archivos de texto.

Referimos al lector a la sección dedicada a la codificación de criptografía simétrica, dentro de este mismo capítulo, para conocer más acerca del modo de utilización de esta librería.

Crypt_RSA.

Clase PEAR (extensión, paquete o *package*) que provee las funcionalidades necesarias para la generación de llaves, encriptación y desencriptación y verificación de firmas del tipo RSA.

Este paquete permite la utilización de dos llaves de criptografía fuerte, como RSA, con longitudes de llaves arbitrarias.

A manera de ejemplo, a continuación se presenta el código PHP que, utilizando la extensión, genera llaves y encripta y desencripta un archivo de texto:

```
require_once 'Crypt/RSA.php';

// Generar el par de llaves
function generar_key_pair()
{
   global $public_key, $private_key;

   $key_pair = new Crypt_RSA_KeyPair(32);

   // Devuelve la llave pública del par
   $public_key = $key_pair->getPublicKey();

   // Devuelve la llave privada del par
   $private_key = $key_pair->getPrivateKey();
}

// Chequeo de errores
function check_error(&$obj)
{
   if ($obj->isError()){
      $error = $obj->getLastError();
      switch ($error->getCode()) {
      case CRYPT_RSA_ERROR_WRONG_TAIL :
         break;
      default:
         // Mostrar mensaje de error y salir
         echo 'error: ', $error->getMessage();
         exit;
      }
   }
}

$archivo = 'archivo.txt';
```

```
generar_key_pair();
$texto_plano = file_get_contents($archivo);

// Obtener cadena de caracteres que representa la
// llave pública
$llave = Crypt_RSA_Key::fromString(
 $public_key->toString());

$rsa_obj = new Crypt_RSA;
check_error($rsa_obj);

// Encripta $texto_plano utilizando $llave
$encriptado = $rsa_obj->encrypt($texto_plano, $llave);

$archivo_enctriptado = @fopen('encriptado.txt', 'w');
$encriptacion_ok = fwrite($archivo_encriptado,
 $encriptado);
if($encriptacion_ok) {
  echo 'Código encriptado en encriptado.txt';
} else {
 echo 'Error al escribir archivo');
}

@fclose($archivo_encriptado);

$texto_cifrado = $encriptado;

// Obtener cadena de caracteres que representa a la
// llave privada
$llave_privada = Crypt_RSA_Key::fromString(
 $private_key->toString());
check_error($llave_privada);

// Chequeo de encriptación y desencriptación
$rsa_obj->setParams(array('dec_key' => $llave_privada));
check_error($rsa_obj);

// Desencriptar $texto_cifrado
$desencriptado = $rsa_obj->decrypt($texto_cifrado);


$nuevo_archivo = @fopen('nuevo_archivo.txt', 'w');
$desencriptacion_ok = @fwrite($nuevo_archivo,
 $ desencriptado);
if($desencriptacion_ok) {
  echo 'Código desencriptado en nuevo_archivo.txt';
} else {
```

```
        echo 'Error al escribir archivo');
      }
      @fclose($nuevo_archivo);

   ?>
```

Listado 5-3. Ejemplo de utilización de librería Crypt_RSA. Generación de llaves, encriptación y desencriptación de archivos de texto.

Referimos al lector a la sección dedicada a la codificación de criptografía asimétrica, dentro de este mismo capítulo, para conocer más acerca del modo de utilización de esta librería.

Crypt_HMAC.

Se trata en este caso también, de una clase PEAR (extensión, paquete o *package* PHP), que provee la implementación de funciones para calcular HMAC según el estándar RFC 2104.

Ejemplo de la creación de un HMAC utilizando la librería:

```
      require_once 'Crypt/HMAC.php';

      // Crear una llave repitiendo el caracter "0x0b"
      // 20 veces
      $llave = str_repeat(chr(0x0b), 20);

      // Crear una instancia de la clase Crypt_HMAC
      $crypt = new Crypt_HMAC($llave, 'md5');

      // Función de hashing
      echo $crypt->hash("Contenido de prueba").'<br />';

      // Reutilización de la instancia
      $llave = str_repeat(chr(0xaa), 10);
      $data = str_repeat(chr(0xdd), 50);
      //Establecer la llave a usar
      $crypt->setKey($llave);
      echo $crypt->hash($data);

   ?>
```

Listado 5-4. Ejemplo de utilización de librería Crypt_HMAC. Generación de un HMAC.

Crypt_DiffieHellman.

Implementación del algoritmo criptográfico o protocolo de Diffie-Hellman para el intercambio de llaves. La librería ha sido encapsulada en una clase PEAR (extensión, paquete o *package* PHP) para PHP5.

Permite a las dos partes involucradas en la comunicación, sin ningún conocimiento previo de cada parte, establecer una llave secreta compartida segura, a través de un canal inseguro.

Veremos ahora el código PHP que utiliza esta extensión para generar una llave secreta entre dos partes que se desconocen:

```
       require_once 'Crypt/DiffieHellman.php';

       // Establecer las opciones requeridas para dos
       // subjects o sujetos
       $subject_1 = array('prime'=>'123', 'generator'=>'7', 'private'=>'3');
       $subject_2 = array('prime'=>'123', 'generator'=>'7',
'private'=>'34');

       //Aplicar algoritmo Diffie Hellman
       $subject_1_GK = new Crypt_DiffieHellman(
          $subject_1['prime'], $subject_1['generator'],
          $subject_1['private']);
       $subject_2_GK = new Crypt_DiffieHellman(
          $subject_2['prime'], $subject_2['generator'],
          $subject_2['private']);

       // Generar llaves
       $subject_1_GK->generateKeys();
       $subject_2_GK->generateKeys();

       // Computar las llaves secretas
       $subject_1_SK = $subject_1_GK->computeSecretKey(
          $subject_2_GK->getPublicKey())->getSharedSecretKey();
       $subject_2_SK = $subject_2_GK->computeSecretKey(
          $subject_1_GK->getPublicKey())->getSharedSecretKey();

       // Mostrar las llaves secretas

       echo('Subject_1_SK:'.$subject_1_SK.'<br />');
       echo('Subject_2_SK:'.$subject_2_SK);
    ?>
```

Listado 5-5. Ejemplo de utilización de librería Crypt_DiffieHellman. Generación de llave secreta.

El siguiente ejemplo realiza las mismas operaciones, pero en modo binario:

```
   require_once 'Crypt/DiffieHellman.php';

   // Establecer las opciones requeridas para dos
   // subjects o sujetos
   $subject_1 = array('prime' =>
      '95680945580498983409350983490 53',
      'generator'=>'2',
      'private' =>
       '22323702772376288232792737237428722893987 23');
   $subject_2 = array('prime' =>
      '95680945580498983409350983490 53',
      'generator'=>'2',
      'private' =>
       '03892372887213239874298343892982324333634 63');

   // Aplicar algoritmo Diffie Hellman
   $subject_1_GK = new Crypt_DiffieHellman(
      $subject_1[ 'prime'], $subject_1[ 'generator'],
      $subject_1[ 'private']);
   $subject_2_GK = new Crypt_DiffieHellman(
      $subject_2[ 'prime'], $subject_2[ 'generator'],
      $subject_2[ 'private']);

   // Generar llaves
   $subject_1_GK->generateKeys();
   $subject_2_GK->generateKeys();

   // Computar las llaves secretas usando modo BINARY
   $subject_1_SK = $subject_1_GK->computeSecretKey(
   $subject_2_GK->getPublicKey(
   Crypt_DiffieHellman::BINARY),
   Crypt_DiffieHellman::BINARY)->
   getSharedSecretKey(Crypt_DiffieHellman::BINARY);
   $subject_2_SK = $subject_2_GK->computeSecretKey(
      $subject_1_GK->getPublicKey(
       Crypt_DiffieHellman::BINARY),
       Crypt_DiffieHellman::BINARY)->
       getSharedSecretKey(Crypt_DiffieHellman::BINARY);

   // Mostrar las llaves secretas
   echo('subject_1_SK:'.$subject_1_SK.'<br />');
   echo('subject_2_SK:'.$subject_2_SK.'<br />');
?>
```

Listado 5-6. Ejemplo de utilización de librería Crypt_DiffieHellman. Generación de llave secreta - modo binario.

Referimos al lector a la sección dedicada a la codificación de criptografía asimétrica, dentro de este mismo capítulo, para conocer más acerca del modo de utilización de esta librería.

Codificación de encriptación simétrica.

El lenguaje PHP no provee funciones incorporadas para el cifrado simétrico de información, salvo por la función `str_rot13()`. Si bien no se utilizan contraseñas, pudiéndolo considerar como un mero proceso de codificación, implementa una sustitución elemental de caracteres, cuya salida comúnmente es referida como datos o información "encriptada" que puede "desencriptarse". Como hemos aclarado en la sección introductoria, este algoritmo se mantiene disponible en sistemas y lenguajes de programación o interpretados por razones históricas y por supuesto, no debe ser considerado seguro bajo ninguna circunstancia.

Aclarado lo anterior, nos referiremos ahora a la opciones disponibles en librerías o *frameworks* adicionales para la codificación de encriptación simétrica en PHP (se recomienda haber leído los apartados MCrypt y Crypt_Blowfish de esta misma sección).

Comenzaremos describiendo el proceso de encriptación simétrica, utilizando la librería MCrypt vista en la sección anterior. La librería Mcrypt puede ser utilizada para encriptar y desencriptar, usando diferentes algoritmos de cifrado.

Si se está utilizando la versión libmcrypt-2.2.X, las funciones importantes serán `mcrypt_cfb()`, `mcrypt_cbc()`, `mcrypt_ecb()` y `mcrypt_ofb()`, que pueden operar en ambos modos, llamados o definidos como `MCRYPT_ENCRYPT` y `MCRYPT_DECRYPT`.

Veamos a continuación un ejemplo para cifrar una cadena de entrada, o texto-plano, con el algoritmo TripleDES para la versión 2.2.X en modo ECB.

```
<?php

$llave = "Llave secreta";
$entrada = "Contenido de prueba";

$encriptado = mcrypt_ecb (MCRYPT_3DES, $llave, $entrada,
 MCRYPT_ENCRYPT);

?>
```

De esta manera, entonces, la variable `$encriptado` contendrá la información cifrada.

Si, en cambio, se ha utilizado la versión libmcrypt 2.4.X o 2.5.X, estas funciones estarán disponibles, pero se recomienda el uso de las funciones avanzadas.

El ejemplo a continuación mostrará cómo realizar la misma tarea (encriptación con algoritmo TripleDES de una cadena de caracteres en modo ECB) para las versiones de libmcrypt 2.4.X y posteriores.

```
<?php

$llave = "Llave secreta";
$entrada = "Contenido de prueba";

$td = mcrypt_module_open('tripledes', '', 'ecb', '');
$iv = mcrypt_create_iv (mcrypt_enc_get_iv_size($td),
 MCRYPT_RAND);
mcrypt_generic_init($td, $llave, $iv);
$encriptado = mcrypt_generic($td, $entrada);
mcrypt_generic_deinit($td);
mcrypt_module_close($td);

?>
```

Como vemos en el ejemplo, quizá la función principal, utilizada para seleccionar el algoritmo y el modo de operación, es `mcrypt_module_open()`. Veamos su definición:

```
    resource mcrypt_module_open ( string $algorithm, string
$algorithm_directory, string $mode, string $mode_directory)
```

Veamos algunos ejemplos concretos de su forma de utilización.

```
<?php

$td = mcrypt_module_open(MCRYPT_DES, '', MCRYPT_MODE_ECB,
 '/usr/lib/mcrypt-modes');

$td = mcrypt_module_open('rijndael-256', '', 'ofb','');

?>
```

La primera línea en este código de ejemplo intentará abrir el cifrador DES desde el directorio o carpeta por defecto y el modo EBC desde el directorio `/usr/lib/mcrypt-modes`.

El segundo ejemplo –la segunda línea– utiliza cadenas de caracteres o *strings* para determinar el cifrador y el modo que se ha de usar; esto sólo funciona cuando la extensión está utilizando (entiéndase que está enlazada) la librería libmcrypt versión 2.4.X o 2.5.X.

Por último, observemos otro ejemplo del uso de `mcrypt_module_open()`, para la posterior encriptación de información.

```
// Abrir el cifrador

$td = mcrypt_module_open('rijndael-256', '', 'ofb', '');

// Crear el IV y determinar la longitud de la llave
// Nota: utilícese MCRYPT_RAND en Windows
$iv = mcrypt_create_iv(mcrypt_enc_get_iv_size($td),
 MCRYPT_DEV_RANDOM);

$ks = mcrypt_enc_get_key_size($td);

// Crear llave
$llave = substr(md5('Llave secreta'), 0, $ks);

// Inicializar encriptación
mcrypt_generic_init($td, $llave, $iv);

// Encriptar
$encriptado = mcrypt_generic($td, "Contenido de prueba");

// Finalizar el 'handler'
mcrypt_generic_deinit($td);

// Inicializar el módulo de encriptación para desencriptar
mcrypt_generic_init($td, $llave, $iv);

// Desencriptar el string encriptado
$desencriptado = mdecrypt_generic($td, $ encriptado);

// Finalizar el 'handler' de desencriptación y cerrar el
// módulo
mcrypt_generic_deinit($td);
mcrypt_module_close($td);

// Mostrar string
echo trim($desencriptado) . "\n";

?>
```

Listado 5-7. Ejemplo de utilización de la función `mcrypt_module_open()` de librería Mcrypt. Encriptación de información.

Otra alternativa a nuestra disposición es utilizar la extensión Crypt_Blowfish (ver sección anterior). La extensión Crypt_Blowfish permite una rápida encriptación de dos vías (simétrica), utilizando el algoritmo Blowfish desarrollado por Bruce Schneier. La encriptación se realiza puramente en PHP, por lo que la extensión no requiere la librería y extensión Mcrypt para ser utilizada.

Veamos a continuación un ejemplo de utilización.

```
<?php

$bf = new Crypt_Blowfish();
$encriptado = $bf->encrypt("Contenido de prueba");
$texto_plano = $bf->decrypt($encriptado);

?>
```

El ejemplo anterior demuestra su utilización básica; por supuesto, la extensión Crypt_Blowfish también puede encriptar basándose en una llave especificada.

```
<?php

$bf = new Crypt_Blowfish('Llave secreta');
$encriptado = $bf->encrypt("Contenido de prueba");
$texto_plano = $bf->decrypt($encriptado);

?>
```

Por último, referiremos un ejemplo de la utilización de esta extensión basada en los tutoriales de PEAR.

```
<?php

$bf =& Crypt_Blowfish::factory('cbc');
if (PEAR::isError($bf)) {
   echo $bf->getMessage();
   exit;
}
$iv = 'abc123+=';
$llave = 'Llave secreta';
$bf->setKey($llave, $iv);
$encriptado = $bf->encrypt("Contenido de prueba");
$bf->setKey($llave, $iv);
$texto_plano = $bf->decrypt($encriptado);
if (PEAR::isError($texto_plano)) {
   echo $texto_plano->getMessage();
   exit;
```

```
}
// El texto a encriptar es 'padeado' o completado previa
// encriptación, por lo que es preciso utilizar trim()
echo 'texto plano: ' . trim($texto_plano);

?>
```

Listado 5-8. Ejemplo de utilización de la extensión o librería Crypt_Blowfish. Encriptación de una cadena de caracteres.

Codificación de encriptación asimétrica.

PHP no dispone de funciones incorporadas para la codificación de funcionalidades criptográficas asimétricas o de llave pública. Aunque, por supuesto, como se ha adelantado en los apartados anteriores, referidos a extensiones o *packages* que implementan RSA o Diffie-Hellman, dentro de esta misma sección, es posible acceder a una variedad de alternativas mediante la incorporación de librerías adicionales.

Retomando lo referido sobre Crypt_RSA en este capítulo, se describirá a continuación la extensión, paquete o *package* de PEAR Crypt_RSA (clase principal y complementarias), que provee las funcionalidades necesarias para la generación de llaves, encriptación y desencriptación y verificación de firmas del tipo RSA.

Concentrándonos entonces en la clase Crypt_RSA, veremos en principio la definición de su constructor:

```
    Crypt_RSA Crypt_RSA( [ mixed $params = null], [ mixed $wrapper_name =
'default'], [ string $error_handler = ''], array $params , string
$wrapper_name )
```

Todos los parámetros son opcionales. El primero, `$params`, corresponde a un *array* asociativo para especificar, por ejemplo, `enc_key`, `dec_key`, `private_key`, `public_key` y `hash_func`. Veremos luego, en la función o método `setParams()`, cuáles son todas las opciones o parámetros que podríamos especificar en este *array*. `$wrapper_name` será el nombre del *wrapper* o envoltura, entendido en este caso como algo que encapsula o normaliza el acceso a las funciones matemáticas correspondientes, para realizar operaciones con números enteros grandes. Por último, `$error_handler` corresponderá al nombre de la función utilizada para manejar los errores.

Veamos ahora una lista de funciones o métodos que el objeto provee:

`setParams($params)`

Establece parámetros para el objeto en uso (instancia).

A través del *array* asociativo se podrán especificar los siguientes parámetros:

- `enc_key` : Llave utilizada para la encriptación en el método `encrypt()`.
- `dec_key` : Llave utilizada para la desencriptación en el método `decrypt()`.
- `public_key` : Llave que será utilizada por el método `validateSign()`.
- `private_key` : Llave que será utilizada por el método `createSign()`.
- `hash_func` : Nombre de la función que será utilizada para crear y validar una firma.

`encrypt($plain_data, $key = null)`

Utilizada para la encriptación de información.

Encriptará `$plain_data` con la llave `$this->_enc_key` o `$key`.

`decrypt($enc_data, $key = null)`

Utilizada para la desencriptación de información cifrada.

Desencriptará `$enc_data` con la llave `$this->_dec_key` o `$key`.

`createSign($doc, $private_key = null)`

Se utilizará para firmar un documento con una llave privada.

Creará la firma para el documento `$document`, utilizando `$this->_private_key` o `$private_key` como llave privada y `$this->_hash_func` o `$hash_func` como función de *hashing*.

`validateSign($doc, $signature, $public_key = null)`

Validar o verificar la firma de un documento.

Validará o verificará la firma `$signature` para el documento `$document` con la llave pública `$this->_public_key` o `$public_key` y la función de *hashing* `$this->_hash_func` o `$hash_func`.

Veamos ahora algunos ejemplos de código de uso concreto de estas funciones para generar llaves, encriptar y desencriptar datos y firmar documentos.

```
    <?php

    require_once 'Crypt/RSA.php';

    // Crear la función para el manejo de errores
    $error_handler    =    create_function('$obj',    'echo    "error:    ",
      $obj->getMessage(), "\n"');

    // Generar el par de llaves de 1024-bit
    $key_pair = new Crypt_RSA_KeyPair(1024);

    // Chequear consistencia de la instancia Crypt_RSA_KeyPair
    $error_handler($key_pair);

    // Crear instancia Crypt_RSA
    $rsa_obj = new Crypt_RSA;

    // Chequear consistencia
    $error_handler($rsa_obj);

    // Establecer la función para el manejo de errores para
    // la instancia de Crypt_RSA
    $rsa_obj->setErrorHandler($error_handler);

    // Encriptar (usualmente utilizando llave pública)
    $enc_data = $rsa_obj->encrypt($plain_data,
     $key_pair->getPublicKey());

    // Desencriptar (usualmente utilizando llave privada)
    $plain_data = $rsa_obj->decrypt($enc_data,
     $key_pair->getPrivateKey());

    // Firmar
    $signature = $rsa_obj->createSign($document,
     $key_pair->getPrivateKey());

    // Chequear firma
    $is_valid     =     $rsa_obj->validateSign($document,     $signature,
$key_pair->getPublicKey());

    // Firma de más de un documento usando la llave privada
    $rsa_obj = new Crypt_RSA(array('private_key' =>
     $key_pair->getPrivateKey()));

    // Chequear consistencia
    $error_handler($rsa_obj);

    // Establecer función para el manejo de errores
    $rsa_obj->setErrorHandler($error_handler);
```

```
    // Firmar multiples documentos
    $sign_1 = $rsa_obj->sign($doc_1);
    $sign_2 = $rsa_obj->sign($doc_2);
    //... $sign_n = $rsa_obj->sign($doc_n);

    // Cambiar a función de hashing por defecto, la cual es
    // usada para firmar y validar
    $rsa_obj->setParams(array('hash_func' => 'md5'));

    // Alternativa utilizando método factory() method en lugar
    // del constructor
    $rsa_obj = &Crypt_RSA::factory(); if (PEAR::isError($rsa_obj)) { echo
"error: ", $rsa_obj->getMessage(), "\n"; }

    ?>
```

Listado 5-9. Ejemplo de utilización de la extensión o librería Crypt_RSA. Métodos del objeto Crypt_RSA.

Como vemos en el código de ejemplo, existen, además del objeto o clase principal `Crypt_RSA`, las clases complementarias `Crypt_RSA_Key` y `Crypt_RSA_KeyPair`; detallemos sus métodos para conocer cómo y para qué las utilizaríamos:

Comencemos por los métodos de la clase `Crypt_RSA_Key`:

- `getKeyLength()`: Retorna la longitud de la llave (cantidad de bits).
- `getExponent()`: Retorna el exponente de la llave (binario).:
- `getModulus()`: Retorna el módulo de la llave (binario).
- `getKeyType()`: Retorna el tipo de la llave (pública o privada).
- `toString()`: Retorna la llave como una cadena de caracteres (“serializada”).
- `fromString($key_str)`: Función estática que retorna la llave “de-serializada” a partir de una cadena de caracteres –parámetro `$key_str`–.
- `isValid($key)`: Función estática utilizada para validar la llave (parámetro `$key`).

En la clase `Crypt_RSA_KeyPair` disponemos de los siguientes métodos:

- `generate($key)`: Función utilizada para generar un nuevo par de llaves.
- `getPublicKey()`: Retorna la llave pública.
- `getPrivateKey()`: Retorna la llave privada.
- `getKeyLength()`: Retorna la longitud de la llave (cantidad de bits).
- `setRandomGenerator($func_name)`: Establece, como generador de números aleatorios, la función parámetro `$func_name`.

- `fromPEMString($str)`: Obtiene el par de llaves a partir de una cadena de caracteres (parámetro `$str`), codificada en PEM (*Privacy Enhanced Mail*).
- `toPEMString()`: Codifica en PEM (*Privacy Enhanced Mail*) el par de llaves.
- `isEqual($keypair2)`: Compara el par de llaves actual contra `$keypair2`.

Codificación de funciones de una vía y *hash*.

A diferencia de los casos anteriores –para la encriptación simétrica y asimétrica– en donde no teníamos herramientas o funcionalidades incorporadas por defecto, la distribución estándar del lenguaje provee quizá todo lo que podríamos necesitar respecto de las funciones de una vía o *hashing*.

Repasando lo visto en el primer apartado de esta sección, sabemos que a partir de la versión 4.0 del lenguaje disponemos de la función `md5()`, que nos permite generar un *hash* a partir de un *string.* La función también devuelve una cadena de caracteres, con el resultado expresado en hexadecimal. El tamaño de esta cadena será de 32 caracteres, ya que el resultado MD5 consta de 16 bytes, para cada uno de los cuales son necesarios dos caracteres hexadecimales.

La definición de la función es la siguiente:

```
string md5 ( string $cadena [ , bool $modo_binario ] )
```

Como hemos visto entonces, para la generación del *hash* criptográfico MD5 de una cadena de caracteres, asignarlo a la variable `$hash` e imprimirlo, codificaríamos:

```
<?php

$cadena = "Contenido de prueba";
$hash = md5($cadena);
echo "El hash MD5 para la cadena '$cadena' es: $hash";

?>
```

Esto imprimirá lo siguiente:

```
El hash MD5 para la cadena "Contenido de prueba" es:
17603a7adfa88737d21c1aded1753fd3
```

Para ejemplificar el uso del `modo_binario` –sólo disponible a partir de la versión 5– generaremos el mismo *hash* y lo imprimiremos sirviéndonos de la función PHP

`bin2hex()`. Téngase en cuenta que el modo binario ha de ser útil en el caso de que necesitemos, por ejemplo, registrar en base de datos el resultado de la función en modo binario; en tal caso, necesitaríamos un campo de 16 bytes –binario, por supuesto–, y no 32 de bytes para la cadena de caracteres, que representa el número en formato hexadecimal. Como veremos, el resultado obtenido a partir de la siguiente prueba será el mismo:

```
<?php

$cadena = "Contenido de prueba";
$hash_binario = md5($cadena, TRUE);
echo "El hash MD5 para la cadena '$cadena' es: ". bin2hex($hash_binario);

?>
```

Código que, de acuerdo con lo esperado, genera un resultado idéntico al anterior:

```
El hash MD5 para la cadena "Contenido de prueba" es:
17603a7adfa88737d21c1aded1753fd3
```

Siguiendo con lo visto al comienzo de este capítulo, nos ocuparemos ahora de la función para la generación de *hashes* mediante el algoritmo SHA-1 llamada, convenientemente, `sha1()`. Nótese que la función se encuentra disponible a partir de la versión 4.3 y el parámetro opcional `$modo_binario`, como con `md5()`, a partir de la versión 5 del lenguaje.

Veamos nuevamente su definición:

```
string sha1 ( string $cadena [ , bool $modo_binario ] )
```

Siendo su utilización muy similar a la de la función md5(), nos ocuparemos de notar sus diferencias. Implementa, por supuesto, el algoritmo SHA-1 en lugar de MD5. Esto significa que la salida o resultado del algoritmo se trata, en este caso, de una cadena de 40 caracteres de longitud, representa en hexadecimal los 20 bytes generados por el algoritmo.

Veamos un ejemplo:

```
<?php

$cadena = "Contenido de prueba";
$hash = sha1($cadena);
echo "El hash SHA-1 para la cadena '$cadena' es: $hash");

?>
```

El resultado obtenido es:

```
    El hash SHA-1 para la cadena "Contenido de prueba" es:
f5d240c860be1a48f01348b01cb55c7577c8e46a
```

Hasta aquí hemos hablado de las funciones incorporadas al lenguaje en su distribución estándar. Como hemos visto sobre Mhash en este capítulo, tenemos la opción de utilizar, como extensión o agregado, la librería Mhash. Ésta provee una interfaz común o uniforme a un gran número de algoritmos de funciones de una vía o *hashing*, como así también provee el soporte para HMAC para la autenticación de mensajes, siguiendo el estándar RFC 2104.

Al momento, la librería soporta los siguientes algoritmos: SHA1, SHA160, SHA192, SHA224, SHA384, SHA512, HAVAL128, HAVAL160, HAVAL192, HAVAL224, HAVAL256, RIPEMD128, RIPEMD256, RIPEMD320, MD4, MD5, TIGER, TIGER128, TIGER160, ALDER32, CRC32, CRC32b, WHIRLPOOL, GOST, SNEFRU128 y SNEFRU256.

Como vemos en la lista de algoritmos soportados, la utilización de esta librería nos daría muchas más opciones (además de MD5 y SHA-1, provistos por PHP) a la hora de implementar generación de *hashes*.

A manera de ejemplo, generaremos nuevamente, a partir de la cadena de caracteres "Contenido de prueba", diferentes resultados de *hashing* utilizando diferentes algoritmos.

```
<?php

$cadena = "Contenido de prueba";

$hash_binario_md5 = mhash(MHASH_MD5, $cadena);
echo "El hash MD5 para la cadena '$cadena' es: ".
  bin2hex($hash_binario_md5);

$hash_binario_sha1 = mhash(MHASH_SHA1, $cadena);
echo "El hash SHA-1 para la cadena '$cadena' es: ".
  bin2hex($hash_binario_sha1);

$hash_binario_sha192 = mhash(MHASH_SHA192, $cadena);
echo "El hash SHA-192 para la cadena '$cadena' es: ".
  bin2hex($hash_binario_sha192);

$hash_binario_tiger160 = mhash(MHASH_TIGER160, $cadena);
echo "El hash TIGER160 para la cadena '$cadena' es: ".
  bin2hex($hash_binario_tiger160);

$hash_binario_haval192 = mhash(MHASH_HAVAL192, $cadena);
echo "El hash HAVAL192 para la cadena '$cadena' es: ".
  bin2hex($hash_binario_haval192);

?>
```

Listado 5-10. Ej. de utilización de la librería Mhash. Generación de *hashes* usando diferentes algoritmos.

