



Seguridad, criptografía y comercio electrónico con Java

Acerca de este documento

Este tutorial pretende recopilar los conceptos y tecnologías que se usan para conseguir crear sistemas informáticos seguros.

Muchos tutoriales de seguridad se centran en el uso de las herramientas administrativas. En este tutorial, aunque se evalúan bastantes herramientas desde el punto de vista administrativo, pretendemos llevar este estudio hasta el punto de vista del programador de estas herramientas. Con este fin hemos elegido las librerías criptográficas de Java, ya que creemos que son unas librerías completas, homogéneas, y que abarcan todos los aspectos de la criptografía moderna. Aunque el estudio se hace sobre Mac OS X, la interoperatividad del lenguaje Java permite llevar estos conceptos a otros sistemas operativos sin problemas.

A lo largo de este documento se empezarán estudiando las técnicas criptográficas convencionales basadas en clave secreta. Después se estudian los métodos criptográficos de clave pública, y se detallan las ventajas que aportan frente a métodos criptográficos convencionales. En una tercera parte se estudiarán las técnicas de gestión de claves. Si queremos garantizar la seguridad de un sistema, estas técnicas son tan importantes como los algoritmos en sí. Por último se estudian las firmas digitales, los certificados digitales, así como su aplicación al comercio electrónico.

Al acabar de leer este documento, esperamos que el lector tenga los conceptos necesarios como para construir sus propias aplicaciones Java que implementen las distintas partes de un sistema informático seguro.

Nota legal

Este tutorial ha sido escrito por Fernando López Hernández para MacProgramadores, y de acuerdo a los derechos que le concede la legislación española e internacional el autor prohíbe la publicación de este documento en cualquier otro servidor web, así como su venta, o difusión en cualquier otro medio sin autorización previa.

Sin embargo el autor anima a todos los servidores web a colocar enlaces a este documento. El autor también anima a bajarse o imprimirse este tutorial a cualquier persona interesada en conocer los conceptos de seguridad, criptografía y comercio electrónico.

Madrid, Febrero del 2007

Para cualquier aclaración contacte con:

fernando@DELIITmacprogramadores.org

Tabla de contenido

TEMA 1: Introducción a la criptografía

1 Terminología básica.....	9
2 Técnicas criptográficas de algoritmo público y privado	11
2.1 Técnicas criptográficas de algoritmo privado	11
2.2 Técnicas criptográficas de algoritmo público.....	11
3 Algoritmos de clave secreta y algoritmos de clave pública	13
3.1 Algoritmos de clave secreta o simétricos	13
3.2 Algoritmos de clave pública o asimétricos.....	14
4 Técnicas de criptoanálisis.....	16
5 One Time Pad	18
6 Esteganografía	19
7 Cifrado por sustitución y por transposición	19
7.1 La sustitución.....	19
7.2 La transposición	20
8 Protocolos.....	21
8.1 Definición	21
8.2 Roles que participan en un protocolo	21
8.3 Protocolo de una comunicación simétrica	23
8.4 Protocolo de partición de un secreto.....	24
8.5 Protocolo de compartición de un secreto.....	25
9 One-way functions	26
9.1 One-way hash functions	26
9.2 MAC (Message Authentication Codes)	28

Tema 2: Criptografía de clave secreta

1 Protocolo	31
1.1 Encriptación de canales y datos.....	31
1.2 Compresión, encriptación y control de errores	34
1.3 Destrucción de la información	35
2 DES (Data Encryption Standard).....	37
2.1 Orígenes de DES	37
2.2 Algoritmo.....	37
2.3 Desencriptación de DES	45
2.4 Situación actual de DES	46
3 Otros algoritmos de clave secreta	48
3.1 Algoritmos de clave secreta más conocidos	48
3.2 Restricciones de exportación de los EEUU	48
4 Modos del algoritmo	49
4.1 Modos en los cifradores de bloque.....	49
4.2 Modos en los cifradores de flujo	55
4.3 Modos mixtos	61

4.4 Qué modo usar	65
5 Padding	66
6 Claves binarias y PBE.....	67
7 Las librerías JCA y JCE	69
7.1 La librería JCA (Java Cryptography API)	69
7.2 La librería JCE (Java Cryptography Extensions).....	72
8 Proveedores de seguridad instalados	74
8.1 La clase Provider	75
8.2 Mostrar los proveedores instalados	76
9 Tipos opacos y especificaciones.....	78
10 Uso de claves desde Java.....	80
10.1 La interfaz Key	80
10.2 PublicKey, PrivateKey y SecretKey	80
10.3 KeyPairGenerator y KeyGenerator.....	81
10.4 KeyFactory y SecretKeyFactory	82
11 Uso de un cifrador	85
11.1 La clase Cipher	85
11.2 Encriptar y desencriptar de un mensaje	88
12 PBE (Password Based Encryption).....	90
13 Los cifradores de streams	98
14 La clase SealedObject	100
15 Implementar un proveedor de seguridad.....	101

TEMA 3: Criptografía de clave pública

1 Introducción	110
2 Sistemas criptográficos híbridos.....	111
3 Algoritmos criptográficos de clave pública más conocidos.....	113
3.1 Los puzzles de Merkle.....	113
3.2 El algoritmo de la mochila.....	114
3.3 RSA.....	121
4 Modo y Padding	130
5 Criptografía de clave pública en Java	131
5.1 Encriptación de una clave de sesión.....	133
5.2 Encriptación de un fichero con RSA	136

TEMA 4: Firmas digitales

1 Funciones hash y MAC en Java.....	146
1.1 Funciones hash	146
1.2 MAC (Message Authentication Codes)	147
1.3 Message Digest Streams	147
2 Firmas digitales	151
2.1 Firma de documentos con criptografía de clave secreta y árbitro	151
2.2 Firma de documentos con criptografía de clave pública	152
2.3 Algoritmos de firmas digitales más usados	154

2.4 Firmar documentos con timestamp	155
2.5 Firmar el hash de un documento	155
2.6 Múltiples firmas.....	156
2.7 Firmas digitales y encriptación	156
2.8 Ataques contra las firmas digitales RSA.....	156
3 Firmas digitales en Java.....	160
3.1 La clase <code>Signature</code>	160
3.2 La clase <code>SignedObject</code>	164

TEMA 5: Gestión de claves

1 Gestión de claves (Key management)	166
1.1 Generación de claves.....	166
1.2 Transferencia de claves	168
1.3 Almacenamiento de claves	169
1.4 Previsión de pérdida de claves	169
1.5 El ciclo de vida de una clave	170
2 Intercambio seguro de claves.....	171
2.1 Intercambio seguro de claves con criptografía simétrica	171
2.2 Intercambio seguro de claves con criptografía asimétrica	171
2.3 El man in the middle attack.....	172
2.4 Evitar el man in the middle attack	172
2.5 Intercambio seguro de claves con firmas digitales	173
3 Diffie-Hellman	174
3.1 Matemáticas previas.....	174
3.2 Algoritmo de Diffie-Hellman	175
3.3 Diffie-Hellman con tres o más participantes.....	176
3.4 Diffie-Hellman para una clave decidida por A.....	178
4 Intercambio seguro de claves en Java.....	179
5 Los certificados digitales	184
6 Los Key Distribution Center (KDC)	185
7 Kerberos.....	186
7.1 Motivación	187
7.2 Qué solución aporta Kerberos	187
7.3 Componentes de Kerberos	189
7.4 El sistema de nombres de Kerberos	190
7.5 Protocolo de Kerberos	191
7.6 Credenciales	192
7.7 Obtener el ticket inicial	193
7.8 Obtener tickets de servidor	194
7.9 Servidor maestro y esclavo	194
8 Certificación distribuida de claves	195
9 GPG (Gnu Privacy Guard).....	196
9.1 Introducción	196
9.2 Instalación y puesta a punto de GPG	199
9.3 Intercambiar la clave pública.....	202
9.4 Firmar un documento con la clave principal.....	206

9.5 Crear claves subordinadas	208
9.6 Encriptar con la clave subordinada	210
9.7 Exportar la clave subordinada	210
9.8 Encriptar ficheros con clave secreta	211
9.9 Añadir más identidades al llavero	212
9.10 Añadir más claves al llavero	214
9.11 Encriptar y firmar con una determinada clave subordinada	216
9.12 Borrar claves del llavero.....	217
9.13 Revocar claves	218
9.14 Usar un servidor de claves	220
9.15 Editar las claves privadas del llavero	221
9.16 Web of trust.....	222
10 Identificación, autentificación y autorización.....	226
10.1 Terminología	226
10.2 Login por password handshake	226
10.3 Login por clave pública	227
11 Sistemas de identificación OTP	229
11.1 Introducción	229
11.2 Tipos de password.....	230
11.3 Instalación	230
11.4 Registrar un usuario OTP en el sistema.....	232
11.5 Identificación en un sistema OTP.....	233
11.6 Generar nuevos OTP.....	234
12 Los keychain	237
13 SSH (Secure SHell)	239
13.1 ¿Qué es SSH?	239
13.2 Servicios y aplicaciones de SSH	240
13.3 Evolución histórica de SSH	243
13.4 Mecanismos de identificación	244
14 VPN	248
14.1 Qué es una VPN	248
14.2 Los firewalls.....	248
14.3 Tunneling	251

TEMA 6: PKI

1 PKI.....	258
1.1 Introducción	258
1.2 Roles.....	260
1.3 Topología	261
1.4 Los certificados X.509.....	266
1.5 Las CRL X.509.....	269
1.6 Los repositorios.....	270
1.7 Validar el camino de certificación.....	271
1.8 Formatos estándar	272
2 Certificados digitales en Java.....	274
2.1 Clases para gestión de certificados	274

2.2 El Keystore de Java	278
3 S/MIME	285
4 IPSec	286
4.1 Visión general	286
4.2 IKE (Internet Key Exchange).....	287
4.3 AH (Authentication Header)	288
4.4 ESP (Encapsulating Security Payload)	289
5 SSL y TLS	290
5.1 Introducción	290
5.2 Evolución histórica	290
5.3 Establecimiento de una conexión.....	291
5.4 Sesiones.....	294
5.5 Identificación de clientes	294
5.6 Rehandshake	294
6 OpenSSL.....	295
6.1 Encriptación de ficheros.....	295
6.2 Pasar el password	296
6.3 Message digest	297
6.4 Certificados.....	298
6.5 Firmar documentos	306
6.6 Documentos S/MIME	307
6.7 Encriptar con RSA	310
6.8 Crear un cliente y servidor SSL.....	311
6.9 La herramienta ssldump.....	313
7 SSL en Java	315
7.1 JSSE.....	315
7.2 Keystores y truststores	316
7.3 Las socket factories	316
7.4 Cliente y servidor SSL.....	318
6.6 Sesiones.....	321
6.7 Identificación del cliente	322
6.8 Depurar el handshake	324
7 HTTPS.....	326
7.1 Introducción	326
7.2 Proxies	326
7.3 Negociación de las conexiones HTTPS	327
7.4 Problemas de seguridad.....	328
8 HTTPS en Java.....	331
8.1 Cliente y servidor HTTPS	331
8.2 No comprobar el CN	334
8.3 Propiedades de HTTPS	335
9 SMTP sobre TLS	336
9.1 Introducción	336
9.2 Protocolo	336
9.3 Tipos de identificación	337

Tema 1

Introducción a la criptografía

Sinopsis:

En este primer tema se presenta una descripción general de algunos conceptos criptográficos que vamos a necesitar en el resto del documento.

Se empiezan dando unas definiciones y nomenclaturas generales, para luego clasificar las principales técnicas criptográficas, así como ejemplos de algunas técnicas criptográficas históricas.

Este primer tema es puramente conceptual con lo que no se trata el tema de la programación con las librerías de seguridad de Java.

1 Terminología básica

Para empezar nuestro estudio vamos a definir una serie de conceptos básicos que necesitaremos usar durante el resto del documento.

Comenzaremos viendo a qué es a lo que se llama criptografía. Para nosotros la **criptografía** va a ser un conjunto de técnicas que nos permiten enviar un mensaje desde un emisor a un receptor sin que nadie que intercepte el mensaje en el camino pueda interpretarlo.

Se llama **criptógrafos** a las personas que estudian y usan la criptografía. El objetivo de los criptógrafos es poder enviar mensajes de forma segura, es decir, sin que ninguna otra persona puede descubrir qué mensaje es el que está enviando por un canal inseguro. Ejemplos de canales de comunicación pueden ser una carta de correo o Internet.

El **criptoanálisis** es una técnica que busca poder descifrar mensajes cifrados. A las personas que realizan el criptoanálisis se les llama **criptoanalistas**.

La **criptología** es la rama de la matemática que se encarga de estudiar tanto la criptografía como el criptoanálisis. Los **criptólogos** son las personas que realizan esta tarea.

La criptología se puede dividir en criptografía y criptoanálisis. En este documento nos vamos a dedicar a estudiar principalmente la **criptografía**, el **criptoanálisis** lo estudiaremos sólo de cuando hablamos de los problemas de seguridad que tienen algunas técnicas criptográficas.

Vamos a ver ahora que elementos intervienen en un sistema criptográfico. La Figura 1.1 resume estos elementos gráficamente:

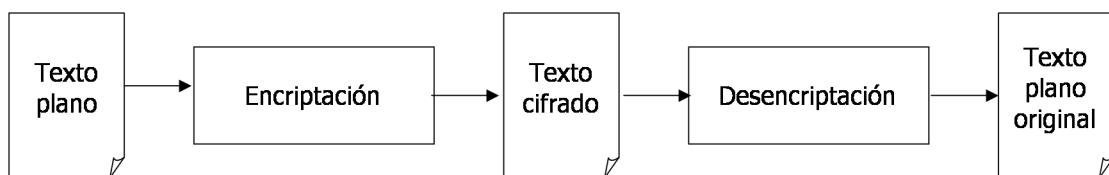


Figura 1.1: Elementos de un sistema criptográfico

El **texto plano (plaintext)** es el mensaje que queremos transmitir de forma confidencial. La **encriptación** es el proceso de transformar el texto plano en un texto cifrado que nadie pueda interpretar. La **desencriptación** es el proceso de volver a transformar el texto cifrado en el texto plano original.

Al **texto cifrado** se le llama también **texto encriptado**, que de hecho es un nombre más correcto, ya que "cifrado" significa que está siendo representado por otro sistema de codificación (normalmente numérico), mientras que "encriptado" significa que no se puede acceder a él. Sin embargo, en los libros se usa con más frecuencia el término "cifrado" (ciphertext) y nosotros lo llamaremos también así.

Normalmente al texto plano se le representa con una M (Mensaje) o una P (Plaintext), y al texto cifrado por una C (Cifrado).

El proceso de la encriptación se representa con una función $E()$ de forma que:

$$C = E(P)$$

Al proceso de la desencriptación se le representa como una función $D()$ de forma que:

$$P = D(C)$$

Para que un sistema criptográfico sea correcto, tiene que cumplirse la propiedad:

$$P = D(E(P))$$

Si este documento le está resultando útil puede plantearse el ayudarnos a mejorarlo:

- Anotando los errores editoriales y problemas que encuentre y enviándlos al sistema de Bug Report de Mac Programadores.
- Realizando una donación a través de la web de Mac Programadores.

2 Técnicas criptográficas de algoritmo público y privado

En este apartado vamos a comentar las diferencias que existen entre las técnicas criptográficas que utilizan un **algoritmo público**, conocido por todo el mundo, y las que utilizan un **algoritmo privado**, que en principio sólo conoce el que hizo el programa de encriptación/desencRIPTACIÓN¹.

2.1 Técnicas criptográficas de algoritmo privado

Las técnicas criptográficas de algoritmo privado son técnicas en las que la confidencialidad de los datos reside en el desconocimiento público de la forma en que se ha codificado un mensaje.

En principio podría parecer una buena idea, ya que al no conocer nadie el algoritmo de encriptación/desencRIPTACIÓN del mensaje, nadie podría desencriptar el mensaje. Por desgracia, las técnicas de criptoanálisis actuales descubren muy fácilmente el algoritmo empleado (especialmente si conocen un trozo del texto plano), con lo que en la práctica han dejado de utilizarse.

2.2 Técnicas criptográficas de algoritmo público

Las técnicas criptográficas de algoritmo público son técnicas en las que la confidencialidad de los datos reside en la clave utilizada, y no en el algoritmo empleado, con lo que el sistema continua siendo igual de seguro si desvelamos el algoritmo.

De hecho, en estas técnicas cuando desvelamos el algoritmo suele aumentar la seguridad, ya que el algoritmo es estudiado por criptólogos de todo el mundo, y si en un tiempo prudencial ninguno ha encontrado un defecto, podemos estar seguros de que el algoritmo es seguro. En el resto de este documento se va a tratar sólo este segundo tipo de criptografía.

En las técnicas criptográficas de algoritmo público, las funciones de encriptar $E()$ y desencriptar $D()$, además de texto plano o cifrado, reciben una **clave K** (key) de forma que:

$$\begin{aligned}C &= E(P, K) \\P &= D(C, K) \\P &= D(E(P, K), K)\end{aligned}$$

¹ El lector no debe confundir las técnicas criptográficas de *algoritmo* público y privado con las técnicas criptográficas de *clave* pública y privada que veremos luego.

Más formalmente se define un **criptosistema** como una quíntupla (P, C, K, E, D) donde:

- P es el conjunto de textos planos
- C es el conjunto de textos cifrados
- K es el conjunto de claves
- $E()$ es la función de encriptación
- $D()$ es la función de desencriptación

Para que a esta definición se la considere un criptosistema se debe cumplir que las funciones $E()$ y $D()$ sean **inyectivas**, es decir, que para cada elemento de P haya un solo elemento de C , y que para cada elemento de C haya un solo elemento de P . Además debe cumplirse que:

$$\forall x \in P, k \in K \exists y \in C / D(y = E(x, k), k) = x$$

$$\text{Es decir, dadas } x_1 \in P, x_2 \in P / x_1 \neq x_2 \Rightarrow \forall k \in K E(x_1, k) \neq E(x_2, k)$$

Más adelante veremos que a veces la clave de encriptación k_1 y de desencriptación k_2 son distintas, tal como muestra la Figura 1.2.