Como regla general, podemos decir que la constante que determina el algoritmo que se ha de utilizar se compone del literal `MHASH_` seguido del nombre del algoritmo. Se recomienda verificar las opciones disponibles en la documentación de Mhash de la versión correspondiente; sin embargo, se refiere aquí la lista de ejemplo obtenida del manual PHP *online* [Consulta: Noviembre 2008].

- `MHASH_ADLER32`
- `MHASH_CRC32`
- `MHASH_CRC32B`
- `MHASH_GOST`
- `MHASH_HAVAL128`
- `MHASH_HAVAL160`
- `MHASH_HAVAL192`
- `MHASH_HAVAL256`
- `MHASH_MD4`
- `MHASH_MD5`
- `MHASH_RIPEMD160`
- `MHASH_SHA1`
- `MHASH_SHA256`
- `MHASH_TIGER`
- `MHASH_TIGER128`
- `MHASH_TIGER160`

Se recomienda visitar el sitio `http://md5.rednoize.com/` para acceder a una base de datos de "resultados" o *hashes* generados a partir de diferentes *strings* o cadenas de caracteres; este tipo de base de datos son las utilizadas a la hora de intentar descifrar o recuperar el texto plano original, por ejemplo, de una contraseña cifrada con MD5 o SHA-1. No se trata exactamente de un ataque por fuerza bruta, ya que los resultados no se generan para compararse especialmente, sino que se encuentran registrados *a priori* en la base de datos.

Figura 5-3. Sitio Web `http://md5.rednoize.com`.

Codificaciones de casos prácticos.

Identificación.

Veremos en este apartado una funcionalidad típica y elemental de las aplicaciones Web, que tiene un punto de contacto con las funciones criptográficas de una vía o *hashing*. Se trata de la identificación de usuarios o *login*. Muchos sitios de Internet fomentan el registro de información personal para acciones de marketing, como ser el envío de información respecto de sus productos y servicios. Nos interesa aquí lo referente al control de acceso o identificación de usuarios, que permitirán el uso de secciones privadas del sitio (acceso al correo electrónico en el caso de un *Web-mail*, a información de la cuenta en caso de un sistema de *e-banking* o banca electrónica o al historial de compras en el caso de una tienda *online*).

Comenzaremos analizando el formulario HTML que visualizará los campos para completar (nombre de usuario y contraseña):

```
<form action="identificacion.php" method="post">
<p><label for='nombre'>Nombre de usuario</label>
<input type='text' name='nombre' id='nombre' />
</p>
<p><label for='contrasena'>Contraseña</label>
<input type='text' name='contrasena' id='contrasena' />
</p>
<p><input type="submit" name="submit" value="Ingresar"/>
</p>
</form>
```

Y aquí veremos el código PHP que procesará esos campos para permitir o no el acceso, al consultar la base de datos, comparando la contraseña ingresada en el formulario HTML con la versión de su *hash* registrada:

```
<?php
$nombre  = $_POST['nombre'];
$contrasena = $_POST['contrasena'];

$contrasena_md5 = md5($contrasena);
$sql = "select nombre, contrasena from usuarios
   where nombre = '$nombre'
   and contrasena = '$contrasena_md5'";
$resultado = mysql_query($sql);

if (mysql_num_rows($resultado)){
```

```
            // hemos encontrado al usuario
    } else{
            // información incorrecta, o no encontrada en base de datos
    }
?>
```

Transferencia segura de parámetros.

La técnica que se describirá a continuación es utilizada para proteger el pasaje de parámetros a través de vínculos o URLs en aplicaciones Web y está basada en la utilización de *hashes*. La técnica aplica a los pasajes entre invocaciones o referencias a páginas dentro de un mismo sitio, hacia otros sitios y, también, para generación de vínculos que serán enviados por *e-mail*.

El problema o tipo de ataque que combatimos con esto es denominado *Web parameter tampering* o manipulación de parámetros Web. Consta de la alteración de parámetros (credenciales de usuario, identificadores de registros, precios, etc.) en la comunicación entre el cliente y el servidor.

Veamos un ejemplo concreto para ilustrar el problema. Imaginemos que nuestra aplicación Web deberá manejar la baja voluntaria de usuarios registrados enviando una confirmación por *e-mail*. Sigamos el proceso paso a paso. El usuario, una vez identificado dentro del sitio, tendrá la opción de –por ejemplo– presionar un botón llamado "eliminar mi cuenta". Al hacerlo, el sistema entonces enviará un *e-mail* donde se informará acerca de la solicitud de baja y proveerá un vínculo, dentro del mismo mensaje, para que el usuario confirme su decisión.

Para evidenciar el problema, propondremos la siguiente implementación vulnerable: Crearemos un archivo PHP, "confirmar_eliminacion.php", que tomará por parámetro GET al valor del identificador (ID) de usuario. Así, entonces, al momento de enviar el *e-mail* para el usuario cuyo identificador sea, por ejemplo, el número 45, la aplicación lo confeccionará con el siguiente vínculo:

```
http://.../aplicacion/confirmar_eliminacion.php?ID=45
```

Como vemos, quien reciba el *e-mail* y detecte que el *script* eliminará la cuenta de usuario, cuyo identificador sea el número 45, podrá utilizar la misma URL para eliminar otras cuentas, cambiando únicamente el número, su número o identificador, por cualquiera de otro usuario.

El problema entonces se resume a que sabemos que debemos enviar por *e-mail* la información que determine qué usuario, en caso de confirmarlo, esto es, de seguir el vínculo, será dado de baja. Sabemos también que deberíamos contemplar el uso indebido o abuso de la aplicación, al cual nos expondríamos en el caso de llevar adelante una implementación insegura como la planteada anteriormente.

Como han de imaginarse, disponemos del auxilio de la criptografía, concretamente, mediante la utilización de funciones de una vía. La técnica consiste en generar un verificador, o *checksum,* de la información codificada en el vínculo, de manera tal que el *script* PHP "confirmar_eliminación.php" pueda verificar que el número identificador no ha sido alterado. Se resuelve esto entonces, para nuestro caso particular, enviando dos parámetros. El ID o identificador de usuario y su verificación, que realizaremos mediante MD5. Esta verificación o *hash* no será únicamente sobre el número, sino sobre la concatenación del número y una clave secreta. Al enviar el *e-mail*, el vínculo será calculado de la siguiente manera:

```
$clave_secreta = 'WrV3dZ';
$idu = $_POST['idu'];
// checksum
$cs = md5("$idu||$clave_secreta");
$link =  "http://…/aplicacion/confirmar_eliminacion.php? ".
 idu=$idu&cs=$cs";
```

Como vemos, el vínculo seguirá conteniendo el número, pero la verificación dependerá de éste y de una clave secreta (inaccesible a quien recibe el *e-mail,* dadas las características de las funciones de una vía). El cálculo verificador será realizado en "confirmar_eliminacion.php" de la misma forma y podrá ser comparado con el valor "cs" recibido. En el caso de que no coincidan, sabremos que el usuario ha alterado el número parámetro.

Nota:

[1] En sus orígenes PHP era por *Personal Home Page*, según Rasmus Lerdorf, su creador. Luego, según el acrónimo recursivo GNU: GNU's Not UNIX, se convirtió en PHP: *Hypertext Preprocessor.*

6

Criptografía en bases de datos

Los motores de bases de datos relacionales más utilizados actualmente proveen, en mayor o menor cantidad en cada caso, implementaciones de diversas funciones criptográficas.

De la misma manera en que incorporan funciones generales como las referentes al manejo de cadenas de caracteres, fechas o funciones matemáticas, desde el lenguaje SQL pueden invocarse funciones para la generación de *hashes* o descifrar un campo de un registro que ha sido cifrado al almacenarse.

Nos concentraremos en este capítulo en los motores de base de datos relacionales actualmente disponibles (quizá los más populares) y en particular, en las versiones actuales y no tan actuales, pero que años atrás ya implementaban una variedad importante de funciones de criptografía. Los servidores que trataremos entonces son: Microsoft SQL Server (versiones 2000, 2005, 2008) de Microsoft Corporation, Oracle (versiones 9i, 10i y 11i) de Oracle Corporation y, por último, MySQL (versiones 3.23, 4.1, 5.0 y 6.0.1) de MySQL AB.

Utilizaremos aquí en mayor medida el término motor aunque, rigurosamente, les cabe a estos servidores la clasificación de gestores. Los sistemas de gestión de base de datos (SGBD) relacionales (RDBMS, siglas de *Relational DataBase Management System*) son sistemas dedicados a suplir servicios de interfaz entre la base de datos (información almacenada) y las aplicaciones y/o usuarios que la utilizan.

Una base de datos relacional almacena datos en tablas independientes, lo que permite obtener una gran velocidad y flexibilidad en el tratamiento de la información almacenada. La parte SQL, por las siglas de *Structured Query Language* que comparte estos motores o gestores de bases de datos que trataremos, se refiere al lenguaje estandarizado para acceder a este tipo de bases de datos y está definido por el estándar ANSI/ISO SQL. El estándar SQL ha evolucionado desde 1986 y existen varias versiones. Por ejemplo, “SQL-92” se refiere al estándar de 1992, “SQL:1999” se refiere a la versión de 1999 y “SQL:2003” se refiere a la versión actual del estándar.

Al margen de lo que acabamos de referir respecto del estándar, hemos de notar que los diferentes motores de bases de datos no mantienen un conjunto de funciones comunes para algunas tareas específicas. Por ejemplo, para obtener la fecha y hora actual del motor de base de datos existen diferentes alternativas: En MySQL usaríamos la función `NOW()`, en Oracle el literal `SYSDATE` y en Microsoft SQL Server la función `GETDATE()`.

Notamos estas diferencias porque veremos más adelante que, respecto de las funciones de criptografía, también tendremos un conjunto de funciones particular para cada motor de base de datos. Si bien, por ejemplo, para las funciones de *hashing*, todos los motores implementan el algoritmo de *hashing* criptográfico SHA-1, cada cual llamó a su función de manera diferente.

Microsoft SQL Server.

El motor SQL producido por Microsoft Corporation, Microsoft SQL Server, es un sistema de gestión de bases de datos (SGBD) relacionales, como hemos descripto anteriormente, que implementa el lenguaje Transact-SQL.

Como algunas de las características principales de Microsoft SQL Server podrían destacarse las siguientes:

- Soporte de transacciones.
- Capacidad de escalar, estabilidad y seguridad.
- Soporta procedimientos almacenados.
- Incluye entorno gráfico de administración, que permite el uso de comandos DDL y DML gráficamente.
- Permite trabajar en modo cliente-servidor, donde la información y datos se alojan en el servidor y las terminales o clientes de la red sólo acceden a la información.
- Permite administrar información de otros servidores de datos.

Ya desde la disponibilidad de la versión 2000, Microsoft establecía que si bien SQL Server fue diseñado para trabajar como el motor de *data storage* (almacenamiento de datos) para miles de usuarios concurrentes conectados a través de una red, también es capaz de trabajar como una base de datos *stand-alone* (independiente), directamente sobre el mismo equipo sobre el que funcionará una aplicación. Las caracteristicas de escalabilidad y facilidad de uso de SQL Server le permiten trabajar eficientemente en una computadora de manera independiente, sin consumir recursos en exceso o requiriendo mayor trabajo administrativo por el usuario individual. Las mismas características le permiten a SQL Server adquirir dinámicamente los recursos necesarios para soportar a miles de usuarios, mientras minimiza la administración y

necesidad de *tunnig* (configuración específica) de la base de datos. El motor relacional de la base de datos realiza automáticamente el *tunnig* por sí mismo para adquirir o liberar los recursos de sistema apropiados para soportar una carga variable de usuarios, accediendo a una instancia de SQL Server en cada momento en particular. Este mismo motor tiene la propiedad de prevenir problemas lógicos que pudiesen ocurrir si un usuario intentara leer o modificar datos concurrentemente utilizados por otros usuarios.

La arquitectura de Microsoft SQL Server está principalmente dividida en tres componentes: SQLOS, que implementa los servicios básicos requeridos por el servidor SQL (incluidos el manejo de *threads* –hilos– el de memoria y el de I/O –entrada/salida–), el motor relacional (que implementa los componentes del sistema de base de datos relacional, tablas, *querys* –o consultas– y *stored procedures* –o procesos almacenados–) y la capa de protocolo, que expone la funcionalidad del servidor SQL.

Con respecto a la historia de este motor de base de datos producido por Microsoft Corporation, podría mencionarse que el código base (anterior a la versión 7.0) fue originario del Sybase SQL Server y constituyó el ingreso de la compañía al mercado de base de datos empresariales, compitiendo con Oracle, IBM y, más adelante, también con Sybase. Microsoft SQL Server 7.0 fue el primer servidor de base de datos con una interfaz gráfica que correspondía a una verdadera reescritura del código original de Sybase. Fue sucedido luego por la versión 2000, primera edición lanzada con una variante para la arquitectura IA-64. SQL Server 2005 fue liberado hacia finales del año 2006, como sucesor de SQL Server 2000. Incluyó soporte nativo para el manejo de datos XML, además de los datos relacionales. La versión actual de SQL Server es la 2008, y apunta, principalmente, al *auto-tunning* (que ya hemos mencionado), la organización y el mantenimiento automático mediante las tecnologías llamadas SQL Server *Always On* para mantener el *downtime* (tiempo de caída o inactividad) cercano a cero.

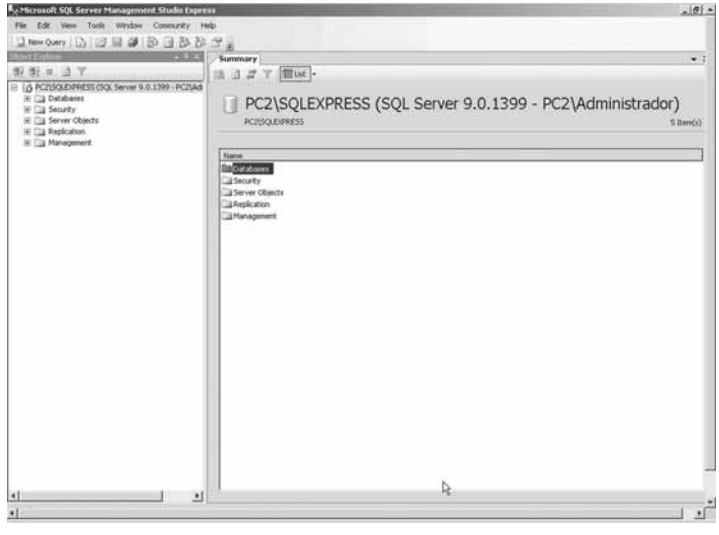

Fig. 6-1. Interfaz gráfica de SQL Server 2005 - SQL *Server Management Studio.*

Funciones para el registro de contraseñas.

Hasta la versión 2005, SQL Server no proveía más funcionalidad criptográfica que dos funciones no documentadas para la encriptación y comparación de constraseñas de usuarios del motor. Si se necesitara implementar, por ejemplo, el algoritmo MD5 en una de estas versiones anteriores a 2005 en base de datos, nos veríamos obligados a utilizar un *stored procedure* (procedimiento almacenado) o a instalar un ActiveX, provistos por terceros (existen alternativas, para ambos casos, descargables desde Internet).

Veamos entonces ahora la manera de implementar *hashing* criptográfico directamente sobre el motor SQL Server, recordando que sólo disponemos de esta funcionalidad a partir de la versión 2005.

Debemos tener en cuenta que se aplican, para el registro de contraseñas, los conceptos vistos respecto del *salt* para evitar la potencial eficacia de un ataque por diccionario. Describiremos las funciones para obtener las salidas de los algoritmos de *hashing* a continuación, pero debe considerarse que queda a cargo del usuario implementar alguna metodología para incluir el uso de *salt*.

La función que utilizaremos es `HashBytes()`. A diferencia de los otros sistemas de base de datos, SQL Server provee esta única función para obtener *hashes* utilizando diferentes algoritmos (esto es establecido mediante el primer parámetro) en lugar de una función para cada uno.

La función se define de la siguiente manera:

```
HashBytes ( '<algorithm>', { @input | 'input' } )

<algorithm>::= MD2 | MD4 | MD5 | SHA | SHA1
```

Como vemos, el primer parámetro determina qué algoritmo se utilizará para retornar el resultado o *hash* de la variable de entrada `@input` o de la cadena de caracteres. La función retorna el valor con el tipo de dato `varbinary`. Nótese que `SHA` y `SHA1` son sinónimos.

Veamos algunos ejemplos de cómo podríamos utilizar esta función.

Para obtener el hash MD5:

```
SELECT HashBytes('M5', "Contenido de prueba")
```

Ahora, para el algoritmo SHA-1. Estas sentencias serán equivalentes:

```
SELECT HashBytes('SHA', "Contenido de prueba")
```

```
SELECT HashBytes('SHA1', "Contenido de prueba")
```

Ejemplo de cómo actualizaríamos la contraseña de un usuario en una tabla de usuarios propia:

```
UPDATE usuarios SET contrasena = HashBytes ('SHA1',
 'contraseña') WHERE email = 'maiorano@gmail.com'
```

Versiones anteriores a la 2005 de SQL Server incluyen, como hemos mencionado más arriba, un par de funciones no documentadas para manejar la información interna de las contraseñas de los usuarios de la base de datos. Las funciones son `pwdencrypt()` y `pwdcompare()`. No se recomienda la utilización de estas funciones en aplicaciones para el registro de contraseñas. Se comenta a continuación, brevemente mediante un ejemplo, cómo funciona cada una.

Declaramos la variable para almacenar la salida de la función; esto es, el *hash* generado a partir de la contraseña:

```
declare @hash varbinary (255)
```

Obtenemos la contraseña cifrada de esta manera (recordamos que se trata de la salida de una función de una vía):

```
SET @hash = pwdencrypt('contraseña')
```

Para una posterior comparación, codificaríamos:

```
SELECT pwdcompare ('contraseña', @hash)
```

Téngase en cuenta que a partir de la versión 2005 de SQL Server, la función `pwdencrypt()` retorna diferentes valores para un mismo valor de entrada –contraseña–. También han cambiado las longitudes de salida entre diferentes versiones. Como se ha mencionado más arriba, estas funciones se considerarían de uso interno y no se recomienda utilizarlas en aplicaciones de usuario.

Funciones para cifrado simétrico y asimétrico de datos.

Los mecanismos de encriptación que provee el motor Microsoft SQL Server, a partir de la versión 2005, son los siguientes:

- Funciones Transact-SQL.
- Llaves asimétricas.

- Llaves simétricas.
- Certificados.
- Encriptación transparente de datos.

Con respecto a las funciones Transact-SQL, usándolas es posible encriptar ítems individuales al insertar o actualizar registros. Las funciones que se describirán en principio, antes de conocer el manejo interno de llaves por parte de SQL Server, son `EncryptByPassPhrase()` y `DecryptByPassPhrase()`. Veremos a continuación cómo utilizarlas, junto con algunos ejemplos de código.

La función para la encriptación se define como:

```
EncryptByPassPhrase ( { 'passphrase' | @passphrase }
        , { 'cleartext' | @cleartext }
    [ , { add_authenticator | @add_authenticator }
        , { authenticator | @authenticator } ] )
```

Conozcamos los parámetros o argumentos de esta función.

- `passphrase` es la frase-clave de contraseña a partir de la cual se generará la llave simétrica, alternativamente `@passphrase` es la variable de tipo `nvarchar, char, varchar, binary, varbinary` o `nchar` que podrá contener esta frase.
- `cleartext` es el texto-plano que deseamos encriptar o cifrar. Alternativamente, como en el caso anterior, puede usarse la variable `@cleartext` (variable de tipo `nvarchar, char, varchar, binary, varbinary` o `nchar`; el tamaño máximo es de 8 000 bytes).
- `add_authenticator` indicará si se cifrará un autenticador junto con el texto sin cifrar. Se utilizará 1 si se va a agregar un autenticador. El mismo valor puede contenerse en la variable `@add_authenticator` alternativamente.
- `authenticator`, o la variable alternativa `@authenticator`, determinará cómo se obtendría un autenticador.

La función retornará un `"varbinary"` con un tamaño máximo de 8 000 bytes.

Veamos algunos ejemplos de uso:

```
SELECT EncryptByPassPhrase('Frase clave secreta', "Contenido de prueba")
```

```
INSERT INTO ArticulosCifrados (Artículo, Contenido)
 VALUES ('Artículo #1',
```

```
 EncryptByPassPhrase('Frase clave secreta',
  'Contenido secreto'))

UPDATE MisPasswords SET Password =
 EncryptByPassPhrase('Frase clave', '12345')
 WHERE Referencia = 'gmail'

UPDATE MisPasswords SET Password =
 EncryptByPassPhrase('Frase clave', 'qwerty')
 WHERE Referencia = 'hotmail'

UPDATE MisPasswords SET Password =
 EncryptByPassPhrase('Frase clave', '98765')
 WHERE Referencia = 'yahoo'
```

Por último veamos un ejemplo basado en la documentación oficial, donde se actualiza un registro de la tabla *SalesCreditCard* y se cifra el valor del número de la tarjeta de crédito almacenado en la columna `CardNumber_EncryptedbyPassphrase`, utilizando la clave principal como un autenticador.

```
USE AdventureWorks;
GO
— Crear columna que almacenará el texto-cifrado.
ALTER TABLE Sales.CreditCard
    ADD CardNumber_EncryptedbyPassphrase varbinary(256);
GO
— Establecer frase-clave.
DECLARE @PassphraseEnteredByUser nvarchar(128);
SET @PassphraseEnteredByUser
    = 'frase-clave de prueba';

— Actualizar registro.
UPDATE Sales.CreditCard
SET CardNumber_EncryptedbyPassphrase =
EncryptByPassPhrase(@PassphraseEnteredByUser
    , CardNumber, 1, CONVERT( varbinary, CreditCardID))
WHERE CreditCardID = '1234';
GO
```

Sigamos ahora con la función que se utilizará para desencriptar lo cifrado con `EncryptByPassPhrase()`. La función se define de esta forma:

```
DecryptByPassPhrase ( {  'passphrase' | @passphrase }
        , {  'ciphertext' | @ciphertext }
    [  , {  add_authenticator | @add_authenticator }
        , {  authenticator | @authenticator } ]  )
```

Los detalles especificados en la descripción de la función de encriptación respecto de los parámetros aplican de la misma forma en esta función.

Veamos los ejemplos que desencriptarían lo cifrado en los ejemplos anteriores:

```
SELECT Articulo, CONVERT(VARCHAR(2000),
 DecryptByPassPhrase ('Frase clave secreta', Contenido))
 FROM ArticulosCifrados
```

```
SELECT Referencia, CONVERT(VARCHAR(2000),
 DecryptByPassPhrase ('Frase clave', Password))
 FROM MisPasswords
```

```
USE AdventureWorks;
— Establecer frase-clave.
DECLARE @PassphraseEnteredByUser nvarchar(128);
SET @PassphraseEnteredByUser
= 'frase-clave de prueba';

— Desencriptar registro.
SELECT CardNumber, CardNumber_EncryptedbyPassphrase
    AS 'Número cifrado', CONVERT(nvarchar,
    DecryptByPassphrase(@PassphraseEnteredByUser,
    CardNumber_EncryptedbyPassphrase, 1
    , CONVERT(varbinary, CreditCardID)))
    AS 'Número cifrado' FROM Sales.CreditCard
    WHERE CreditCardID = 1234;
GO
```

Al hablar de los mecanismos de llaves simétricas y asimétricas implementados en SQL Server 2005, deberemos conocer lo referente a la jerarquía de llaves incorporada dentro del motor.

Este servidor hace uso de esta jerarquía para proteger las llaves –almacenadas por él mismo– que han de utilizarse para la encriptación y desencriptación de información

por el motor. Podríamos visualizar un poco mejor estos conceptos imaginándolos como una serie de capas, en donde cada una encripta lo correspondiente a la capa superior.

Veamos una figura ilustrativa de la documentación oficial:

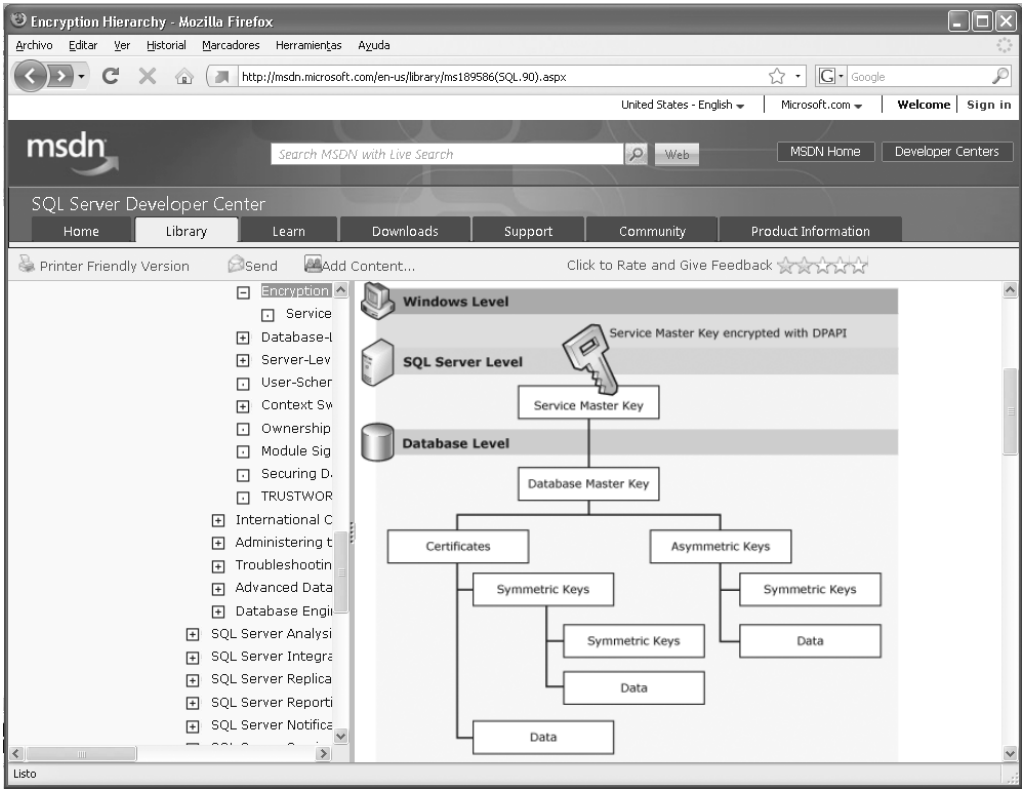

Fig. 6-2. Jerarquía de llaves implementada en Microsoft SQL Server 2005.

Encontramos en el nivel más alto de la jerarquía a la *Data Protection API* (API de protección de datos), que provee estos servicios mediante llamadas o funciones de sistema, es decir, a nivel de sistema operativo. Hacia abajo encontramos la *Service Master Key (SMK)* o llave maestra de servicios, una llave simétrica generada al momento de instalación. Luego, si dentro de una base de datos en particular un usuario necesita encriptar utilizando una llave simétrica, una asimétrica o un certificado, sin especificar una contraseña o frase-clave para generar la llave, una *Database Master Key (DMK)* o llave maestra de base de datos debe ser creada. Cada una de estas llaves, luego de haberse creado, es encriptada con la SMK. Sabemos entonces que esta DMK es la llave que podríamos utilizar para encriptar y/o desencriptar llaves que se almacenarán en base de datos. Si, en cambio, usáramos frases-clave cada vez para la ge-

neración de llaves a la hora de encriptar o desencriptar, no será necesario crear una DMK.

Acerca del mecanismo referido a los certificados, sabemos por lo visto en los capítulos introductorios que son información firmada digitalmente, que relaciona una llave pública a la identidad de una persona, dispositivo o servicio dueño de la correspondiente llave privada. Los certificados *self-signed* (firmados por él mismo), creados por SQL Server, siguen el estándar X.509 y soportan los campos del X.509 versión 1.

Por último, siguiendo con los mecanismos de encriptación implementados a partir de SQL Server 2005, debemos saber que la encriptación transparente de datos, o *Transparent Data Encryption* (TDE), es un caso especial de encriptación utilizando una llave simétrica. TDE encripta una base de datos completa usando la llave simétrica de encriptación de la base de datos. Como vimos anteriormente, esta llave está protegida por otras llaves o certificados que, a su vez, son protegidos por la llave maestra de la base de datos o por una llave asimétrica almacenada en un módulo EKM.

Explicados estos conceptos respecto de los mecanismos de encriptación implementados en SQL Server 2005, abordaremos las funciones Transact-SQL no tratadas hasta el momento, para la encriptación y desencriptación simétrica y asimétrica, junto con algunos ejemplos.