En estos casos:

$$\begin{aligned} C &= E(P, k_1) \\ P &= D(C, k_2) \\ P &= D(E(P, k_1), k_2) \end{aligned}$$

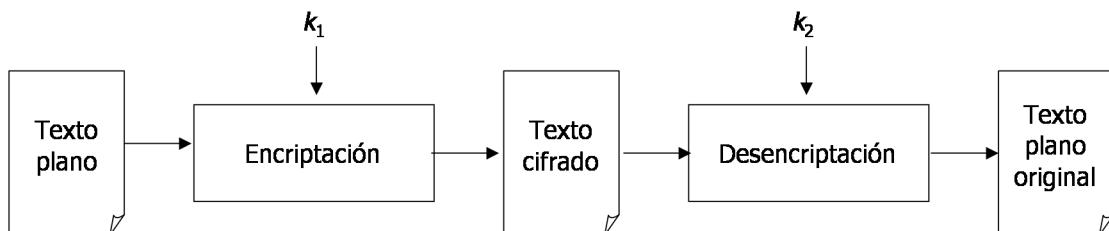


Figura 1.2: Claves de encriptación/desencriptación distintas

3 Algoritmos de clave secreta y algoritmos de clave pública

Las técnicas criptográficas de *algoritmo* público (que son las que nosotros vamos a estudiar), son técnicas en las que la seguridad de los datos reside en la clave.

Los algoritmos públicos a su vez se subclasifican en dos:

1. Algoritmos de *clave* secreta o simétricos
2. Algoritmos de *clave* pública o asimétricos

A continuación vamos a detallar cada uno de ellos.

3.1 Algoritmos de clave secreta o simétricos

En estos algoritmos es necesario que antes de empezar a transmitir datos de forma segura, el emisor y el receptor se pongan de acuerdo en la clave a utilizar.

A estos algoritmos se les llama de **clave secreta**, y no de **clave privada**, porque la clave no la conoce un sólo individuo, sino dos.

Estos algoritmos son los más clásicos, y fáciles de entender, pero tienen una limitación importante, y es que el emisor y el receptor se tienen que haber puesto en contacto previamente para acordar la clave secreta, lo cual es posible en algunos casos (p.e. comunicación militar, cartas de amor, encriptar un fichero), pero no en otros (p.e. comunicación segura entre un browser y un servidor web).

Los algoritmos de clave secreta se subdividen en dos tipos, en función de la forma en que se codifica y descodifica el mensaje.

Algoritmos de flujo (stream algorithms). Son algoritmos en los que las funciones $E()$ y $D()$ reciben el mensaje a codificar o descodificar bit a bit. En consecuencia, se hace una llamada a función por cada bit a codificar o descodificar. Por su naturaleza, son más apropiados para implementar en hardware (circuitos diseñados para encriptar y desencriptar).

Algoritmos de bloque (block algorithms). Son algoritmos en los que las funciones $E()$ y $D()$ no reciben un sólo bit, sino un bloque de bits (normalmente 64 bits ó 128 bits). En estos algoritmos habitualmente durante la última llamada hay que meter un relleno (padding) para que completemos los 64 ó 128 bits. Son más apropiados para implementar en software.

3.2 Algoritmos de clave pública o asimétricos

Esta nueva forma de criptografía fue propuesta por primera vez por Whitfield Diffie y Martin Hellman, dos profesores de la Universidad de Stanford en 1976, estos señores propusieron un algoritmo de encriptación llamado algoritmo de la mochila, que aunque actualmente está roto, sentó las bases de una nueva forma de pensar, en la que se basaron otros algoritmos, como por ejemplo el algoritmo RSA.

En estas mismas fechas Ralph Merkle, un profesor de la Universidad de Berkeley (que por aquel entonces era todavía estudiante de postgrado), había propuesto estas mismas ideas con otro algoritmo llamado “Los puzzles de Merkle”, pero finalmente se atribuyó la invención de los algoritmos de clave pública a Whitfield Diffie y Martin Hellman que fueron los primeros en publicarla.

La gran novedad que introducen los algoritmos de clave pública es que permiten que emisor y receptor se comuniquen de forma segura sin haberse puesto de acuerdo previamente en una clave secreta (p.e. un browser se comunica con un servidor web de forma segura a través de HTTPS sin que previamente se conocieran para haber acordado una clave secreta).

Aquí, para que el emisor pueda enviar información al receptor de forma segura, el receptor debe disponer de dos claves:

- **Clave pública.** Esta clave la debe conocer todo el mundo, y sólo vale para encriptar mensajes.
- **Clave privada.** Esta clave sólo la conoce el receptor (a diferencia de la clave secreta que la conocían emisor y receptor), y sólo vale para desencriptar mensajes.

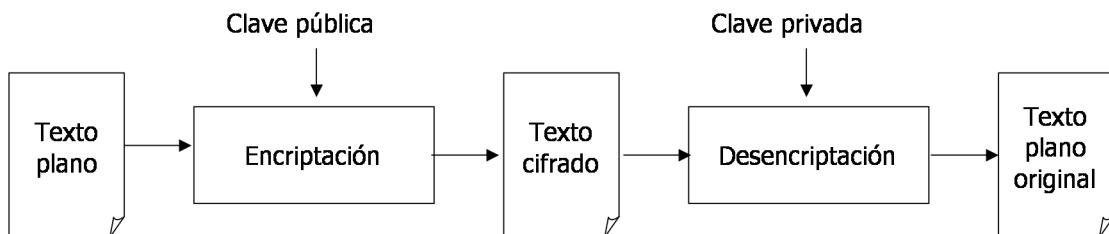


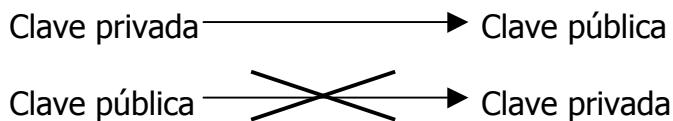
Figura 1.3: Claves para la encriptar y desencriptar en algoritmos de clave pública

Tal como muestra la Figura 1.3, de esta forma el emisor puede encriptar mensajes, y sólo el receptor puede desencriptarlos. Por ejemplo, supongamos que un banco tiene una clave privada y entrega la correspondiente clave pública a sus clientes. En este caso los clientes pueden enviar mensajes al banco encriptados con la clave pública del banco, y tienen la seguridad de que sólo el banco los sabrá desencriptar.

Para que un posible espía que haya en medio no pueda desencriptar los mensajes encriptados con la clave pública, pero sí que los pueda desencriptar el receptor, lo único que se exige es que:

1. Dada la clave privada sea posible calcular la clave pública.
2. Dada la clave pública sea imposible deducir la clave privada.

Es decir:



En concreto, el receptor elige una clave privada, calcula su clave pública, y pide al emisor que le mande el mensaje codificado con esa clave pública.

Bajo estas condiciones, si el espía (que sólo dispone de la clave pública) no puede deducir la clave privada, no puede desencriptar el mensaje.

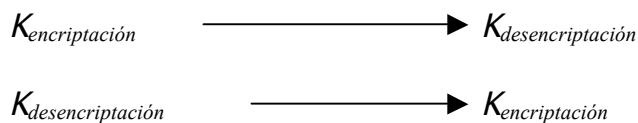
Es decir, todo el mundo puede hacer:

$$C = E(P, K_{publica})$$

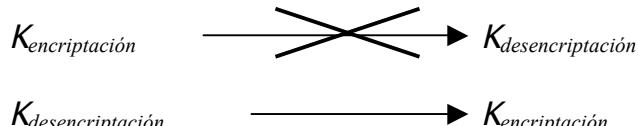
Pero sólo el receptor puede hacer:

$$P = D(C, K_{privada})$$

A los algoritmos de clave secreta se les llama también algoritmos **simétricos**, ya que, como muestra la Figura 1.2, o bien utilizan la misma clave para encriptar y desencriptar, o bien utilizan claves distintas, pero siempre se cumple que:



A los algoritmos de clave pública se les llama **asimétricos** porque:



4 Técnicas de criptoanálisis

Como se indicó en el apartado 1, el **criptoanálisis** es la técnica de intentar descifrar mensajes cifrados. En concreto el criptoanalista intenta recuperar, o bien el texto plano, o bien la clave.

Las técnicas de criptoanálisis más utilizadas son seis:

1. Criptoanálisis de texto cifrado. El criptoanalista tiene uno o más mensajes cifrados con el mismo algoritmo y clave¹. Su tarea es recuperar el texto plano de uno o más mensajes, o mejor aun, deducir la clave utilizada.

2. Criptoanálisis de texto plano no elegido. El criptoanalista tiene acceso, no sólo a varios mensajes cifrados, sino también a sus correspondientes textos planos.

Esto ocurre cuando tenemos acceso al módulo de encriptación, pero el texto a encriptar no lo decide el criptoanalista. También ocurre si por ejemplo encriptamos un fichero .zip, aquí el criptoanalista sabe cuál es la cabecera de un fichero .zip. Su misión es deducir la clave usada para encriptar los mensajes.

3. Criptoanálisis de texto plano parcialmente elegido. El criptoanalista no sólo tiene acceso al texto cifrado y sus correspondientes textos planos, sino que además puede elegir parcialmente los textos planos a encriptar. Esto ocurre cuando tenemos acceso al módulo de encriptación, y el texto a encriptar lo puede elegir parcialmente el criptoanalista. P.e. un usuario de un servicio de red encriptado, puede elegir los mensajes que quiere enviar. En consecuencia, este ataque es más efectivo que el anterior. La misión del criptoanalista es deducir la clave usada para encriptar los mensajes.

Este tipo de ataque es especialmente efectivo si hay relativamente pocos mensajes que se pueden enviar (como suele pasar en los protocolos de comunicación), entonces el criptoanalista puede sacar los mensajes encriptados que corresponden a cada mensaje. Además, muchas veces basta con saber que un texto cifrado no corresponde a un texto plano.

4. Criptoanálisis de texto plano elegido. Este es un caso muy usado del tipo anterior en el que el criptoanalista puede elegir qué mensaje encriptar en función de los mensajes previamente encriptados.

En este caso el criptoanalista suele elegir encriptar mensajes muy parecidos para observar las diferencias entre los mensajes cifrados.

¹ Cuando se cifra dos veces el mismo texto plano con el mismo algoritmo y clave no tiene por qué (y de hecho no suele) obtenerse el mismo texto cifrado.

5. Criptoanálisis de texto cifrado elegido. El criptoanalista puede elegir diferentes textos cifrados a desencriptar, y tiene acceso sus correspondientes textos planos. Esto ocurre cuando tenemos acceso al módulo de desencriptación. Su trabajo es deducir la clave. Este ataque se utiliza sobre todo en algoritmos de clave pública, como veremos más adelante.

6. Criptoanálisis de textos elegidos. Así se llama al criptoanálisis en el que podemos usar a la vez criptoanálisis de texto plano elegido y criptoanálisis de texto cifrado elegido. Esto ocurre cuando tenemos acceso a los módulos de encriptación y desencriptación.

Lars Knudsen en su libro "Block Ciphers: Analysis, Design, Applications" clasifica la resistencia ante un criptoanálisis de los algoritmos en:

1. **Totalmente roto.** El criptoanalista encontró la clave.
2. **Deducción global.** El criptoanalista encontró un algoritmo que le da el texto plano de cualquier texto cifrado.
3. **Deducción local.** El criptoanalista encontró el texto plano de un mensaje cifrado concreto.
4. **Deducción de información.** El criptoanalista encontró alguna información que puede ser usada más tarde para romper el algoritmo.

Se dice que un algoritmo criptográfico es **incondicionalmente seguro**, si independientemente de la cantidad de texto plano y cifrado que el criptoanalista obtenga, no puede deducir el texto plano de otro texto cifrado que el sistema genere. Es decir, un algoritmo criptográfico es incondicionalmente seguro si un usuario de un sistema criptográfico, por mucho texto que encripte y desencripte, no puede obtener el texto plano de mensajes cifrados enviados por otros usuarios.

El único algoritmo de encriptación que se conoce con estas características es el One Time Pad (que explicaremos en el siguiente apartado), que además es el único que no se puede romper con un tiempo y recursos infinitos. Todos los demás algoritmos criptográficos se pueden romper, con un criptoanálisis de texto cifrado, o mediante la técnica de **fuerza bruta**, que consiste en probar todas las posibles claves, una a una, hasta encontrar las que dan un texto plano con sentido.

Decimos que un algoritmo es **computacionalmente seguro** si no se puede romper con los recursos computacionales presentes o futuros en un tiempo menor a la utilidad del mensaje encriptado.

5 One Time Pad

Como hemos comentado en el apartado anterior, el único algoritmo de encriptación perfecto que se conoce es el One Time Pad, inventado en por Major Joseph Manborgue en 1917.

Para aplicarlo necesitamos disponer de una larga serie de caracteres aleatorios (la clave) y el receptor debe disponer de la misma serie.

A cada carácter que queremos enviar le hacemos un XOR con su correspondiente carácter de la clave, de forma que la clave que usamos para encriptar el mensaje no se vuelve a usar para encriptar otros mensajes.

Para obtener el mensaje original, el receptor lo único que tiene que hacer es volver a hacer un XOR a cada carácter encriptado con el carácter de la clave.

P.e.: el emisor hace:

Texto plano	H O L A
Clave	@ á \$ % (XOR)
Texto cifrado	~ # B]

Y después el receptor hace:

Texto cifrado	~ # B]
Clave	@ á \$ % (XOR)
Texto plano	H O L A

Ya que todos los caracteres cifrados tienen la misma probabilidad, no hay forma de que el criptoanalista sepa a qué mensaje corresponde, es decir, la probabilidad de HOLA es la misma que la de MATA y la misma de @ | & r.

Es importante que la clave sea totalmente aleatoria, y no se haga con un generador de números pseudoaleatorio, o el criptoanalista podría deducir la serie pseudoaleatoria.

Esta técnica se usó durante la segunda guerra mundial, y actualmente por la CIA. Como la clave es muy grande, normalmente se transporta en un CD.

6 Esteganografía

La **esteganografía** consiste en enviar un mensaje secreto escondido en otro mensaje no secreto.

Esta técnica la inventaron los poetas, que metían un mensaje oculto usando cada una de las primeras letras de cada verso (línea) de un poema.

Una versión moderna de esta técnica es utilizar el bit menos significativo de una imagen, y hay programas freeware que lo hacen.

El principal problema de esta técnica es que, como técnica criptográfica de algoritmo privado que es, si alguien sospecha que se están enviando mensajes así, le resultará fácil desencriptarlos. Otro problema es que necesitamos un mensaje grande para esconder el mensaje secreto. Por ejemplo, en un bitmap necesitaríamos 8 píxeles para esconder un byte.

7 Cifrado por sustitución y por transposición

En la antigüedad se utilizaban técnicas criptográficas de algoritmo privado y sin clave. En la actualidad, sin embargo, predominan las técnicas criptográficas de algoritmo público y con clave.

Las dos técnicas criptográficas más utilizadas en la antigüedad fueron la sustitución y la transposición, en este apartado vamos a comentar con más detalle estas dos técnicas.

7.1 La sustitución

El **cifrado por sustitución** consiste en reemplazar cada carácter del texto plano por otro carácter en el texto cifrado, y para desencriptar se sigue el proceso inverso.

Un ejemplo de cifrado por sustitución muy conocido es el "Cifrado del César", que lo usó para enviar mensajes a sus generales en el frente. Consistía en que sustituían cada letra del mensaje por tres letras más adelante en el vocabulario en módulo 26, tal como muestra el siguiente ejemplo:

Texto plano	→	H U Y E
Texto cifrado	→	K X B H

Para descifrarlo sólo había que sustituir las letras cifradas por tres letras atrás en el vocabulario, es decir:

Texto cifrado	\longrightarrow	K X B H
Texto plano	\longrightarrow	H U Y E

En los grupos de news es muy típico esconder mensajes obscenos o soluciones a puzzles encriptados con ROT13, una técnica que consiste en que para encriptar un mensaje sumamos 13 en módulo 26 a las letras mayúsculas y minúsculas (no a los símbolos). Después para decodificar los mensajes volvemos a sumar 13 en módulo 26 a cada una de las letras, es decir:

$$P = \text{ROT13}(\text{ROT13}(P))$$

7.2 La transposición

Los **cifradores por transposición** simplemente cambian el orden de las letras.

El cifrado por transposición más común es el **cifrado por columnas**, que consiste en colocar el texto en filas de ancho predeterminado y leerlo por columnas.

P.e. si queremos transmitir:

Texto plano: NOS ATACAN CON CARBUNCO

N O S _ A
T A C _ A N
— C O N —
— C A R B U —
N C O — —

Texto cifrado: NT CN OACACSCORO ANB AN U

Como ya habíamos indicado, estas técnicas criptográficas se encuentran en la actualidad superadas. El libro: "Cryptanalysis" H.F. GAINES Ed. American Photographic Press, muestra la fácil que es encontrar el algoritmo privado utilizado cuando se utilizan técnicas de sustitución, transposición o ambas combinadas.

8 Protocolos

8.1 Definición

En criptografía llamamos **protocolo** a el orden en que tenemos que ejecutar los algoritmos criptográficos para resolver un problema criptográfico de forma segura.

Los protocolos existen porque a veces, si estos pasos se ejecutan de forma incorrecta, se puede romper la seguridad del sistema criptográfico, aunque todos los algoritmos utilizados sean algoritmos totalmente seguros.

Los protocolos han sido diseñados por expertos y publicados en muchos libros sin que nadie haya encontrado un agujero de seguridad en un sistema que funcione tal como dice el protocolo.

8.2 Roles que participan en un protocolo

En los protocolos que vamos a estudiar van a participar varios tipos de personajes que vamos a comentar a continuación.

8.2.1 Los interlocutores

En un protocolo puede participar un único individuo *A* (p.e. encriptar un fichero), dos individuos *A*, *B* (p.e. en una comunicación cifrada), o incluso más de dos interlocutores, en cuyo caso los llamaremos *A*, *B*, *C*, *D*...

8.2.2 Los árbitros

Cuando los individuos que participan en un protocolo no confían el uno en el otro, pueden necesitar la ayuda de un **árbitro**, que es un tercer participante en el que ambos confían, es decir, el árbitro tiene que ser imparcial, y ambos se someten a las resoluciones del árbitro.

P.e. Si *A* quiere vender un coche a *B*, pero, por un lado *B* quiere pagar con cheque a *A*, y *A* no quiere dar las llaves del coche a *B* hasta que haya comprobado que el cheque es válido, puede ejecutar un protocolo con árbitro como el siguiente:

1. *A* da las llaves del coche al árbitro.
2. *B* da el cheque a *A*.
3. *A* deposita en cheque en el banco.

4. Transcurrido el tiempo pactado el árbitro da las llaves a B , pero si en ese tiempo A demuestra al árbitro que el cheque no tenía fondos, entonces el árbitro devuelve las llaves a A .

Un tipo de árbitro son los **expendedores de certificados**, que son árbitros que pueden firmar documentos para demostrar su autenticidad.

P.e. un expendedor de certificados puede ser un banco. El banco puede expedir un cheque certificado, en el que garantiza que ese dinero está en la cuenta de B , y que B no lo puede sacar. En este caso el protocolo para el problema de que A venda un coche a B podría ser:

1. B pide un cheque certificado al banco
2. A da las llaves a B , y B da el cheque certificado a A
3. A cobra el cheque sin problemas, ya que está certificado por el banco

Otro tipo de árbitro son los **notarios** cuya misión es tomar nota de que una operación entre A y B se ha realizado, en caso de que posteriormente hubiese una disputa entre A y B , el notario diría qué es lo que realmente pasó.

Debido a que el coste económico de un árbitro suele ser grande, muchos protocolos utilizan **jueces**, que son árbitros que sólo actúan en caso de que haya una disputa.

Si usamos un juez, el protocolo general sería:

1. A y B negocian los términos del contrato
2. A firma el contrato
3. B firma el contrato

Si más tarde hubiera una disputa:

4. A y B presentan el contrato al juez
5. El juez resuelve la disputa

Otro tipo de protocolo que estudiaremos son los **protocolos mutuamente impuestos**, que son protocolos en los que por la naturaleza del protocolo ninguno puede mentir, y si intenta hacerlo se detecta antes de acabar el protocolo. Estos protocolos son más baratos, pero por desgracia no siempre se pueden usar.

8.2.3 Los atacantes

Los **atacantes** son individuos que no deberían formar parte del protocolo.

Un posible atacante es el **espía**, que es un atacante que no modifica el protocolo, sino que sólo intenta acceder a los datos intercambiados. A este tipo de atacante también se le llama **atacante pasivo**.

Otro tipo de ataque más peligroso es el **ataque activo**, en el que el atacante puede ver y modificar los datos que se intercambian en el protocolo.

En caso de que el atacante activo sea uno de los participantes, se le llama **participante mentiroso**, y también es un ataque muy peligroso.

8.3 Protocolo de una comunicación simétrica

La **comunicación simétrica** es la que se realiza utilizando algoritmos criptográficos de clave secreta.

En este tipo de comunicación el protocolo que se utiliza tiene la siguiente forma:

1. A y B se ponen de acuerdo en el sistema criptográfico a utilizar.
2. A y B se ponen de acuerdo en la clave a utilizar.
3. A encripta un mensaje usando el algoritmo y clave acordados.
4. A envía el mensaje encriptado a B .
5. B desencripta el mensaje usando el algoritmo y clave acordados.

En este caso un espía puede ver el mensaje que se está transmitiendo en el paso 4, y puede intentar criptoanalizarlo, en cuyo caso estaría realizando un criptoanálisis de texto cifrado que, como vimos en el apartado 4, es el más difícil de criptoanalizar.

En la actualidad los sistemas criptográficos son lo suficientemente seguros a un criptoanálisis de texto cifrado por fuerza bruta, pero si el espía no es estúpido puede intentar espiar el paso 2, en cuyo caso el ataque es más peligroso. Por esta razón es por lo que hoy en día es tan importante lo que se denomina la gestión de claves (key management) que estudiaremos en el Tema 5.

Un atacante activo, sin embargo, puede realizar otro tipo de ataque llamado **Denial of Service attack (DoS)**, que consiste en romper la comunicación en el paso 4 para que los interlocutores no se puedan comunicar. Este ataque es un ataque moderadamente efectivo, ya que cuando A y B descubran que la comunicación de red está rota procederán a su reparación.

Un DoS más efectivo consiste en cambiar los mensajes encriptados por otros cualquiera de forma que cuando los reciba y desencripte B , no encontrará más que basura. Muchas veces esto es suficiente para que el programa que está recibiendo los mensajes pierda el control de flujo y explote. Este es un

ataque más difícil de detectar ya que seguramente B pensará que lo que está pasando es que su programa tiene un bug.

De este estudio concluimos, que los sistemas de comunicación mediante criptografía simétrica tienen dos problemas (que solucionará la criptografía asimétrica):

1. Si alguien se apodera de la clave romperá todo el sistema.
2. Si queremos montar una red que utilice criptografía simétrica, el número de claves que necesitamos para comunicar de forma segura cada puesto con los demás aumenta rápidamente, en concreto, una red de n ordenadores necesita $(n*(n-1))/2$ claves diferentes.

A estos dos problemas, debemos añadir el problema de que emisor y receptor se tienen que poner previamente de acuerdo en una clave secreta, tal como comentamos en el apartado 3.1.

8.4 Protocolo de partición de un secreto

La partición de un secreto consiste en dividir un mensaje en piezas, consiguiendo que, de forma individual no signifique nada, pero si unimos todas las piezas tenemos el mensaje original.

P.e. si tenemos la fórmula de la Coca-Cola, un árbitro la puede partir en dos, y dar una pieza a A y otra a B , siguiendo el protocolo que vamos a detallar:

1. El árbitro genera una serie aleatoria de bits R , que tendrá la misma longitud que el mensaje M .
2. El árbitro hace un XOR de M y R para obtener S :

$$M \wedge R = S$$
3. El árbitro da R a A , y S a B .
4. Después, para reconstruir el mensaje se reúnen A , B y hacen un XOR a sus partes, para obtener así el mensaje original: $M = R \wedge S$

Si el árbitro es de confianza, la serie aleatoria es totalmente aleatoria, y no se repite, estamos usando un cifrado one-time pad, que es incondicionalmente seguro.

Esta técnica se puede extender fácilmente a más de dos participantes generando el árbitro más series aleatorias.

P.e si fueran cuatro participantes:

1. El árbitro genera 3 series aleatorias R , S , T .
2. El árbitro hace un XOR a M con las tres series aleatorias:

$$M \wedge R \wedge S \wedge T = U$$

3. El árbitro da R a A , S a B , T a C y U a D . Cuando quieran recuperar el secreto, se reúnen los cuatro y hacen un XOR a sus series, para obtener el mensaje original: $R \wedge S \wedge T \wedge U = M$

Lógicamente, si el mensaje es muy grande, es mejor compartir una clave secreta que luego se use para encriptar y desencriptar todo el mensaje.

Sin embargo, partir un secreto tiene un serio inconveniente: Si uno de los miembros muere (o se pasa a la competencia) el secreto se pierde para siempre.

8.5 Protocolo de compartición de un secreto

El inconveniente de la técnica de partición de un secreto se puede solucionar con técnicas de compartición de un secreto, en las que ya no es necesario que todos los miembros estén presentes para desvelar un secreto, sino que sólo es necesario que se alcance un umbral mínimo.

P.e. si estamos diseñando un sistema de lanzamiento de misiles nucleares, podríamos exigir que al menos 3 de los 5 generales del Pentágono estén de acuerdo para que los misiles se lancen. Incluso lo podríamos complicar más y exigir que si un general no está disponible, su voto pueda ser sustituido por el de 5 coronelos. Para ello utilizamos un sistema de participaciones, de forma que los generales tienen 5 participaciones, y los coronelos 1. Ahora fijamos un sistema de 20/30 participaciones para que se inicie el lanzamiento de los misiles. Los algoritmos de compartición de secreto están descritos en el libro: "An Introduction to Shared Secrets and/or Shared Control Schemes" GUS SIMMONS. Ed IEEE Press.

Los algoritmos de partición y compartición de secretos tienen otros inconvenientes como son:

1. Es fácil que un miembro minta. P.e. si C es un pacifista puede introducir un número erróneo, y aunque todos los demás metan el número correcto, el resultado es incorrecto. Lógicamente, también se han inventado algoritmos que evitan esto.
2. Cuando se reúnen para reconstruir el secreto todos tienen que desvelar sus números. También se han inventado formas de que los números sigan siendo secretos.

9 One-way functions

Una **one-way function** o **función unidireccional** es una función cuyo valor es fácil de calcular, pero es muy difícil calcularlo en el sentido inverso, es decir, dado x es fácil calcular $f(x)$, pero si tenemos $f(x)$ es muy difícil calcular a qué x corresponde.

P.e. Dado x es relativamente fácil calcular x^2 , pero dado x^2 es relativamente más difícil calcular su raíz.

Matemáticamente hablando, no existen pruebas de que existan las one-way functions, sino que se utilizan funciones que parecen difíciles de invertir, pero nadie asegura que alguien pueda alguna vez descubrir una forma fácil de invertir esa función.

Obsérvese, que en principio las one-way functions tal cual no tienen mucha aplicación, ya que una vez que encriptamos un mensaje usando una one-way function, nadie incluido el receptor lo puede desencriptar. La utilidad la encontraremos cuando las combinemos con otras técnicas.

9.1 One-way hash functions

Vamos a explicar que son las **funciones hash unidireccionales**, también llamadas **funciones de compresión** o **message digest**. Estas funciones tienen muchas aplicaciones en criptografía, como iremos viendo a lo largo del documento.

Estas funciones reciben un texto de longitud variable y devuelven otro de longitud fija (generalmente más pequeño).

Una función hash muy simple sería una que recibe un texto y devuelve el XOR de todos sus bytes. Por desgracia esta función hash no sería unidireccional, ya que es muy fácil encontrar otro texto de entrada que devuelva la misma salida.

Una buena función hash es aquella para la que es muy difícil encontrar un texto de entrada que dé la misma salida. Obsérvese que, de hecho, tiene que haber muchas entradas que den la misma salida, ya que el tamaño de la entrada suele ser mucho mayor al de la salida, pero lo difícil es que dada una salida encontremos otra entrada que dé esa misma salida.

Además para que la función hash sea buena los valores de las salidas deben estar uniformemente distribuidos, con el fin de minimizar las colisiones.

También hay que tener en cuenta que las funciones hash son públicas, es decir, todo el mundo sabe cómo calcular el hash de un texto. Su fuerza reside en la imposibilidad de encontrar otra entrada que produzca esa salida.

Las dos funciones hash más usadas se llaman: MD5 y SHA1.

Como digimos en el apartado 8.3, un DoS muy efectivo consiste en cambiar los mensajes encriptados que viajaban por la red con el fin de que la aplicación receptora pierda el control de flujo y explote. Una de las aplicaciones de las funciones hash unidireccionales es evitar este problema, es decir comprobar que el mensaje encriptado llega sin alteraciones, y si las ha sufrido, detectarlo.

Para ello, tal como ilustra la Figura 1.4, se utiliza el siguiente protocolo:

1. *A* escribe un mensaje *M*.
2. *A* calcula el hash de *M*.
3. *A*, usando la clave secreta, encripta *M* concatenado con su hash para obtener *C*.
4. *A* envía a *B* el mensaje encriptado *C*.

Ahora en recepción:

5. *B* desencripta *C* con la clave secreta.
6. *B* comprueba que el hash concatenado *H* corresponda al mensaje *M*.
7. *B* utiliza el mensaje plano *M* recibido.

Obsérvese que si ahora un atacante activo modificara el texto cifrado *C* durante su viaje por la red, *B* detectaría que el hash *H* del mensaje recibido *M* no es correcto, y no lo usaría.

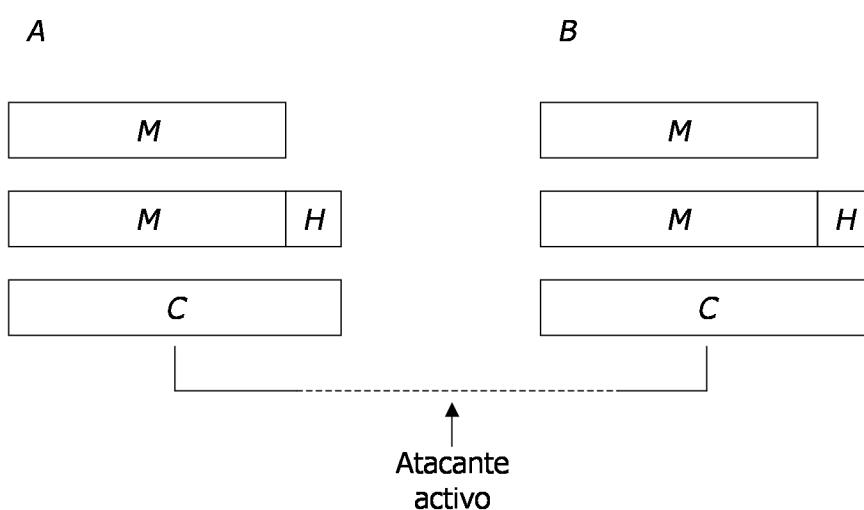


Figura 1.4: Uso de función hash para detectar mensajes encriptados inválidos

9.2 MAC (Message Authentication Codes)

Las funciones hash las hemos usado para garantizar que el *texto cifrado* no ha sido modificado en la transmisión. Los MAC van un paso más alla y nos garantizan que un *texto plano* no ha sido modificado por un atacante activo durante su transmisión. Los MAC se utilizan cuando no nos interesa ocultar el mensaje, sólo nos interesa que nadie lo pueda modificar.

Para garantizar la autenticidad, los MAC, tal como muestra la Figura 1.5, utilizan el siguiente protocolo:

1. A escribe un mensaje M .
2. A concatena al mensaje M una clave secreta K y calcula su hash H .
3. A envía el mensaje M y su hash H (pero no la clave secreta K) a B.

Cuando lo recibe B hace lo siguiente:

4. B concatena al mensaje recibido M su clave secreta K y calcula su hash H' .
5. Si el hash calculado H' coincide con el H recibido, B sabe que el mensaje no ha sido modificado.

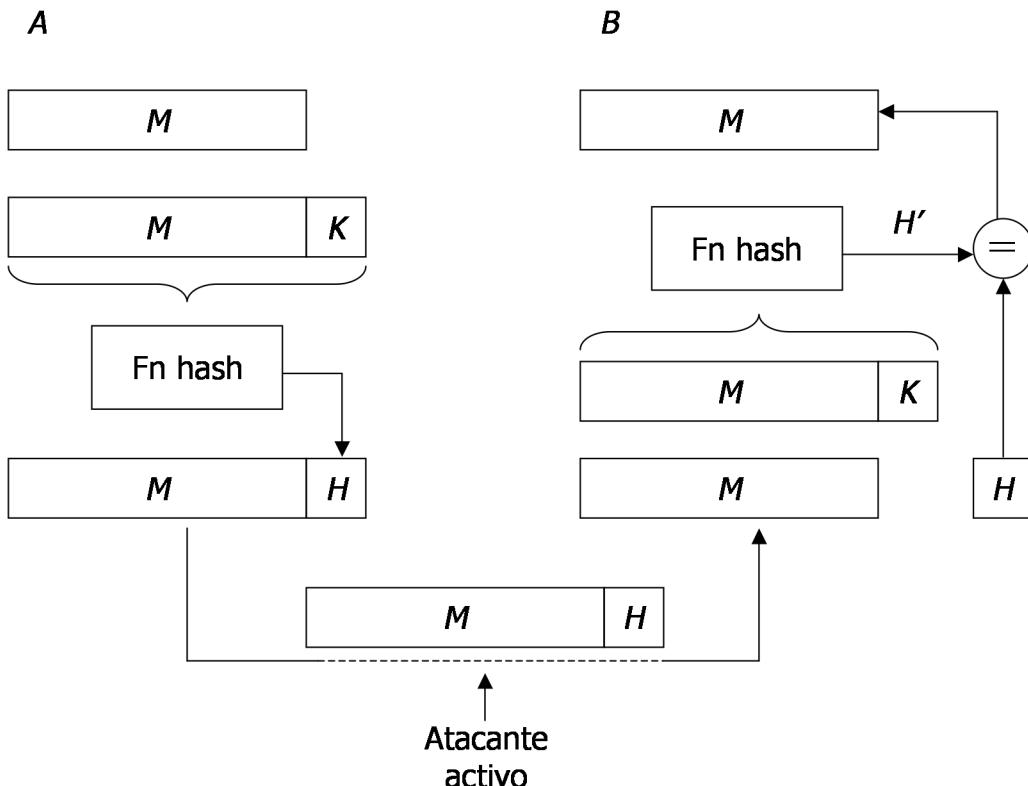


Figura 1.5: Uso de MAC para detectar texto plano inválido

Ahora, si el atacante modifica el mensaje, no puede calcular el hash del nuevo mensaje al no conocer la clave secreta.

En los MAC, para comprobar la autenticidad de un mensaje hay que conocer la clave secreta. En el Tema 4 veremos otra técnica llamada firmas digitales, en la que podemos comprobar la autenticidad de un mensaje, incluso sin necesidad de acordar una clave secreta, sino conociendo sólo la clave pública del emisor.

Tema 2

Criptografía de clave secreta

Sinopsis:

En este tema vamos a estudiar detenidamente todos los aspectos referidos a la criptografía de clave secreta.

El tema se puede dividir en dos partes principales: La primera parte que abarca los apartados 1 al 6 se estudian los aspectos teóricos fundamentales para la criptografía de clave secreta.

En la segunda parte se inicia un estudio de las librerías criptográficas de Java, y en concreto se estudian todos los aspectos relevantes de la criptografía de clave secreta con estas librerías.

1 Protocolo

Como ya comentamos en el apartado 8.3 del Tema 1, el protocolo de la criptografía de clave secreta es el siguiente:

6. A y B se ponen de acuerdo en el sistema criptográfico a utilizar.
7. A y B se ponen de acuerdo en la clave a utilizar.
8. A encripta un mensaje usando el algoritmo y clave acordados.
9. A envía el mensaje encriptado a B.
10. B desencripta el mensaje usando el algoritmo y clave acordados.

En este apartado vamos a comentar más aspectos relativos a este protocolo.

1.1 Encriptación de canales y datos

La criptografía de clave secreta se puede usar básicamente para uno de estos dos fines:

1. Encriptar canales de comunicación
2. Encriptar datos para su almacenamiento

Vamos a comentar más detalladamente cómo se realizan cada una de estas tareas.

1.1.1 Encriptación de canales de comunicación

Aunque en teoría la encriptación se puede realizar a cualquier nivel de la capa OSI, en la práctica se suele realizar en uno de estos cuatro niveles:

a) Encriptación a nivel de enlace

Podemos usar tarjetas de red que transmitan los datos encriptados tal como muestra la Figura 2.1.