Para la encriptación simétrica, utilizando claves almacenadas en base de datos, disponemos de estas dos funciones: `EncryptByKey()` y `DecryptByKey()`. Recuérdese que los parámetros de texto-plano y autenticador aplican de la misma forma que en las funciones, ya descriptas, `EncryptByPassPhrase()` y `DecryptByPassPhrase()`. Téngase en cuenta que las llaves que se han de utilizar deben abrirse antes de usarse. Veamos cómo se definen.

```
EncryptByKey (key_GUID , { 'cleartext' | @cleartext }
        [ , { add_authenticator | @add_authenticator }
          , { authenticator | @authenticator } ] )
```

```
DecryptByKey ( { 'ciphertext' | @ciphertext }
    [ , add_authenticator,
    { authenticator | @authenticator } ] )
```

En cambio, para la encriptación asimétrica, disponemos de las funciones `EncryptByAsymKey()` y `DecryptByAsymKey()` y, de igual forma para los certificados, las funciones `EncryptByCert()` y `DecryptByCert()`. Las definiciones formales de cada una de ellas:

```
EncryptByAsymKey ( Asym_Key_ID ,
    { 'plaintext' | @plaintext } )
```

```
DecryptByAsymKey (Asym_Key_ID ,
    { 'ciphertext' | @ciphertext }
    [ , 'Asym_Key_Password' ] )
```

```
EncryptByCert ( certificate_ID ,
    { 'cleartext' | @cleartext } )
```

```
DecryptByCert (certificate_ID ,
    { 'ciphertext' | @ciphertext }
    [ , { 'cert_password' | @cert_password } ] )
```

Empecemos ahora con algunos ejemplos para comprender mejor cómo usar estas funciones. Creamos nuestra llave simétrica, que estará protegida por la contraseña "contraseña":

```
CREATE SYMMETRIC KEY NuestraLlaveSimetrica
        WITH ALGORITHM = DESX
        ENCRYPTION BY PASSWORD = N'Contraseña';
```

Podemos verificar que la llave haya sido efectivamente creada mediante la siguiente sentencia:

```
SELECT * FROM sys.symmetric_keys;
```

Entre los resultados obtenidos de esta consulta, podremos conocer el *global unique identifier (GUID)*, o identificador único global, dato que utilizaremos en las llamadas a funciones de encriptación y desencriptación, usando esta llave almacenada.

De manera muy similar para el caso de llaves asimétricas, podríamos crear una llave (o par de llaves) de esta manera:

```
CREATE ASYMMETRIC KEY NuestraLlaveAsimetrica
  WITH ALGORITHM = RSA_2048
  ENCRYPTION BY PASSWORD = N'Contraseña';
```

Nótese que, por supuesto, se especifica otro algoritmo. En este caso, RSA, con longitud de llave de 2 048 bits.

Confirmaríamos la creación de la llave con la sentencia:

```
SELECT * FROM sys.asymmetric_keys;
```

De acuerdo con los ejemplos presentados en el artículo de referencia de Peterson y Li[1], se expone la forma en que se utilizan estas funciones para encriptar y desencriptar información, mediante llaves almacenadas en la base de datos –que se crearán como hemos visto, antes de la ejecución de las pruebas– para los cifrados simétrico y asimétrico. Veremos también los ejemplos del uso de certificados y cómo podríamos implementar el proceso de firma digital.

Ejemplo para la encriptación simétrica:

```
— use the database tempdb
USE tempdb;

— create permanent temp table
CREATE TABLE SymmetricTempTable
(
    Id INT IDENTITY(1,1) PRIMARY KEY,
    PlainText NVARCHAR(100),
    CipherText VARBINARY(MAX)
);

— create symmetric key 'SecureSymmetricKey'
— using the DESX encryption algorithm
— and encypt the key using the password
— 'StrongPassword'
CREATE SYMMETRIC KEY SecureSymmetricKey
   WITH ALGORITHM = DESX
   ENCRYPTION BY PASSWORD = N'StrongPassword';

— must open the key if it is not already
OPEN SYMMETRIC KEY SecureSymmetricKey
   DECRYPTION BY PASSWORD = N'StrongPassword';

— declare and set varible @str to store plaintext
DECLARE  @str NVARCHAR(100)
SET @str = N'Hello DESX';

— encrypt @str and store in TempTable
INSERT INTO SymmetricTempTable (PlainText, CipherText)
VALUES (
```

```
   @str,
   EncryptByKey(Key_GUID('SecureSymmetricKey'), @str)
);

— select data from TempTable
SELECT * FROM SymmetricTempTable;

— decrypt CipherText column and display it
SELECT CONVERT(NVARCHAR(100),
               DecryptByKey(CipherText)) AS PlainText
FROM SymmetricTempTable;

— close the key and drop it
CLOSE SYMMETRIC KEY SecureSymmetricKey;
DROP SYMMETRIC KEY SecureSymmetricKey;
DROP TABLE SymmetricTempTable;
```

Ejemplo para la encriptación asimétrica:

```
— use the database tempdb
USE tempdb;

— create permanent temp table
CREATE TABLE SymmetricTempTable
(
    Id INT IDENTITY(1,1) PRIMARY KEY,
    PlainText NVARCHAR(100),
    CipherText VARBINARY(MAX)
);

— create symmetric key 'SecureSymmetricKey'
— using the DESX encryption algorithm
— and encypt the key using the password
— 'StrongPassword'
CREATE SYMMETRIC KEY SecureSymmetricKey
   WITH ALGORITHM = DESX
   ENCRYPTION BY PASSWORD = N'StrongPassword';

— must open the key if it is not already
OPEN SYMMETRIC KEY SecureSymmetricKey
   DECRYPTION BY PASSWORD = N'StrongPassword';

— declare and set varible @str to store plaintext
DECLARE  @str NVARCHAR(100)
SET @str = N'Hello DESX';
```

```
— encrypt @str and store in TempTable
INSERT INTO SymmetricTempTable (PlainText, CipherText)
VALUES (
   @str,
   EncryptByKey(Key_GUID('SecureSymmetricKey'), @str)
);

— select data from TempTable
SELECT * FROM SymmetricTempTable;

— decrypt CipherText column and display it
SELECT CONVERT(NVARCHAR(100),
               DecryptByKey(CipherText)) AS PlainText
FROM SymmetricTempTable;

— close the key and drop it
CLOSE SYMMETRIC KEY SecureSymmetricKey;
DROP SYMMETRIC KEY SecureSymmetricKey;
DROP TABLE SymmetricTempTable;
```

Ejemplo de la utilización de certificados:

```
— use the database tempdb
USE tempdb;

— create permanent temp table
CREATE TABLE CertificateTempTable (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    PlainText NVARCHAR(100),
    CipherText VARBINARY(MAX)
);

— create self signed certificate encrypting the private
— key with the supplied — password
CREATE CERTIFICATE SelfSignedCertificate
   ENCRYPTION BY PASSWORD = 'CertificateStrongPassword'
   WITH SUBJECT = 'Self Signed Certificate',
   EXPIRY_DATE = '12/01/2030';

— declare and set plaintext to be encrypted
DECLARE @str NVARCHAR(100);
SET @str = 'Secret information...shhhhhh';

— insert plaintext and encrypted data into the temp table,
```

```
— using the public key of the specified certificate
INSERT INTO CertificateTempTable (PlainText, CipherText)
    VALUES(@str,
    EncryptByCert(Cert_ID('SelfSignedCertificate'), @str));

— display data in table
SELECT * FROM CertificateTempTable;

— decrypt data and display
SELECT CONVERT(NVARCHAR(MAX),
    DecryptByCert(Cert_Id('SelfSignedCertificate'),
    CipherText, N'CertificateStrongPassword')) As PlainText
FROM CertificateTempTable;

— delete certificate and drop table
DROP CERTIFICATE SelfSignedCertificate;
DROP TABLE CertificateTempTable;
```

Ejemplo para la firma digital de información:

```
— use the database tempdb
USE tempdb

— create symmetric key AnotherAsymmetricKey
— using the 2048-bit RSA encryption algorithm
— and encypt the key using the password
— 'VeryVeryStrongPassword'
CREATE ASYMMETRIC KEY AnotherAsymmetricKey
   WITH ALGORITHM = RSA_2048
   ENCRYPTION BY PASSWORD = N'VeryVeryStrongPassword';

— create temp table for inserting data
CREATE TABLE AsymmetricTemp (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    PlainText NVARCHAR(100),
    CipherText VARBINARY(MAX),
    Signature VARBINARY(MAX)
);

— declare and set varible @str to store plaintext
DECLARE @str NVARCHAR(100);
SET @str = N'60000';

— insert data into AsymmetricTemp
INSERT INTO AsymmetricTemp (PlainText, CipherText, Signature)
```

```
VALUES (
   @str,
   EncryptByAsymKey(AsymKey_ID('AnotherAsymmetricKey'), @str),
   SignByAsymKey(AsymKey_ID('AnotherAsymmetricKey'),
                          @str, N'VeryVeryStrongPassword')
);

— display data in table
SELECT * FROM AsymmetricTemp;

— check the integrity of the data stored by checking
— the signature against the plaintext
SELECT PlainText,
       CONVERT(NVARCHAR(100),
       DecryptByAsymKey(AsymKey_ID('AnotherAsymmetricKey'),
              CipherText, N'VeryVeryStrongPassword')) AS Decrypted,
       CASE
         WHEN VerifySignedByAsymKey(AsymKey_Id('AnotherAsymmetricKey'),
                                    PlainText, Signature) = 1
            THEN N'The data has not been changed.'
          ELSE N'The data has been modified!'
       END AS IntegrityCheck
FROM AsymmetricTemp;

— add a '0' to the end of the plaintext
UPDATE AsymmetricTemp SET PlainText = PlainText + '0';

— check the integrity of the data stored by checking the
— signature against the plaintext
SELECT PlainText,
       CONVERT(NVARCHAR(100),
       DecryptByAsymKey(AsymKey_ID('AnotherAsymmetricKey'),
              CipherText, N'VeryVeryStrongPassword')) AS Decrypted,
       CASE
         WHEN VerifySignedByAsymKey(AsymKey_Id('AnotherAsymmetricKey'),
                                    PlainText, Signature) = 1
            THEN N'The data has not been changed.'
          ELSE N'The data has been modified!'
       END AS IntegrityCheck
FROM AsymmetricTemp;

— delete key and table
DROP ASYMMETRIC KEY AnotherAsymmetricKey;
DROP TABLE AsymmetricTemp;
```

ORACLE Server.

Oracle Database, Oracle RDBMS (como hemos visto, siglas de *Relational DataBase Management System*) o simplemente Oracle es un sistema de gestión de base de datos (SGBD) relacional producido por Oracle Corporation.

Un sistema de base de datos Oracle comprende, al menos, una instancia de la aplicación –procesos del sistema operativo y estructuras de memoria–, interactuando con el segundo componente principal, siendo este la información o conjunto de datos almacenados.

Veamos un diagrama, tomado de la documentación oficial –*Oracle9i Database Online Documentation; Oracle9i Database Getting Started; Oracle9i Architecture on Windows*–, de la arquitectura del servidor Oracle en Windows. Los procesos de *background*, o procesos de fondo o en segundo plano, leen y escriben sobre los diferentes *datafiles*, dependiendo de la configuración particular. Las referencias se explican a continuación.

- `DBW0`: Proceso que escribe en base de datos.
- `LGWR`: Corresponde al proceso que escribe el log o bitácora.
- `PMON`: Es el monitor de procesos; `SMON` el monitor de sistema.
- `CKPT`: El proceso de puesto de control.
- `ARCH0`: Proceso de archivo.
- `RECO`: El proceso distribuido de recuperación.

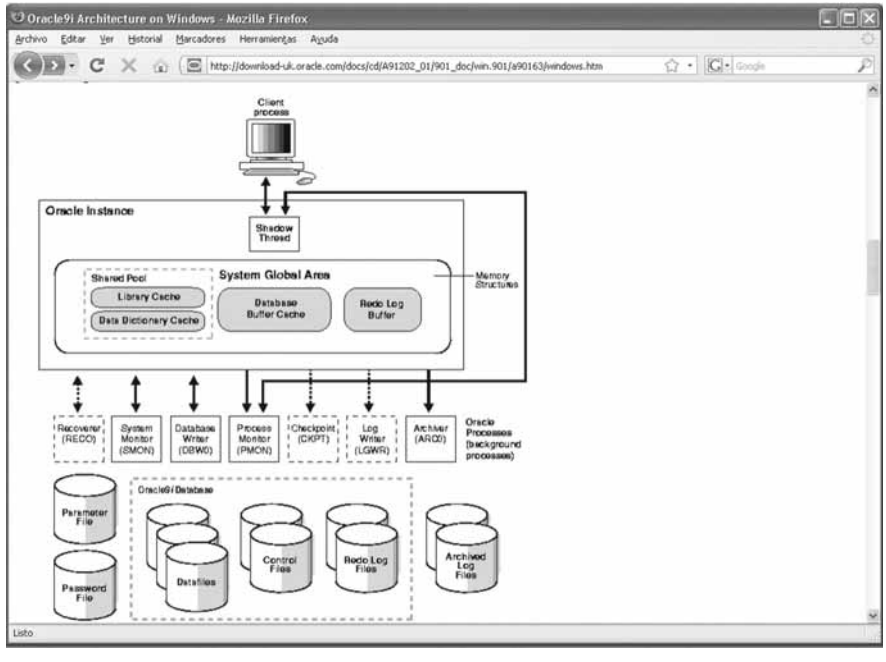

Fig. 6-3. Arquitectura de Oracle en Windows.

La última versión comercial del producto es la 11g, liberada en el año 2007 (la versión 10g se comercializó a partir del año 2003 y la 9i en el año 2001). Estas tres versiones son las más utilizadas actualmente y se implementan, principalmente, en entornos corporativos. En el año 2005, Oracle Corporation libera su primera base de datos libre o gratuita, Oracle Database 10g Express Edition (XE).

Con respecto a la historia de este motor de RDBMS, nos referiremos a la información publicada por el sitio Oracle para Latinoamérica bajo el título de "La historia de Oracle: Innovación, liderazgo y resultados"; hemos de disculpar los agregados o adecuaciones a lo rigurosamente histórico:

> *"Hace tres décadas, Larry Ellison vio una oportunidad que otras empresas no supieron apreciar al descubrir la descripción de un prototipo de trabajo para una base de datos relacional y enterarse de que ninguna empresa se había comprometido a comercializar la tecnología. Ellison y sus cofundadores, Bob Miner y Ed Oates, se dieron cuenta del gran potencial económico que ofrecía el modelo de base de datos relacional, pero no se dieron cuenta de que ellos cambiarían la informática empresarial para siempre.*
>
> *Con la agilidad de una empresa mucho más pequeña, Oracle ha demostrado, gracias a su historial, que puede construir para el futuro sobre la base de años de innovación, el gran conocimiento de los éxitos y desafíos de sus clientes, y los mejores talentos en el área técnica y comercial alrededor del mundo. La empresa ha demostrado no sólo su capacidad de aprovechar al máximo su gran tamaño y sus virtudes para servir a sus clientes, sino también su capacidad de tomar decisiones que eliminen las creencias convencionales y lleven sus productos y servicios hacia una nueva dirección.*
>
> *Después de 30 años, Oracle sigue siendo el estándar de oro para las aplicaciones y la tecnología de base de datos de empresas de todo el mundo: La compañía es proveedora líder mundial de software para la administración de la información, y la segunda empresa de software independiente más grande del mundo. La tecnología de Oracle puede encontrarse en casi todos los sectores, y en los centros de datos de 98 de las 100 empresas Fortune. Oracle es la primera empresa de software en desarrollar e implementar software empresarial 100 por ciento activado por Internet en toda su línea de productos: Base de datos, aplicaciones comerciales y herramientas para el soporte de decisiones y el desarrollo de aplicaciones.*
>
> *Es la innovación la que impulsa el éxito de Oracle. Oracle fue una de las primeras empresas en lograr que sus aplicaciones comerciales*

estén disponibles en Internet, una idea hoy en día dominante. Con el lanzamiento de Oracle Fusion Middleware, Oracle comienza a lanzar nuevos productos y funcionalidades, los cuales reflejan el objetivo de la empresa: Conectar todos los niveles de tecnología empresarial para ayudar a los clientes a acceder al conocimiento que necesitan para responder con velocidad y agilidad ante los requisitos del mercado. Hoy, Oracle Real Application Clusters, Oracle E-Business Suite, Oracle Grid Computing (inglés), soporte de Enterprise Linux, y Oracle Fusion fomentan el compromiso hacia la innovación y los resultados, lo cual ha perfilado a Oracle durante 30 años.

¿Qué planeamos para el futuro? Lucharemos por ser #1 en Middleware y #1 en aplicaciones, del mismo modo que lo hicimos con la base de datos. Nuestro objetivo es continuar innovando y liderando el sector, focalizándonos siempre en solucionar los problemas de los clientes que confían en nuestro software."

Agregaremos únicamente que el nombre Oracle proviene del nombre en código de un proyecto financiado por la CIA, en el cual Larry trabajó mientras era empleado de la compañía Ampex.

Funciones para el registro de contraseñas.

Respecto de las funciones para generación de *hashes* criptográficos disponibles en el motor RDBMS Oracle, nos referiremos en primera instancia a las funcionalidades provistas por el *package* o paquete `DBMS_CRYPTO`, disponible a partir de la versión 10g.

Como para el caso de SQL Server (también para MySQL o para cualquier otro motor de base de datos), se recomienda la utilización de un *salt* para registro de contraseñas para evitar la eficacia de un ataque de diccionario. Veremos a continuación cómo usar las funciones para obtener las salidas de los algoritmos de *hashing* en este motor particularmente, pero no debemos olvidar que queda a cargo del usuario implementar alguna metodología para incluir el uso de *salt*.

`DBMS_CRYPTO` provee implementaciones de los algoritmos MD4, MD5 y SHA-1. La función del paquete para acceder a estas implementaciones es `Hash()`. El algoritmo para utilizar se especifica por parámetro. Para tal fin se definen las siguientes constantes: `HASH_MD4`, `HASH_MD5` y `HASH_SH1`.

Veamos la sintaxis de la función en tres versiones, de acuerdo con el tipo de dato utilizado para la entrada:

```
DBMS_CRYPTO.Hash (
   src IN RAW,
   typ IN PLS_INTEGER)
 RETURN RAW;
```

```
DBMS_CRYPTO.Hash (
   src IN BLOB,
   typ IN PLS_INTEGER)
 RETURN RAW;
```

```
DBMS_CRYPTO.Hash (
   src IN CLOB CHARACTER SET ANY_CS,
   typ IN PLS_INTEGER)
 RETURN RAW;
```

El parámtro `src` contendrá los datos a partir de los cuales deseamos obtener el *hash*; el parámetro `typ` se utilizará para determinar qué algoritmo utilizar. Como hemos visto, las constantes son `HASH_MD4, HASH_MD5` y `HASH_SH1` y corresponden a los algoritmos MD4, MD5 y SHA-1, respectivamente. Incluimos el detalle de los valores de estas constantes en las siguientes definiciones de ejemplo:

```
HASH_MD4 CONSTANT PLS_INTEGER := 1;
HASH_MD5 CONSTANT PLS_INTEGER := 2;
HASH_SH1 CONSTANT PLS_INTEGER := 3;
```

Veamos un ejemplo de cómo utilizar esta función:

```
SQL> l
1 declare
2 l_in_val varchar2(2000) := "Contenido de prueba";
3 l_hash raw(2000);
4 begin
5 l_hash := dbms_crypto.hash (
6 UTL_I18N.STRING_TO_RAW (l_in_val, 'AL32UTF8'),
7 dbms_crypto.hash_sh1
8 );
9 dbms_output.put_line('Hash='||l_hash);
10* end;
```

Veamos otro ejemplo que hace uso de los tres algoritmos disponibles:

```
set serveroutput on

DECLARE
 l_input VARCHAR2(100) := "Contenido de prueba";
 l_input_raw RAW(128) := utl_raw.cast_to_raw(l_input);
 l_encrypted_raw RAW(2048);
BEGIN

  dbms_output.put_line('entrada:  ' || l_input_raw);

  l_encrypted_raw := dbms_crypto.hash(l_input_raw, 1);
  dbms_output.put_line('MD4: ' || l_encrypted_raw);

  l_encrypted_raw := dbms_crypto.hash(l_input_raw, 2);
  dbms_output.put_line('MD5: ' || l_encrypted_raw);

  l_encrypted_raw := dbms_crypto.hash(l_input_raw, 3);
  dbms_output.put_line('SHA-1: ' || l_encrypted_raw);
END;
/
```

Para versiones anteriores a la 10g, utilizaremos el *package* o paquete `DMBS_OBFUSCATION_TOOLKIT`, que dispone de una implementación (en una función del mismo nombre) del algoritmo MD5.

Detallamos la sintaxis de esta función:

```
DBMS_OBFUSCATION_TOOLKIT.MD5(
   input            IN   RAW,
   checksum         OUT  raw_checksum);
```

```
DBMS_OBFUSCATION_TOOLKIT.MD5(
   input_string     IN   VARCHAR2,
   checksum_string  OUT  varchar2_checksum);
```

```
DBMS_OBFUSCATION_TOOLKIT.MD5(
   input         IN  RAW)
  RETURN raw_checksum;
```

```
DBMS_OBFUSCATION_TOOLKIT.MD5(
   input_string  IN  VARCHAR2)
  RETURN varchar2_checksum;
```

Los parámetros de `input` o `input_string` se utilizan para especificar la entrada o información a partir de la cual deseamos obtener el *hash* criptográfico MD5. Los parámetros (o valores retornados) `raw_checksum` y `varchar2_checksum` son, como se habrá adivinado, el resultado o *hash* MD5 generado.

Veamos un ejemplo de cómo podríamos utilizar una de estas funciones:

```
SQL> create or replace function md5( input varchar2 ) return
sys.dbms_obfuscation_toolkit.varchar2_checksum as
begin
  return sys.dbms_obfuscation_toolkit.md5( input_string => input );
 end;
/

SQL> select md5("Contenido de prueba") from dual;
```

Funciones para cifrado simétrico de datos.

Para la encriptación y desencriptación simétrica, las implementaciones de los algoritmos a las cuales nos referiremos en este apartado, como con lo visto en el apartado anterior, se encuentran en los *packages* o paquetes `DBMS_CRYPTO` (disponible a partir de la versión 10g) y `DBMS_OBFUSCATION_TOOLKIT` (en la versión 8i se proveía la implementación del algoritmo DES y en la versión 9i se agregó la implementación de TripleDES).

Se verán primero las funciones o procedimientos que provee el paquete `DBMS_CRYPTO`. Las funciones y procedimientos fueron llamados `ENCRYPT` y `DECRYPT`.

```
DBMS_CRYPTO.ENCRYPT(
   src IN RAW,
   typ IN PLS_INTEGER,
   key IN RAW,
   iv  IN RAW          DEFAULT NULL)
 RETURN RAW;
```

```
DBMS_CRYPTO.ENCRYPT(
   dst IN OUT NOCOPY BLOB,
   src IN            BLOB,
   typ IN            PLS_INTEGER,
   key IN            RAW,
   iv  IN            RAW          DEFAULT NULL);
```

```
DBMS_CRYPTO.ENCRYPT(
   dst IN OUT NOCOPY BLOB,
   src IN            CLOB         CHARACTER SET ANY_CS,
   typ IN            PLS_INTEGER,
   key IN            RAW,
   iv  IN            RAW          DEFAULT NULL);
```

```
DBMS_CRYPTO.DECRYPT(
   src IN RAW,
```

```
    typ IN PLS_INTEGER,
    key IN RAW,
    iv  IN RAW          DEFAULT NULL)
 RETURN RAW;
```

```
DBMS_CRYPTO.DECRYPT(
   dst IN OUT NOCOPY BLOB,
   src IN            BLOB,
   typ IN            PLS_INTEGER,
   key IN            RAW,
   iv  IN            RAW          DEFAULT NULL);
```

```
DBMS_CRYPT.DECRYPT(
   dst IN OUT NOCOPY CLOB         CHARACTER SET ANY_CS,
   src IN            BLOB,
   typ IN            PLS_INTEGER,
   key IN            RAW,
   iv  IN            RAW          DEFAULT NULL);
```

El parámetro `src` contendrá la información que se ha de encriptar; `typ` determinará el algoritmo, modo y *padding* para utilizar. El argumento o parámetro `key` especificará la llave.

El parámetro `iv` contendrá el vector de incialización. Para los procedimientos, que utilizan el parámetro `dst`, éste corresponde al LOB de la salida, valor que será sobreescrito.

Los algoritmos disponibles –listaremos las constantes, ya que convenientemente fueron nombradas como los nombres código de cada algoritmo– son los siguientes:

- `ENCRYPT_DES`
- `ENCRYPT_3DES_2KEY`
- `ENCRYPT_3DES`
- `ENCRYPT_AES128`
- `ENCRYPT_AES192`
- `ENCRYPT_AES256`
- `ENCRYPT_RC4`

Veamos un ejemplo de cómo se codificaría la encriptación de nuestra cadena de caracteres usada como ejemplo a lo largo del libro, pero ahora recortada a “Contenido de pru”, para mantener una longitud de 16 bytes (requerimiento de múltiplo de 8). Lo haremos mediante el algorimo AES, en modo CBC y sin *padding* (por lo cual necesita-

mos utilizar para la entrada una longitud múltiplo del tamaño de bloque del algoritmo). Se utilizará como llave a “1234567890123456”, cadena de caracteres de 16 bytes de longitud (128 bits).

```
  1  declare
  2    l_key     varchar2(2000) := '1234567890123456';
  3    l_in_val  varchar2(2000) := "Contenido de pru";
  4    l_mod     number := dbms_crypto.ENCRYPT_AES128
  5                       + dbms_crypto.CHAIN_CBC
  6                       + dbms_crypto.PAD_NONE;
  7    l_enc     raw (2000);
  8  begin
  9    l_enc := dbms_crypto.encrypt
 10      (
 11         UTL_I18N.STRING_TO_RAW (l_in_val, 'AL32UTF8'),
 12         l_mod,
 13         UTL_I18N.STRING_TO_RAW (l_key, 'AL32UTF8')
 14      );
 15    dbms_output.put_line ('Cifrado='||l_enc);
 16* end;
SQL> /
```

Refiriéndonos ahora al paquete `DBMS_OBFUSCATION_TOOLKIT`, a partir de la versión 9i, contamos con una implementación del algoritmo Triple-DES, a través de las funciones o procedimientos `DES3Encrypt` y `DES3Decrypt`. Veamos las diferentes alternativas que diponemos para invocar esta funcionalidad.

Estas son las alternativas para `DES3Encrypt`:

```
DBMS_OBFUSCATION_TOOLKIT.DES3Encrypt(
   input           IN     RAW,
   key             IN     RAW,
   encrypted_data  OUT    RAW,
   which           IN     PLS_INTEGER  DEFAULT TwoKeyMode
   iv              IN     RAW          DEFAULT NULL);
```

```
DBMS_OBFUSCATION_TOOLKIT.DES3Encrypt(
   input_string      IN     VARCHAR2,
   key_string        IN     VARCHAR2,
   encrypted_string  OUT    VARCHAR2,
   which             IN     PLS_INTEGER  DEFAULT TwoKeyMode
   iv_string         IN     VARCHAR2     DEFAULT NULL);
```

```
DBMS_OBFUSCATION_TOOLKIT.DES3Encrypt(
```

```
    input        IN RAW,
    key          IN RAW,
    which        IN PLS_INTEGER DEFAULT TwoKeyMode
    iv           IN RAW         DEFAULT NULL)
  RETURN RAW;
```

```
DBMS_OBFUSCATION_TOOLKIT.DES3Encrypt(
   input_string  IN VARCHAR2,
   key_string    IN VARCHAR2,
   which         IN PLS_INTEGER DEFAULT TwoKeyMode
   iv_string     IN VARCHAR2    DEFAULT NULL)
  RETURN VARCHAR2;
```

Estas son las alternativas para `DES3Decrypt`:

```
DBMS_OBFUSCATION_TOOLKIT.DES3Decrypt(
   input             IN   RAW,
   key               IN   RAW,
   decrypted_data    OUT  RAW,
   which             IN   PLS_INTEGER  DEFAULT TwoKeyMode
   iv                IN   RAW          DEFAULT NULL);
```

```
DBMS_OBFUSCATION_TOOLKIT.DES3Decrypt(
   input_string      IN   VARCHAR2,
   key_string        IN   VARCHAR2,
   decrypted_string  OUT  VARCHAR2,
   which             IN   PLS_INTEGER  DEFAULT TwoKeyMode
   iv_string         IN   VARCHAR2     DEFAUTL NULL);
```

```
DBMS_OBFUSCATION_TOOLKIT.DES3Decrypt(
   input         IN  RAW,
   key           IN  RAW,
   which         IN  PLS_INTEGER DEFAULT TwoKeyMode
   iv            IN  RAW         DEFAULT NULL)
  RETURN RAW;
```

```
DBMS_OBFUSCATION_TOOLKIT.DES3Decrypt(
   input_string  IN  VARCHAR2,
   key_string    IN  VARCHAR2,
   which         IN  PLS_INTEGER DEFAULT TwoKeyMode
   iv_string     IN  VARCHAR2    DEFAULT NULL)
  RETURN VARCHAR2;
```

La implementación de Oracle de TripleDES (o 3DES) soporta tanto implementaciones de dos (se deberá pasar como llave una de 128 bits de longitud) como de tres

llaves (se pasará una de 192 bits de longitud), en modo CBC. La entrada debe ser múltiplo de 8 bytes.