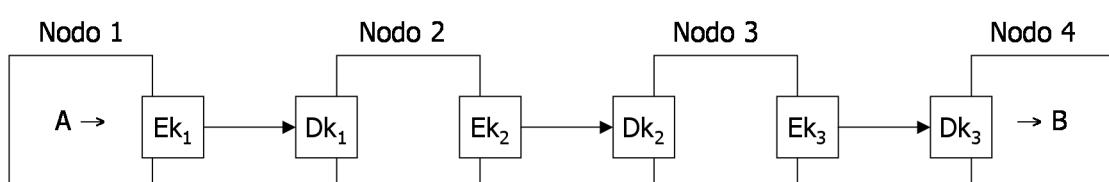


Figura 2.1 Encriptación a nivel de enlace

En este caso tendremos que utilizar una clave distinta para cada enlace. Esta configuración es posible usarla en comunicaciones por cable punto a punto,

pero en las redes de broadcast (p.e. Ethernet) tendremos que elegir entre una de estas opciones:

1. Para cada par de ordenadores de la LAN utilizar una clave distinta, de esta forma ningún otro ordenador de la LAN puede interceptar la comunicación de los demás. Sin embargo esta solución tiene el inconveniente de que el número de pares de claves aumenta rápidamente. En concreto el número de pares de claves sería $(n*(n-1))/2$.
2. Que un administrador configure la misma clave para todos los ordenadores de la red. Aunque un ordenador ajeno a la red que se enchufará a la LAN no podría entender el tráfico que circula por la red, esta solución tiene el inconveniente de que todos los ordenadores configurados para trabajar en esa red pueden desencriptar el tráfico de red de los demás usuarios legítimos.

Debido a estas restricciones, este tipo de encriptación sólo se suele usar en las comunicaciones wireless, como la del protocolo 802.11. En las redes wireless es muy fácil que cualquiera se conecte al medio, ya que al ser el medio aéreo de radiofrecuencia, no es necesario conectarse físicamente a una boca de la red.

La principal ventaja que tiene encriptar a nivel de enlace es que el criptoanalista no sólo no sabe los datos que se transmiten, sino que tampoco sabe quien está hablando con quien. A veces esta sola información puede ser útil para un criptoanalista.

b) Encriptar a nivel de red

En la red IPv4 actual no existe la encriptación a nivel de IP, pero dentro de IPv6 se ha metido una opción llamada IPsec para soportar la encriptación directamente sobre IP. Con esta opción los paquetes IP viajan encriptados desde el emisor al receptor.

El inconveniente de encriptar a nivel de red es que la información de routing (direcciones IP, puertos, etc) no se encripta. La ventaja es que la encriptación/desencriptación es transparente a las aplicaciones.

c) Virtual Private Networks

Las VPN nos permiten conectar LANs que no necesitan estar encriptadas, a través de una red insegura (p.e. Internet) donde la información se transmite encriptada.

Al canal de comunicación que se abre entre las LAN se le llama **tunneling**. Las VPN las vamos a estudiar en el Tema 5.

d) Encriptación a nivel de transporte

En este caso, al igual que en los dos anteriores, la información se encripta sólo en los extremos de la comunicación tal como muestra la Figura 2.2.

También, al igual que en los dos casos anteriores, tiene el inconveniente de que no se encripta la información de routing.

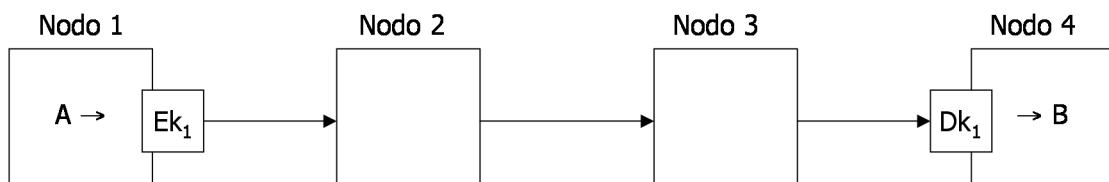


Figura 2.2 Encriptación a nivel de transporte

Un ejemplo de encriptación a nivel de transporte es SSL (Socket Secure Layer), que realiza la encriptación a nivel TCP.

e) Encriptación a nivel de aplicación

La encriptación a nivel de aplicación se realiza antes de que los datos salgan de la aplicación, y no se desencriptan hasta haber llegado a la aplicación receptora.

Un ejemplo de encriptación a este nivel es S/MIME (Secure MIME), un protocolo para envío de correo seguro. Otro ejemplo es el telnet kerberizado donde los datos se envían cifrados. Ambas aplicaciones las veremos en el Tema 5.

1.1.2 Encriptación de datos para su almacenamiento

La encriptación de datos para su almacenamiento puede modelizarse como un envío de datos desde *A* hasta *B*, pero en este caso el emisor y el receptor son los mismos.

Sin embargo, aparecen algunos problemas nuevos que no existían antes.

1. En el caso de las bases de datos encriptadas no es eficiente desencriptar toda una base de datos, ya que para acceder a un registro tendríamos que desencriptar toda la base de datos, luego la encriptación se realiza por registros individuales. Otro problema son las búsquedas, en las que habría que ir desencriptando todos los registros. Para evitarlo se utilizan índices.
2. En la encriptación de ficheros existen dos opciones: Encriptar a nivel de unidad de disco, o encriptar a nivel de fichero. La segunda opción tiene el inconveniente de que si el criptoanalista ve los nombres de los ficheros, puede obtener mucha información sin tener acceso a su contenido. Por el contrario, la primera opción tiene el inconveniente de que si se corrompe la unidad se pierden todos los datos.

1.2 Compresión, encriptación y control de errores

En la práctica es muy típico usar algoritmos de encriptación junto con algoritmos de compresión. Esto es así por dos razones:

1. Una técnica de criptoanálisis muy usada consiste en buscar redundancias en el mensaje cifrado, ya que los mensajes planos suelen tener redundancias. Si comprimimos antes de encriptar estamos eliminando estas redundancias.
2. La encriptación es un proceso costoso que se puede reducir si primero comprimimos el mensaje.

Es importante que la compresión se realice antes de la encriptación tal como muestra la Figura 2.3, o de lo contrario no podríamos aprovechar las ventajas anteriores.

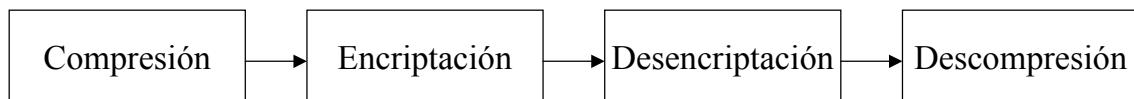


Figura 2.3 Orden correcto de compresión/enciptación

Además, los textos encriptados se comprimen muy poco, ya que uno de los objetivos que busca un algoritmo de encriptación es que el mensaje encriptado se asemeje a una serie aleatoria de bits.

Por otro lado, podemos añadir un sistema de control de errores, en cuyo caso, tal como muestra la Figura 2.4, se recomienda añadirlo después de la encriptación:

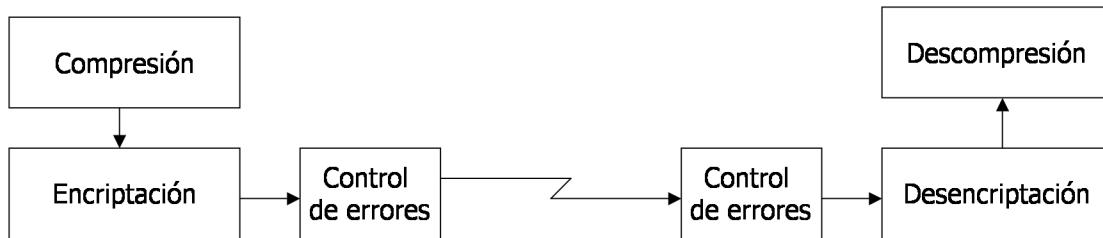


Figura 2.4 Orden correcto de control de errores con compresión/enciptación

De esta forma, si el control de errores detecta que los datos han llegado mal, no hace falta ejecutar el proceso costoso de desencriptación.

1.3 Destrucción de la información

Cuando borramos un fichero del disco, lo único que se borra es una entrada en el sistema de ficheros. Para borrar realmente los datos se necesita sobrescribirlos.

Hay herramientas como PGP Wipe que nos permiten borrar los ficheros sobrescribiéndoles previamente con basura para que sea totalmente imposible la recuperación de su contenido.

Otro problema que surge durante la destrucción de la información es la memoria virtual que usan los sistemas operativos modernos. Cuando un sistema operativo usa memoria virtual, los datos que tenemos en memoria se pueden haber pasado a disco sin que nosotros lo hayamos pedido, y otro programa puede escanear el fichero de swap en busca de estos datos.

En el diseño de las API de Java existen algunas medidas destinadas a paliar estos efectos que vamos a comentar:

En Java, cuando creamos una variable, esta siempre tiene que estar inicializada (si declaramos una variable sin asignarla nada la inicializa a cero), con lo que otro trozo del programa Java no puede escanear la memoria que reserva en busca de datos, como podría hacerse en otros lenguajes más convencionales como C o C++.

Sin embargo, aunque un programa Java no puede espiar a otro programa Java, si que un programa C puede espiar a un programa Java, ya que Java cuando deja de utilizar una variable, no la borra de memoria, con lo que luego un programa C podría escanear la memoria en busca de esa información.

Para evitar este riesgo se recomienda sobrescribir manualmente la variable donde hemos guardado datos confidenciales antes de eliminarla.

Por ejemplo:

```
public class ClaveSecreta
{
    private byte[] clave;
    public void clear()
    {
        for (int i=0;i<clave.length;i++)
            clave[i] = 0;
    }
    // Otros métodos
    .....
}
```

Un problema mayor le encontramos en el uso de clases inmutables como `String` o `BigInteger`, que no se van a poder sobrescribir. Para evitar este problema Java no usa la clase `String` para guardar claves, sino que usa siempre arrays de bytes.

2 DES (Data Encryption Standard)

2.1 Orígenes de DES

DES es un algoritmo de encriptación inventado inicialmente por IBM en 1970 bajo el nombre "Lucifer".

En 1977 fue adoptado por el NIST (National Institute of Standards and Technology) y por la NSA (National Security Agency) como su estándar nacional. El algoritmo está publicado en [DES].

En 1981 este algoritmo fue adoptado también por ANSI.

2.2 Algoritmo

DES es un cifrador de bloque con tamaño de bloque de 64 bits. Para cada bloque de 64 bits de entrada se obtienen 64 bits a la salida, es decir, el tamaño del fichero resultante no crece ni decrece.

La clave es de 56 bits. Se suele expresar como un número de 64 bits, pero el último bit de cada byte es de paridad, y se quita antes de aplicar el algoritmo.

2.2.1 Visión general del algoritmo

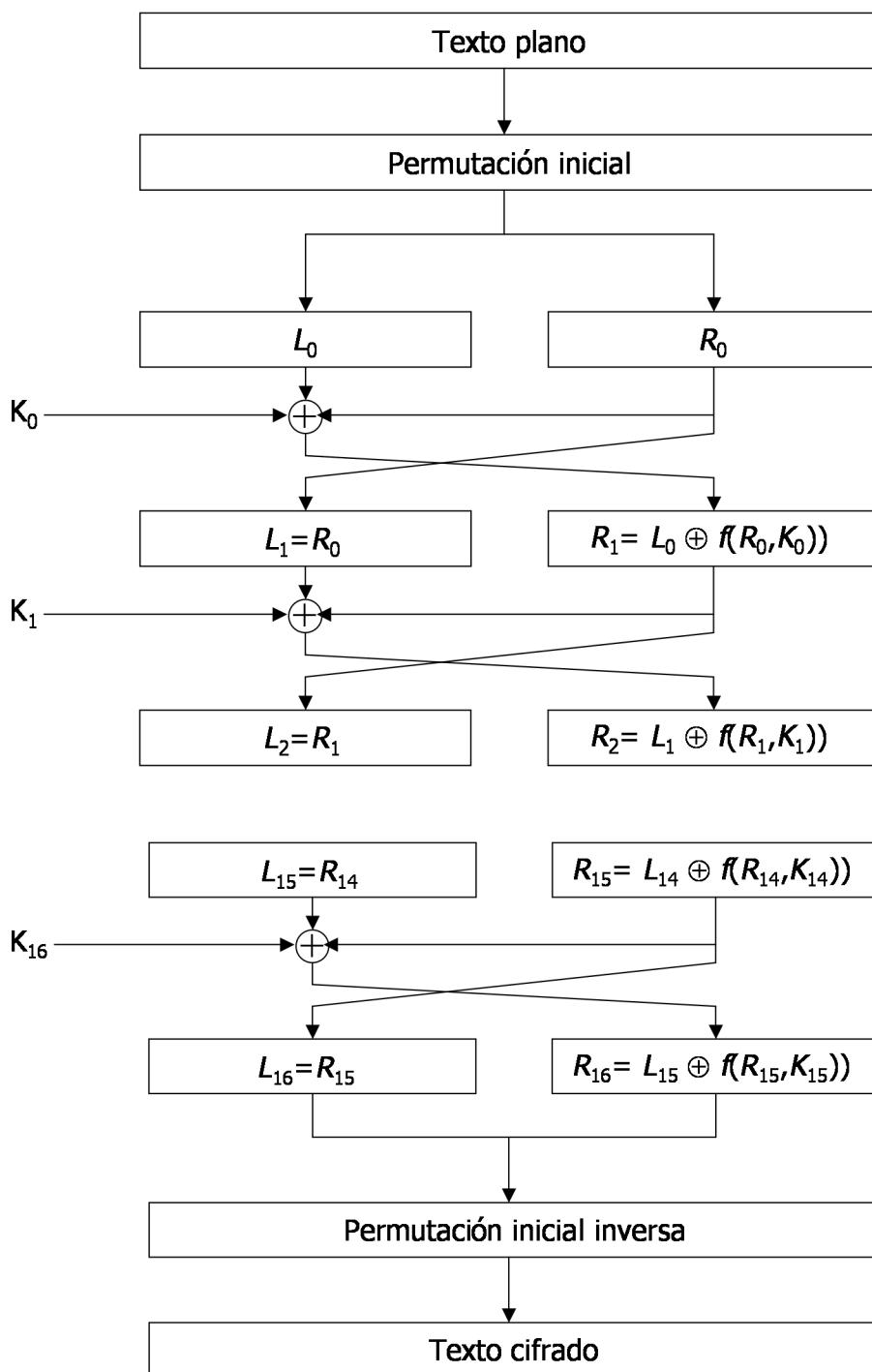
En esta primera sección vamos a empezar describiendo las partes principales de las que consta el algoritmo, para luego comentar más detalladamente cada una de ellas.

El algoritmo se limita a aplicar sustituciones (cambios de unos valores por otros) y permutaciones (cambios en la posición que ocupan los bits).

Tal como muestra la Figura 2.5, el proceso de encriptación se realiza en 16 **rounds**, y cada round lleva a cabo la misma serie de sustituciones y permutaciones.

En el apartado 7 del Tema 1 se indicó que las operaciones de permutación y sustitución actualmente no son seguras, y de hecho la seguridad de DES se debe a que en cada round se va introduciendo una clave K_i .

Después de la permutación inicial, el bloque se parte en dos bloques de 32 bits (L_0 y R_0), se aplican los 16 rounds, y al final se vuelven a juntar las mitades y se aplica una última permutación que es la inversa de la permutación inicial.

**Figura 2.5:** Algoritmo general de DES

En cada round (ver Figura 2.6) se cogen 48 bits (permutación compresiva) de los 56 bits de la clave que se combinan con XOR con el registro de la derecha previamente expandido de 32 a 48 bits (permutación expansiva).

A los 48 bits resultantes se les pasa por las S-Box y por las P-Box, para finalmente hacerles un XOR con los bits del registro de la izquierda.

En cada round se intercambian los bits de los registros de la derecha y la izquierda.

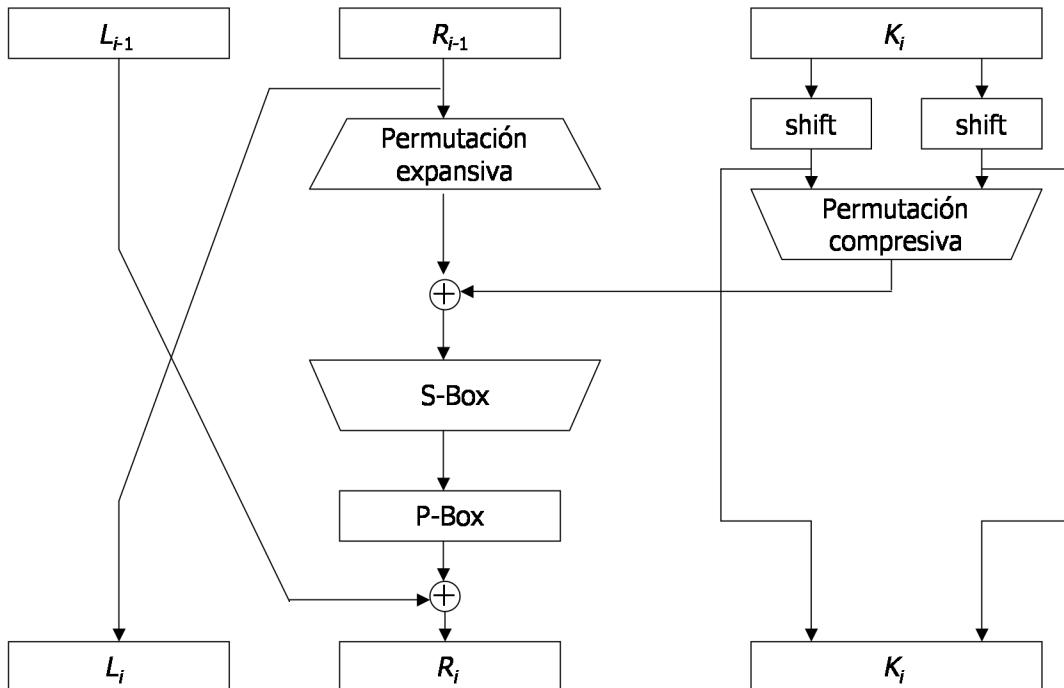


Figura 2.6: Un round de DES

2.2.2 La permutación inicial

Antes de empezar los rounds se realiza la llamada permutación inicial (ver Figura 2.5). Esta, permuta los bits tal como muestra la Tabla 2.1. Es decir, el bit 1 del texto plano se pone en la posición 58, el bit 2 se pone en la segunda posición 50, y así sucesivamente.

Desde	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hasta	58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4

Desde	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Hasta	62	54	46	28	30	22	14	6	64	56	48	40	32	24	26	8

Desde	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
Hasta	57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3

Desde	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
Hasta	61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Tabla 2.1: Permutación inicial de un bloque DES

Las permutaciones inicial y final no añaden seguridad a DES. Su propósito era cargar los datos en un registro hardware de 64 bits. Esto se hace así porque en los tiempos de DES los sistemas de encriptación solían ser hardware con lo que las permutaciones eran triviales. En las implementaciones software, más comunes en la actualidad, esto sólo enlentece el proceso.

2.2.3 La transformación de claves

Como hemos dicho, DES tiene una clave de 64 bits, de los cuales sólo se usan 56 bits, ya que el octavo bit de cada byte se usa para paridad, y se ignora por parte del algoritmo.

El proceso de transformación de claves es el siguiente: Inicialmente se cargan los 56 bits de la clave también permutados (al igual que hemos hecho con el bloque) de acuerdo a la Tabla 2.2.

Desde	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hasta	57	49	41	33	25	17	9	1	58	50	42	34	26	18	10	2

Desde	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Hasta	59	51	43	35	27	16	11	3	60	52	44	36	63	55	47	39

Desde	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
Hasta	31	23	15	7	62	54	46	38	30	22	14	6	61	53	45	37

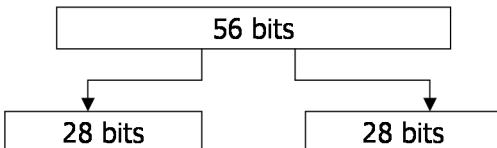
Desde	49	50	51	52	53	54	55	56								
Hasta	29	21	13	5	28	20	12	4								

Tabla 2.2: Permutación inicial de la clave DES

Después de colocar los 56 bits en este orden, se va a coger una subclave de 48 bits para cada uno de los 16 rounds de DES. A estas subclaves se las llama K_i y se obtienen de la siguiente manera:

1. Se dividen los 56 bits en dos mitades de 28 bits, es decir, se usan registros de 28 bits como muestra la Figura 2.7.
2. Los registros de 28 bits se desplazan a la izquierda 1 ó 2 posiciones en cada round, de acuerdo a la Tabla 2.3.
3. En cada round de los 56 bits se eligen 48 según la Tabla 2.4. P.e. el bit 45 de los registros de desplazamiento se obtiene en la posición 34 de la salida, y el bit 18 de los registros de desplazamiento es ignorado. A esta operación se la llama permutación compresiva, porque no sólo permutamos, sino que de los 56 bits elegimos sólo 48 bits.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Desplazamiento	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Tabla 2.3: Desplazamiento de bits de los registros de la clave de un round**Figura 2.7** División de la clave en bloques de 28 bits

Entrada	14	17	11	24	1	5	3	28	25	6	21	10	23	19	12	4
Salida	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Entrada	26	8	16	7	27	20	13	2	41	52	31	37	47	55	30	40
Salida	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Entrada	51	45	33	48	44	49	39	56	34	53	46	42	50	36	29	32
Salida	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48

Tabla 2.4: Permutación compresiva de la clave

Obsérvese que debido a los desplazamientos, en cada round se elige para formar la subclave un subconjunto distinto de posiciones de la clave, y cada bit se usa aproximadamente 14 de las 16 veces. Pero no todos los bits se usan exactamente el mismo número de veces.

2.2.4 La permutación expansiva de los datos

Esta operación expande el lado derecho de los datos de 32 bits a 48 bits. El objetivo es conseguir que el lado derecho tenga el mismo tamaño que la clave para poder aplicar la operación XOR. En la Tabla 2.5 se muestra cómo se debe realizar esta expansión.

Entrada	32	1	2	3	4	5	4	5	6	7	8	9	8	9	10	11
Salida	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Entrada	12	13	12	13	14	15	16	17	16	17	18	19	20	21	20	21
Salida	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Entrada	22	23	24	25	24	25	26	27	28	29	28	29	30	31	32	1
Salida	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48

Tabla 2.5: Permutación expansiva de los datos

Por ejemplo, el bit de la posición 3 de entrada se mueve a la posición 4 de salida, y el bit de la posición 21 de entrada se mueve a las posiciones 30 y 32 de salida.

Obsérvese que aunque la salida es mayor a la entrada, cada bloque de entrada genera un bloque de salida distinto.

2.2.5 La sustitución en las S-Box

Los 48 bits resultantes del XOR entran en las S-Box (Sustitution Box) para realizar sobre ellos una operación de sustitución.

Existen 8 S-Box, cada una con 6 bits de entrada y 4 de salida, tal como muestra la Figura 2.8, con lo que a la salida se obtienen 32 bits, es decir, se produce ahora una compresión.

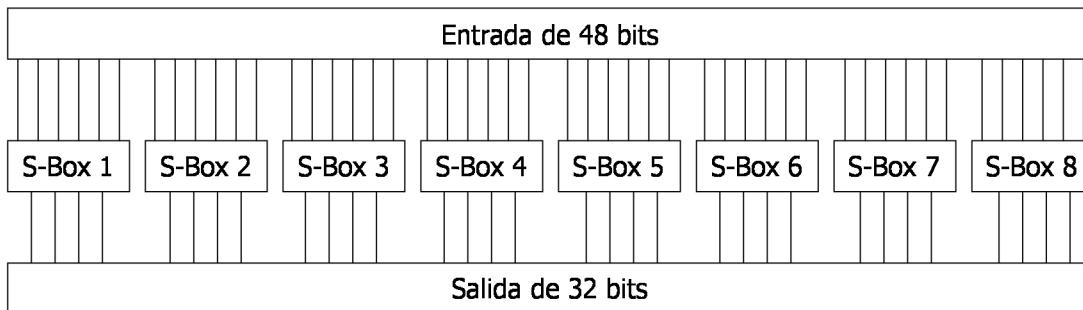


Figura 2.8: Estructura de las S-Box

Cada S-Box opera tal como muestra la Tabla 2.6.

S-Box 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-Box 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S-Box 3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
3	1	13	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S-Box 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S-Box 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-Box 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S-Box 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S-Box 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Tabla 2.6: Funcionamiento de las S-Boxes

De todos los números que hay en la tabla de cada S-Box, sólo se elige un número de salida en función de los valores de entrada. Para elegir cual de ellos se pone en la salida se sigue el siguiente criterio:

Supongamos que los bits de entrada son $b_0, b_1, b_2, b_3, b_4, b_5$. Los bits b_0 y b_5 (los extremos) se usan para formar un número de 2 bits que indica la fila a elegir (de 0 a 3) de la tabla correspondiente. Los otros cuatro bits se usan para formar un número de 4 bits (del 0 al 15) para indicar la columna de la tabla correspondiente. Dada la fila y la columna, y tenemos el valor de salida

Por ejemplo, si en la sexta S-Box entran los valores:

$$\begin{array}{cccccc} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \\ \hline 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

Para elegir la fila tenemos:

$$\begin{array}{cc} b_0 & b_5 \\ \hline 1 & 1 \end{array} = 3$$

Y para elegir la columna tenemos:

$$\begin{array}{cccc} b_1 & b_2 & b_3 & b_4 \\ \hline 1 & 0 & 0 & 1 \end{array} = 9$$

Si ahora miramos en la tabla, en la fila=3, columna=9 tenemos el valor de salida 14.

La sustitución con las S-Box es la operación más importante de DES, ya que las demás operaciones son lineales, y por lo tanto fáciles de criptoanalizar.

Los 32 bits de la salida se forman volviendo a juntar los 4 bits de salida de cada una de las 8 S-Box.

2.2.6 La permutación con las P-Box

Los 32 bits del paso anterior se vuelven a permutar pasándolos por la llamada P-Box (Permutation Box), de acuerdo a la permutación de la Tabla 2.7.

Desde	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hasta	16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10

Desde	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Hasta	2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Tabla 2.7: Permutación realizada por la P-Box

2.2.7 Vuelta a repetir otro round

Finalmente a la salida de las P-Box se le hace un XOR con los 32 bits del registro de la izquierda, se intercambian los contenidos de los registros de la izquierda y la derecha (ver Figura 2.6), y comienza un nuevo round.

2.3 Desencriptación de DES

Después de todas las extrañas transformaciones que hemos visto que se hacen durante la encriptación, podemos pensar que la desencriptación implica otro montón de operaciones aún mayor. Sin embargo, las operaciones están elegidas para que el mismo algoritmo sirva para la encriptación y la desencriptación, es decir, podemos usar la misma función para encriptar y para desencriptar.

La única diferencia es que la clave debe usarse en orden inverso, es decir, si las claves usadas en la encriptación fueron: $k_1, k_2, k_3, \dots, k_{16}$ en la desencriptación usaremos $k_{16}, k_{15}, k_{14}, \dots, k_1$.

El algoritmo que se usa para generar las claves también es circular, con lo que podemos calcular las claves al revés realizando desplazamientos hacia la derecha en vez de hacia la izquierda tal como muestra la Tabla 2.8.

Encriptación

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Desplazamiento	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Desencriptación

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Desplazamiento	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1	1

Tabla 2.8: Desplazamientos hacia la derecha en la desencriptación

2.4 Situación actual de DES

El punto débil de DES no está en el algoritmo, sino en el tamaño de la clave, que es sólo de 56 bits, lo cual da lugar a demasiadas pocas claves posibles y es susceptible de sufrir un ataque por fuerza bruta. Actualmente un algoritmo de clave secreta se considera fuerte si tiene una clave de 128 bits o mayor.

Esto ha dado lugar a que desde hace años estén surgiendo sistemas que desencriptan DES en tiempo cada vez menor. Para atacar DES se han usado básicamente dos aproximaciones:

1. Sistemas distribuidos
2. Máquinas especiales diseñadas por la NSA que tardan menos de 1 hora en desencriptar DES

Para acabar con este problema han surgido varias variantes de DES que sí que son seguras.

La más conocida de estas es **TripleDES** (también llamado DESede), que no es más que DES aplicado tres veces con tres claves distintas. La primera vez en modo encriptación con la primera clave, la segunda en modo de desencriptación con la segunda clave, y la tercera en modo de encriptación con la tercera clave. La Figura 2.9 muestra gráficamente este proceso.

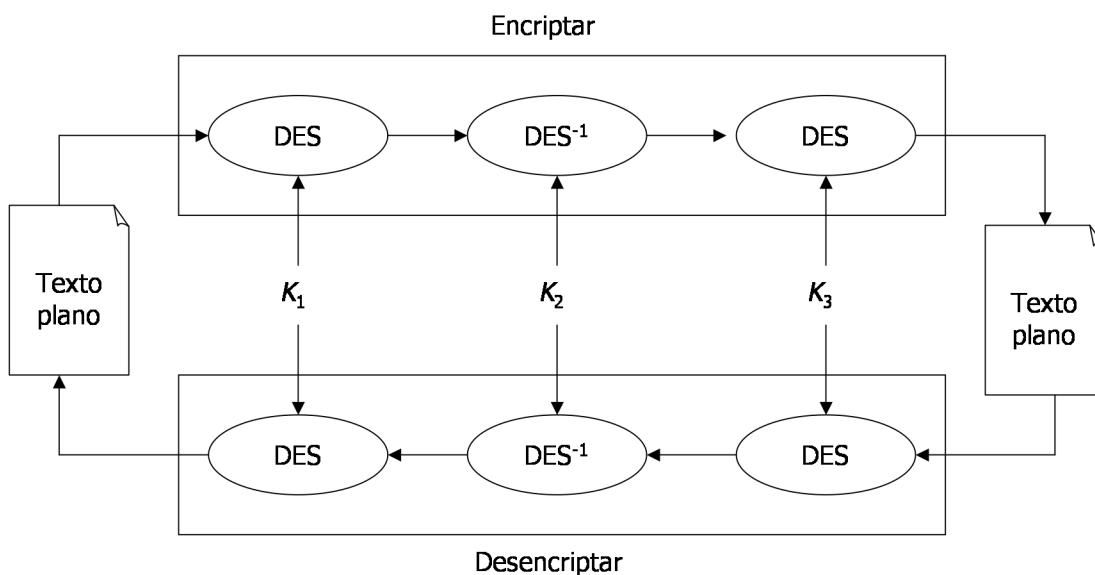


Figura 2.9: Proceso de encriptación y de desencriptación con TripleDES

El nombre DESede viene justo de la forma en que se aplica el algoritmo, (Encryption, Decryption, Encryption).

En TripeDES el tamaño de la clave es de 168 bits (3*56 bits) que es suficientemente segura como para eludir un ataque por fuerza bruta actual.

Existe otra variante que usa claves de 112 bits, y donde la primera clave se usa la primera y tercera vez.

También han surgido otras variantes de DES como son DESX, RDES o Generalized DES.

3 Otros algoritmos de clave secreta

3.1 Algoritmos de clave secreta más conocidos

En esta sección vamos a comentar cuáles son los algoritmos de clave secreta más conocidos:

Blowfish. Es un algoritmo inventado por Bruce Schneier en 1993. No está patentado, sino que es de dominio público. Respecto a sus principales características, destaca que permite usar una clave variable de hasta 448 bits, y que está optimizado para microprocesadores de 32 bits y de 64 bits.

IDEA (International Data Encryption Algorithm) inventado en Zurich (Suiza) en 1990. Usa una clave de 128 bits, y es el que usa PGP para encriptar. Se supone que es seguro, pero por desgracia ha sido patentado por una empresa Suiza.

RC2 (Rivest's Code). Fue desarrollado por Ronald Rivest de la empresa RSA Data Security, y el algoritmo es propiedad intelectual de la empresa RSA Data Security, la cual lo vende junto con su implementación. Por desgracia fue revelado en un envío anónimo a Usenet. En la versión de exportación era vendido con un tamaño de clave de 40 bits que lo hace inseguro.

RC4. Una nueva implementación que también fue desvelada en un envío anónimo a Usenet. También su versión de exportación fue vendida con un tamaño de clave de 40 bits.

RC5. Otra nueva implementación que actualmente vende RSA Data Security, y que todavía no ha sido desvelada. Su principal característica es que permite usar claves de longitud variable.

AES (Advanced Encryption Standard), es el algoritmo elegido en Octubre del 2000 por el NIST como sustituto a DES. Utiliza un algoritmo llamado Rijndael, claves de 128, 192 y 256 bits, y tamaños de bloque de 128, 192 y 256 bytes respectivamente.

3.2 Restricciones de exportación de los EEUU

Hasta el año 2003, por razones de seguridad militar el gobierno de los EEUU no dejaba exportar software de encriptación seguro, sólo dejaba exportar software con DES y algoritmos de 40 bits como RC2 y RC4.

Esto dio lugar a que el software de encriptación se empezara a desarrollar en otros países. Actualmente se ha quitado la restricción excepto para exportar a algunos países como Irán, Irak o Afganistán.

4 Modos del algoritmo

Como hemos introducido en el apartado 3.1 del Tema 1, existen dos **tipos** de algoritmos de encriptación: cifradores en bloque y cifradores en flujo.

Otra clasificación de los algoritmos de cifrado es en función del modo. El **modo** nos indica cómo se mezcla el texto plano con la clave, y opcionalmente con el texto plano o cifrado anterior, para ir formando el texto cifrado. Como vamos a ver ahora, usar un texto plano (o cifrado) anterior dificulta la capacidad del atacante de desencriptar un texto cifrado aunque conozca partes del texto plano.

4.1 Modos en los cifradores de bloque

4.1.1 ECB. Electronic Code Book mode

Esta es la forma más sencilla de hacer un cifrador de bloque: Cada bloque de texto plano encipta en un bloque de texto cifrado.

En este modo, como el mismo texto plano siempre da lugar al mismo texto cifrado, es posible hacer un **code book**, es decir, una tabla que nos diga para cada texto plano cuál es su correspondiente texto cifrado.

En principio este code book sería lo suficientemente grande como para no poder implementarse un ataque con esta técnica. Por ejemplo, para bloques de 64 bits tendríamos un code book de 2^{64} bits, para cada clave. Si la clave tiene un tamaño de K bits, el code book tendría un tamaño de $2^{64} \times 2^K = 2^{(64+K)}$ bits.

Como muchas veces se usa la misma clave para encriptar una transmisión, en realidad necesitamos un code book de 2^{64} bits solamente, que todavía es mucho más grande que un disco duro actual.

Aun así, como los mensajes se suelen repetir, al criptoanalista le bastaría con crear una base de datos con parte del code book. Además muchos mensajes suelen tener cabeceras comunes que son fáciles de detectar por el criptoanalista.

Por otro lado, una ventaja del modo ECB es que, al no depender un bloque de los demás bloques, podemos encriptar o desencriptar partes aisladas de un fichero, lo cual es muy útil para aplicaciones como las bases de datos encriptadas, en las que vamos a poder acceder a sus registros aleatoriamente.

Otra ventaja es que el proceso de encriptación se puede paralelizar.

Una tercera ventaja está en que, si hay un error en la comunicación, sólo perdemos el bloque afectado por el error, ya que los bloques son independientes unos de otros. Sin embargo, si accidentalmente se añade o pierde un bit, se estropea todo el mensaje a partir de ese bit.

Block replay

El problema más serio con el que se encuentra ECB es que el atacante puede enviar mensajes sin conocer la clave, que es a lo que se llama un **ataque por block replay** (o también **playback attack**).

Por ejemplo, supongamos que una transferencia bancaria consta de una serie de campos de los que conocemos su correspondiente tamaño, tal como muestra la Tabla 2.9. En este caso, el atacante puede interceptar transferencias de las que sabe el texto plano y su correspondiente texto cifrado, y luego puede:

1. Retransmitir paquetes con transferencias a su favor
2. Cambiar el bloque que corresponde al destinatario

El primer ataque se puede evitar poniendo un timestamp o número de serie al mensaje, pero el segundo no.