El parámetro `input` es la información que se ha de cifrar o encriptar; se utilizará `key` para especificar la llave; `encrypted_data` es el parámetro de salida para el texto-cifrado; `which` determinará si se usará el modo de dos llaves o de tres; `iv` será el vector de inicialización para utilizar; `input_string` la entrada, o información para cifrar, como cadena de caracteres; `key_string` será la llave pero como cadena de caracteres; `encrypted_string` la salida o texto-cifrado como cadena de caracteres; y, por último, `iv_string` el vector de inicialización como cadena de caracteres.

Teniendo en cuenta que en la versión 8i posee las funciones o procedimientos `DESEncrypt` y `DESDecrypt` del paquete `DBMS_OBFUSCATION_TOOLKIT` para la encriptación simétrica, implementando el algoritmo DES, se incluyen las definiciones correspondientes.

Para `DBMS_OBFUSCATION_TOOLKIT.DESEncrypt` las opciones son:

```
DBMS_OBFUSCATION_TOOLKIT.DESEncrypt(
   input            IN    RAW,
   key              IN    RAW,
   encrypted_data   OUT   RAW);
```

```
DBMS_OBFUSCATION_TOOLKIT.DESEncrypt(
   input_string     IN    VARCHAR2,
   key_string       IN    VARCHAR2,
   encrypted_string OUT   VARCHAR2);
```

```
DBMS_OBFUSCATION_TOOLKIT.DESEncrypt(
   input         IN  RAW,
   key           IN  RAW)
  RETURN RAW;
```

```
DBMS_OBFUSCATION_TOOLKIT.DESEncrypt(
   input_string  IN  VARCHAR2,
   key_string    IN  VARCHAR2)
  RETURN VARCHAR2;
```

Para `DBMS_OBFUSCATION_TOOLKIT.DESDecrypt` las opciones son:

```
DBMS_OBFUSCATION_TOOLKIT.DESDecrypt(
   input             IN   RAW,
   key               IN   RAW,
   decrypted_data    OUT  RAW);
```

```
    DBMS_OBFUSCATION_TOOLKIT.DESDecrypt(
       input_string      IN   VARCHAR2,
       key_string        IN   VARCHAR2,
       decrypted_string  OUT  VARCHAR2);
```

```
    DBMS_OBFUSCATION_TOOLKIT.DESDecrypt(
       input            IN  RAW,
       key              IN  RAW)
      RETURN RAW;
```

```
    DBMS_OBFUSCATION_TOOLKIT.DESDecrypt(
       input_string     IN  VARCHAR2,
       key_string       IN  VARCHAR2)
      RETURN VARCHAR2;
```

Los parámetros, al margen de que resulten evidentes, aplican de la misma manera que en los otros procedimientos o funciones referidos en este apartado. Recuérdese que en este caso la llave debe ser de 8 bytes y la entrada múltiplo de 8 bytes.

Veremos ahora el ejemplo de la documentación oficial para mostrar la forma de usarlas, en el caso de que se precise trabajar con el servidor más antiguo de la línea. Por supuesto, este código funcionará también en versiones posteriores, pero en tales versiones se recomienda el uso de algunos de los algoritmos más modernos.

```
    DECLARE
       input_string        VARCHAR2(16) := 'tigertigertigert';
       raw_input           RAW(128) := UTL_RAW.CAST_TO_RAW(input_string);
       key_string          VARCHAR2(8)  := 'scottsco';
       raw_key             RAW(128) := UTL_RAW.CAST_TO_RAW(key_string);
       wrong_input_string  VARCHAR2(25) := 'not_a_multiple_of_8_bytes';
       wrong_raw_input     RAW(128) : =
       UTL_RAW.CAST_TO_RAW(wrong_input_string);
       wrong_key_string    VARCHAR2(8)  := 'scottsco';
       wrong_raw_key      RAW(128) := UTL_RAW.CAST_TO_RAW(wrong_key_string);
       encrypted_raw               RAW(2048);
       encrypted_string            VARCHAR2(2048);
       double_encrypted_raw        RAW(2048);
       double_encrypted_string     VARCHAR2(2048);
       decrypted_raw               RAW(2048);
       decrypted_string            VARCHAR2(2048);
       error_in_input_buffer_length EXCEPTION;
       PRAGMA EXCEPTION_INIT(error_in_input_buffer_length, -28232);
       INPUT_BUFFER_LENGTH_ERR_MSG VARCHAR2(100) :=
       '*** DES INPUT BUFFER NOT A MULTIPLE OF 8 BYTES - IGNORING EXCEPTION
***';
       double_encrypt_not_permitted EXCEPTION;
```

```
       PRAGMA EXCEPTION_INIT(double_encrypt_not_permitted, -28233);
       DOUBLE_ENCRYPTION_ERR_MSG VARCHAR2(100) :=
        '*** CANNOT DOUBLE ENCRYPT DATA - IGNORING EXCEPTION ***';

    — 1. Begin testing raw data encryption and decryption
       BEGIN
       dbms_output.put_line('> ========= BEGIN TEST RAW DATA =========');
       dbms_output.put_line('> Raw input                          : ' ||
                     UTL_RAW.CAST_TO_VARCHAR2(raw_input));
       BEGIN
          dbms_obfuscation_toolkit.DESEncrypt(input => raw_input,
                   key => raw_key, encrypted_data => encrypted_raw );
          dbms_output.put_line('> encrypted hex value              : ' ||
                   rawtohex(encrypted_raw));
          dbms_obfuscation_toolkit.DESDecrypt(input => encrypted_raw,
                   key => raw_key, decrypted_data => decrypted_raw);
          dbms_output.put_line('> Decrypted raw output             : ' ||
                        UTL_RAW.CAST_TO_VARCHAR2(decrypted_raw));
          dbms_output.put_line('>  ');
          if UTL_RAW.CAST_TO_VARCHAR2(raw_input) =
                        UTL_RAW.CAST_TO_VARCHAR2(decrypted_raw) THEN
                dbms_output.put_line('> Raw DES Encyption and Decryption
successful');
          END if;
       EXCEPTION
          WHEN error_in_input_buffer_length THEN
                 dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
       END;
       dbms_output.put_line('>  ');

    — 2. Begin testing raw data double encryption prevention
       BEGIN
          dbms_output.put_line('> testing double encryption prevention');
          dbms_output.put_line('>  ');
          dbms_obfuscation_toolkit.DESEncrypt(input => raw_input,
                   key => raw_key, encrypted_data => encrypted_raw );
          dbms_output.put_line('> input hex value : ' ||
                   rawtohex(encrypted_raw));
          dbms_obfuscation_toolkit.DESEncrypt(
                   input => encrypted_raw,
                   key => raw_key,
                   encrypted_data => double_encrypted_raw );
          dbms_output.put_line('> double encrypted hex value : ' ||
                        rawtohex(double_encrypted_raw));
       EXCEPTION
          WHEN  double_encrypt_not_permitted THEN
                 dbms_output.put_line('> ' || DOUBLE_ENCRYPTION_ERR_MSG);
```

```
    END;
    dbms_output.put_line('>  ');

— 3. Begin testing wrong raw input length values for encrypt operation
    BEGIN
       dbms_output.put_line('> Wrong Raw input for encryption   : ' ||
                     UTL_RAW.CAST_TO_VARCHAR2(wrong_raw_input));
       dbms_obfuscation_toolkit.DESEncrypt(
                input => wrong_raw_input,
                key => raw_key,
                encrypted_data => encrypted_raw );
       dbms_output.put_line('> encrypted hex value : ' ||
                     rawtohex(encrypted_raw));
    EXCEPTION
       WHEN error_in_input_buffer_length THEN
             dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
    END;
    dbms_output.put_line('>  ');

— 4. Begin testing wrong raw input length values for decrypt operation
    BEGIN
       — testing wrong input values for decyrpt operation
       dbms_output.put_line('> Wrong Raw input for decryption   : ' ||
                     UTL_RAW.CAST_TO_VARCHAR2(wrong_raw_input));
       dbms_obfuscation_toolkit.DESDecrypt
           (input => wrong_raw_input,
            key => raw_key,
            decrypted_data => decrypted_raw );
       dbms_output.put_line('> decrypted hex value  : '
                || rawtohex(decrypted_raw));
    EXCEPTION
       WHEN error_in_input_buffer_length THEN
             dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
    END;
    dbms_output.put_line('>  ');
    dbms_output.put_line('> ========= END TEST RAW DATA =========');

— 5. Begin testing string data encryption and decryption
    dbms_output.put_line('> ========= BEGIN TEST STRING DATA =========');

    BEGIN
       dbms_output.put_line('> input string : '
                           || input_string);
       dbms_obfuscation_toolkit.DESEncrypt(
                input_string => input_string,
                key_string => key_string,
                encrypted_string => encrypted_string );
```

```
         dbms_output.put_line('> encrypted hex value : ' ||
                     rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_string)));
         dbms_obfuscation_toolkit.DESDecrypt(
                  input_string => encrypted_string,
                  key_string => key_string,
                  decrypted_string => decrypted_string );
         dbms_output.put_line('> decrypted string output : ' ||
                    decrypted_string);
         if input_string = decrypted_string THEN
            dbms_output.put_line('> String DES Encyption and Decryption
    successful');
         END if;
      EXCEPTION
         WHEN error_in_input_buffer_length THEN
                dbms_output.put_line(' ' || INPUT_BUFFER_LENGTH_ERR_MSG);
      END;
      dbms_output.put_line('>  ');

   — 6. Begin testing string data double encryption prevention
      BEGIN
         dbms_output.put_line('> testing double encryption prevention');
         dbms_output.put_line('>  ');
         dbms_obfuscation_toolkit.DESEncrypt(
                  input_string => input_string,
                  key_string => key_string,
                  encrypted_string => encrypted_string );
         dbms_output.put_line('> input hex value : ' ||
                     rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_string)));
         dbms_obfuscation_toolkit.DESEncrypt(
                  input_string => encrypted_string,
                  key_string => key_string,
                  encrypted_string => double_encrypted_string );
         dbms_output.put_line('> double encrypted hex value       : ' ||

rawtohex(UTL_RAW.CAST_TO_RAW(double_encrypted_string)));
      EXCEPTION
         WHEN  double_encrypt_not_permitted THEN
                dbms_output.put_line('> ' || DOUBLE_ENCRYPTION_ERR_MSG);
      END;
      dbms_output.put_line('>  ');

   — 7. Begin testing wrong string input length values for encyrpt
operation
      BEGIN
         dbms_output.put_line('> testing wrong input values for encyrpt
    operation');
         dbms_output.put_line('> Wrong Raw input for encryption   : ' ||
```

```
                        wrong_input_string);
         dbms_obfuscation_toolkit.DESEncrypt(
                  input_string => wrong_input_string,
                  key_string => wrong_key_string,
                  encrypted_string => encrypted_string );
         dbms_output.put_line('> encrypted hex value : ' ||
                     rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_string)));
      EXCEPTION
         WHEN error_in_input_buffer_length THEN
               dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
      END;
      dbms_output.put_line('>  ');

   — 8. Begin testing wrong string input length values for decrypt
operation
      BEGIN
         — testing wrong input values for decyrpt operation
         dbms_output.put_line('> Wrong Raw input for encryption   : ' ||
                     wrong_input_string);
         dbms_obfuscation_toolkit.DESDecrypt(
                  input_string => wrong_input_string,
                  key_string => wrong_key_string,
                  decrypted_string => decrypted_string );
              dbms_output.put_line('>  decrypted  string  output  :  '  ||
decrypted_string);
      EXCEPTION
         WHEN error_in_input_buffer_length THEN
               dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
      END;
      dbms_output.put_line('>  ');
       dbms_output.put_line('> ========= END TEST STRING DATA =========');
      END;
   /
```

MySQL Server.

Su *slogan* comercial es *The world's most popular open source database* (La base de datos *open-source* más popular del mundo) y, efectivamente, es quizá MySQL el motor de base de datos *open-source,* o código abierto o libre, más popular en la actualidad.

En combinación con el sistema operativo Linux, el servidor Web Apache y el intérprete PHP (formando el entorno LAMP) se han establecido como un estándar para la implementación de sitios Web. Cabe aclarar que el uso se ha popularizado en sitios

de pequeña o mediana envergadura, acaso por la disponibilidad del entorno y su fácil utilización, aunque también existen experiencias positivas de la implementación de estas tecnologías para sitios sofisticados en cuanto al modelado y/o alto volumen de la información que manejan, soportando una gran cantidad de carga con un buen rendimiento.

Entre los usuarios de MySQL AB se encuentran Alcatel-Lucent, Amazon.com, Google, Digg, Facebook, Nokia, Yahoo y YouTube, entre otros.

La compañía comercial que lo desarrolla y distribuye, bajo el nombre MySQL AB, de los fundadores de MySQL y principales desarrolladores, se estableció originalmente en Suecia por David Axmark, Allan Larsson y Michael "Monty" Widenius.

La parte "AB" del nombre de la compañía es el acrónimo sueco "*aktiebola*", o *stock company* o sociedad anónima. Se traduce como MySQL Inc o MySQL SA. De hecho, MySQL Inc. y MySQL GmbH son ejemplos de empresas subsidiarias de MySQL AB. Están establecidas en los E.E. U.U. y Alemania, respectivamente.

Esta compañía posee el *copyright* o derecho de copia del código fuente de MySQL, de su logo y de sus manuales.

En enero de 2008, la compañía Sun Microsystems anunció un acuerdo para adquirir la empresa MySQL AB. El acuerdo fue hecho por un total de 1 000 millones de dólares, aproximadamente, que se incluirán al mercado de base de datos estimado en 15 000 millones de dólares que tiene Sun. Entonces, Jonathan Schwartz (CEO y presidente de Sun Microsystems) dijo: *"Apoyando nuestro plan de crecimiento general, la adquisición de MySQL amplifica nuestras inversiones en las tecnologías demandadas por los titanes de los medios en Internet"*.

Los aspectos destacados de MySQL son:

- Es *Open Source*; lo que significa que es posible para cualquier persona utilizar y modificar el software. Puede bajarse desde Internet y usarse sin costo, puede estudiarse el código fuente y modificarlo para adapatarlo a necesidades particulares. El software MySQL usa la licencia GPL (GNU *General Public License*), aunque también se ofrece una licencia comercial.
- Es rápido, fiable y fácil de usar. MySQL Server se desarrolló, originalmente, para tratar grandes bases de datos mucho más rápido que soluciones existentes y ha sido usado con éxito en entornos de producción de alto rendimiento durante varios años. MySQL Server ofrece hoy en día una gran cantidad de funciones. Su conectividad, velocidad y seguridad hacen que MySQL Server sea altamente apropiado para acceder a bases de datos en Internet.
- Es un sistema cliente/servidor, que consiste en un servidor SQL *multi-threaded* que trabaja con diferentes *bakends*, programas y bibliotecas cliente, herramientas administrativas y un amplio abanico de interfaces de programación para aplicaciones (APIs).

También, a partir de la documentación oficial, dentro de lo que se informa como las características principales del motor de base de datos MySQL, destacaremos las siguientes:

- Escrito en C y en C++.
- Probado con un amplio rango de compiladores diferentes.
- Funciona en diferentes plataformas.
- APIs disponibles para C, C++, Eiffel, Java, Perl, PHP, Python, Ruby y Tcl.
- Uso completo de *multi-threaded* mediante *threads* del *kernel*. Pueden usarse fácilmente multiples CPUs si están disponibles.
- Proporciona sistemas de almacenamiento transaccionales y no transaccionales.
- Usa tablas en disco *B-tree* (MyISAM) muy rápidas con compresión de índice.
- Relativamente sencillo de añadir otro sistema de almacenamiento. Esto es útil si desea añadir una interfaz SQL para una base de datos propia.
- Un sistema de reserva de memoria muy rápido basado en *threads*.
- *Joins* muy rápidos usando un *multi-join* de un paso optimizado.
- Tablas *hash* en memoria, que son usadas como tablas temporales.
- Las funciones SQL están implementadas usando una librería altamente optimizada y deben ser tan rápidas como sea posible. Normalmente, no hay reserva de memoria tras toda la inicialización para consultas.
- El código MySQL se prueba con Purify (un detector de memoria perdida comercial) así como con Valgrind, una herramienta.
- El servidor está disponible como un programa separado para usar en un entorno de red cliente/servidor. También está disponible como biblioteca y puede ser incrustado (vinculado o enlazado) en aplicaciones autónomas. Estas aplicaciones pueden usarse por sí mismas o en entornos donde no hay red disponible.
- Un sistema de privilegios y contraseñas que es muy flexible y seguro y que permite verificación basada en el *host*. Las contraseñas son seguras porque todo el tráfico de contraseñas está encriptado cuando se conecta con un servidor.
- Soporte a grandes bases de datos. MySQL Server es utilizado con bases de datos que contienen 50 millones de registros. También se conocen usuarios que usan MySQL Server con 60 000 tablas y cerca de 5 000 000 000 000 de registros.
- Se permiten hasta 64 índices por tabla (32 antes de MySQL 4.1.2). Cada índice puede consistir desde 1 hasta 16 columnas o partes de columnas.

- Los clientes pueden conectar con el servidor MySQL usando *sockets* TCP/IP en cualquier plataforma. En sistemas Windows de la familia NT (NT,2000,XP o 2003), los clientes pueden usar *named pipes* para la conexión. En sistemas Unix, los clientes pueden conectar usando archivos socket Unix.
- En MySQL 5.0, los servidores Windows soportan conexiones con memoria compartida.
- La interfaz para el conector ODBC (MyODBC) proporciona a MySQL soporte para programas clientes que usen conexiones ODBC (*Open Database Connectivity*).
- La interfaz para el conector J MySQL proporciona soporte para clientes Java que usen conexiones JDBC.
- El servidor puede proporcionar mensajes de error a los clientes en muchos idiomas.
- Soporte completo para distintos conjuntos de caracteres, incluyendo latin1 (ISO-8859-1), german, big5, ujis y más.
- Todos los datos se guardan en el conjunto de caracteres elegido. Todas las comparaciones para columnas normales de cadenas de caracteres son *case-insensitive*.
- El ordenamiento se realiza acorde con el conjunto de caracteres elegido.
- MySQL server tiene soporte para comandos SQL para corroborar, optimizar y reparar tablas.

Fig. 6-4. Logo de MySQL. El nombre del delfín es “Sakila”, elegido por los usuarios en un concurso.

Funciones para el registro de contraseñas.

Con respecto a funciones para la encriptación de contraseñas disponibles en este motor de base de datos, se aclara, en primera instancia, que MySQL provee una fun-

ción llamada `PASSWORD()`, pero ésta se utiliza para manejar los *passwords* o contraseñas de los usuarios del sistema (registrados en la columna *password* de la tabla *user*, es decir, dentro del sistema de autenticación del servidor MySQL).

La documentación recomienda no utilizarla en aplicaciones propias. Sugiere el uso de las funciones `MD5()` o `SHA1()` –que veremos más adelante– en su lugar.

Aunque no sea recomendada para usos distintos de la actualización o manejo de la información de autenticación de MySQL, con el ánimo de no omitir los detalles de una función relacionada con el tema en cuestión, definiremos la función y daremos un ejemplo.

La función se define de la manera siguiente:

```
PASSWORD(str)
```

Retorna la cadena de caracteres de *password* o contraseña cifrada, a partir del texto-plano de la contraseña en el parámetro `str`. La encriptación es de una vía (no reversible).

Ejemplo:

```
SELECT PASSWORD('Contraseña');
```

Existe además una función llamada `OLD_PASSWORD()` para contemplar la codificación o encriptación utilizada en versiones de MySQL anteriores a la 4.1. Su utilización es idéntica a la de la función anterior. Veamos un ejemplo:

```
mysql> SELECT OLD_PASSWORD('Contraseña');
        -> '49b282fb660b3b7c'
```

Antes de pasar a describir las funciones de *hashing* disponibles en MySQL, lo recomendado para el registro de *passwords* o contraseñas, veremos rápidamente la función `ENCRYPT()`. MySQL la incorpora para mantener compatibilidad con los sistemas de contraseñas de los sistemas Unix; en Windows, por ejemplo, esta función devolverá siempre NULL. Su definición es como sigue:

```
ENCRYPT(str [ ,salt])
```

Encriptará la cadena de caracteres `str` usando la llamada al sistema (Unix) `crypt()` y retornará la cadena encriptada. El argumento opcional `salt` deberá ser una cadena de, al menos, dos caracteres de longitud. Si no se especifica, un *salt* aleatorio será utilizado.

Ejemplo de uso:

```
SELECT ENCRYPT('Contraseña');
```

Veamos ahora sí las funciones que hemos de utilizar a la hora de registrar contraseñas de nuestras aplicaciones en MySQL. Éstas no son otras que las funciones de *hashing* criptográfico o funciones de una vía que implementan los algoritmos MD5 y SHA-1 llamadas, convenientemente, `MD5()` y `SHA1()` o `SHA()` –las dos últimas son sinónimos–.

La documentación de MySQL hace una nota respecto de los *exploits* (códigos o programas que explotan una vulnerabilidad o problema de seguridad) publicados respecto de los agoritmos MD5 y SHA-1, que hemos comentado en los capítulos introductorios, y advierte al usuario que considere la utilización de la alternativa SHA-2, disponible a partir de la versión 6 de MySQL.

Además, debe tenerse en cuenta que aquí también aplican los conceptos vistos respecto al *salt* para evitar la eficacia de ataques por diccionario, aunque queda a cargo del usuario implementar alguna metodología que se valga de ese mecanismo.

```
MD5(str)
```

Esta es la definición de la función `MD5()`. Calculará la salida de 128 bits generada a partir de la cadena de caracteres pasada como parámetro en `str`. El valor es retornado como una cadena de 32 caracteres de longitud (como es convención, en formato hexadecimal). Retornará NULL en el caso de que el parámetro sea NULL. La documentación aclara que se trata de una implementación del "*RSA Data Security Inc. MD5 Message-Digest Algorithm*."

Veamos un ejemplo:

```
SELECT MD5('Contenido de prueba');
```

Veamos ahora la definición de la función `SHA1()` (o `SHA()`, que se utiliza como sinónimo):

```
SHA1(str), SHA(str)
```

Calculará la salida del algoritmo SHA-1, de acuerdo con la RFC 3174 (*Secure Hash Algorithm*), de 160 bits de longitud. Retornará este valor como una cadena de 40 caracteres en formato hexadecimal. Como con la función `MD5()`, si el parámetro es NULL, la función retornará NULL.

Ejemplos:

```
SELECT SHA1('Contenido de prueba');
```

```
SELECT SHA('Contenido de prueba');
```

Por último veremos lo referente a la función `SHA2()`, disponible a partir de la versión 6.0.5 de MySQL. Como queda establecido en la documentación oficial, esta función puede ser considerada criptográficamente más segura que `MD5()` y `SHA1()`.

```
SHA2(str, hash_length)
```

La función calculará el *hash* criptográfico utilizando un algoritmo de la familia SHA-2 (SHA-224, SHA-256, SHA-384 y SHA-512). El primer argumento o parámetro es la cadena de caracteres a partir de la cual se pretende obtener el *hash* (`str`); el segundo indica la longitud, en bits, del resultado deseado, que deberá ser alguno de los siguientes valores: 224, 256, 384 o 512. Si alguno de los parámetros es NULL, o se especifica una longitud de salida no válida, la función retornará NULL. Caso contrario, retornará el valor resultado, representado en una cadena de caracteres en formato hexadecimal.

Ejemplo de uso:

```
SELECT SHA2('Contenido de prueba', 224);
```

Nótese que esta función se ejecutará sólo si MySQL fue configurado con soporte para SSL.

Funciones para cifrado simétrico de datos.

MySQL provee, para la encriptación y desencriptación simétrica, las implementaciones de dos algorirmos: TripleDES y AES. Estas implementaciones constan de un par de funciones (una para encriptar y otra para desencriptar) para cada uno de los dos algoritmos. Se describirán a continuación.

```
DES_ENCRYPT(str [ ,{ key_num|key_str}] )
```

Encriptará la cadena de caracteres con la contraseña, llave también en este caso, usando el algoritmo triple-DES. En caso de error, retornará NULL.

Nótese que esta función ejecutará sólo si MySQL fue configurado con soporte para SSL.

La contraseña o llave para encriptar se determina según el segundo argumento de `DES_ENCRYPT()`: Si no se ha especificado ninguno, se usará la primera contraseña del archivo de contraseñas DES. Si fuese un número (0-9), se utilizará esa contraseña, es decir, la correspondiente a ese número, definida también en el archivo DES. Si se usa una cadena de caracteres, se usará la misma para encriptar el parámetro `str`.

El archivo de contraseñas o claves puede especificarse con la opción de servidor `--des-key-file`.

La cadena de caracteres retornada tiene como primer carácter un `CHAR(128 | key_num)`. Se añade el número 128 para hacer más sencillo reconocer una contraseña o clave encriptada. Si se usa una cadena de caracteres, `key_num` es 127.

La longitud de la cadena para el resultado es `new_len = orig_len + (8-(orig_len % 8))+1`.

Cada línea en el archivo de contraseñas o claves DES tiene el siguiente formato:

```
key_num des_key_str
```

Cada `key_num` debe ser un número en el rango de 0 a 9. Las líneas en el archivo pueden estar en cualquier orden; `des_key_str` es la cadena de caracteres que se usa para encriptar la información. Entre el número y la clave debe haber un espacio como mínimo. La primera clave es aquella usada por defecto si no especifica ningún argumento o parámetro de clave para `DES_ENCRYPT()`.

Se puede especificar al motor MySQL que lea un nuevo valor de clave del archivo clave con el comando `FLUSH DES_KEY_FILE`. Esto necesita el privilegio `RELOAD`.

Un beneficio de tener un conjunto de contraseñas o claves por defecto es que se les permite a las aplicaciones una forma de corroborar la existencia de valores de columna encriptados, sin dar al usuario la posibilidad de desencriptarlos.

Ejemplo para un comprobación de este tipo:

```
mysql> SELECT campo1 FROM tabla1
     > WHERE campo2 =
     > DES_ENCRYPT('1234');
DES_DECRYPT(crypt_str [ ,key_str])
```

Desencriptará una cadena encriptada con `DES_ENCRYPT()`. En caso de error, esta función retornará NULL.

Como con la función anterior, debe notarse que esta función actuará sólo si MySQL fue configurado con soporte para SSL.

Si no se especifica el parámetro `key_str, DES_DECRYPT()` examina el primer byte de la cadena encriptada para determinar el número de clave DES que se usó para encriptar la cadena original y, luego lee la clave del archivo de contraseñas o claves DES para desencriptar la información. Para que esto funcione, el usuario debe tener el privilegio `SUPER`. El archivo puede especificarse con la opción del servidor `--des-key-file`.

Si se especifica el argumento `key_str`, esta cadena se usa como la contraseña (clave o llave, en este caso) para desencriptar la información.

Si el argumento `crypt_str` no parece una cadena encriptada, MySQL retorna `crypt_str`.

```
AES_ENCRYPT(str,key_str)
```

Esta función encriptará datos utilizando el algoritmo oficial AES (*Advanced Encryption Standard*), conocido anteriormente como "Rijndael". Se implementó por defecto una encriptación con una clave o llave de 128 bits de longitud, pero es posible ampliarlo hasta 256 bits modificando el código fuente. La documentación oficial aclara que se ha seleccionado una longitud de 128 bits por cuestiones de velocidad y considera que, de momento, sería suficientemente seguro.