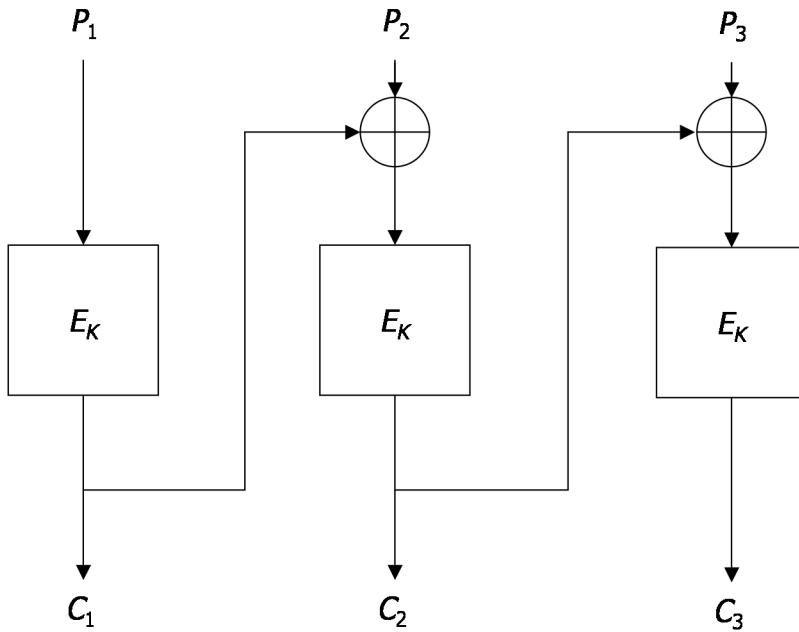
Campo	Tamaño en bloques
Banco emisor	1,5 bloques
Banco receptor	1,5 bloques
Nombre receptor	6 bloques
Cuenta receptor	2 bloques
Importe	1 bloque

Tabla 2.9: Campos de un mensaje de transferencia bancaria

4.1.2 CBC. Cipher Block Chaining mode

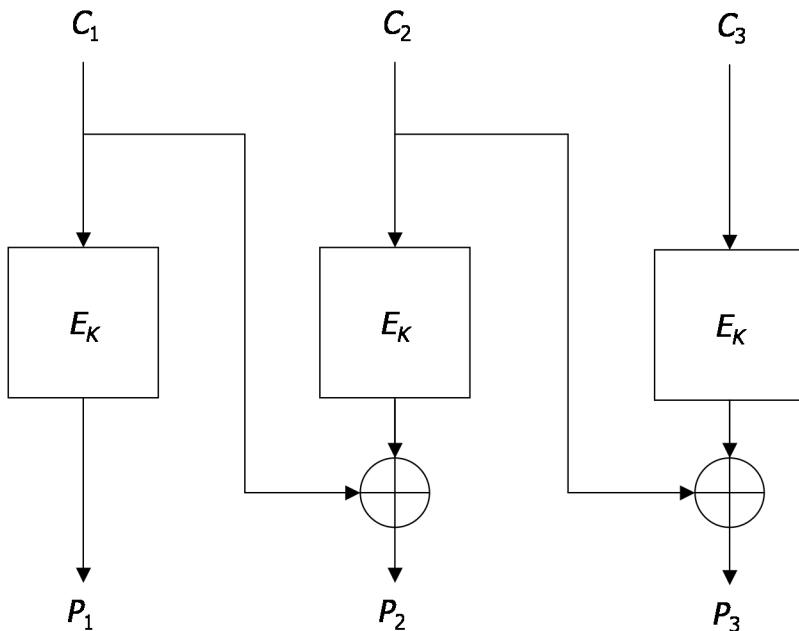
Para solucionar el problema del modo anterior, este modo lo que hace es que cada vez que encriptamos un bloque, el texto cifrado no sólo depende de la clave y del algoritmo utilizado, sino que también depende de los anteriores bloques que hemos encriptado.

Para ello hacemos un XOR al texto plano del bloque actual con el anterior bloque cifrado tal como muestra la Figura 2.10.

**Figura 2.10:** Proceso de encriptación en CBC

Obsérvese que el primer bloque se encripta normalmente, pero al segundo se le hace un XOR con el primer bloque cifrado antes de encriptarlo.

Ahora, para desencriptar, el primer bloque se desencripta normalmente, pero al segundo después de desencriptarlo se le hace un XOR con el primer bloque cifrado anterior tal como muestra la Figura 2.11.

**Figura 2.11:** Proceso de desencriptación en CBC

El orden en que se realizan los XOR con el bloque cifrado anterior es importante para que al encriptar y desencriptar se obtengan los mismos resultados. La Figura 2.12 pretende representar el orden en que se realizan estas operaciones.

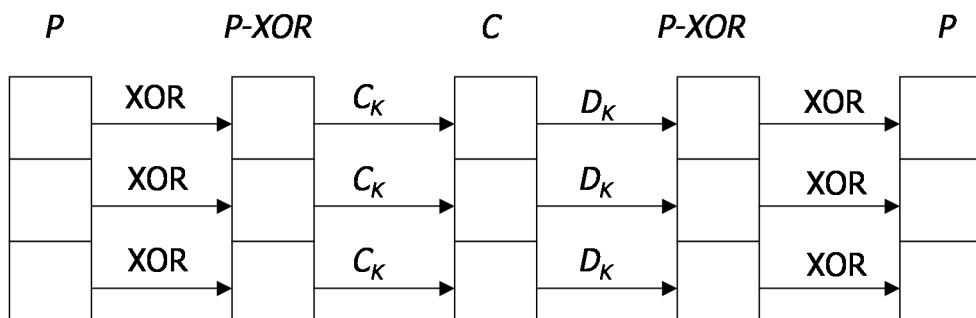


Figura 2.12: Orden correcto de aplicación de los XOR en CBC

En cada fila de la Figura 2.12 el bloque cifrado anterior con el que hacemos el XOR es el mismo. En la encriptación los XOR con el bloque cifrado anterior se aplican antes de encriptar, y en la desencriptación después de encriptar.

Es decir:

$$\begin{aligned} C_i &= E(K, P_i \oplus C_{i-1}) \\ P_i &= C_{i-1} \oplus D(K, C_i) \end{aligned}$$

El vector de inicialización

Ahora, usando el modo CBC conseguimos que bloques de texto plano idénticos den lugar a diferentes bloques cifrados, con la única condición de que los bloques anteriores sean distintos. Pero todavía si encriptamos dos mensajes idénticos obtenemos el mismo mensaje cifrado.

También si dos mensajes comienzan igual dan lugar al mismo texto cifrado hasta que haya la primera diferencia en el mensaje, con lo que las cabeceras de los mensajes (que suelen ser siempre iguales) pueden dar al criptoanalista información útil.

Para prevenir esto vamos a usar un **vector de inicialización (IV Initialization Vector)**, que es un texto aleatorio que se pone en el primer bloque. Ahora, cada vez que encriptemos, idénticos textos planos darán lugar a textos cifrados completamente distintos.

Conviene hacer un par de apreciaciones: La primera es que el IV no tiene que ser secreto, ya que el conocimiento del IV no le sirve al atacante, ni para desencriptar el mensaje, ni para cambiarlo. La segunda es que el IV debe ser único en cada mensaje, o el atacante puede usar la técnica de block replay para mensajes con el mismo IV.

Propagación de errores

Para ver cómo afectan los errores de transmisión a un mensaje cifrado con CBC, vamos a estudiar tres situaciones:

1. Si se produce un error en un bit del texto plano
2. Si se produce un error en un bit del texto cifrado
3. Si se añade o pierde un bit del texto cifrado

1. Se produce un error en un bit del texto plano

En principio, un error en un bit del texto plano afecta tanto a la encriptación del bloque donde está el bit, como a todos los siguientes, pero curiosamente la desencriptación deshace este efecto, y cuando desencriptamos obtendremos el texto plano con un solo error en el bit afectado. Esto se debe a que realmente lo que estamos haciendo es encriptar el texto plano del bit defectuoso y después desencriptarlo.

2. Se produce un error en un bit del texto cifrado

Más problemático es si se produce un error en un bit del texto cifrado (p.e. se modifica un bit de un fichero encriptado que estamos transmitiendo). En este caso el error en un bit de un bloque cifrado hace que el descifrado de ese bloque entero sea basura, y que el siguiente bloque se vea afectado sólo en el bit que está en la misma posición. Pero CBC se recupera a partir del tercer bloque.

Para analizar este efecto con más detalle vamos a hacer un ejemplo. Para simplificar supongamos que el proceso de encriptación consiste en sumar 1 al bloque, y el proceso de desencriptación en restar 1 al bloque, y queremos encriptar el texto plano:

Texto plano

1	0	0	1
0	1	1	0
0	1	0	1
0	1	1	1

Al cifrar el primer bloque obtendríamos:

1 0 0 1 → cifrado → 1 0 1 0

Al cifrar el segundo bloque, tenemos que hacer un XOR con el primero y obtendríamos:

1 0 1 0 **0 1 1 0** → XOR → **1 1 0 0** → **cifrado** → **1 1 0 1**

Al cifrar el tercer bloque:

1 1 0 1 **0 1 0 1** → XOR → **1 0 0 0** → **cifrado** → **1 0 0 1**

Al cifrar el cuarto bloque:

1 0 0 1 **0 1 1 1** → XOR → **1 1 1 0** → **cifrado** → **1 1 1 1**

Luego el texto cifrado sin errores sería:

Texto cifrado
1 0 1 0
1 1 0 1
1 0 0 1
1 1 1 1

Ahora supongamos que se produce un error en un bit del segundo bloque:

Texto cifrado
con error
1 0 1 0
1 0 0 1
1 0 0 1
1 1 1 1

Al descifrar el texto cifrado con error:

Primer bloque cifrado:

1 0 1 0 → descifrado → **1 0 0 1**

Segundo bloque cifrado:

1 0 0 1 → descifrado → **1 0 0 0** **1 0 1 0** → XOR → **0 0 1 0**

Como vemos hemos obtenido basura en todo el segundo bloque:

Tercer bloque cifrado:

$$\begin{array}{r} 1 \ 0 \ 0 \ 1 \rightarrow \text{descifrado} \rightarrow 1 \ 0 \ 0 \ 0 \\ 1 \ 0 \ 0 \ 1 \end{array} \quad \boxed{\begin{array}{l} \\ \nearrow \\ \end{array}} \rightarrow \text{XOR} \rightarrow 0 \ 0 \ 0 \ 1$$

Observe que tras desencriptar el tercer bloque, sólo se modifica el bit afectado en el bloque cifrado anterior. Esto se debe a que nuestro bloque hace un XOR con el bloque cifrado anterior. Lo malo hubiera sido si hubiéramos hecho un XOR con el bloque descifrado anterior ya que el resultado de descifrar un bloque cifrado modificado no resulta sólo en la modificación de un bit, sino que puede llegar a modificar todos los bits.

Cuarto bloque cifrado:

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \rightarrow \text{descifrado} \rightarrow 1 \ 1 \ 1 \ 0 \\ 1 \ 0 \ 0 \ 1 \end{array} \quad \boxed{\begin{array}{l} \\ \nearrow \\ \end{array}} \rightarrow \text{XOR} \rightarrow 0 \ 1 \ 1 \ 1$$

A partir de aquí se descifrarán correctamente el resto de bloques, ya que para desencriptar no se tiene en cuenta para nada el bloque defectuoso.

Obsérvese que peor hubiera sido si dependiéramos de todos los bloques descifrados anteriores, en cuyo caso arrastraríamos el error para siempre.

3. Se añade o pierde un bit del texto cifrado

Mientras que CBC se recupera rápidamente de un error en un bit, no se recupera en absoluto de un error de sincronismo, sino que a partir del error se produce basura indefinidamente.

4.2 Modos en los cifradores de flujo

Los **cifradores de flujo** se caracterizan por transformar el texto plano en texto cifrado bit a bit. Vamos a ver ahora en qué modos se pueden implementar estos cifradores.

Para realizar el cifrado vamos a necesitar un **keystream generator** (también llamado running-key generator), que no es más que un generador de bits pseudoaleatorios k_1, k_2, \dots, k_n , a los cuales se les hace luego un XOR con el texto plano p_1, p_2, \dots, p_n para obtener el texto cifrado c_1, c_2, \dots, c_n . Es decir¹:

$$c_i = k_i \oplus p_i$$

¹ Vamos a usar letras mayúsculas para referirnos a los bloques de un cifrador de bloque, y letras minúsculas para referirnos a los bits de un cifrador de flujo.

Para desencriptar en el receptor volvemos a usar el mismo keystream tal como muestra la Figura 2.13:

$$p_i = c_i \oplus k_i$$

Ya que:

$$c_i \oplus k_i = k_i \oplus p_i \oplus k_i = p_i$$

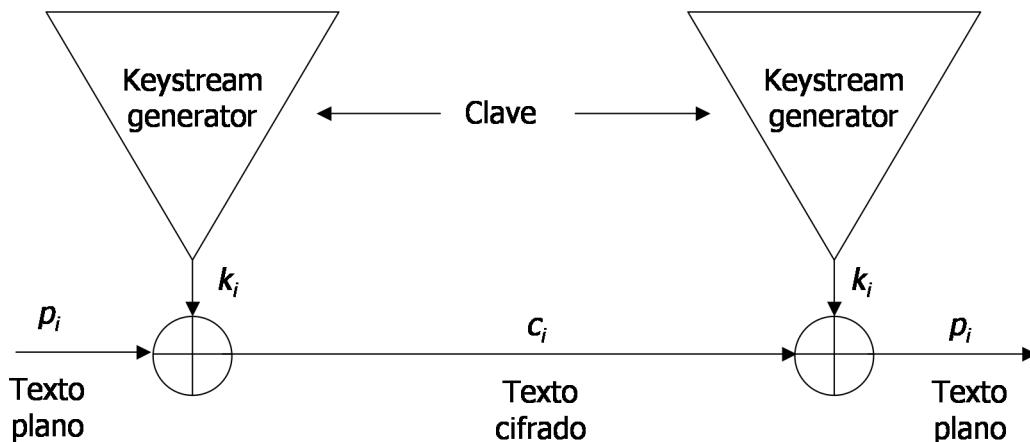


Figura 2.13: Cifrado y descifrado con un keystream generator

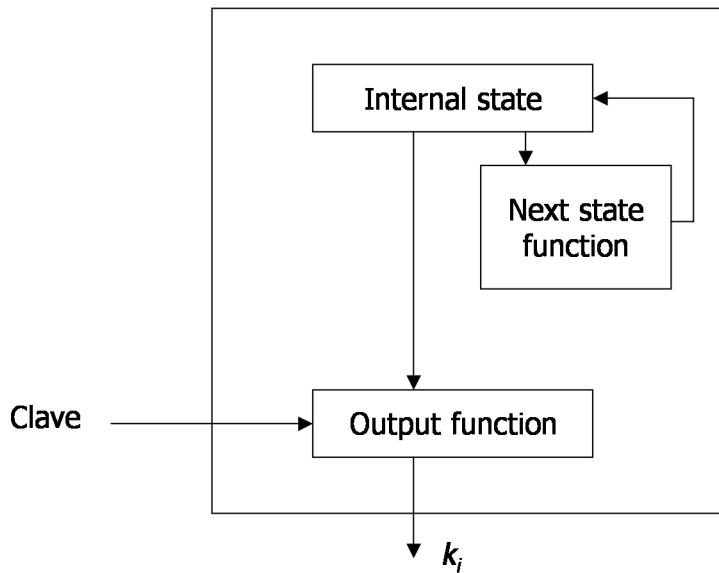
Obsérvese que la seguridad del sistema depende íntegramente de la seguridad del keystream. Si alguien descubre la forma de obtener la serie pseudoaleatoria del keystream, rompería el algoritmo.

Si el keystream usa un algoritmo One-Time Pad, sería irrompible, pero en la práctica el keystream se basa en una clave (y opcionalmente el texto anteriormente cifrado) para generar la serie pseudoaleatoria.

Tal como pretende indicar la Figura 2.14, un keystream generator tiene tres partes básicas:

1. **Internal state.** Un registro que almacena el estado actual del keystream generator.
2. **Next state function.** Función que genera un nuevo estado a partir del estado actual.
3. **Output function.** Función que a partir del estado interno y de la clave genera el keystream de salida.

No hay nada de complejo en el estado interno ni en la next state function (que muchas veces es un mero contador), sino que toda la complejidad para el criptoanálisis recae en la output function, y en concreto en el hecho de adivinar la clave.

**Figura 2.14:** Componentes de un keystream generator

La output function con el mismo estado interno y la misma clave genera siempre el mismo keystream.

Esta apreciación hace que un punto débil de los cifradores de flujo sea que si el keystream generator produce la misma serie de bits cada vez que arrancamos el cifrador se vuelve trivial romper el sistema criptográfico: Para ello simplemente tenemos que disponer de un texto cifrado y su correspondiente texto plano (criptoanálisis de texto plano no elegido, como lo llamamos en el apartado 4 del Tema 1), y entonces podemos sacar el keystream haciendo a ambos un XOR, es decir:

$$k_i = p_i \oplus c_i$$

Y después usamos este keystream para descifrar otros mensajes que se transmitan.

De hecho, existe otro posible ataque en el que el atacante no tiene que conocer un texto plano y su correspondiente texto cifrado (lo que llamábamos criptoanálisis de texto cifrado). En este caso el criptoanalista hace un XOR a dos mensajes cifrados con el mismo keystream (dos mensajes procedentes del mismo arranque de máquina) para obtener dos textos planos hechos un XOR (los keystream se anulan al ser los mismos en ambos mensajes). Ahora es fácil romper la seguridad porque partes de uno de los mensajes son fáciles de intuir (p.e. cabeceras), y sólo tiene que probar hasta encontrar que el otro mensaje también tiene sentido. Después el atacante puede obtener el keystream haciendo un XOR a un texto cifrado con su correspondiente texto plano, y a partir de ese momento puede desencriptar todos los mensajes que se encripten con este keystream.

Por esta razón los stream ciphers se suelen usar en sistemas que no hay que reiniciar, como pueda ser una línea de comunicaciones T-1.

Lógicamente, una solución sencilla si hay que reiniciar una comunicación es usar un IV.

4.2.1 Cifradores de flujo autosincronizados

Como muestra la Figura 2.15, en estos cifradores cada bit del keystream está en función de número un fijo de bits previos del texto cifrado.