Los parámetros o argumentos de entrada pueden ser de cualquier longitud. Si algún argumento es NULL, el resultado de esta función también es NULL.

Debido a que AES es un algoritmo de cifrado por bloques, se utilizará *padding* o relleno para cadenas de longitud no correspondiente y así la longitud de la cadena resultante puede calcularse como `16 * (trunc(string_length / 16) + 1)`.

Si `AES_DECRYPT()` detecta datos inválidos o un *padding* incorrecto, retornará NULL. También es posible que `AES_DECRYPT()` retorne un valor no NULL (posiblemente basura) si los datos de entrada o la clave/llave son inválidos.

Es posible utilizar la función `AES_ENCRYPT()` para almacenar datos de forma encriptada modificando las sentencias SQL. Por ejemplo:

```
INSERT INTO tabla1 VALUES (1, AES_ENCRYPT('texto',
  'contraseña'));
```

Es posible obtener, incluso, mejor seguridad si no se transfiere la contraseña a través de la conexión por cada consulta; esto puede hacerse almacenando la clave en una variable del servidor al realizar la conexión. Por ejemplo:

```
SELECT @contrasena:='contraseña';
INSERT INTO tabla1 VALUES (1, AES_ENCRYPT('texto',
 @contrasena));
```

```
AES_DECRYPT(crypt_str,key_str)
```

Esta función posibilitará la desencriptación de la información usando el algoritmo oficial AES (*Advanced Encryption Standard*) que, probablemente –salvo que utilice una implementación fuera de la provista por la base de datos–, ha sido encriptada previamente mediante la función `AES_ENCRYPT()`, descripta anteriormente.

Según la documentación oficial, `AES_ENCRYPT()` y `AES_DECRYPT()` pueden considerarse las funciones de cifrado simétrico criptográficamente más seguras disponibles en MySQL.

[1] Peterson, Erich; LI, Siqing. *An Overview of Cryptographic Systems and Encrypting Database Data* [en línea]. 4 Guys From Rolla. Febrero de 2007.
<http://aspnet.4guysfromrolla.com/articles/021407-1.aspx>
[Consulta: Junio 2008].

7

Criptografía en el nivel de aplicación

Veremos en este capítulo algunas aplicaciones o programas informáticos que hacen uso de diferentes técnicas criptográficas, incluyendo la encriptación de archivos, canales o túneles seguros e implementaciones de cifrado asimétrico para correo electrónico.

Se incluye esta información con el ánimo de que el lector no reinvente la rueda en el caso de verse en la necesidad de implementar un sistema de características similares a los programas que se detallarán brevemente a continuación. Son programas de uso libre, y si bien a lo largo del libro se ha descripto cómo es posible incorporar funcionalidades criptográficas a nuestras aplicaciones desarrolladas en Java, .NET y/o PHP, es posible también –y de hecho muy común en la práctica– hacer que nuestra aplicación se valga de programas externos, generalmente por *command line* o línea de comandos, para realizar, por ejemplo, una comunicación segura (SSH), la transferencia de un archivo sobre un canal encriptado (SSH/SCP/SFTP) o el envío de un *e-mail* usando criptografía asimétrica (PGP).

Si bien se harán referencias a algunas alternativas comerciales, el foco estará puesto sobre las aplicaciones, servicios o programas *open-source* o de código abierto, disponibles gratuitamente en Internet, de uso libre.

Al hablar de criptografía en el nivel de aplicación, también podrían considerarse a los protocolos criptográficos estándares de alto nivel, basados en aquellos otros que ya hemos visto. Estos protocolos de alto nivel, entonces, determinarán cómo han de comunicarse las partes, qué algoritmos de cifrado podrán utilizarse, especificaciones de formato de mensajes, etc. Dado el carácter práctico del libro, este capítulo, en realidad, tratará principalmente sobre aplicaciones que implementaron estos protocolos, algunos de los cuales han sido definidos formalmente o estandarizados luego o sobre la marcha, es decir, después de aparecido en primera instancia el programa que lo implementaba.

SSL/TLS, SET y OpenSSL toolkit.

El crecimiento explosivo de la *World Wide Web* para el comercio electrónico y la publicación de información de la más variada índole generó la necesidad de implementar diversos mecanismos de seguridad, varios de ellos basados en técnicas de criptografía moderna.

Existen en la actualidad, en relación con lo anterior, dos estándares principales: *Secure Sockets Layer* (SSL) o Capa Segura de Sockets y *Secure Electronic Transaction* (SET) o Transacción Electrónica Segura.

OpenSSL se trata de un proyecto *open-source* o de código abierto, libre, que además de implementar un *toolkit* (kit o caja de herramientas) para los protocolos SSL v2/v3 y TLS v1, implementa una librería criptográfica de uso general sin restricciones.

SSL / TLS.

Secure Sockets Layer (SSL) provee servicios de seguridad entre la capa TCP y las aplicaciones que hacen uso de esa capa. La versión para Internet de este protocolo, que sucede a la anterior, actualmente es llamada *Transport Layer Service* (TLS) o Capa de Servicio de Trasporte.

Las diferencias entre el protocolo SSL 3.0 o v3 (por versión 3) –desarrollado por Netscape y publicado en el año 1996– y el protocolo TLS v1 (versión 1) –estándar del IETF, RFC 2246– son menores, por lo que se tratarán de aquí en más indistintamente.

SSL/TLS provee entonces confidencialidad, lográndola con criptografía simétrica y controlando la integridad de los datos, ya que utiliza un MAC (*Message Authentication Code* o Código de Autenticación de Mensaje). El protocolo incluye, además, la funcionalidad para permitir a dos usuarios de la capa TCP determinar cuáles mecanismos de seguridad y servicios serán utilizados.

El proceso de comunicación del protocolo establece, en primera instancia, el acuerdo o negociación entre las partes de los algoritmos que serán utilizados. Luego, se procede al intercambio de llaves públicas y a la autenticación basada en certificados digitales para, por último, cifrar de manera simétrica los datos o información para transferir.

En cuanto a los protocolos –de nivel inferior o sobre los cuales SSL se apoya– que serán utilizados, las opciones son varias. Recuérdese que, en las fases iniciales, el cliente y el servidor negociarán qué algoritmos criptográficos utilizarán para comunicarse o autenticarse. Las opciones en funciones de *hash* son MD5 y SHA; para criptografía simétrica son opciones los algoritmos RC2, RC4, IDEA, DES, TripleDES y AES; y, por último, para la asimétrica, RSA, Diffie-Hellman, DSA y Fortezza.

SET.

SET (por *Secure Electronic Transaction*) se trata, en cambio, de una especificación de seguridad abierta, diseñada para proteger las transacciones realizadas con tarjetas de crédito a través de Internet.

El protocolo se especificó como estándar de seguridad luego de la solicitud por parte de VISA y de MasterCard en el año 1996.

En resumen, el protocolo implementa técnicas criptográficas entre las que se incluyen los certificados digitales y criptografía de llave pública, que posibilitan a las entidades las autenticaciones necesarias y la comunicación segura.

Si bien el protocolo pareció en camino a establecerse como un estándar de facto hacia el año 2000, cuestiones inherentes a la especificación –necesidad de instalar un software cliente por ejemplo– hicieron que las compañías de tarjetas de crédito empezaran a promocionar otras alternativas.

En la actualidad, a consecuencia de su simplicidad y bajo costo principalmente, las implementaciones de comercio electrónico en Internet, en su gran mayoría, se basan en la seguridad proporcionada por el protocolo SSL.

OpenSSL.

En este apartado se tratará este proyecto *open-source* o de código abierto, que implementa funciones criptográficas sin limitaciones dentro de una librería y que provee, además, una utilidad (un programa específicamente) para que, por línea de comandos, sea posible realizar diversas tareas, como encriptar y desencriptar un archivo, calcular su *hash*, etc. Veremos ejemplos de cómo usar esta herramienta más adelante.

El sitio del proyecto es `http://www.openssl.org/` y, como desde allí se detalla, OpenSSL es un esfuerzo colaborativo para el desarrollo, un *toolkit* o caja de herramientas, *open-source*, robusta, de grado comercial y de funcionalidad completa, que implementa los protocolos SSL v2/v3 y TLS v1, así como una librería o biblioteca criptográfica de propósito general. El proyecto se gestiona a través de una comunidad mundial de voluntarios que utilizan Internet para comunicarse, planear y desarrollar el conjunto de herramientas y su documentación.

OpenSSL se basa en la librería SSLeay desarrollada por Eric A. Young y Tim J. Hudson. El conjunto de herramientas OpenSSL está bajo una licencia del estilo Apache, lo que básicamente significa que seremos libres de obtener y utilizar OpenSSL con fines comerciales y no comerciales, estando sujetos a las condiciones de la licencia.

Por último, desde el sitio del proyecto se aclara que el software utiliza criptografía “fuerte” y que, aunque éste es creado, mantenido y distribuido en y desde países de Europa donde no es ilegal hacerlo, OpenSSL puede estar bajo ciertas restricciones de exportación, de importación y/o de uso en algunas otras partes del mundo.

Veamos ahora algunas formas de utilización de la herramienta para realizar tareas criptográficas desde la línea de comandos, que podríamos incluir sencillamente dentro de nuestros desarrollos, aplicaciones o servicios. Esto en tanto, por supuesto, esté el *toolkit* instalado en el equipo en el cual se ejecute (también podríamos distribuir OpenSSL junto con nuestro programa, en tanto mantengamos las referencias a sus creadores).

El programa para realizar estas tareas desde el *shell* se llama "openssl" y, a través de él, es posible:

- Crear y manejar llaves privadas, públicas y parámetros.
- Realizar operaciones criptográficas de llave pública.
- Crear certificados X.509 (CSRs y CRLs).
- Calcular resúmenes de mensajes (*hashing*).
- Encriptar y desencriptar con diferentes algoritmos de criptografía simétrica.
- Testear clientes y servidores SSL/TLS.
- Manejar *e-mails* firmados o encriptados S/MIME.
- Requerir, generar y verificar fechados.

Ejecutamos el *shell* de Windows, línea de comandos o símbolo de sistema. Nos posicionamos sobre el directorio "bin" dentro de la carpeta seleccionada para la instalación ("C:\OpenSSL" por defecto) para ejecutar el programa, aunque por supuesto, puede incluirse a esta carpeta en la variable de entorno PATH. Veremos la versión instalada y luego qué parámetros y algoritmos soporta.

Una lista completa de todos los comandos soportados puede obtenerse desde `http://www.openssl.org/docs/apps/openssl.html`. Aquí, a continuación, veremos algunos ejemplos generales para describir algunos de los posibles usos de esta herramienta.

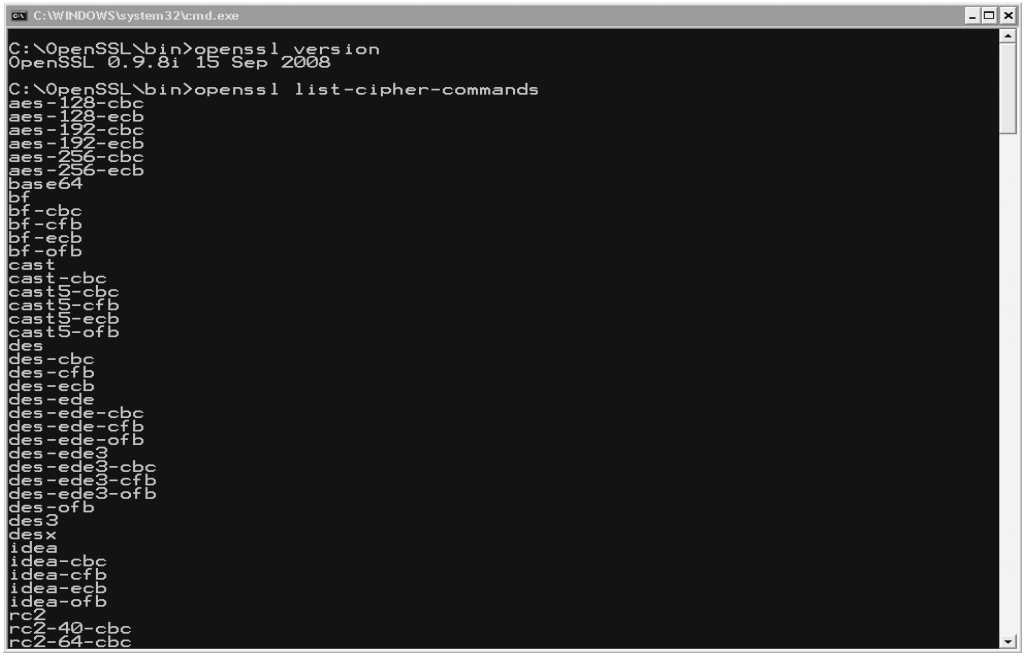

Fig. 7-1. Utilitario OpenSSL para línea de comandos: Versión, funciones y algoritmos soportados para el cifrado simétrico.

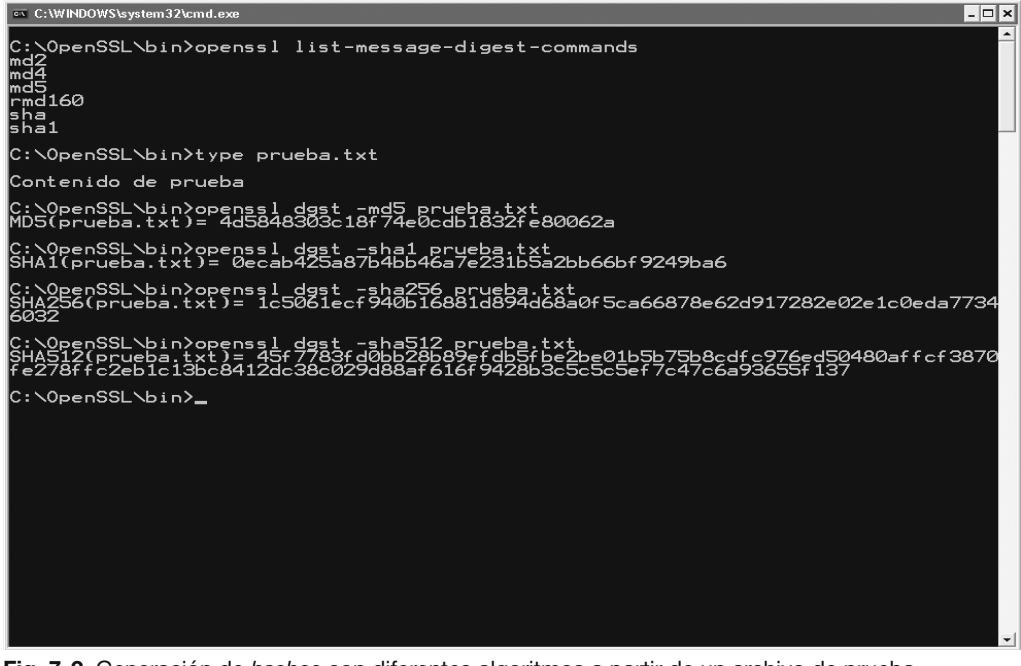

Fig. 7-2. Generación de *hashes* con diferentes algoritmos a partir de un archivo de prueba.

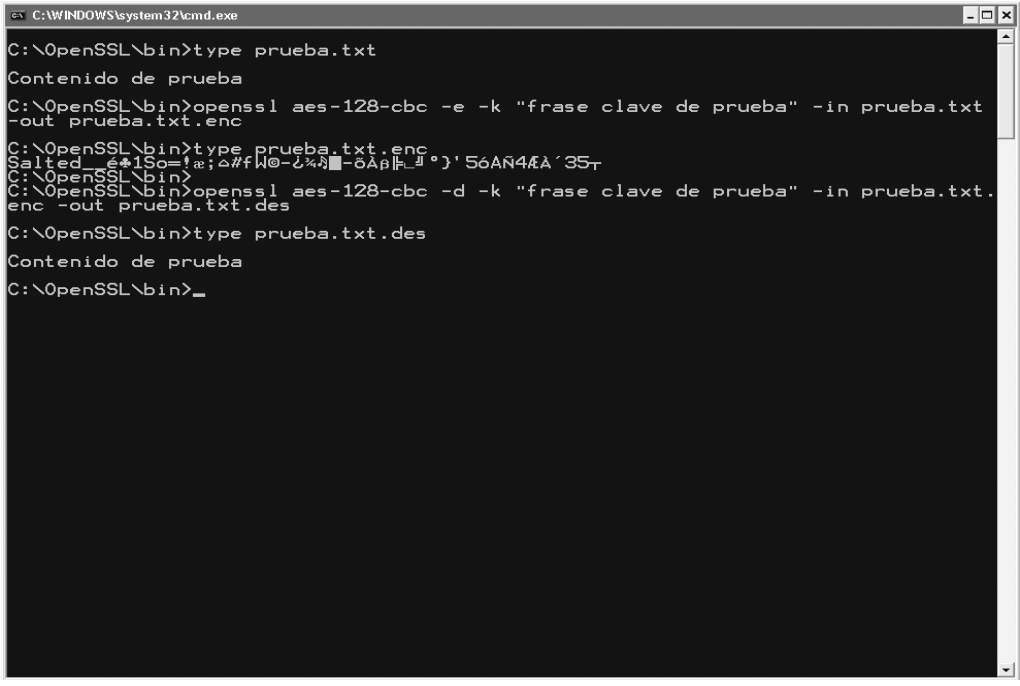

Fig. 7-3. Encriptación utilizando el algoritmo AES con una llave de 128 bits de longitud en modo CBC.

PGP, estándar OpenPGP y GnuPG.

PGP.

La historia se remonta al año 1991, cuando Phil Zimmermann autorizó la publicación de sistemas BBSs y foros de USENET, un programa de dominio público llamado *Pretty Good Privacy* (PGP).

El programa incluía una implementación del algoritmo de RSA. El autor la había desarrollado basándose en la información, abiertamente publicada del algoritmo, unos diez años atrás. La compañía RSA amenazó con demandar a Zimmermann, por lo que éste prometió no continuar con el desarrollo de PGP. Sin embargo, programadores de diferentes partes del mundo lo continuaron y lo portaron o implementaron en múltiples plataformas y sistemas operativos.

Más adelante, el MIT decide negociar con Zimmermann y RSA una licencia para el uso no comercial de PGP, lo que da lugar a la versión 2.5, pero que no podía ser exportada.

Stale Schumacher crea posteriormente la versión internacional de PGP, compatible con la de MIT.

Siguieron diversos acuerdos entre el autor original, el equipo que participó en el desarrollo, otras compañías comerciales y el MIT. Actualmente, el nombre PGP lo utiliza PGP Corporation, desde donde es posible descargar gratuitamente PGP *Desktop Trial Software*, software de evaluación de PGP. Pasado este tiempo de evaluación –30 días–, esta herramienta sólo posibilita las opciones básicas de PGP, debiendo el usuario pagar la licencia en el caso de querer utilizar las funcionalidades extra agregadas. Más información se puede hallar en el sitio Web oficial `http://www.pgp.com/`.

A su vez, para consultar las últimas novedades y tener acceso a las últimas versiones disponibles de la versión internacional de PGP, puede consultarse el siguiente sitio: `http://www.pgpi.org/` (The International PGP Home Page).

OpenPGP.

Fue en julio de 1997 cuando PGP Inc. propone a la IETF que se desarrolle un estándar, llamado OpenPGP, para la normalización del protocolo y evitar el potencial caos dada la creciente aparición de diversas implementaciones de PGP. La compañía le cedió a la IETF el permiso para utilizar el nombre OpenPGP para describir esta nueva norma, así como cualquier programa que la soportara. El IETF aceptó la propuesta e inició el Grupo de Trabajo OpenPGP.

OpenPGP es el estándar de cifrado de correo electrónico más ampliamente utilizado en la actualidad. Se define y mantiene por el grupo de trabajo OpenPGP de la *Internet Engineering Task Force* (IETF), en el documento RFC 4880. Esta norma estuvo originariamente basada en derivados del programa PGP, visto anteriormente en este capítulo, creado por Phil Zimmermann en el año 1991.

The OpenPGP Alliance, o Alianza OpenPGP, es un grupo creciente de empresas y otras organizaciones que llevan adelante e implementan la norma propuesta para OpenPGP. La Alianza trabaja para facilitar la interoperabilidad técnica y la comercialización de la sinergia entre diferentes implementaciones de OpenPGP.

El protocolo OpenPGP define los formatos estándares de cifrado de mensajes, firmas y certificados para el intercambio de llaves públicas.

Es posible consultar más información en el sitio Web oficial de la Alianza OpenPGP en la siguiente dirección: `http://www.openpgp.org/`

GnuPG.

GnuPG, por *GNU Privacy Guard*, es la implementación completa y libre del estándar OpenPGP por parte del proyecto GNU, tal como se define en la RFC 4880. GnuPG permite cifrar y firmar datos y comunicaciones. Cuenta además con un versátil sistema

de gestión de llaves, así como con módulos de acceso para todo tipo de directorios de llave pública. GnuPG, también conocido como GPG, es una herramienta que se opera por línea de comandos con las características necesarias para que la integración con otras aplicaciones resulte muy sencilla. Este es el aspecto destacable en lo que al objetivo de este libro se refiere. Podemos hacer uso de esta herramienta desde nuestras aplicaciones y conformar con la norma vigente más utilizada actualmente para el envío de correo electrónico de manera segura. Paralelamente, provistas por terceros, existe una gran cantidad de aplicaciones gráficas y librerías que hacen uso de esa interfaz. La versión 2 de GnuPG también soporta a S/MIME. GnuPG es software libre; puede ser libremente utilizado, modificado y distribuido bajo los términos de la licencia GNU (*General Public License)*.

El programa encripta los mensajes usando pares de llaves asimétricas individualmente generados por los usuarios. Las llaves públicas pueden ser intercambiadas con otros usuarios en una variedad de formas, tales como los servidores de llaves disponibles en Internet. Estas llaves deben ser intercambiadas con ciertos recaudos para evitar la suplantación o impostura de identidad por corrupción de las correspondencias entre identidades y llaves públicas. También, es posible añadir una firma digital criptográfica a un mensaje, de modo que el mensaje del remitente y la integridad pueden ser verificadas.

GnuPG no se basa en algoritmos patentados o restringidos de modo alguno. En lugar de ello, puede utilizar, por ejemplo, los algoritmos CAST5, Triple DES, AES, Blowfish y Twofish.

Éste es un software de cifrado híbrido, en tanto que el programa utiliza algoritmos de criptografía simétrica por cuestiones de velocidad, como así también a la criptografía de llave pública, para asegurar la facilidad de intercambio de llaves, normalmente mediante el uso de la llave pública del destinatario para cifrar una clave de sesión que sólo se utilizará una vez. Este modo de operación es parte del estándar OpenPGP y ha sido parte del PGP desde su primera versión.

Más información y la descarga del programa, desde el sitio Web oficial de The GNU Privacy Guard (`http://www.gnupg.org/`).

Veamos, por último, algunas capturas de pantalla que ejemplifican cómo se crean las llaves (llave pública y llave privada) y cómo se firma un documento (archivo en este caso); siempre mediante el utilitario de línea de comandos.

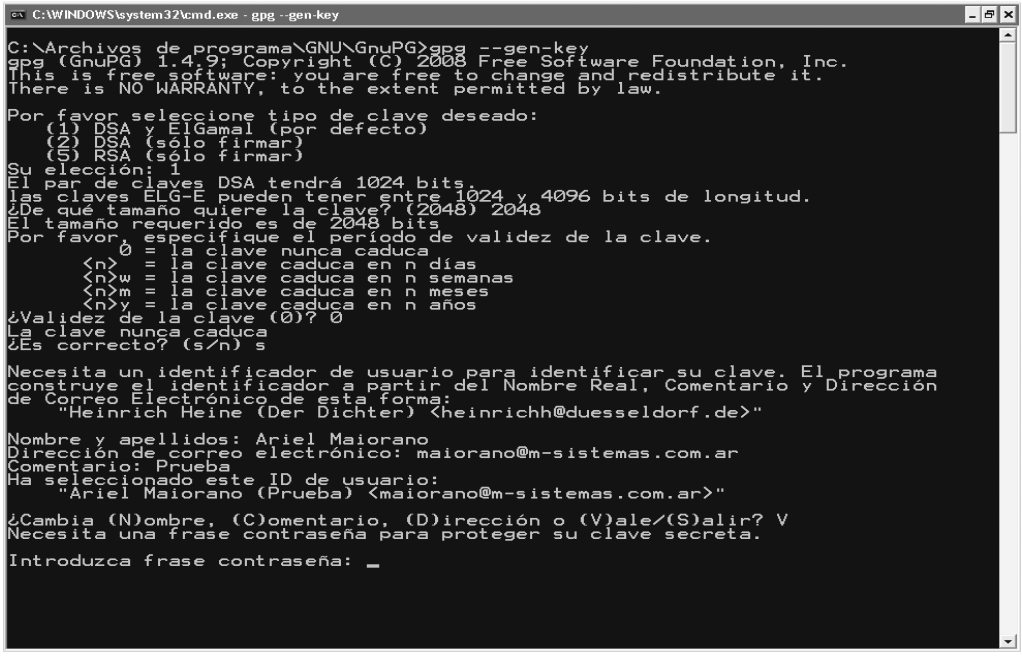

Fig. 7-4. Generación del par de llaves (pública y privada) para la encriptación, desencriptación y firma digital.

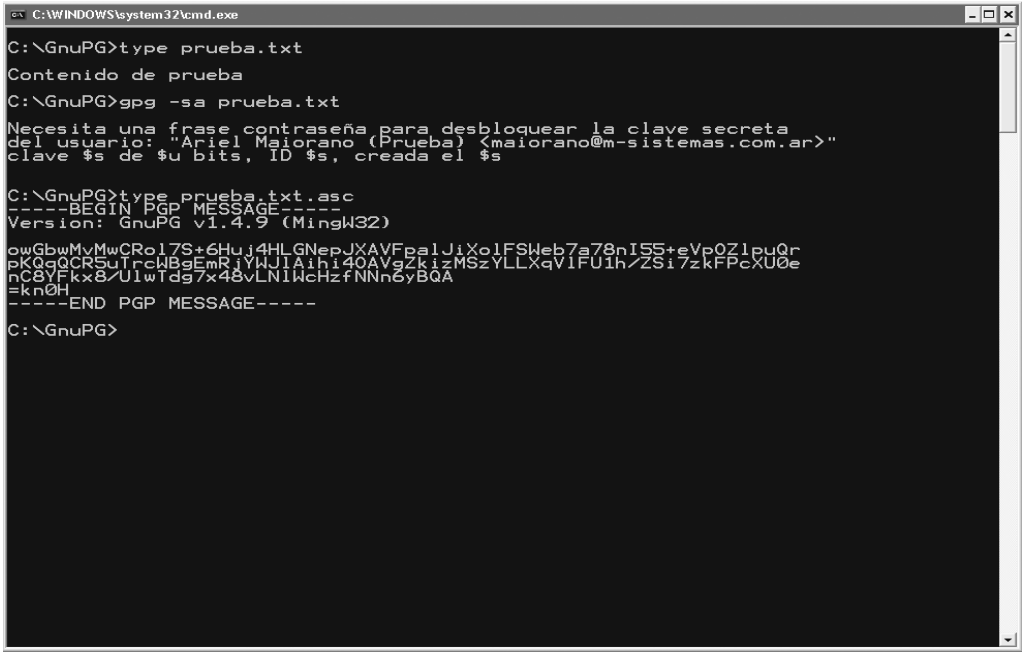

Fig. 7-5. Firma digital de un archivo de prueba; formato ASCII.

SSH y Herramientas OpenSSH.

SSH.

SSH es una manera segura de conectarse a otro sistema informático. Se trata de un protocolo que implementa una alternativa segura al tradicional servicio “telnet” de los sistemas Unix. Comúnmente, es referido simplemente como “ssh”, tanto al aplicativo como al protocolo de comunicación subyacente. Se entienden, por lo general, como parte del paquete “ssh” a un utilitario para la copia segura de archivos (“scp”) y un cliente ftp seguro (“sftp”).

El protocolo SSH también permite establecer conexiones seguras con otros programas o protocolos mediante el uso de la redirección de puertos, también llamados túneles. Esto significa que los protocolos de correo por ejemplo, entre otros programas, pueden utilizar mecanismos de cifrado entre las dos partes involucradas en la comunicación.

SSH utiliza diversas técnicas de cifrado o encriptación que hacen que la información que viaja por el medio de comunicación inseguro lo haga de manera aparentemente no legible y ninguna tercera persona pueda descubrir el usuario y la contraseña de la conexión, ni lo que se escribe durante la sesión.

La versión 2 del protocolo (vigente actualmente) se encuentra especificada, en diferentes capas, en los RFCs 4251, 4252 y 4253. Los algoritmos de encriptación y de compresión pueden negociarse en cada comunicación. Además del basado en *passwords* o contraseñas, ssh permite la utilización de llaves DSA y RSA para la autenticación. Utiliza Diffie-Hellman para el acuerdo de llaves, salvando la necesidad de un servidor de llaves o de distribución de llaves.

OpenSSH.

OpenSSH es una versión libre y *open-source* de las herramientas SSH para diferentes plataformas Unix. OpenSSH ofrece, además, la capacidad de generar túneles de seguridad y varios métodos de autenticación y soporta todas las versiones actuales del protocolo SSH.

La serie de programas de los que se compone OpenSSH, que reemplazarían a las más antiguas como rlogin y/o telnet, rcp, ftp, son, respectivamente, ssh (cliente), scp y sftp. El servicio se incluye como el programa sshd, este es el daemon o servicio desatendido que proporciona el acceso al equipo local. Por último, se encuentran utilitarios extra, de uso general, como ssh-add, ssh-agent, ssh-keysign, ssh-keyscan, ssh-keygen y sftp-server.

OpenSSH es desarrollado y mantenido por el proyecto OpenBSD (sistema operativo Unix BSD orientado hacia la seguridad –`http://www.openbsd.org/`–). El software se desarrolla en países que permiten la exportación de criptografía. Puede ser libremente utilizado bajo una licencia BSD.

Puede consultarse más información al respecto desde el sitio Web oficial en `http://www.openssh.org/`.

Existe una versión para Windows de estas herramientas, distribuidas por un tercero (persona no participante del proyecto OpenSSH original), disponible en el sitio Web OpenSSH for Windows (`http://sshwindows.sourceforge.net/`).

OpenSSH para Windows es un paquete gratuito que instala las herramientas de cliente y servidor del software original de OpenSSH, mediante un subconjunto mínimo del paquete Cygwin (*framework* para la ejecución de programas originalmente desarrollados para Unix en entornos Windows), sin la necesidad de contar con una instalación completa de Cygwin.

OpenSSH para Windows proporciona el paquete completo de las utilidades ssh, scp y sftp. La terminal de ssh proporciona una interfaz similar al símbolo de sistema de Windows. Para los estilos de carpetas o directorios, en scp o sftp, se mantiene la convención de Unix o Cygwin.

El programa ha sido testeado con las siguientes versiones de Windows:

- Windows NT 4.0 Workstation con SP6a.
- Windows 2000 Professional con SP3.
- Windows 2000 Server con SP3.
- Windows XP Professional.
- Windows 2003 Server, Enterprise Edition.

Veremos a continuación algunas capturas de pantallas que darán una idea de cómo funcionan o podrían utilizarse estas herramientas.

Fig. 7-6. Conexión a sistema Linux remoto (aceptación de llaves).

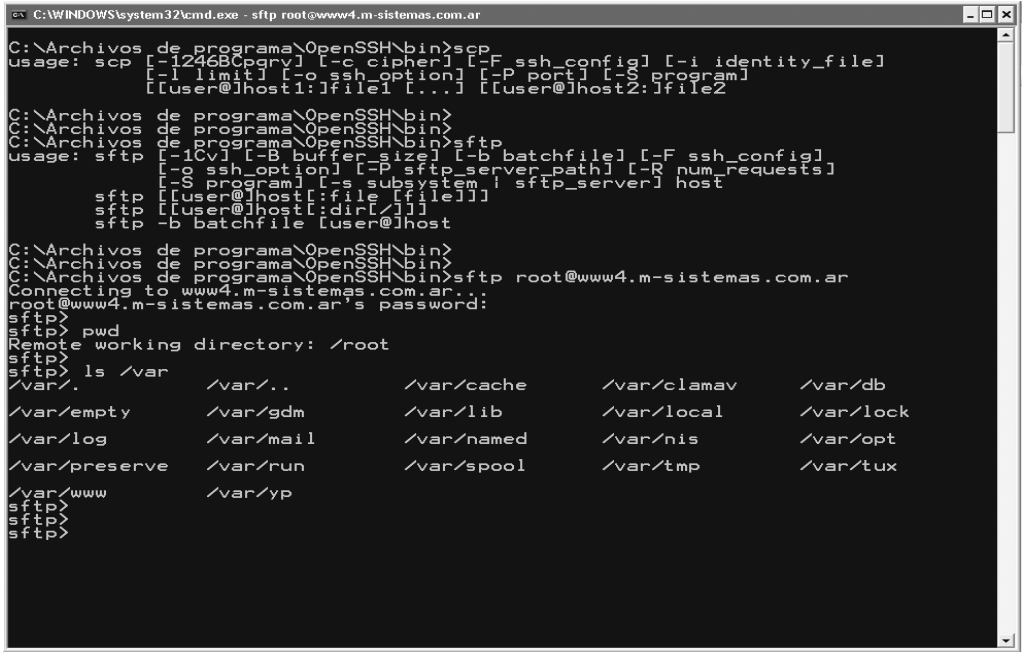

Fig. 7-7. Ayuda para la invocación de los comandos “scp” y “sftp”.

Kerberos.

Veremos en esta sección una breve introducción a Kerberos. Se trata de un protocolo de autenticación de red y está diseñado para proporcionar autenticación para aplicaciones de tipo cliente/servidor utilizando criptografía simétrica. Una implementación libre, de código abierto, de este protocolo ha sido desarrollada por el Instituto de Tecnología de Massachussets (MIT).

Muchos protocolos, aún hoy día utilizados en Internet, no proporcionan ninguna seguridad. Herramientas para hacer *sniffing* u oler las contraseñas fuera de la red son de uso común por usuarios malintencionados. Por lo tanto, las aplicaciones que envían una contraseña sin cifrar a través de la red son extremadamente vulnerables. Otras aplicaciones de tipo cliente/servidor se basan en el programa cliente para asegurar el carácter de "honestidad" sobre la identidad del usuario que lo está operando. Otras aplicaciones también se basan en el cliente para restringir sus actividades a las que le corresponde hacer, sin ningún otro control de seguridad por parte del servidor.

En muchas ocasiones se ha intentado resolver estos problemas de seguridad mediante la utilización de *firewalls*. Lamentablemente, asumir que personas malintencionadas estarán sólo en el exterior es a menudo una muy mala suposición. La mayoría de los incidentes realmente perjudiciales de los delitos informáticos es llevada a cabo por personas con información privilegiada. Los *firewalls* también tienen una importante desventaja en el sentido de que restringen a los usuarios el uso de Internet. Kerberos fue creado por el MIT como una solución a estos problemas de seguridad de la red. El protocolo Kerberos utiliza criptografía "fuerte" para que un cliente pueda probar su identidad a un servidor (y viceversa), a través de una conexión de red insegura. También pueden cifrar todas sus comunicaciones para asegurar la privacidad y la integridad de los datos.

El protocolo se basa en la criptografía de llave simétrica y requiere de un tercero de confianza. Otras extensiones de Kerberos pueden prever el uso de la criptografía de llave pública durante ciertas fases de la autenticación.

Kerberos utiliza como base el protocolo Needham-Schroeder. Se hace uso de un tercero de confianza, un denominado centro de distribución de claves, o *key distribution center* (KDC), que consta de dos partes separadas lógicamente: Un servidor de autenticación, o *Authentication Server* (AS) y un servidor de tickets garantidos, o *Ticket Granting Server* (TGS). Kerberos funciona sobre la base de estos tickets, que sirven para demostrar la identidad de los usuarios.

El KDC mantiene una base de datos de llaves secretas; cada entidad en la red –sea un cliente o un servidor– comparte una llave secreta conocida sólo por él mismo y por el KDC. El conocimiento de esta llave sirve para probar la identidad de una entidad. Para la comunicación entre dos entidades, el KDC genera una llave de sesión que pueden utilizar para garantizar sus comunicaciones.

Kerberos está disponible gratuitamente, descargable desde el MIT, bajo restricciones de permisos de derechos de autor muy similares a los utilizados para el sistema operativo BSD y el sistema X Window. La implementación de MIT de Kerberos se distribuye en forma de código fuente.

En resumen, Kerberos proporciona las herramientas de autenticación y criptografía "fuerte" para ayudar a garantizar la seguridad en sistemas de información dentro de Internet o dentro de una red no confiable.

Puede obtenerse más información en el sitio que el MIT mantiene para el proyecto en `http://web.mit.edu/kerberos/www/`.

Otras aplicaciones criptográficas.

TrueCrypt.

Se trata ésta de una utilidad libre, *open-source*, para encriptar discos o volúmenes en sistemas Windows Vista/XP, Mac OS X y Linux. Es una de las aplicaciones más populares en la materia.

Sus características principales incluyen: La posibilidad de creación de un disco virtual encriptado dentro de un archivo y la posibilidad de montarlo como uno real. Puede encriptarse una partición completa o un dispositivo de almacenamiento como un *pendrive*. Además, puede encriptar la partición donde el sistema Windows esté instalado; la encriptación es automática y transparente para el usuario. Realiza estas tareas criptográficas valiéndose de los algoritmos de cifrado simétrico AES-256, Serpent y Twofish.

Respecto de nuestro interés particular como desarrolladores, el programa provee una interfaz por línea de comandos, de manera que sería sencillo incorporarlo a, o utilizarlo desde, nuestras aplicaciones frente a la necesidad de guardar, por ejemplo, más de un archivo de manera segura –dentro de un volumen–, liberándonos de la programación del cifrado de los mismos.

Puede consultarse más información referente al programa en su sitio Web: `http://www.truecrypt.org/`. Desde allí también, por supuesto, es posible descargarlo.

Veamos a continuación algunas capturas de pantalla del programa para darnos una idea acerca de cómo funciona y cómo se configura.

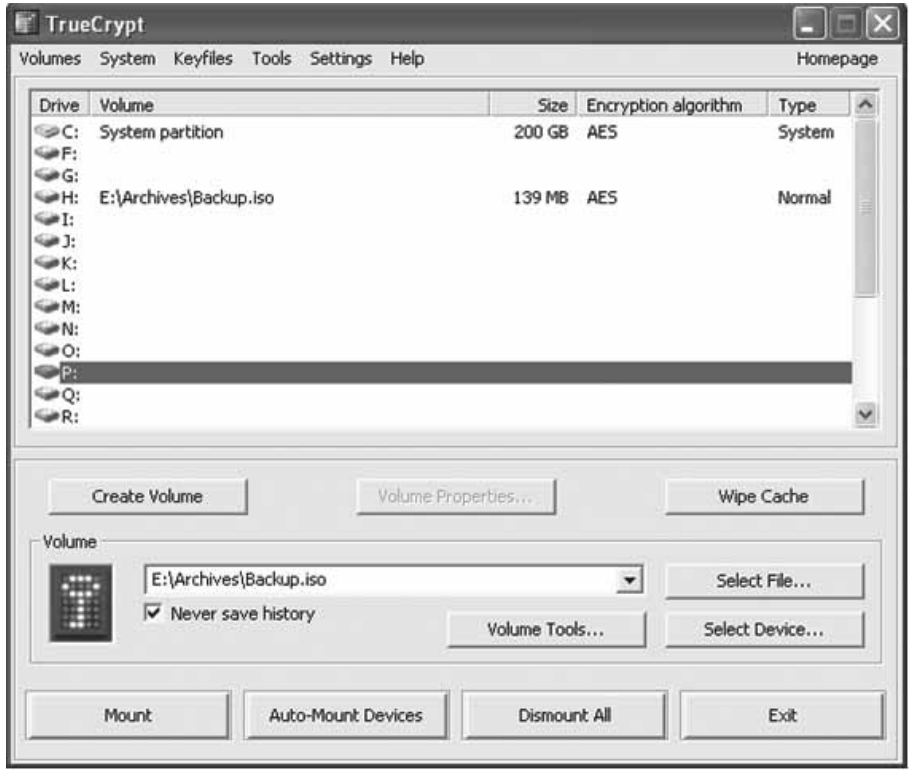

Fig. 7-8. Ventana principal de TrueCrypt: Listado de volúmenes.

Fig. 7-9. Selección de un algoritmo criptográfico para la creación de un volumen.

AxCrypt.

De mano de la empresa Axantum Software AB, desarrolladora del programa para la encriptación de archivos llamado AxCrypt, disponemos de una útil herramienta para la encriptación de archivos integrada con el sistema operativo Windows.

El programa permite hacer clic derecho sobre un archivo y debido a su integración con el explorador de Windows, AxCrypt posibilita de forma muy sencilla encriptar (y desencriptar) archivos individuales. Luego, también, puede hacerse doble clic sobre archivos cifrados. Se hace casi tan fácil trabajar con archivos cifrados como con archivos de otro tipo. Siguiendo en la línea de mantener la operación lo menos complicada posible para el usuario, cuenta además con un proceso de instalación muy sencillo. AxCrypt se encuentra disponible en muchos idiomas, incluyendo el español.

El programa utiliza llaves de 128 bits de longitud, generadas a partir de una frase-clave o *passphrase* provista por el usuario. Para el cifrado de los archivos implementa el algoritmo AES en modo CBC. Para las verificaciones de integridad se basa en el algoritmo HMAC-SHA1-128.

AxCrypt es software libre. Puede redistribuirse y modificarse bajo los términos de la licencia GNU (*General Public Licence*).

Se puede obtener más información respecto del programa, y el programa mismo, en el sitio Web `http://www.axantum.com/AxCrypt/`.

Veamos en la siguiente captura una de las funcionalidades clave del programa, integrado al menú de contexto sobre archivos en sistemas Windows:

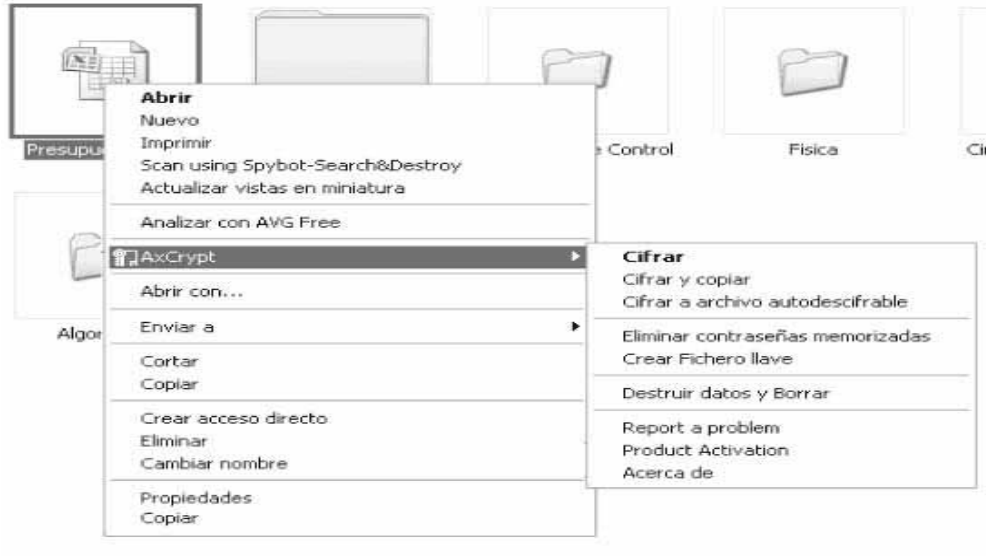

Fig. 7-10. Menú de contexto para encriptación o desencriptación de archivos.

STunnel.

El sitio Web de este programa se encuentra en `http://www.stunnel.org/`. Se trata de *Wrapper* (o enmascarador, nivel de indirección, capa intermedia, etc.), que permitirá encriptar conexiones TCP dentro de, o aplicando el protocolo, SSL. Se encuentra disponible tanto para Unix como para entornos Windows.

El programa permite convertir, o mejor dicho introducir en canales seguros las comunicaciones de protocolos que en principio trabajarían sobre un canal abierto –inseguro– como por ejemplo POP, IMAP, LDAP, etc. Stunnel es el programa que provee el túnel de encriptación sobre el que se transferirán los datos, sin necesidad de cambios en las aplicaciones o servicios existentes.

Se trata de una alternativa muy interesante para considerar a la hora de desarrollar nuestras aplicaciones y servicios, ya que en lugar de incorporar funcionalidades criptográficas para la encriptación de la información que se ha de transmitir, nos podríamos valer de esta herramienta externa a tal efecto.

Stunnel trabaja con la librería OpenSSL descripta anteriormente, aunque puede trabajar también con SSLeay; como se aclara desde el sitio Web, esto quiere decir que Stunnel soportará las mismas funcionalidades que la librería de base SSL soporte.

El código fuente de Stunnel se establece bajo la licencia GNU (*General Public Licence*), lo que implica que es libre para usos no comerciales y también comerciales, en tanto se provea el código fuente en las posteriores distribuciones.

OpenVPN.

Se comentará ahora brevemente acerca de una alternativa para la implementación de una VPN (*Virtual Private Network* o Red Privada Virtual). Esta metodología posibilita muchas veces establecer un canal seguro para nuestras aplicaciones distribuidas, para hacerlo a través de Internet.

En lugar de pensar en la incorporación de criptografía en nuestra aplicación para la comunicación segura, puede considerarse la opción de establecer dentro de una red privada virtual, o VPN, a las computadoras que ejecuten nuestra aplicación y así conseguir un medio seguro para la comunicación.

Desde el sitio Web del proyecto OpenVPN (`http://openvpn.net/`) nos enteramos que es una solución VPN SSL de código abierto, completamente equipada, que se adapta a una amplia gama de configuraciones, incluyendo:

- Acceso remoto.
- Seguridad Wi-Fi.
- Recupero por fallas.
- Controles de acceso específicos.
- VPN de sitio a sitio.
- Soluciones empresariales de acceso remoto con balanceo de carga.

El proyecto tiene como lema la premisa fundamental de que la complejidad es enemiga de la seguridad. OpenVPN ofrece una alternativa con una relación costo-eficiencia atractiva frente a otras tecnologías de VPN. Se comenta, además, que estaría orientado tanto a grandes empresas como a pequeñas y medianas.

Se agrega, además, que el diseño ligero o liviano de OpenVPN soluciona o simplifica muchas de las complejidades que caracterizan a otras implementaciones VPN. En OpenVPN el modelo de seguridad está basado en SSL, protocolo estándar de la industria para comunicaciones seguras a través de Internet. OpenVPN implementa la ampliación de redes seguras de las capas OSI 2/3 mediante la utilización del protocolo SSL / TLS; también soporta métodos de autenticación basados en certificados, tarjetas inteligentes (*smart-cards*) y permite políticas de control de acceso para el usuario o grupo específico, usando reglas de *firewall* que se aplican a la interfaz virtual de la VPN.

La compañía que lo desarrolla y mantiene, OpenVPN Technologies Inc., fue cofundada por Francis Dinha y James Yonan, autor del paquete de software OpenVPN. El modo de licenciamiento de OpenVPN queda comprendido bajo la licencia de GNU/GPL.

Si bien se trata de un servicio (*user-land* o en campo de usuario, es decir, no de kernel o sistema operativo), que funciona sin interfaz gráfica y se controla con programas por línea de comando, principalmente para la parte cliente, también existen aplicaciones gráficas a tal efecto. Veremos un ejemplo desarrollado por un tercero que muestra el estado de una conexión y el pedido de la contraseña.

Fig. 7-11. Página Web de OpenVPN.

8

Aspectos legales y estandarización

A continuación se retomará en parte lo abordado en el apartado de protocolos de firma digital del segundo capítulo. De un país a otro, las legislaciones en la materia –la criptografía: Su uso, importación, exportación, patentamiento, etc.– varían considerablemente. Por ejemplo, no muchos años atrás, Francia prohibió todas las formas de cifrado o encriptación (con la excepción de las firmas digitales desde 1990) ya que los políticos entendieron que los ciudadanos no debían tener acceso a las herramientas criptográficas utilizadas por los militares. EE.UU., por su parte, años atrás, mantuvo una política de control de exportación que no permitía, por ejemplo, la descarga de aplicaciones que hacían uso de criptografía "fuerte" desde otro país.

En este capítulo, entonces, se tratará lo referido a las patentes de algunos algoritmos a lo largo del tiempo y, también, a las reglas de importación y de exportación de criptografía, hoy con menos restricciones que antaño, pero aún con algunas limitaciones para tener en cuenta.

Como en el caso de las firmas digitales, siempre será conveniente el asesoramiento legal profesional. Corresponderá asegurarse, antes de la selección de un algoritmo criptográfico –o programa que lo utilice– para implementar en nuestros sistemas, de que no se estén infringiendo normas de derecho de uso y/o importación y exportación. Al descargar estos programas o librerías, *frameworks*, o en la misma documentación de los diferentes entornos de programación, por lo general, se encontrará un detalle a este respecto que nos permitirá confirmar que, según su finalidad, forma de distribución, país o países en donde funcionará, cuándo lo hará, etc., no estemos violando alguna ley o patente vigente.

Con respecto a los estándares, en primera instancia hemos de abordar el trabajo hecho por la ISO (*International Organization for Standardization*) y, luego, por las principales entidades de registro y estandarización de los EE.UU. con relación a cuestiones tecnológicas. Estas últimas son entonces la ANSI (*American National Standards Institute*) y la FIPS (*Federal Information Processing Standards*). Estas tres organizaciones, principalmente, son la referencia (en lo que a criptografía se refiere) en materia de estandarización.

Patentes.

Al hablar de patentes que podrían restringir la disponibilidad de implementaciones de algoritmos criptográficos actuales, empezaremos comentado un caso muy conocido respecto de la compañía RSA Data Security Inc.

Para conocer cómo eran las cosas algunos años atrás, referimos aquí parte de la documentación adjunta a la distribución de PGP versión 1.0, donde el autor del programa –Philip Zimmermann– aclara algo así como que el algoritmo está licenciado y que el uso del mismo, por parte del usuario (lo que en ciertos casos implicaría una falta), es responsabilidad del usuario y no del autor del programa:

> *"El criptosistema de clave pública RSA fue desarrollado en el MIT con fondos federales mediante subvenciones de la National Science Foundation [Fundación Nacional para la Ciencia] y la Armada. Está patentado por MIT (patente U.S. nº 4,405,829, concedida el 20 de septiembre de 1983). Una compañía llamada Public Key Partners (PKP) tiene licencia comercial exclusiva para vender y sub-licenciar el criptosistema de clave pública RSA. Para detalles de licencias sobre el algoritmo RSA, puede vd. contactar con Robert Fougner de PGP, en el 408/735-6779. El autor de esta implementación en software del algoritmo RSA provee esta implementación únicamente para uso educacional. Licenciar este algoritmo en PKP es responsabilidad de usted, el usuario, no Philip Zimmermann, autor de esta implementación de software. El autor no asume responsabilidades por ninguna violación de la ley de patentes que resultare del uso sin licencia, por parte del usuario, del algoritmo RSA usado en este software."*

En la actualidad, PGP es legal tanto dentro como fuera de los EE.UU.; se utilizará la versión pgp2.6.2 dentro de este país y la versión pgp2.6.2i fuera de él (nótese la letra "i", por internacional). Sin embargo, existen limitaciones; veamos ahora un fragmento de la documentación de la versión pgp2.6.2i:

> *"Ascom-Tech AG ha dado permiso para que la versión gratuita [freeware] de PGP use el cifrado IDEA en usos no comerciales, en todas partes. En EEUU y Canadá, todos los usuarios comerciales o gubernamentales deben obtener una versión, con licencia, de ViaCrypt, quienes tienen una licencia de Ascom-Tech para el cifrado IDEA."*

Esto quiere decir que, para el uso comercial fuera de los EE.UU. y Canadá, RSA podrá ser utilizado gratuitamente ya que no está patentado, pero se requiere una licencia de Ascom Systec para IDEA. La patente de RSA caducaba en el año 2003, pero fue entregado al dominio público en el año 2000.

Como vemos, en el caso del algoritmo IDEA o *International Data Encryption Algorithm* (Algoritmo Internacional de Cifrado de Datos), inventado por Xuejia Lai y James Massey en la Escuela Politécnica Federal de Zúrich, su patente, actualmente, es propiedad de Ascom-Tech.

IDEA fue diseñado en contrato con la Fundación Hasler, la cual se hizo parte de Ascom-Tech AG. IDEA es libre para uso no comercial, aunque fue patentado y sus patentes se vencerán en 2010 y 2011. El nombre IDEA es una marca registrada y está licenciado mundialmente por MediaCrypt.

El lector puede encontrar más información al respecto de PGP en el capítulo anterior, donde abordamos específicamente la implementación OpenPGP, libre de limitaciones.

Veremos algunas otras patentes registradas en los EE.UU. a manera de ejemplo –se notará cuáles están actualmente caducas–:

Nombre del algoritmo	Año de envío	Año de patente	Nro. de patente	Inventores	Asignación	Patente caduca
DES	1975	1976	3,962,539	Ehrsam et al.	IBM	SI
Diffie-Hellman	1977	1980	4,200,770	Hellman, Diffie, y Merkle	Universidad de Stanford	SI
RSA	1977	1983	4,405,829	Rivest, Shamir, y Adleman	MIT	SI
IDEA	1992	1993	5,214,703	Lai y Massey AG	Ascom Tech	NO
DSA	1991	1993	5,231,668	Kravitz	EE.UU.	NO

Tabla 8-1. Patentes registradas en E.E.U.U.

Reglas de importación y exportación.

Se tratará en esta sección los aspectos relativos a las reglas o legislaciones de los diferentes países que controlan la importación y la exportación de software que implementa algoritmos criptográficos.

Se recomienda especialmente la consulta de un sitio Web, mantenido por Bert-Jaap Koops, con el título de *crypto law survey* o encuesta de leyes criptográficas; su dirección es `http://rechten.uvt.nl/koops/cryptolaw/`. Se trata de una excelente referencia, de actualización periódica, que detalla qué está sucediendo en la actualidad en materia de legislación internacional –y nacional para el caso de gran cantidad de países– en cuestiones que involucran a la criptografía en general.

En esta sección veremos los aspectos principales del acuerdo más importante respecto de regulaciones de importación y exportación (general, no únicamente de criptografía) y cuáles son, a grandes rasgos, las legislaciones vigentes para los EE.UU. y para los países de Latinoamérica.

COCOM y Acuerdo de Wassenaar.

Establecido en el año 1996, en la ciudad holandesa de Wassenaar, este acuerdo es sucesor del Comité de Coordinación de Control Multilateral de Exportaciones (*Coordinating Committee for Multilateral Export Controls* o COCOM), vigente durante la Guerra Fría. Hoy, no está activo, aunque los países integrantes en ese entonces acordaron mantener el *status quo,* quedando los productos criptográficos comprendidos dentro de las listas de control de exportación.

El acuerdo de Wassenaar (del inglés *Wassenaar Arrangement*, siendo su nombre completo *The Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies,* es decir, acuerdo de Wassenaar sobre los controles de exportación para armas convencionales y de bienes o productos de doble uso y tecnologías), es un régimen multilateral de controles de exportación del que participan 40 países.

Se entiende por bienes de doble uso a aquellos que pueden ser utilizados tanto para propósitos militares como civiles. La criptografía, por ejemplo, es considerada un "bien" de este tipo.

La secretaría que se encarga de la administración del acuerdo de Wassenaar se encuentra en Viena, Austria.

De acuerdo con la información provista por la administración, el acuerdo se ha establecido con el fin de contribuir a la seguridad regional e internacional y a la estabilidad, mediante la promoción de transparencia y una mayor responsabilidad en transferencias de armas convencionales y bienes de doble uso y tecnologías, lo que impide la acumulación desestabilizadora. Los países participantes buscan, a través de sus políticas nacionales, el fin de asegurar que las transferencias de estos elementos no contribuyan al desarrollo o a la mejora de las capacidades militares que socaven estos objetivos y que tales capacidades no se desvíen en el apoyo.

Los países participantes en el acuerdo de Wassenaar, a la fecha, son: Argentina, Australia, Austria, Bélgica, Bulgaria, Canadá, Croacia, República Checa, Dinamarca, Estonia, Finlandia, Francia, Alemania, Grecia, Hungría, Irlanda, Italia, Japón, Letonia, Lituania, Luxemburgo, Malta, Países Bajos, Nueva Zelandia, Noruega, Polonia, Portugal, República de Corea, Rumania, Rusia, Eslovaquia, Eslovenia, Sudáfrica, España, Suecia, Suiza, Turquía, Ucrania, Reino Unido y Estados Unidos.