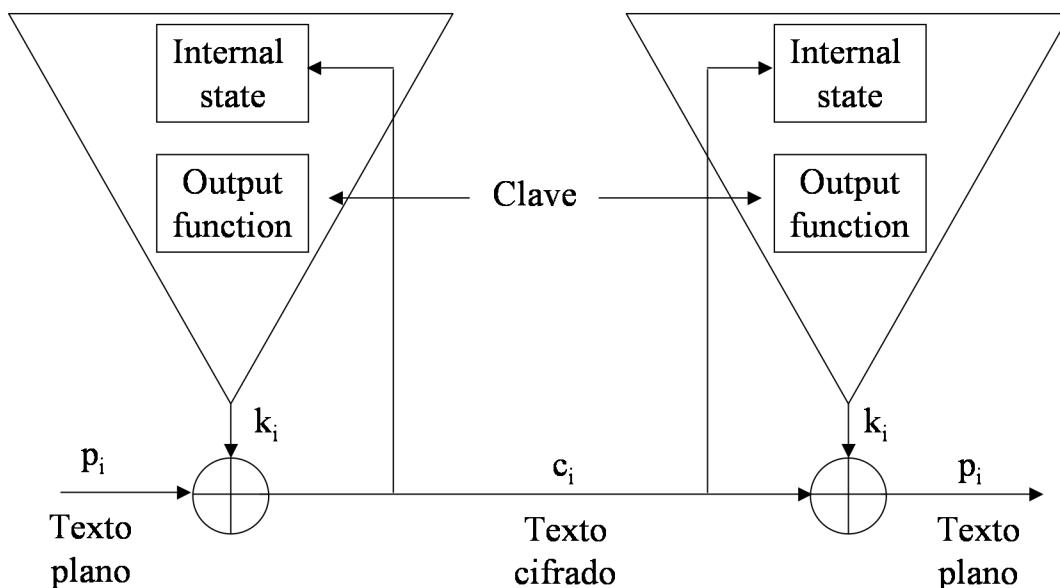


Figura 2.15: Cifrador de flujo autosincronizado

En este cifrador el internal state depende únicamente de R bits previos. Como dijimos antes, la complejidad criptográfica no está en el internal state, sino en la output function que depende del internal state y de la clave.

A estos cifradores se les llama **autosincronizados** porque emisor y receptor no tienen que sincronizar sus keystream, sino que, tras transmitir R bits, cifrador y descifrador quedan sincronizados. Esto se debe a que el internal state depende exclusivamente de los R anteriores bits cifrados. Es decir, el mensaje comienza con R bits basura tras los cuales emisor y receptor se sincronizan. Para que el sincronismo se produzca, la única condición es que emisor y receptor tengan la misma clave.

Propagación de errores

En estos cifradores siempre que se produce un error en un bit (bien sea alteración del bit o añadir o quitar un bit) el error se propaga durante los R siguientes bits que se descifran como basura, y tras los cuales cifrador y descifrador vuelven a estar sincronizados.

Problemas de seguridad

Los cifradores de flujo autosincronizados son vulnerables al playback attack. El atacante puede hacer que el receptor tras recibir R bits basura vuelva a recibir el mismo mensaje.

Este ataque es muy efectivo para comunicaciones de audio y video ya que el atacante puede capturar un mensaje de voz o audio, y tras una pequeña interferencia, reintroducirlo en la comunicación.

Otro ataque que es muy efectivo es el que se produce cuando el atacante conoce exactamente que un trozo de texto plano va en una determinada posición del texto cifrado, en este caso puede sustituir este texto plano por cualquier otro con sólo hacer un XOR con el texto plano que sabe que está en esa posición, y que quiere quitar, y otro XOR con el texto plano que quiere poner. En este caso después del mensaje modificado aparecerán R bits con basura, pero el daño puede estar ya hecho.

Este ataque es especialmente útil en protocolos en los que se saben exactamente qué comandos van en qué posición, o en comunicaciones multimedia donde puede cambiar un trozo de mensaje por otro.

4.2.2 Cifradores de flujo síncronos

En estos cifradores el keystream es generado independientemente del mensaje cifrado. Para ello es necesario que emisor y receptor tengan sincronizados sus keystream generators, con el fin de que los bit del keystream del emisor coincidan con los bits del keystream del receptor.

La principal ventaja que ofrecen los cifradores de flujo síncronos frente a los cifradores de flujo autosincronizados es que se adaptan mucho mejor a los problemas de transmisión en tiempo real (p.e. audio o videoconferencia), ya que el keystream puede ser precalculado tanto en emisión como en recepción, y cuando llega el momento de transmitir el texto sólo hay que hacerle un XOR con el keystream.

Problemas de seguridad

Un inconveniente que presentan los cifradores síncronos es que, como el keystream generator debe generar la misma salida en el emisor que en el receptor, el keystream generator debe ser determinista, y como está implementado en una máquina de estados finitos (el ordenador) la secuencia acabará repitiéndose, a no ser que el keystream generator use un one-time pad (una clave del tamaño del mensaje).

Al número de estados entre repetición se le llama **periodo**, y se procura diseñar sistemas con el máximo periodo, ya que, como dijimos antes cuando estudiamos los problemas de los cifradores de flujo, si se repiten los keystream era fácil criptoanalizar el sistema. El criptoanálisis es especialmente fácil si se conoce la longitud del periodo, ya que en este caso se podía hacer un XOR de dos mensajes que aparezcan en la misma posición de un periodo, obteniendo así dos textos planos hechos un XOR, y sacar luego el keystream correspondiente.

Para evitar este problema, la clave se debe cambiar periódicamente. Por ejemplo, si el internal state es un registro de 64 bits, existen next state functions que generan 2^{64} estados distintos (p.e. sumar 1 al registro). Si tenemos una línea T-1 que transmite 2^{37} bits a la hora, deberíamos de cambiar la clave antes de que pasen 2^{27} horas.

La principal ventaja de los cifradores síncronos frente a los autosincronizados es que están protegidos frente a un playback attack, ya que la inserción (que no la sustitución) de un solo bit en el mensaje provoca una pérdida de sincronismo que será inmediatamente detectado.

Propagación de errores

Si uno de ellos se desfasa, todo el mensaje a partir de allí se transmite mal, siendo necesario volver a sincronizar emisor y receptor para restaurar la comunicación.

A cambio, una ventaja de los cifradores síncronos es que si se altera un bit por acción del ruido en el canal de comunicación, este será el único bit que reciba mal el receptor.

4.2.3 Cifradores de flujo en modo contador

Como dijimos antes, la seguridad de los keystream generator se encuentra en la output function y no en el internal state (véase Figura 2.14), con lo que en el internal state podemos usar un simple contador que se vaya incrementando de uno en uno, es decir, la next state function lo que hace es incrementar uno al internal state.

Con un cifrador de flujo en modo contador es posible generar la i -esima clave k_i sin haber generado previamente todas las claves anteriores. Simplemente ponemos el contador en el i -ésimo internal state y generamos el bit. Esto es especialmente útil en el acceso aleatorio a ficheros de datos encriptados (p.e. bases de datos encriptadas), sin necesidad de desencriptar todo el fichero.

Problemas de seguridad

Un posible error sería empezar cifrando todos los ficheros con el mismo internal state (p.e. 0), ya que en este caso estaríamos usando siempre el mismo keystream y se podría hacer un XOR de dos ficheros encriptados para obtener dos ficheros de texto plano hechos un XOR.

Para evitarlo se recomienda empezar a encriptar cada fichero con un número distinto y poner este número en la cabecera del fichero.

En principio, el número por el que empezamos puede guardarse en la cabecera sin encriptar, pero en este caso debemos tener cuidado de que los números por los que empezamos a encriptar cada fichero sean distintos y no se solapen, ya que si no el criptoanalista puede juntar dos trozos de fichero cifrado que coincidan en la numeración, y hacerles un XOR para obtener los textos planos hechos un XOR.

4.3 Modos mixtos

Los modos de cifrado en bloque tienen la ventaja de que se implementan mejor en software, mientras que los modos de cifrado en flujo se implementan mejor en hardware (ya que procesan bit a bit).

Por contra, los modos de cifrado en bloque tienen el inconveniente de que los datos se cifran en unidades de tamaño de bloque (típicamente 64 bits) con lo que, a diferencia de los modos de cifrado en flujo, no se adaptan bien a problemas en los que hay que transmitir unidades pequeñas de información. Por ejemplo, terminales seguros como SSH (Secure SHell).

Los modos mixtos cogen lo mejor de ambos mundos, ya que usan un algoritmo de cifrado en bloque (que son más fáciles de implementar en software), y permiten transmitir pequeñas unidades de información (como los algoritmos de cifrado en flujo).

Estos algoritmos están pensados para cifrar byte a byte, pero se podrían modificar fácilmente para encriptar en otros tamaños distintos a 8 bits. Para reflejar el hecho de que en los cifradores mixtos trabajamos con un byte en vez de con un bit, a las variables las llamaremos P_i , K_i y C_i en vez de p_i , k_i y c_i .

A continuación vamos a explicar dos modos mixtos de cifrar: CFB y OFB.

4.3.1 CFB. Cipher FeedBack mode

En nuestra explicación vamos a suponer que tenemos un algoritmo de cifrado en bloque de 64 bits. El cifrador que vamos a implementar es parecido al cifrador de flujo autosincronizado, sólo que encripta byte a byte, en vez de bit a bit.

Tal como muestra la Figura 2.16, necesitamos una cola del tamaño del bloque (64 bits), representada por el registro de desplazamiento. La cola se inicializa con un IV.

Empezamos encriptando la cola con el algoritmo de bloque y la clave, para generar un bloque cifrado del cual sólo cogemos el byte más a la izquierda para obtener K_i (un keystream de 8 bits) al que hacemos un XOR con los 8 bits de P_i para obtener C_i .

Después movemos el byte C_i a la derecha del registro de desplazamiento y descartamos el byte más a la izquierda del registro de desplazamiento.

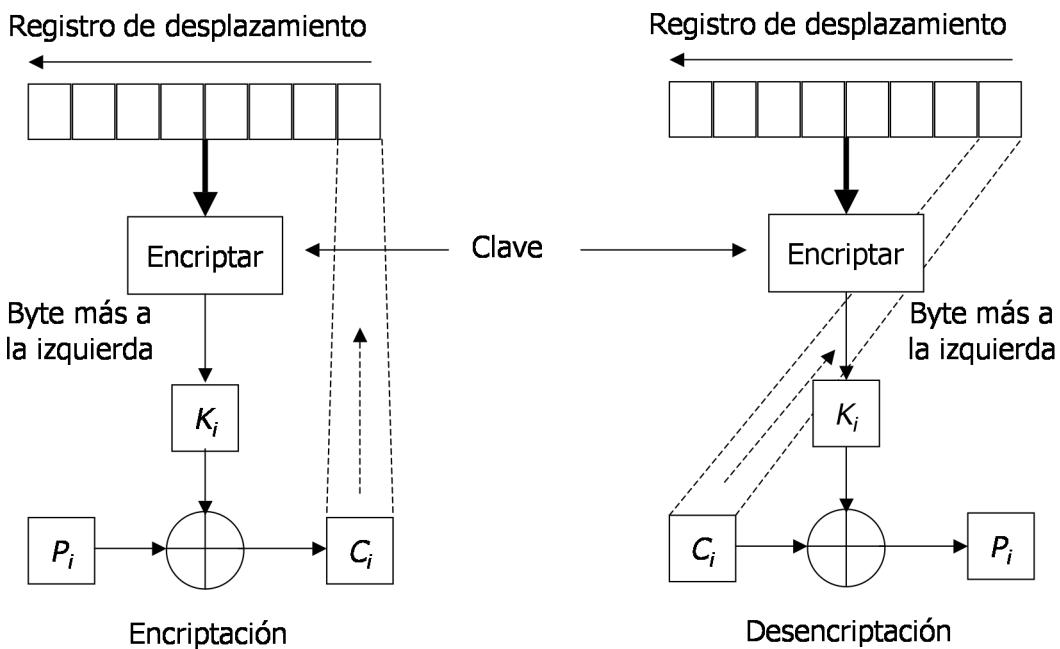


Figura 2.16: Cifrador mixto CFB

El algoritmo del receptor es similar al del emisor. En recepción también se usa el algoritmo de bloque en modo de encriptación con el fin de ir generando el mismo keystream que el emisor. La única diferencia es que ahora es el byte C_i al que hacemos un XOR con K_i para obtener P_i .

Lo importante es que en recepción, los 64 primeros bits se descifran como basura, tras lo cual el registro de desplazamiento queda sincronizado con el emisor.

Matemáticamente hablando, debido a que pasados los 64 bits iniciales, la clave K_i será la misma en emisión y recepción. La operación que se ejecuta en el emisor es:

$$C_i = P_i \oplus K_i$$

Y la que se ejecuta en el receptor:

$$P_i = C_i \oplus K_i$$

Si S_i es el estado del registro de desplazamiento en el paso i -ésimo, la forma en que van cambiando K_i y S_i viene dada por las operaciones:

$$\begin{aligned} K_i &= E(K, S_{i-1}) \\ S_i &= (S_{i-1} \ll 8) || C_{i-1} \end{aligned}$$

Obsérvese que en el receptor también es C_i el byte que usamos para cargar el registro de desplazamiento. Esto se hace con el fin de obtener el mismo keystream que en emisión.

Problemas de seguridad

Un inconveniente de seguridad del modo CFB es que, como cifrador de flujo autosincronizado que es, es propicio a un playback attack. Para evitarlo podemos poner un número consecutivo distinto a cada mensaje. Este número puede empezar a partir del valor del IV, ya que el IV no es necesario que sea secreto, pero sí que debería comprobar el receptor que sea mayor al último recibido para evitar un playback attack.

También, como algoritmo de cifrado en flujo autosincronizado que es, no existe el problema de que se criptoanalicen dos trozos que se sepa que fueron encriptados con el mismo internal state, ya que el IV o texto anterior distinto garantiza que los keystreams usados para cifrar cada mensaje son distintos.

Propagación de errores

Aquí un error en el texto plano da lugar a la generación de un texto cifrado distinto a partir del error, pero como hemos explicado ya otras veces, la desencriptación cancela este efecto, y al desencriptar sólo se obtiene el error que tenía el texto plano del emisor.

Más curioso es un error de cambio de un bit en el texto cifrado, que da lugar a que ese bit se desencripte mal sólo en el bit afectado (al hacer el XOR con K_i). A partir de este momento C_i entrará en el registro de desplazamiento y provocará la generación de basura hasta que sale por el otro extremo, es

decir, deja los 8 bytes siguientes a él con basura. Luego al final son 9 los bytes inutilizados por la alteración de un solo bit del texto cifrado.

Al igual que ocurría con los cifradores de flujo autosincronizados, CFB se sabe recuperar de errores de sincronismo (añadir o quitar un bit). El error entra en el registro de desplazamiento, y cuando sale de él, después de haber dejado 9 bytes erróneos, el sistema se vuelve a recuperar.

4.3.2 OFB. Output FeedBack mode

Este modo supone que disponemos de un cifrador en bloque de 64 bits, a partir del cual vamos a construir un cifrador que encripta byte a byte, como si fuera un cifrador en flujo síncrono (el keystream no depende del mensaje, con lo que el keystream se puede dejar precalculado).

Como muestra la Figura 2.17, el algoritmo es igual a CFB, excepto que el registro de desplazamiento se carga a partir de K_i en vez de a partir de C_i .

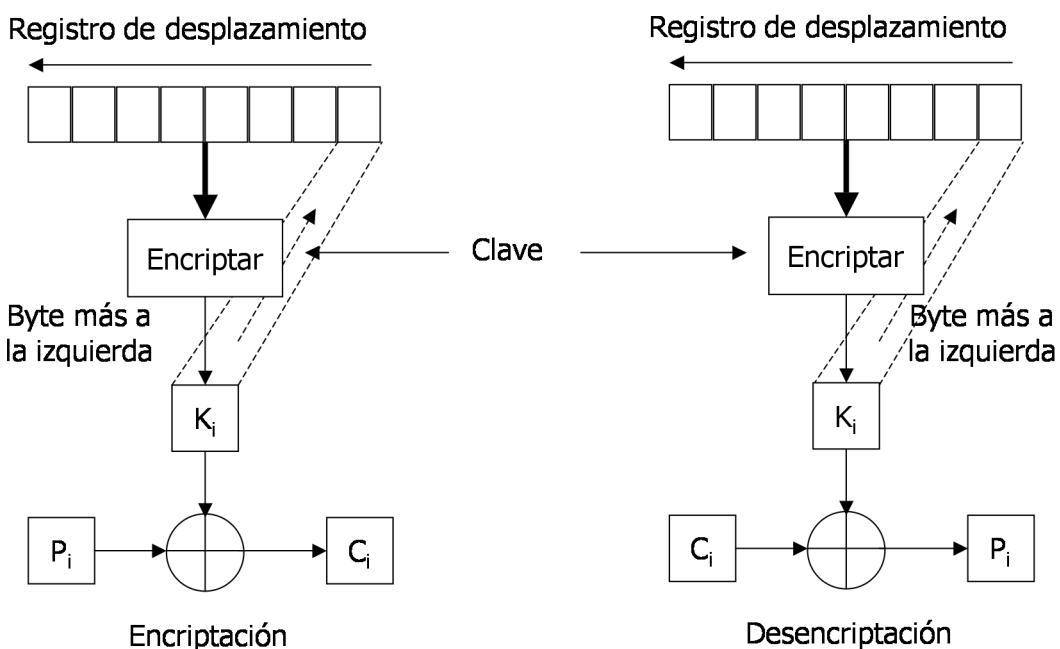


Figura 2.17: Cifrador mixto OFB

Ahora, el algoritmo que siguen emisor y receptor es totalmente idéntico.

Matemáticamente hablando:

$$C_i = P_i \oplus K_i \quad P_i = C_i \oplus K_i \quad K_i = E(K, S_{i-1}) \quad S_i = (S_{i-1} \ll 8) || K_i$$

Siendo S_i el estado del registro de desplazamiento del paso i -ésimo.

En OFB, al igual que en CFB, el registro de desplazamiento tiene que quedar cargado con un IV, que debe ser único aunque no tiene porque ser secreto.

Problemas de seguridad

El único problema de seguridad se produce si el IV no es único, ya que como explicamos en el apartado 4.2, es fácil criptoanalizar el mensaje cifrado a partir de otro mensaje cifrado con el mismo IV del que conozcamos el texto plano.

Propagación de errores

La propagación de errores en OFB es análoga a la propagación de errores en los cifradores síncronos: La alteración de un bit sólo afecta a ese mismo bit en recepción. A cambio, una perdida de sincronismo es fatal y no se recupera nunca. Por eso estos sistemas criptográficos deben disponer de un mecanismo de detección de pérdida de sincronismo.

4.4 Qué modo usar

ECB es el más fácil de usar, y el más rápido, pero el más vulnerable. En consecuencia ECB no se recomienda más que para encriptar pequeños mensajes aleatorios como por ejemplo claves. No se recomienda para encriptar ficheros (las cabeceras comunes se pueden criptoanalizar) ni para enviar mensajes (sufre de playback attack).

Para encriptar ficheros se recomienda usar uno de los otros modos: CBC, CFB, OFB.

Para grandes cantidades de información es más rápido CBC, pero si son mensajes pequeños (tráfico interactivo) tendremos que usar CFB o OFB.