Es importante notar que se establece que cada país debe procurar el seguimiento de estos lineamientos acordados mediante su legislación local en particular.

En lo que a criptografía se refiere, se destaca que los productos de criptografía que implementen cifrado simétrico con llave de hasta 56 bits de longitud son libres de exportación. Para los de cifrado asimétrico, el límite es de 512 bits. En productos criptográficos de uso masivo, en cuanto al cifrado simétrico, el límite fue de 64 bits, hasta el año 2000, cuando se canceló esa limitación. Al margen, todo software criptográfico de dominio público es libre de exportación, aunque no se ha especificado nada acerca de exportación electrónica (a través de Internet, por ejemplo).

Es posible consultar más información acerca del acuerdo en el sitio Web mantenido para informar `http://www.wassenaar.org/`.

Exportación e importación en los EE.UU.

Con respecto a la importación de criptografía, a la fecha, en los EE.UU. no hay restricciones legisladas.

Al hablar de la exportación, el país ha firmado el acuerdo de Wassenaar descripto anteriormente, pero mantiene controles más estrictos. Un sector del Departamento de Comercio y del Departamento de Justicia se encargan de las regulaciones al respecto.

Publicar criptografía en Internet o sistemas BBS es considerado exportación, salvo que se realicen controles o se implementen medidas que aseguren la exclusión de accesos externos.

Las reglas para la exportación diferencian cinco categorías o implementaciones de criptografía, a saber: Software de uso masivo, productos que permitirían a una tercera parte –en este caso, al gobierno– recuperar la información cifrada, licencias de prueba temporarias, otros ítems de criptografía y tecnología de encriptación.

A partir de las nuevas regulaciones del año 2000, se estableció principalmente que:

1. Criptografía sin límites de longitudes de llave podía ser exportada bajo una licencia de excepción, luego de una revisión técnica, a cualquier usuario final (no gobiernos) y a cualquier país, salvo los considerados países terroristas.
2. Código fuente de criptografía no-restringida, incluyendo al software *open-source*, puede ser exportado a cualquier usuario final sin necesidad de revisión técnica.
3. Las regulaciones implementan los cambios al acuerdo de Wassenaar del año 1998.

Puede consultarse el sitio Web del Acuerdo de Wassenaar para obtener la lista de países considerados terroristas por los EE.UU. en la actualidad.

Para tener en cuenta que el gobierno de los EE.UU. toma en serio estos controles, podrían citarse dos casos popularmente conocidos. En el año 2001, dos personas fueron arrestadas y acusadas de exportación de dispositivos criptográficos a China. Al año siguiente, el Departamento de Comercio sancionó con una multa de 95 000 dólares a una empresa de San Diego por exportar ilegalmente software de encriptación (128 bits) a Corea del Sur.

Exportación e importación en países de Latinoamérica.

En la siguiente tabla veremos, de aquellos países latinoamericanos para los cuales existe información al respecto, las correspondientes normas de importación y exportación, como las regulaciones locales en materia de criptografía.

	Regulaciones para la exportación	Regulaciones para la importación	Regulaciones locales
Argentina	Firmó el acuerdo Wassenaar, por lo tanto aplican tales controles de exportación.	Sin regulaciones.	Sin regulaciones.
Brasil	Sin regulaciones referentes a la exportación pero se está trabajando en la materia.	Sin regulaciones referentes a la importación pero se está trabajando en la materia.	Sin regulaciones.
Chile	-	Sin regulaciones.	Sin regulaciones.
Colombia	-	Sin regulaciones.	Sin regulaciones.
Costa Rica	-	-	Sin regulaciones.
México	Sin regulaciones.	Sin regulaciones.	Sin regulaciones.
Perú	Sin regulaciones.	Sin regulaciones.	Sin regulaciones.
Puerto Rico	-	Sin regulaciones.	Sin regulaciones.
Uruguay	-	Sin regulaciones.	Sin regulaciones.
Venezuela	-	-	Sin regulaciones.

Tabla 8-2. Normas de importación, exportación y regulaciones locales para algunos países latinoamericanos.

Organismos de estandarización.

ISO.

Las famosas siglas que identifican a esta organización corresponden a *International Organization for Standardization* (Organización Internacional para la Estandarización). Está constituida por los institutos u organizaciones –en relación con los estándares propios nacionales– de los diferentes países adheridos. La organización produce estándares industriales y comerciales a nivel mundial.

Gran parte de la producción de ISO, particularmente en la materia que nos ocupa, se realizó en conjunto con la IEC –*International Electrotechnical Comission* o Comisión Electrotécnica Internacional–, responsable de la estandarización de equipo eléctrico.

De los primeros estándares publicados por el organismo, muchos ya han sido retirados, es decir, han caducado y ya no son válidos o son continuados y/o ampliados en otros estándares, con otros números de referencia. Entre estos estándares ya no vigentes encontramos por ejemplo, los siguientes:

- ISO 8372: Estándar retirado en el año 2004; especificaba los cuatro modos de operación de un algoritmo de cifrado simétrico por bloques de 64 bits de longitud. Definía lo referente al uso de variables de inicialización y de parámetros.
- ISO/IEC 9979: De los primeros pasos en materia de estandarización de criptografía por la ISO, esta norma trató –en la actualidad ya ha caducado– del registro de algoritmos. Cada organismo nacional podía enviar un algoritmo para que fuese registrado, especificando sus procesos –en los casos no privados o secretos–, aplicaciones, parámetros, modos, etc. Esto no implicaba cosa alguna salvo el registro del algoritmo: No era dado ningún respaldo por parte de ISO respecto de su seguridad o calidad. Los algoritmos registrados incluyeron, por ejemplo, a los algoritmos DES e IDEA. La norma fue retirada en febrero de 2006, principalmente por el hecho de que el registro de algoritmos era redundante al existir ya la norma ISO/IEC 18033.

Se detallarán a continuación los estándares relacionados con protocolos y algoritmos criptográficos más importantes, actualmente en vigencia:

- ISO/IEC 9796: Estándar compuesto por tres partes (dos de ellas actualmente disponibles) que trata o especifica cuestiones relativas a la generación de llaves criptográficas en el uso de cifrados asimétricos.
  - ISO/IEC 9796-2: Segunda parte del estándar que especifica mecanismos basados en el factoreo de números enteros. Describe tres esquemas de firma digital, particularmente, lo relacionado con la generación de llaves para cada uno de estos tres esquemas.

  - ISO/IEC 9796-3: Tercera parte del estándar 9796 que detalla los mecanismos basados en el problema del logaritmo discreto de un campo finito o curva elíptica.

- ISO/IEC 9797: Estándar compuesto por dos partes (dos números de referencia), referido a la generación de códigos de autenticación de mensajes o *message autentication codes* (MAC).
  - ISO/IEC 9797-1: Esta parte define lo referente a la generación de códigos de autenticación de mensajes, mediante algoritmos de cifrado simétrico por bloques.
  - ISO/IEC 9797-2: Esta parte define lo referente a la generación de códigos de autenticación de mensajes, utilizando funciones de *hashing* criptográfico.

- ISO/IEC 9798: Estándar de seis partes referente a la autenticación de entidades.
  - ISO/IEC 9798-1: Primera parte del estándar que trata o especifica generalidades y terminología utilizada.
  - ISO/IEC 9798-2: Segunda parte del estándar que describe mecanismos de autenticación basados en algoritmos de cifrado simétrico.
  - ISO/IEC 9798-3: Tercera parte del estándar que detalla mecanismos de autenticación basados en la firma digital.
  - ISO/IEC 9798-4: Cuarta parte del estándar que describe mecanismos de autenticación basados en funciones de verificación criptográficas.
  - ISO/IEC 9798-5: Quinta parte del estándar que describe mecanismos de autenticación basados en técnicas de cero-conocimiento.
  - ISO/IEC 9798-6: Sexta y última parte del estándar que describe mecanismos de autenticación basados en técnicas de transferencia manual de datos.

- ISO/IEC 10116: Estándar que especifica los modos de operación de un algoritmo de cifrado simétrico por bloques, de cualquier cantidad de bits de longitud. Se especifican cinco modos de operación: ECB (*Electronic Codebook*), CBC (*Cipher Block Chaining*), CFB (*Cipher Feedback*), OFB (*Output Feedback*) y CTR (*Counter*).

- ISO/IEC 11770: En este estándar se especifican cuestiones relativas al manejo de llaves criptográficas. Su categorización por ISO lo establece con el nombre de *Key management* (manejo de llaves), dentro de *Security techniques* (técnicas de seguridad), a su vez dentro de *Information technology* (tecnología de la información). Este estándar también se divide en cuatro partes:
  - ISO/IEC 11770-1:1996: Parte 1, *Framework*: Define un modelo general de manejo de llaves independientemente del algoritmo utilizado. Trata respecto del objetivo del manejo de llaves y conceptos básicos.

  - ISO/IEC 11770-2:2008: Parte 2: Mecanismos utilizando técnicas simétricas. Se especifican aquí trece mecanismos para la implementación de llaves secretas compartidas utilizando criptografía simétrica.
  - ISO/IEC 11770-3:2008: Parte 3: Mecanismos utilizando técnicas asimétricas. Se especifican aquí mecanismos para el manejo de llaves utilizando criptografía simétrica (acuerdo y transporte de llaves secretas, disponibilidad de llaves públicas).
  - ISO/IEC 11770-4:2006: Parte 4: Mecanismos basados en secretos débiles (*weak secrets*). Se especifican aquí mecanismos para la implementación de llaves que pueden ser memorizadas por una persona (esto es lo que se entiende aquí por *weak secrets*), es decir, con un rango menor de posibilidades.

- ISO/IEC 13888: Estándar dedicado a describir las técnicas criptográficas de no-repudio. Está compuesto por tres partes:
  - ISO/IEC 13888-1: Primera parte que incluye una introducción y la descripción de generalidades al respecto de las técnicas de no-repudio que serán abordadas en las partes 2 y 3.
  - ISO/IEC 13888-2: Segunda parte del estándar que describe mecanismos que utilizan técnicas basadas en algoritmos simétricos.
  - ISO/IEC 13888-3: Tercera y última parte del estándar que describe mecanismos que utilizan técnicas basadas en algoritmos asimétricos.

- ISO/IEC 14888: Estándar compuesto por un documento principal y tres apéndices, dedicado a los mecanismos de firma digital.
  - ISO/IEC 14888-1: Primera parte del estándar que describe principios generales y requerimientos para este tipo de firmas digitales.
  - ISO/IEC 14888-2: Segunda parte del estándar que trata acerca de mecanismos para la firmas digitales por apéndices basados en el factoreo de números.
  - ISO/IEC 14888-3: Tercera y última parte del estándar que describe mecanismos basados en el problema del logaritmo discreto en un campo finito.

- ISO/IEC 18033: Este estándar especifica sistemas de encriptación para la confidencialidad de información. Su categorización por ISO lo establece con el nombre de *Encryption algorithms* (algoritmos de encriptación), dentro de *Security techniques* (técnicas de seguridad), a su vez dentro de *Information technology* (tecnología de la información). El estándar se divide en cuatro partes:
  - ISO/IEC 18033-1:2005: Parte 1: General. Aquí se especifican lo términos utilizados a lo largo del estándar, el propósito de la encriptación de información, las diferencias entre algoritmos de cifrado simétrico y asimétrico,

asuntos relativos al manejo de llaves, los usos y propiedades de criptografía y el criterio para la inclusión de algoritmos dentro del estándar.

- ISO/IEC 18033-2:2006: Parte 2: Cifrado asimétrico. Este estándar especifica la interfaz funcional de un esquema de llave pública o cifrado asimétrico y, adicionalmente una serie de esquemas en particular, aparentemente seguros respecto de ataques de texto-cifrado conocido o elegido (*chosen ciphertext attack*).
- ISO/IEC 18033-3:2005: Parte 3: Cifrado en bloque. Aquí se detalla lo referente a este tipo de cifrado. Se especifican los siguientes algoritmos: De 64 bits de longitud de bloque, TDEA, MISTY1 y CAST-128; de 128 bits de longitud, AES, Camellia y SEED.
- ISO/IEC 18033-4:2005: Parte 4: Cifrado en flujo. Este estándar especifica algoritmos de cifrado en flujo. Se especifican las siguientes alternativas para generar un *keystream* o flujo de llave: OFB, CTR y CFB (basados en cifradores por bloques) y MUGI y SNOW 2.0 (generadores dedicados). También se especifican los siguientes modos: Funciones de salida MULTI-S01 y *binary-additive* (adición binaria).

ANSI.

Las siglas corresponden a *American National Standards Institute* o Instituto Nacional de Estándares Americano. Se trata de una organización o asociación privada y sin fines de lucro de los E.E.U.U. Produce estándares industriales y es un miembro de la ISO y del IEC. La organización desarrolla estándares a través de diferentes comités o *Accredited Standards Committees* (ASCs).

Los comités relacionados con estandarizaciones vinculadas a la criptografía están organizados dentro de diferentes áreas, principalmente dentro del área financiera. Estos son: El ASC X3 –*Information Processing Systems* o Sistemas de Procesamiento de Información–, el ASC X9 –*Financial Services* o Servicios Financieros– y el ASC X12 –*Electronic Business Data Interchange* o Intercambio de Datos de Comercio Electrónico–.

Las publicaciones o estándares, a su vez, se subdividen en otros números, que representan al sub-comité que se encargó de emitirlo. Vemos, a este respecto, algunos ejemplos como:

- ANSI X3.92: Estándar que especifica al algoritmo DES, que aquí es referenciado como *Data Encryption Algorithm* (DEA).
- ANSI X3.106: Este estándar especifica los modos del algoritmo DES (o DEA en la terminología de ANSI).
- ANSI X9.9: Es un estándar para la banca mayorista nacional para la autenticación de las transacciones financieras. Aborda dos cuestiones: El formato de

mensajes y el algoritmo de autenticación de mensajes. El algoritmo definido por el estándar es el llamado DES-MAC, basado en el algoritmo DES.

- ANSI X9.19: Similar o de aplicaciones similares al anterior, más detallado, pero apuntando a la banca minorista.
- ANSI X9.17: Es el estándar referido a la administración de llaves. En él se definen los protocolos que han de ser utilizados por las instituciones financieras, tales como bancos, para la transferencia de llaves criptográficas. El protocolo se basa en técnicas de distribución de llaves secretas, utilizando cifrado simétrico.
- ANSI X9.30: Estándar de la industria financiera para la firma digital basada en su *Digital Signature Algorithm* (DSA) o Algoritmo de Firma Digital. Requiere el algoritmo de *hashing* SHA-1.
- ANSI X9.31: Contraparte de la norma para la firma digital, pero basada en el algoritmo RSA. El estándar requiere el algoritmo de *hashing* MDC-2.
- ANSI X9.42: Proyecto de norma para el acuerdo de llaves sobre la base del algoritmo Diffie-Hellman. El estándar está destinado a especificar las técnicas para obtener una llave secreta compartida.
- ANSI X9.44: Proyecto de norma para el transporte de llaves basado en el algoritmo RSA. Se especificarán las técnicas para el transporte de una clave secreta con el algoritmo RSA.

FIPS.

Entre los FIPS –siglas de *Federal Information Processing Standards* o Estándares de Proceso de Información Federales– de la NIST (*National Institute of Standards and Technology* o Instituto Nacional de Estándares y Tecnología) encontramos las publicaciones de estándares criptográficos más populares. Son estándares de uso público, elaborados para ser utilizados por entidades gubernamentales no militares de los EE.UU. y sus contratistas.

NIST fue quien estandarizó, en primera instancia, al algoritmo DES como el estándar de encriptación de los EE.UU. y quien también lo sustituyó luego por el algoritmo AES.

Veamos algunas de estas publicaciones detalladas brevemente:

- FIPS PUB 31: Guía para la seguridad física del proceso automático de datos y control o manejo del riesgo. Año 1974.
- FIPS PUB 73: Guía para la seguridad de aplicaciones de computadoras. Año 1980.
- FIPS PUB 74: Guía para la implementación y uso del estándar de encriptación NBS. Año 1981.

- FIPS PUB 81: Modos de operación del algoritmo DES. Año 1980.
- FIPS PUB 102: Guía para la certificación y acreditación de la seguridad informática. Año 1983.
- FIPS PUB 112: Utilización de palabras clave. Año 1985.
- FIPS PUB 113: Autenticación de datos informáticos. Año 1985.
- FIPS PUB 140-2: Requerimientos de seguridad para módulos criptográficos. Año 2001.
- FIPS PUB 171: Manejo de llaves utilizando el estándar ANSI X9.17. Año 1992.
- FIPS PUB 180-2: Algoritmos de *hashing* de la familia SHA –llamados aquí *Secure Hash Standard* (SHS)–. Año 2002.
- FIPS PUB 181: *Automated Password Generator* (APG) o generación automática de claves. Año 1993.
- FIPS PUB 185: *Escrowed Encryption Standard* (EES) o Estándar de Encriptación Garantido. Año 1994.
- FIPS PUB 186-2: *Digital Signature Standard* (DSS). Año 2000.
- FIPS PUB 190: Guía para el uso de alternativas avanzadas de tecnologías de autenticación. Año 1994.
- FIPS PUB 191: Guía para el análisis de seguridad de redes de área local. Año 1994.
- FIPS PUB 196: Autenticación de entidades utilizando criptografía de llave pública. Año 1997.
- FIPS PUB 197: Algoritmo AES. Año 2001.
- FIPS PUB 198: Algoritmo HMAC o *Keyed-Hash Message Authentication Code*. Año 2002.

Apéndice
Tablas de referencia

Implementaciones criptográficas en Java.

Se incluyen en la siguiente tabla a los algoritmos –nombres en código para el método `getInstance()` las clases `MessageDigest`, `Cipher`, etc.– soportados según versión de la JDK (*Java Development Kit* o kit de desarrollo) o JVM (*Java Virtual Machine* o máquina virtual), considerando las implementaciones provistas, es decir, los *providers* o proveedores incluidos y registrados en las distribuciones de Sun Microsystems.

	JDK 1.3	JDK 1.4/ J2SE 1.4.2	J2SE 5.0	Java SE 6
Proveedor "SUN" *MessageDigest*: MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512 *Signature*: NONEwithDSA, SHA1withDSA *KeyFactory*: DSA *KeyPairGenerator*: DSA	-	X	X	X
Proveedor "SunJCE" *Cipher*: AES, Blowfish, DES, DESede, RC2, PBEWithMD5AndDES, PBEWithMD5AndTripleDES, PBEWithSHA1AndDESede, PBEWithSHA1AndRC2_40, RSA *KeyAgreement*: DiffieHellman *KeyFactory*: DiffieHellman *KeyGenerator*: AES, ARCFOUR, Blowfish, DES, DESede, HmacMD5, HmacSHA1, HmacSHA256, HmacSHA384, HmacSHA512, RC2 *KeyPairGenerator*: DiffieHellman *Mac*: HmacMD5, HmacSHA1, HmacSHA256, HmacSHA384, HmacSHA512	-	X	X	X

	JDK 1.3	JDK 1.4/ J2SE 1.4.2	J2SE 5.0	Java SE 6
Proveedor "SunJSSE" *KeyFactory*: RSA *KeyPairGenerator*: RSA *Signature*:MD2withRSA, MD5withRSA, SHA1withRSA	-	X	X	X
Proveedor "SunRsaSign" *KeyFactory*: RSA *KeyPairGenerator*: RSA *Signature*: MD2withRSA, MD5withRSA, SHA1withRSA, SHA256withRSA, SHA384withRSA, SHA512withRSA	-	-	X	X

Tabla A-1. Funcionalidades de criptografía disponibles en Java. Referencia: "-" no disponible; "X" disponible.

Implementaciones criptográficas en .NET.

Se detallan en esta tabla las funcionalidades criptográficas provistas o implementadas en el *namespace* `System.Security.Cryptography`, según diferentes versiones del *Framework* .NET:

	1.0	1.1	2.0	3.0	3.5
MD5CryptoServiceProvider (Algoritmo MD5)	X	X	X	X	X
SHA1CryptoServiceProvider (Algoritmo SHA-1)	X	X	X	X	X
SHA256CryptoServiceProvider (Algoritmo SHA-256 de familia SHA-2)	-	-	-	-	X
SHA384CryptoServiceProvider (Algoritmo SHA-384 de familia SHA-2)	-	-	-	-	X
SHA512CryptoServiceProvider (Algoritmo SHA-512 de familia SHA-2)	-	-	-	-	X
DESCryptoServiceProvider (Algoritmo DES)	X	X	X	X	X
TripleDESCryptoServiceProvider (Algoritmo TripleDES)	X	X	X	X	X

	1.0	1.1	2.0	3.0	3.5
AesCryptoServiceProvider (Algoritmo AES)	-	-	-	-	X
RC2CryptoServiceProvider (Algoritmo RC2)	X	X	X	X	X
RSACryptoServiceProvider (Algoritmo RSA)	X	X	X	X	X
DSACryptoServiceProvider (Algoritmo DSA)	X	X	X	X	X

Tabla A-2. Funcionalidades de criptografía disponibles en .NET. Referencia: "-" no disponible; "X" disponible.

Implementaciones criptográficas en PHP.

En la siguiente tabla se detallan las funciones criptográficas incorporadas al lenguaje, según versiones del intérprete:

	PHP 4	PHP 5
md5() (Algoritmo MD5)	X	X
md5_file() (Algoritmo MD5)	X (4.2+)	X
sha() (Algoritmo SHA-1)	X (4.3+)	X
sha_file() (Algoritmo SHA-1)	X (4.3+)	X
crypt() (Algoritmos de *hashing* dependientes del S.O.)	X	X

Tabla A-3. Funcionalidades de criptografía en PHP. Referencia: "-" no disponible; "X" disponible.

Esta tabla muestra las librerías adicionales que implementan diferentes algoritmos criptográficos según versiones de PHP:

	PHP 4	PHP 5
Mcrypt (Algoritmos 3-WAY, DES, TripleDES, Blowfish, Rijndael, Safer, Twofish, IDEA, RC2, RC4, RC6, Serpent y GOST)	X (4.0.2+)	X
MHash (Algoritmos SHA1, SHA160, SHA192, SHA224, SHA384, SHA512, HAVAL128, HAVAL160, HAVAL192, HAVAL224, HAVAL256, RIPEMD128, RIPEMD256, RIPEMD320, MD4, MD5, TIGER, TIGER128, TIGER160, ALDER32, CRC32, CRC32b, WHIRLPOOL, GOST, SNEFRU128 y SNEFRU256)	X	X
Crypt_Blowfish (Algoritmo Blowfish)	X (4.2+)	X
Crypt_RSA (Algoritmo RSA)	X	X
Crypt_HMAC (Algoritmo HMAC)	X	X
Crypt_DiffieHellman (Algoritmo Diffie-Hellman)	-	X

Tabla A-4. Funcionalidades de criptografía disponibles en librerías adicionales PHP. Referencia: "-" no disponible; "X" disponible.

Implementaciones criptográficas en bases de datos.

Microsoft SQL Server.

La siguiente tabla muestra las funcionalidades criptográficas implementadas en diferentes versiones del servidor SQL Server de Microsoft:

	2000	2005	2008
HashBytes() (Algortimos MD2, MD4, MD5 y SHA-1)	-	X	X
EncryptByPassPhrase(), **DecryptByPassPhrase()** (Algoritmos de cifrado simétrico disponibles de acuerdo con versión de Windows, por ejemplo DES)	-	X	X
EncryptByKey(), **DecryptByKey()** (Algoritmos de cifrado simétrico disponibles de acuerdo con versión de Windows, por ejemplo DES)	-	X	X
EncryptByAsymKey(), **DecryptByAsymKey()** (Algoritmos de cifrado asimétrico disponibles de acuerdo con versión de Windows, por ejemplo RSA)	-	X	X

Tabla A-5. Funcionalidades de criptografía disponibles en MS-SQL Server. Referencia: "-" no disponible; "X" disponible.

Oracle.

Se detallan en esta tabla los *packages* y funciones de criptografía disponibles en diferentes versiones del servidor de base de datos Oracle:

	8i	9i	10g	11g
DBMS_OBFUSCATION_TOOLKIT.MD5() (Algoritmo MD5)	X	X	X	X
DBMS_OBFUSCATION_TOOLKIT.DESEncrypt(), **DBMS_OBFUSCATION_TOOLKIT.DESDecrypt()** (Algoritmo DES)	X	X	X	X
DBMS_OBFUSCATION_TOOLKIT.DES3Encrypt(), **DBMS_OBFUSCATION_TOOLKIT.DES3Decrypt()** (Algoritmo TripleDES)	-	X	X	X
DBMS_CRYPTO.Hash() (Algoritmos MD4, MD5, SHA-1)	-	-	X	X
DBMS_CRYPTO.ENCRYPT(), **DBMS_CRYPTO.DECRYPT()** (Algoritmos DES, TripleDES, AES y RC4)	-	-	X	X

Tabla A-6. Funcionalidades de criptografía disponibles en Oracle. Referencia: "-" no disponible; "X" disponible.

MySQL.

Se detallarán en la siguiente tabla las funciones criptográficas implementadas según versión del servidor de base de datos MySQL:

	3.23	4.0/4.1	5.0/5.1	6.0/6.0.5
MD5() (Algoritmo MD5)	X	X	X	X
SHA1() (Algoritmo SHA-1)	-	X	X	X
SHA2() (Algoritmos SHA-224, SHA-256, SHA-384, SHA-512)	-	-	-	X
DES_ENCRYPT(), **DES_DECRIPT()** (Algoritmo DES)	-	X	X	X
AES_ENCRYPT(), **AES_DECRYPT()** (Algoritmos TripleDES y AES)	-	X	X	X

Tabla A-7. Funcionalidades de criptografía disponibles en MySQL. Referencia: "-" no disponible; "X" disponible.

Bibliografía y Referencias

Referencias bibliográficas.

Hook, David. *Beginning Cryptography with Java*. Wrox, EE.UU., 2005.

Kahn, David. *The Codebreakers - The Story of Secret Writing*. [Abridged] ed. Weidenfeld and Nicolson, Inglaterra, 1974.

Knudsen, Jonathan. *Java Cryptography*. O'Reilly, EE.UU., 1998.

Mcclure, Stuart; Kurtz, George; Scambray, Joel. *Hacking Exposed: Network Security Secrets & Solutions*. 2da ed. McGraw-Hill, EE.UU., 2000.

Menezes, Alfred J.; Van Oorschot, Paul C.; Vanstone, Scott A. *Handbook of Applied Cryptography*. 5ta ed. CRC Press, EE.UU., 2001.

Pfleeger, Charles P.; Pfleeger, Shari Lawrence. *Security in Computing*. 4ta ed. Prentice Hall, EE.UU., 2006.

Schneier, Bruce. *Applied Cryptography*. 2da ed. John Wiley & Sons, EE.UU., 1996.

Shiflett, Chris. *Essential PHP Security*. O'Reilly, EE.UU., 2005.

Stallings, William. *Cryptography and Network Security Principles and Practices*. 4ta ed. Prentice Hall, EE.UU., 2005.

Stinson, Douglas R. *Cryptography, theory and practice*. 2da ed. CRC Press, EE.UU., 2002.

Tena Ayuso, Juan. *Protocolos Criptográficos y seguridad en redes*. Servicio de publicaciones Universidad de Cantabria, España, 2003.

Referencias a recursos electrónicos.

Libros, manuales, publicaciones, artículos y papers.

Anghel, Octavia Andrea. *A Guide to Cryptography in PHP* [en línea]. DevX.com. Mayo de 2008.
<http://www.devx.com/webdev/Article/37821>
[Consulta: Junio 2008].

Bromberg, Peter A. *RSA Encryption in .NET — Demystified!* [en línea]. EggHeadCafe.com.
<http://www.eggheadcafe.com/articles/20020630.asp>
[Consulta: Junio 2008].

Curphey, Mark; Endler, David; Hau, William; Taylor, Steve; Smith, Tim; Russell, Alex; Mckenna, Gene; Parke, Richard; Mclaughlin, Kevin. *A Guide to Building Secure Web Applications* [en línea]. The Open Web Application Security Project. Ver. 1.1 Final. 22 sep. 2002.
<http://www.cgisecurity.com/owasp/html/index.html>
[Consulta: Junio 2008].

Esposito, Dino. *Cryptography the .NET Way* [en línea]. DevX.com. Agosto de 2003.
<http://www.devx.com/codemag/Article/16747>
[Consulta: Junio 2008].

Gilmore, W. J. *PHP's Encryption Functionality* [en línea]. O'Reilly OnLamp.com PHP DEVCENTER. Julio de 2001.
<http://www.onlamp.com/pub/a/php/2001/07/26/encrypt.html>
[Consulta: Junio 2008].

Hudson, Paul. *Advanced symmetric encryption* [en línea]. Practical PHP Programming.
<http://www.hudzilla.org/phpbook/read.php/17_3_5>
[Consulta: Junio 2008].

Kimmel, Paul. *Encrypting Data in .NET* [en línea]. Developer.com.
<http://www.developer.com/net/net/article.php/3077901>
[Consulta: Junio 2008].

Microsoft. *Create symmetric key (Transact-SQL)* [en línea]. SQL Server 2008 Books Online, SQL Server Developer Center.
<http://msdn.microsoft.com/en-us/library/ms188357.aspx>
[Consulta: Junio 2008].

Microsoft. *Cryptographic Functions (Transact-SQL)* [en línea]. SQL Server 2008 Books Online, SQL Server Developer Center.
<http://msdn.microsoft.com/en-us/library/ms173744.aspx>
[Consulta: Junio 2008].

Microsoft. *Encryption Hierarchy* [en línea]. SQL Server 2008 Books Online, SQL Server Developer Center.
<http://msdn.microsoft.com/en-us/library/ms189586.aspx>
[Consulta: Junio 2008].

Microsoft *.NET Framework Cryptography Model* [en línea]. MSDN - .NET Framework Developer's Guide.
< http://msdn.microsoft.com/en-us/library/aa720325(VS.71).aspx >
[Consulta: Junio 2008].

Microsoft. *System.Security.Cryptography Namespace* [en línea]. MSDN - .NET Framework Class Library.
<http://msdn.microsoft.com/en-us/library/system.security.cryptography.aspx >
[Consulta: Junio 2008].

MySQL AB. *MySQL Reference Manual* [en línea]. MySQL Documentation.
<http://dev.mysql.com/doc/>
[Consulta: Agosto 2008].

Nanda, Arup. *Protect from Prying Eyes: Encryption in Oracle 10g* [en línea]. dbazine.com.
<http://www.dbazine.com/olc/olc-articles/nanda11>
[Consulta: Agosto 2008].

Oracle Corporation. *DBMS_CRYPTO* [en línea]. Oracle® 10g Database PL/SQL Packages and Types Reference.
<http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_crypto.htm>
[Consulta: Agosto 2008].

Oracle Corporation. *DBMS_OBFUSCATION_TOOLKIT* [en línea]. Oracle8i Supplied PL/SQL Packages Reference.
<http://download-west.oracle.com/docs/cd/A87860_01/doc/appdev.817/a76936/dbms_obf.htm>
[Consulta: Agosto 2008].

Oracle Corporation. *La Historia de Oracle: Innovación, Liderazgo y Resultados* [en línea]. Oracle Latinoamérica.
<http://www.oracle.com/global/lad/corporate/story.html>
[Consulta: Agosto 2008].

Oracle Corporation. *Oracle9i Architecture on Windows* [en línea]. Oracle9i Database Online Documentation.
<http://download-uk.oracle.com/docs/cd/A91202_01/901_doc/win.901/a90163/windows.htm>
[Consulta: Agosto 2008].

Peterson, Erich; LI, Siqing. *An Overview of Cryptographic Systems and Encrypting Database Data* [en línea]. 4 Guys From Rolla. Febrero de 2007.
<http://aspnet.4guysfromrolla.com/articles/021407-1.aspx>
[Consulta: Junio 2008].

PHP Group. *Cryptography Extensions* [en línea]. PHP Manual. Reference.
<http://www.php.net/manual/es/refs.crypto.php>
[Consulta: Junio 2008].

Policht, Marcin. *SQL Server 2005 Security - Part 3 Encryption* [en línea]. Database Journal. Febrero de 2005.
<http://www.databasejournal.com/features/mssql/article.php/3483931>
[Consulta: Junio 2008].

Ramió Aguirre, Jorge. *Libro Electrónico de Seguridad Informática y Criptografía* [en línea]. Versión 4.1. Escuela Universitaria de Informática de la Universidad Politécnica de Madrid, España. Sexta edición de 1 de Marzo de 2006.
<http://www.criptored.upm.es/guiateoria/gt_m001a.htm>
[Consulta: Junio 2008].

Sun Microsystems. *Java Cryptography Architecture - API Specification & Reference*. Última modificación del 4 de agosto de 2002.
<http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html>
[Consulta: Junio 2008].

Sun Microsystems. *Java Cryptography Architecture (JCA) Reference Guide for Java Platform Standard Edition 6*.
<http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>
[Consulta: Junio 2008].

Sun Microsystems. *JavaTM Cryptography Extension (JCE) - Reference Guide - for the JavaTM 2 Platform Standard Edition Development Kit (JDK) 5.0*.
<http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html>
[Consulta: Junio 2008].

Talens-Oliag, Sergio. *Seguridad en JAVA* [en línea]. Instituto Tecnológico de Informática (ITI). Diciembre de 1999.
<http://www.uv.es/sto/cursos/seguridad.java/html/>
[Consulta: Junio 2008].

Librerías, *frameworks* y herramientas de desarrollo.

Bouncy Castle Libraries - The Legion of the Bouncy Castle -
http://www.bouncycastle.org/

Crypt_Blowfish - PEAR :: Package :: Crypt_Blowfish -
http://pear.php.net/package/Crypt_Blowfish

Crypt_DiffieHellman - PEAR :: Package :: Crypt_DiffieHellman -
http://pear.php.net/package/Crypt_DiffieHellman

Crypt_HMAC - PEAR :: Package :: Crypt_HMAC -
http://pear.php.net/package/Crypt_HMAC

Crypt_RSA - PEAR :: Package :: Crypt_RSA -
http://pear.php.net/package/Crypt_RSA

Cryptix Library - The Cryptix Project -
http://www.cryptix.org/

MCrypt Library -
http://mcrypt.sourceforge.net/

MHash Library -
http://mhash.sourceforge.net/

Aplicaciones y herramientas – software.

AxCrypt - Axantum Software AB - File Encryption Software -
http://www.axantum.com/AxCrypt/

Kerberos - MIT - The Network Authentication Protocol -
http://web.mit.edu/kerberos/www/

OpenVPN -
http://www.openvpn.net/

PGP - GPG/GnuGP - The GNU Privacy Guard -
http://www.gnupg.org/

PGP - OpenPGP - The OpenPGP Alliance Home Page -
http://www.openpgp.org/

PGP - PGP Corporation -
http://www.pgp.com/

PGP - The International PGP Home Page -
http://www.pgpi.org/

SSH - OpenSSH Project -
http://www.openssh.org/

SSL - OpenSSL - The Open Source toolkit for SSL/TLS -
http://www.openssl.org/

Stunnel -
http://www.stunnel.org/

TrueCrypt - Free Open-Source On-The-Fly Disk Encryption Software -
http://www.truecrypt.org/

Organizaciones normativas.

ANSI - *American National Standards Institute* -
http://www.ansi.org/

IEEE - *Institute of Electrical and Electronics Engineers* -
http://www.ieee.org/

IETF - *Internet Engineering Task Force* -
http://www.ietf.org/

ISO - *International Organization for Standardization* -
http://www.iso.org/

NIST - *National Institute of Standards and Technology* -
http://www.nist.gov/

NSA - *National Security Agency* -
http://www.nsa.gov/

The Wassenaar Arrangement -
http://www.wassenaar.org/

Glosario

Por último, se incluye un breve glosario general con el ánimo de proveer un rápido acceso a una descripción –y referencia dentro del libro, si el tema ha sido abordado– respecto de los términos de uso más común en el tratamiento de cuestiones relacionadas con la criptografía o con la implementación de la criptografía en sistemas informáticos.

Se incluyen conceptos criptográficos elementales, términos matemáticos, nombres de algoritmos, referencias a estándares, a organizaciones, etc. Cada término que haya sido tratado a lo largo del libro contendrá la referencia al capítulo, sección o apartado correspondiente.

Se ha decidido no evitar una buena cantidad de referencias cruzadas. Para mantener la posibilidad de la búsqueda alfabética en ambos idiomas, algunos términos ingleses se incluyen con instrucciones para la consulta del término en idioma español o por sus siglas, código o abreviación si así correspondiese.

Abeliano, grupo (matemática): Grupo abstracto (álgebra abstracta) con una operación binaria conmutativa. Estos grupos también son llamados “grupos conmutativos”.

Acuerdo de llave: Proceso por el cual dos o más partes acuerdan una llave simétrica secreta.

Acuerdo de Wassenaar: *The Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies*, o acuerdo de Wassenaar sobre los controles de exportación para armas convencionales y de bienes o productos de doble-uso y tecnologías. Acuerdo internacional que contempla regulaciones de exportación de criptografía. Más información al respecto en el capítulo 8.

AES (*Advanced Encryption Standard*): Algoritmo de criptografía simétrica, de cifrado por bloques. Fue seleccionado por la NIST para reemplazar al algoritmo DES en el año 2000. Más información al respecto en el capítulo 2, sección de algoritmos criptográficos, apartado "AES".

Algoritmo: Detalle de pasos específicos, ordenados y finitos para la solución de un problema.

Algoritmo criptográfico: Algoritmo cuyo objetivo será la encriptación, firma digital, generación de números aleatorios, etc.; generalmente basado en una función matemática. Puede consultarse el apartado a este respecto en el capítulo 2.

Alice: Nombre inglés tradicionalmente utilizado a la hora de describir protocolos criptográficos. A lo largo del libro se ha utilizado también como "parte A". Se trata de persona, proceso o entidad en general, que realiza el envío y/o primer paso de la serie correspondiente al protocolo. Véase también "Bob".

ANSI (*American National Standards Institute*): Instituto Nacional de Estándares Americano. Organización o asociación privada y sin fines de lucro de los E.E.U.U. que produce estándares industriales. Es un miembro de la ISO y del IEC. Más información en el capítulo 8, sección de estándares.

API (*Application Programming Interface*): Interfaz de Programación de Aplicaciones. Definición de funciones, procedimientos, métodos, objetos, etc. que ofrece un sistema operativo, una librería o un *framework* por ejemplo, que serán utilizados por un programador en la codificación de un programa.

Ataque criptográfico: Intento de compromiso de una parte o de la totalidad de la seguridad de un cripto-sistema.

Ataque del cumpleaños: Ataque por fuerza bruta utilizado para encontrar colisiones. Recibe su nombre del hecho de que la probabilidad de que dos o más personas, en un grupo de 23 personas, compartan el mismo día de cumpleaños es mayor al 50%.

Ataque del diccionario: Ataque de fuerza bruta que probará contraseñas y/o llaves de una lista prefijada de valores.

Ataque por fuerza bruta: Ataque que requiere la prueba de todos (o una fracción importante) los valores posibles hasta que el correcto sea encontrado. También es llamado de búsqueda exhaustiva.

Autenticación: Proceso para verificar la identidad de un usuario, dispositivo u otra entidad en un sistema informático. Además, se entiende también al proceso utilizado para la verificación del origen de la información transmitida; esto es, que esta información

esté correctamente identificada. Más información en el primer capítulo del libro (apartado "Usos o aplicaciones de la criptografía").

Autoridad Certificante: Entidad que maneja las solicitudes, certificación (autenticación del solicitante), emisión y revocación de certificados digitales. Actúa como una tercera parte "de confianza" para la administración de los certificados de llaves públicas que emite.

***Big-O* (notación):** Usada en la teoría de la complejidad de algoritmos. Notación utilizada para cuantificar el tiempo de dependencia de un algoritmo con respecto al tamaño de la entrada.

Birthday attack: Véase "Ataque del cumpleaños".

Bit: Dígito binario; los valores posibles son 0 y 1.

Block Cipher: Véase "Cifrado por bloques".

Bloque: Secuencia de bits de una longitud determinada. Por lo general, secuencias de mayor longitud son divididas en bloques del tamaño particular para la aplicación de diferentes procesos (por ejemplo, el de cifrado por bloques).

Blowfish: Algoritmo simétrico de cifrado por bloques. Más información en el capítulo 2, sección de algoritmos.

Bob: Nombre inglés tradicionalmente utilizado a la hora de describir protocolos criptográficos. A lo largo del libro se ha utilizado también como "parte B". Se trata de persona, proceso o entidad en general, que recibe el envío y/o realiza el segundo paso de la serie correspondiente al protocolo. Véase también "Alice".

Break: Véase "Quiebre o ruptura".

Brute force attack: Véase "Ataque por fuerza bruta".

Byte*:* Conjunto de 8 bits. También llamado octeto. Un byte puede representar 256 valores diferentes o símbolos.

Canal: Camino por el que circula o se transmite información en un sistema.

Campo (matemática): Estructura consistente en un conjunto finito o infinito, junto con dos operaciones binarias llamadas de adición y de multiplicación. Ejemplos típicos incluyen al conjunto de los números reales, al de los números racionales y al de los enteros módulo p.

Certificado: En criptografía, un documento electrónico que enlaza una información –como la identidad de un usuario– y una llave pública. Véase también "Autoridad Certificante".

Certificate Authority (CA): Véase "Autoridad Certificante".

Channel: Véase "Canal".

Checksum: Véase "Suma de verificación".

Cifrado: Véase "Encriptación".

Cifrado en flujo: Cifrado simétrico que encriptará un mensaje de un bit (quizá, hasta de a un byte o una palabra) a la vez.

Cifrado por bloques (o en bloques): Cifrado simétrico que encriptará un mensaje dividiéndolo en bloques de una longitud determinada de bits. Véase también "Bloque".

Cifrador: Algoritmo para la encriptación y/o la desencriptación. Véase también "Encriptación" y "Desencriptación".

Cipher: Véase "Cifrador".

Ciphertext: Véase "Texto-cifrado".

COCOM: *Coordinating Committee for Multilateral Export Controls* o Comité de Coordinación de Control Multilateral de Exportaciones. Comité internacional para la regulación de exportaciones. Más información en el capítulo 8.

Código de Autenticación de Mensaje: Función que tomará una entrada (mensaje) de longitud variable y producirá una salida de una longitud fija determinada. Se utilizará en el proceso una llave secreta, de manera que otras partes –conocedoras también de la llave– podrán verificar la autenticidad del mensaje. Más información en el capítulo 2, sección de algoritmos de funciones *hash*, apartado referido a "MAC" y "HMAC".

Códigos de autenticación de mensajes por *hash:* Combinación de una llave secreta y una función de *hash* diseñada para operar más rápidamente que una MAC y proveer la misma seguridad.

Colisión: Dada una función de –supuestamente– una vía, F, es la situación dada cuando F(x) = F(y), siendo x distinto de y.

Complejidad computacional: Cantidades de espacio o memoria y tiempo, requeridas para la resolución de un problema.

Contraseña: Véase "Palabra clave".

Cripto-análisis: Disciplina que estudia el quiebre o ruptura de un cifrador para recuperar información, o la alteración o generación de información cifrada o texto-cifrado que pudiera ser aceptada como auténtica.

Criptografía: Ciencia que estudia el diseño de algoritmos para la encriptación y desencriptación, principalmente para el aseguramiento de la privacidad y/o autenticación de mensajes o información. Más información en los capítulos 1 y 2 del libro.

Criptología: Se trata del estudio de las comunicaciones seguras; comprende a la criptografía y al cripto-análisis.

Cripto-sistema: Primera acepción: Un algoritmo de encriptación y desencriptación (cifrador), junto con todos los posibles valores de texto-plano, de texto-cifrado y llaves. Segunda acepción: Elementos involucrados en la implementación de un proceso de encriptación y desencriptación.

Cripto-sistema de curva elíptica: Un cripto-sistema de llave pública basado en las propiedades de las curvas elípticas. Véase también "Curvas elípticas".

Curvas elípticas: Conjunto de puntos (x, y) tales que satisfagan una ecuación de la forma $y^2 = x^3 + ax + b$.

DES (*Data Encryption Standard*): Algoritmo de encriptación simétrica de cifrado en bloque. Definido y adoptado por el gobierno de los EE.UU., en el año 1977, como estándar oficial. Desarrollado por IBM. Aún en la actualidad es ampliamente utilizado. Más información en el capítulo 2, sección de algoritmos criptográficos simétricos.

Descifrado: Véase "Desencriptación".

Desencriptación: Proceso inverso a la encriptación, donde, a partir del texto-cifrado, se obtiene el texto-plano original. Como con la encriptación, típicamente, el proceso es función de, o emplea, una llave criptográfica.

Diffie-Hellman: Primer algoritmo criptográfico de llave pública para el acuerdo de llaves secretas sobre un canal inseguro. Es ampliamente utilizado en la actualidad (en Internet por ejemplo, dentro del protocolo SSL). Más información en el capítulo 2, sección de algoritmos criptográficos asimétricos.

DSA (*Digital Signature Algorithm*): Algoritmo de llave pública para la generación de firmas digitales. Puede utilizarse para la autenticación, verificación de integridad y no-repudio de mensajes o información. Más detalles en el capítulo 2, sección de algoritmos criptográficos asimétricos.

***Digital Signature Standard* (DSS):** Estándar que especifica al algoritmo *Digital Signature Algorithm* (DSA). Véase también la entrada de este algoritmo.

ECC (Elliptic Curve Cryptosystem): Véase “Cripto-sistema de curva elíptica”.

ElGamal: Algoritmo de llave pública, utilizado para el acuerdo de llaves secretas, la firma digital y la encriptación de información. Más información en el segundo capítulo, sección de algoritmos.

Encriptación: Proceso por el cual se transforma un mensaje o información (texto-plano) en otra información (texto-cifrado), en apariencia ininteligible, con la intención de ocultar el contenido original. Típicamente, el proceso de transformación o reemplazo es función de una llave criptográfica, o está condicionado por el empleo de ésta.

Espacio de llave: La colección de todas las llaves posibles para un cripto-sistema.

Esteganografía: Técnicas que permiten el ocultamiento de mensajes dentro de otros, de manera que no se note la existencia de los primeros. Más información en el capítulo 1.

Factor (matemática): Dado un entero n, cualquier número que lo divida es llamado un factor de n. Por ejemplo, 7 es un factor de 91, ya que 91 divido 7 da como resultado 13, un entero.

Factoreo (matemática): Descomposición de un entero en sus factores primos.

Feistel, cifrado o cifrador: Clase especial de cifrado por bloques donde el texto-cifrado es computado a partir del texto-plano repitiendo la aplicación de la misma transformación (llamada función de ronda, o simplemente ronda).

FIPS (*Federal Information Processing Standards*): Una traducción posible: Estándares Federales de Procesamiento de la Información. Se trata de estándares anunciados públicamente desarrollados por el gobierno de los E.E. U.U. para que sean utilizados por las agencias del gobierno no militares y por los contratistas del gobierno. Los emite la NIST. Véase también “NIST”.

Frase Clave: Similar a una palabra clave, pero puede estar compuesta por varias palabras.

Galois, campo de (matemática): Campo con un número finito de elementos. El tamaño de un campo finito debe ser potencia de un número primo.

Grafo (matemática): Dentro de la teoría de grafos, conjunto de objetos (“vértices”, también llamados “nodos”) unidos por “aristas” para representar relaciones binarias entre los elementos de un conjunto.

Grupo (matemática): Dentro del álgebra abstracta, conjunto en que se define una operación binaria que debe cumplir con ciertas propiedades (asociatividad, existencia del elemento neutro, y existencia del elemento opuesto).

Hash: Término abreviado de *hashing* criptográfico; resultado o salida de una función de este tipo. Véase "*Hash*, función de".

Hash, función de: *Message digest* o resumen de mensaje. Tipo especial de una función de una vía, en la cual la entrada será de longitud variable y la salida de un tamaño fijo. Recuperar la entrada original a partir del resultado u obtener una entrada que genere el mismo resultado será inviable.

HMAC (*Hashed Message Authentication Code*): Véase "Código de autenticación de mensajes por *hash*".

IDEA: Algoritmo simétrico de cifrado en bloques. Más información en el capítulo 2, sección de algoritmos.

Identificación: Proceso que posibilita el reconocimiento de una entidad por un sistema, generalmente a través del uso de nombre de usuario único.

IEEE (*Institute of Electrical and Electronics Engineers*): Institute of Electrical and Electronics Engineers, o Instituto de Ingenieros Eléctricos y Electrónicos. Asociación mundial de estandarización sin fines de lucro creada en el año 1884. Cuenta con 360 000 miembros en 175 países.

IETF (*Internet Engineering Task Force*): Grupo abierto de técnicos y compañías interesados en la operación armónica de Internet. Muchos estándares (el x.509 por ejemplo) provienen de esta organización.

Impostura, *Impersonation:* Ocurre cuando una entidad pretende ser alguien o algo que no es.

Infraestructura de llave pública: *Framework* para la administración de certificados digitales. Los más populares son x.509 y PGP.

Integridad, control de: Uso o aplicación de la criptografía; asegura que la información o los datos no hayan sido alterados intencional o accidentalmente.

Intercambio de llave: Proceso por el cual dos o más partes intercambian llaves criptográficas dentro de un cripto-sistema.

IPsec (*Internet Protocol Security*): *Internet Protocol Security* o Protocolo de seguridad de Internet. Es un protocolo que autentica la información entrante y encripta la saliente de una computadora de manera transparente para el usuario.

ISO (*International Standards Organization*): *International Standards Organization* u Organización de Estándares Internacional. Genera estándares mundiales y está compuesta por las organizaciones estandarizadoras de los distintos países miembro.

Kerberos: Protocolo de autenticación de red. Desarrollado por el MIT. Utiliza el algoritmo DES para la encriptación y autenticación. Más información en el capítulo 7, sección Kerberos.

Key: Véase "Llave".

Key Agreement: Véase "Acuerdo de llave".

Key Exchange: Véase "Intercambio de llave".

Keyspace: Véase "Espacio de llave".

Llave, Llave criptográfica: Conjunto o cadena de bits utilizada para la encriptación y desencriptación de información. La llave determinará la salida o texto-cifrado después del proceso de encriptación sobre el texto-plano.

Llave privada: En la criptografía asimétrica o de llave pública, llave que utiliza una parte para desencriptar la información que otra parte ha encriptado. También es utilizada para firmar digitalmente una información determinada.

Llave pública: En la criptografía asimétrica o de llave pública, llave que se utilizará para encriptar información. También es utilizada para verificar firmas digitales.

Llave secreta: En la criptografía simétrica o de llave secreta, llave que permitirá la encriptación y la posterior desencriptación de una información determinada. La llave debe mantenerse confidencialmente entre las partes.

MAC (*Message Authentication Code*): Véase "Código de Autenticación de Mensaje".

MD5: Una de las más populares funciones de *hashing* criptográfico. Genera una salida de 128 bits de longitud. Puede consultarse más información en el capítulo 2, sección de algoritmos de funciones *hash*. Véase también "Hash, función de".

Message Digest: Véase "Hash, función de".

***Message Integrity Codes* (MIC):** Véase "*Hash*, función de".

***Modification Detection Codes* (MDC):** Véase "*Hash*, función de".

NIST: *National Institute of Standards and Technology* (Instituto Nacional de Estándares y Tecnología). Produce los estándares o publicaciones denominados FIPS. Más información en el capítulo 8, sección de estándares criptográficos.

NSA: *National Security Agency*. Agencia de Seguridad Nacional de los EE.UU.

One-Way Functions (OWF): Véase "Hash, función de".

Padding: Información que se concatena a una clave, llave o texto-plano, para completar el tamaño de entrada requerido por el algoritmo que se ha de utilizar.

Palabra Clave: Grupo de caracteres que será utilizado para el acceso a una información o recurso determinado. También llamada "contraseña". No confundir con "llave", aunque a veces se utilice como tal en forma directa.

Passphrase: Véase "Frase clave".

Password: Véase "Palabra clave".

PGP: Pretty Good Privacy. Aplicación para el manejo de *e-mail* de manera segura. Más información en el capítulo 7, sección PGP.

PKCS: *Public-Key Cryptography Standards*. Conjunto de estándares criptográficos referidos a cuestiones relativas al cifrado asimétrico o de llave pública, publicados por RSA Laboratories.

Plaintext: Véase "Texto-plano".

Pseudoaleatorios, números: Números con un patrón detectable muy bajo pero no completamente aleatorios. Más información en el capítulo 2, sección de protocolos.

Protocolo: Conjunto de procedimientos que involucra a dos o más partes y describe las relaciones entre ellas.

Protocolo criptográfico: Protocolo que implementa algoritmos criptográficos en sus procedimientos.

***Public Key Infrastructures* (PKI):** Véase "Infraestructura de llave pública".

Quiebre o ruptura (criptográfico): Resultado exitoso de un ataque criptoanalítico. Un algoritmo o cripto-sistema se considerará roto o quebrado cuando el mensaje original puede ser recuperado sin la utilización de la llave, o cuando la llave misma puede ser recuperada.

RC4: Algoritmo simétrico de cifrado en flujo. Más información en el capítulo 2, sección de algoritmos.

Resumen de mensaje: Véase "*Hash*".

Ronda: Cada interacción o aplicación de las técnicas de "confusión" y "difusión" en un algoritmo de cifrado.

RSA: Protocolo de llave pública, utilizable tanto para la autenticación como para la encriptación y desencriptación. Inventado en el año 1977 por Rivest, Shamir, y Adelman (de allí las siglas). Más información en el capítulo 2, sección de algoritmos de criptografía asimétrica.

***Secure Hash Algorithm* (SHA-1):** Una de las más populares funciones de hashing criptográfico. Genera una salida de 160 bits de longitud. Puede consultarse más información en el capítulo dos, sección de algoritmos de funciones hash. Véase también "Hash, función de".

Seed: Véase "Semilla".

Semilla: Típicamente información aleatoria utilizada para generar otra información, generalmente de mayor longitud, pseudoaleatoria.

SET: *Secure Electronic Transaction*, especificación de seguridad abierta, diseñada para proteger las transacciones realizadas con tarjetas de crédito. Más información en el capítulo siete.

SSH (*Secure Shell*): Protocolo para el acceso seguro a sistemas remotos. Más información en el capítulo siete, sección acerca SSH y OpenSSH.

SSL (*Secure Socket Layer*): Protocolo que funciona sobre TCP y provee funcionalidades de autenticación, confidencialidad y control de integridad de la información transferida. Más información en el capítulo siete.

Stream Cipher: Véase "Cifrado en flujo".

Suma de verificación: Utilizada en la detección de errores, una suma de verificación o *checksum* es un cálculo realizado sobre el mensaje y transmitido con el mensaje, para proteger o verificar la integridad del mensaje. Es una forma de control de redundancia.

Sustitución: Técnica de cifrado en la cual porciones del texto-plano o sus símbolos (bits, bytes o caracteres o palabras) son reemplazadas por el símbolo correspondiente de texto-cifrado.

Texto-cifrado: Salida del proceso de encriptación, obtenida a partir del texto-plano. Versión encriptada del texto-plano.

Texto-plano: Mensaje o información inicial, original, que será encriptada para la obtención del texto-cifrado.

TLS (*Transport Layer Service*): Nuevo estándar sobre, o basado en, *Secure Socket Layer (*SSL) que provee intercambio seguro de llaves entre el navegador de Internet y

un servidor. Publicado en el año 1999, las diferencias entre SSL v3 y TLS versión 1 son menores. Véase entonces también “SSL”.

Transposición: Técnica de cifrado en la cual porciones del texto-plano (bits, bytes, o caracteres) son cambiados de posición.

TripleDES, 3DES: Algoritmo de cifrado simétrico por bloques. Basado en DES, en su implementación más sencilla aplica tres veces el algoritmo original. Más información al respecto en el capítulo 2, sección de algoritmos criptográficos.

***Turing*, máquina de:** Modelo teórico de un dispositivo computacional o computadora, desarrollado por Alan Turing.

Twofish: Algoritmo simétrico de cifrado en bloques. Más información en el capítulo 2, sección de algoritmos.

Wassenaar, Acuerdo de: Véase “Acuerdo de Wassenaar”.

XOR: Operador binario. Dados los dos bits operandos, la operación resultará en un 1 si los valores son diferentes y en 0 si son iguales. XOR es una abreviación de *exclusive-OR* u OR exclusivo.