En el caso de usar el cifrador para transmitir información multimedia (donde el contenido multimedia puede sufrir pequeñas alteraciones sin que sean perceptibles por el usuario). Entonces, si no hay problemas de sincronismo es mejor usar CBC. Si hay errores de sincronismo es mejor CFB porque sabe recuperarse de los errores de sincronismo aunque es más lento. OFB es la peor opción ya que tiene problemas de sincronismo y es lento para tráfico máxivo.

Por el contrario, OFB es la mejor opción para tráfico interactivo (p.e. SSH) ya que tiene menos problemas de seguridad de CFB.

Los cifradores en flujo puros se reservan para las aplicaciones hardware.

5 Padding

Los cifradores de bloque suelen trabajar con tamaños de bloque de 64 ó 128 bits, sin embargo los datos a encriptar no siempre son múltiplos de este tamaño.

Para resolver este problema hay dos soluciones:

1. Exigir que el usuario pase al cifrador unidades de información múltiplo del tamaño de bloque.
2. **Padding.** Es decir, que nuestro algoritmo criptográfico rellene el último bloque con datos de relleno antes de encriptarlo, y luego los quite en desencriptación.

La ventaja de usar padding es que es transparente al usuario.

De las técnicas de padding, la más utilizada en los algoritmos de clave secreta es PKCS #5 (Public Key Cryptography Standard 5). La técnica consiste en llenar los bytes restantes del último bloque con un número, que es el número de bytes que han quedado sin llenar en el último bloque tal como muestra la Figura 2.18.



Figura 2.18: Padding PKCS #5

Si resultara que los datos a encriptar fueran múltiplos del tamaño de bloque, se deja un bloque más entero lleno con ochos, esto se hace así porque si no no habría forma de que el receptor detectase el padding.

6 Claves binarias y PBE

Hasta ahora, los algoritmos de clave secreta que hemos utilizado usan una **clave binaria**, que es una clave con un contenido aleatorio y un tamaño predeterminado de bits. Debido a su aleatoriedad, estas claves son difíciles de recordar por las personas, con lo que es muy típico que el usuario utilice otra clave más sencilla de recordar, a la que llamamos **password**.

El password estará formado por una o más palabras, y después de pedírselo al usuario lo transformaremos en una clave binaria que es la que vamos a pasar al algoritmo. En concreto, vamos a pasar al password por una función hash para obtener una clave binaria, que es la que acabaremos dando al algoritmo de encriptación tal como muestra la Figura 2.19. Se llama **Password Based Encryption (PBE)** a las técnicas criptográficas que obtienen la clave binaria a partir de un password.

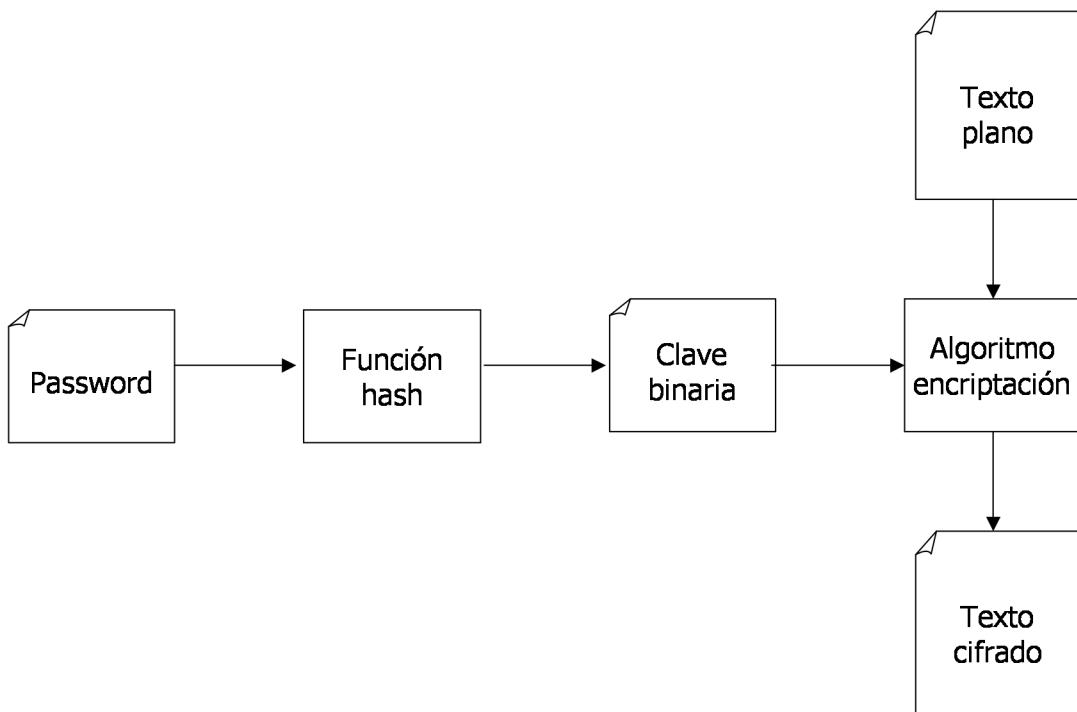


Figura 2.19: Obtención de una clave binaria a partir del password

Un problema que tienen los passwords es que son mucho más fáciles de atacar por fuerza bruta que las claves binarias. Por ejemplo, mientras que una clave binaria de 128 bits tiene 2^{128} combinaciones distintas, un password típico de 8 letras tiene 26^8 combinaciones (26 letras) que es aproximadamente 2^{33} combinaciones distintas.

Además, a esto se une el hecho de que el usuario tiende a utilizar palabras fáciles del idioma, con lo que se vuelve muy efectivo un **ataque por diccionario**. Para dificultar este ataque se suelen usar tres estrategias:

1. **Passphrase.** El programa puede sugerir al usuario que en vez de dar una palabra, de una frase entera, con el fin de aumentar el número de combinaciones.
2. **Salt (Sal).** El salt es un valor aleatorio que se concatena al password antes de pasarlo por la función hash que usamos para obtener la clave binaria. Si no usamos salt, el password `sesamo` siempre da lugar a la misma clave binaria, con el salt tenemos 2^{64} claves binarias distintas para el mismo password. De esta forma dificultamos los ataques por diccionario, obligando al atacante a calcular el hash de todas las claves con todos los salt.

El salt se almacena sin encriptar junto con el mensaje encriptado, ya que es necesario conocerlo para calcular la clave binaria que se está usando.

Un ejemplo de salt lo encontramos en el sistema operativo UNIX, donde al password que se va a almacenar en disco (normalmente en el fichero `/etc/passwd` o en `/etc/passwd.shadow`) se le añade un salt y se le pasa por una función MD5¹ para calcular la clave binaria. Después en disco se guarda salt+clave binaria.

De esta forma se consiguen dos ventajas: Por un lado el password real no se almacena en disco, con lo que ni siquiera el administrador de la máquina conoce el password que tienen sus usuarios. Es importante que los administradores no conozcan el password de sus usuarios para que no puedan utilizarlo para intentar acceder a otro sistema donde alguno de sus usuarios es usuario legítimo, pero el administrador no lo es. Por otro lado, al tener el password un salt, se complica el ataque por diccionario, ya que el atacante tiene que disponer de un diccionario completo precalculado para cada posible valor del salt.

3. **Iteration count.** Es una técnica utilizada con el fin de aumentar el tiempo necesario para calcular el hash de cada password.

El iteration count es el número de veces que hay que hacer hash a el salt junto con el password para calcular la clave binaria. Por ejemplo, si ponemos un iteration count de 1000, el ataque por fuerza bruta se vuelve 1000 veces más costoso.

¹ Hace unos años la función `crypt()` que usaba UNIX para calcular la clave binaria estaba basada en DES. Después se pasó a usar MD5, ya que DES puede ser atacado por fuerza bruta con los recursos hardware actuales.

7 Las librerías JCA y JCE

Debido a las restricciones de exportación de software criptográfico que existían en EEUU cuando se diseñaron las librerías criptográficas de Java, JavaSoft descompuso sus API criptográficas en dos librerías:

- La librería JCA (Java Cryptography Architecture)
- La librería JCE (Java Cryptography Extensions)

La librería JCA forma parte del runtime de la máquina virtual de Java a partir de la versión 1.2, bajo el paquete `java.security`. La librería JCA dispone de elementos de seguridad que no están sujetos a restricciones de exportación (p.e. firmas digitales o funciones hash).

En la librería JCE se incluyen clases de cifrado y descifrado de mensajes que son las que el gobierno de los EEUU no dejaba exportar. Debido a que cuando se creó la librería JCE, sólo se podía obtener en EEUU y Canadá, otros fabricantes de otros países se pusieron a hacer implementaciones de la librería JCE que distribuían gratuitamente al resto del mundo a través de Internet.

Entre ellos destacan:

<http://www.bouncycastle.org>
<http://www.cryptix.org>

A partir del JDK 1.4 cambiaron las leyes, y Sun pudo exportar la librería JCE como parte de su máquina virtual. Aun así los demás proveedores siguen siendo útiles, ya que disponen de más algoritmos que la librería JCE de Sun.

La librería JCE se distribuye como una extensión estándar. Las clases de la librería JCE están bajo el nombre `javax.crypto.*`, y como veremos más adelante, sus clases tienen los mismos privilegios que las del paquete `javax.*` (trusted class), cuando se instalan siguiendo unas determinadas reglas.

7.1 La librería JCA (Java Cryptography API)

La librería JCA está formado por dos grupos de clases:

1. Las **clases de la librería JCA**, que son clases que actúan como interfaces en las que se definen una serie de operaciones estándar.
2. Los **proveedores del seguridad**, que son terceras partes que implementan esas interfaces con algoritmos criptográficos propios.

En Java llamaremos **engine** o **servicio** a cada una de las operaciones criptográficas que proporcionan las librerías criptográficas de Java (p.e. funciones hash, encriptación, firmas digitales).

Como muestra la Figura 2.20, cada servicio está representado por una clase abstracta. Por ejemplo, `MessageDigest`, `Cipher` o `Signature`. La implementación de estos servicios se realiza por clases derivadas que implementan los proveedores de seguridad.

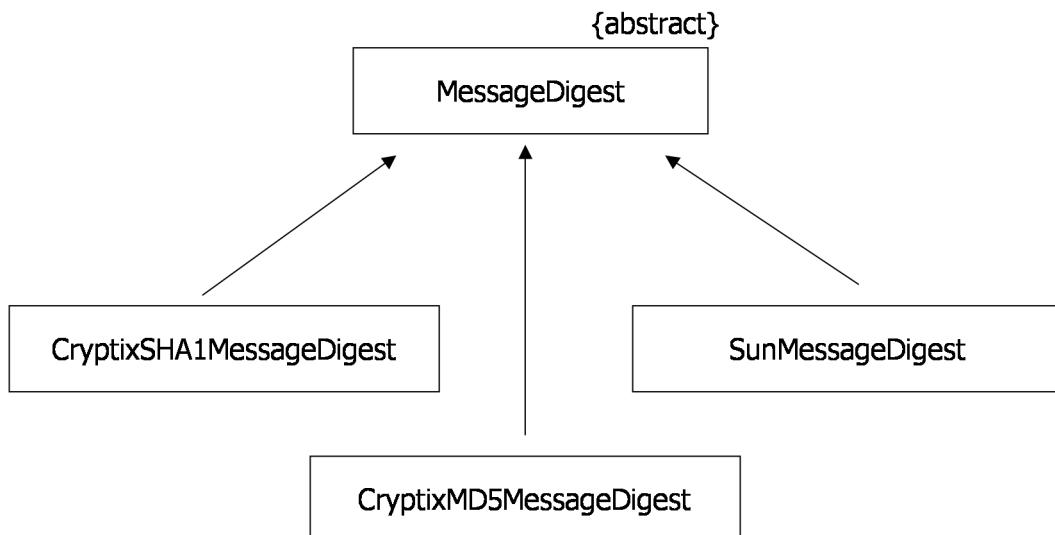


Figura 2.20: Implementación de servicios por clases derivadas

La relación entre la clase abstracta y las clases del proveedor no es 1 a 1, ya que para un mismo proveedor, un mismo servicio puede estar implementado por varias clases diferentes.

A cada implementación se le llama **algoritmo**. Por ejemplo, Cryptix implementa `MessageDigest` con los algoritmos SHA-1 y MD5.

La razón por la que se implementaron las librerías criptográficas de esta forma fueron dos:

1. Que un mismo servicio se pueda implementar con varios algoritmos.
2. Separar las interfaces del servicio, de la implementación que hacen los proveedores (para evitar los problemas de exportación).

En la clase abstracta siempre encontramos el método estático `getInstance()` al cual podemos pedir un objeto que implemente esa misma clase. Para ello debemos indicar el algoritmo y el proveedor. Por ejemplo, para `MessageDigest` haríamos:

```
MessageDigest md =
    MessageDigest.getInstance("SHA1", "SUN");
```

Algoritmo Proveedor

Desde el principio, la máquina virtual de Sun siempre ha traído dos proveedores para la librería JCA: "SUN" y "RSAJCA". Estos proveedores sólo implementan la librería JCA. Como veremos en el apartado 7.2 hay otros proveedores que implementan la librería JCE, o bien otros algoritmos para la librería JCA.

El orden de prioridad de estos proveedores se especifica en el fichero `java.security`:

En Mac OS X este fichero se encuentra en:

```
$JAVA_HOME/lib/security/java.security
```

En la implementación de Java para Linux que proporciona Sun se encuentra en:

```
$JAVA_HOME/jre/lib/security/java.security
```

Y en Windows se encuentra en:

```
%JAVA_HOME%\jre\lib\security\java.security
```

El Listado 2.1 muestra un ejemplo del contenido de este fichero. El número indica el orden de preferencia de la búsqueda de un proveedor que implemente el algoritmo pedido. Esto se hace así con el fin de que si omitimos el proveedor:

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

El método `getInstance()` elija el primer proveedor de la lista que implemente ese algoritmo.

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.apple.crypto.provider.Apple
security.provider.3=sun.security.rsa.SunRsaSign
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
```

Listado 2.1: Contenido del fichero `java.security`

7.2 La librería JCE (Java Cryptography Extensions)

La librería JCE, al igual que la librería JCA se puede dividir por dos partes:

1. Las clases de la librería JCE
2. Los proveedores de seguridad

7.2.1 JCE bajo restricciones de exportación

Hasta la versión Java 1.3 inclusive, Sun no podía exportar sus librerías de JCE, razón por la que pusieron dos restricciones:

1. Las clases de la librería JCE se exportaban a todo el mundo pero, para que Sun pudiera exportar la librería JCE fuera de los EEUU, tuvo que meter algunas restricciones de seguridad en ésta, las cuales hacían imposible el usar algoritmos "fuertes" de otros proveedores.
2. Sólo se exportaba la librería JCE, pero no los algoritmos criptográficos

Para solucionar el primer problema los proveedores crearon sus propias implementaciones de la librería JCE que no tienen estas restricciones, y a la que llamanon **JCE clean room**. El segundo problema se solucionaba proporcionando las clases que implementaban los algoritmos criptográficos.

Si disponemos de una máquina virtual Java 1.3 o inferior debemos obtener una implementación de la librería JCE que disponga de la librería JCE clean room, e instalarlo como vamos a explicar a continuación.

La librería JCE clean room son un conjunto de clases, bajo el paquete `javax.crypto.*`, que se debe descomprimir bajo el directorio de Mac OS X:

```
$JAVA_HOME/lib/ext/
```

O su equivalente en otro sistema operativo.

El descomprimirlo bajo este directorio se hace para que la librería JCE se instale como una extensión estándar.

Lo otro que tenemos que hacer es añadir las clases del proveedor (p.e. `org.bouncycastle.*`) al `CLASSPATH`. Para ello podemos descomprimir estas clases en un directorio y añadirlas al `CLASSPATH`, pero si se distribuyen en un fichero `.jar` también podemos añadirlas al `CLASSPATH` sin descomprimir el fichero con sólo ejecutar el siguiente comando (o su equivalente en otro sistema operativo):

```
$ export CLASSPATH=$CLASSPATH:.../bcprov-jdk13-121.jar
```

7.2.2 JCE liberado

Con la llegada de Java 1.4 desaparecen todas las restricciones de seguridad y Java viene con su propio proveedor de seguridad para la librería JCE llamado "SunJCE", el cual dispone de los principales algoritmos criptográficos. Además podemos instalarnos libremente otros proveedores de seguridad que traigan más algoritmos criptográficos. Para ello simplemente tenemos que añadir las librerías criptográficas del proveedor al CLASSPATH así:

```
$ export CLASSPATH=$CLASSPATH:.../bcprov-jdk14-121.jar
```

O su equivalente en otro sistema operativo.

8 Proveedores de seguridad instalados

En Java, además de las clases abstractas que representan los servicios, y las clases derivadas de las abstractas que representan los algoritmos, hay otras dos clases importantes:

`Provider` Cada instancia de esta clase representa a un proveedor de seguridad que tenemos instalado en el sistema.

`Security` Esta clase actúa como gestor de proveedores. La clase es final y tiene un constructor privado con lo que no se puede instanciar, pero tiene métodos estáticos que gestionan a los proveedores instalados. También tiene un bloque estático que es el que carga a los proveedores del fichero `java.security`. Podemos obtener los proveedores instalados con el método:

```
static Provider[] <Security> getProviders()
```

También podemos añadir o quitar proveedores en tiempo de ejecución con los métodos:

```
static void <Security> addProvider(Provider provider)
static void <Security> removeProvider(String name)
```

Lo cual es muy útil para no tener que pedir al usuario de nuestro programa que modifique el fichero `java.security`

Cuando ejecutamos el método estático `getInstance()`, que explicamos en el apartado 7.1, indirectamente estamos pidiendo a `Security` un algoritmo:

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

Tal como muestra la Figura 2.21, esta sentencia pide a `Security` un proveedor que implemente este servicio, y luego pregunta al proveedor qué clase instanciar para que le dé este servicio.

La forma que tiene `getInstance()` de pedir un servicio a `Security` es pasarle un `String` con el nombre del servicio y el algoritmo de la forma:

```
"MessageDigest.SHA1"
```

Y el método que utiliza para hacer esta petición es:

```
static Provider[] <Security> getProvider(String filter)
```

En `filter` pasamos el texto anterior y nos devuelve un array con todos los proveedores que implementan ese servicio ordenado por prioridad.

Después, `getInstance()` va al primero de los proveedores y le pide una clase que implemente ese servicio, una vez que tiene el nombre de la clase, la instancia para obtener ese servicio. Este último paso lo vamos a detallar en el siguiente apartado.

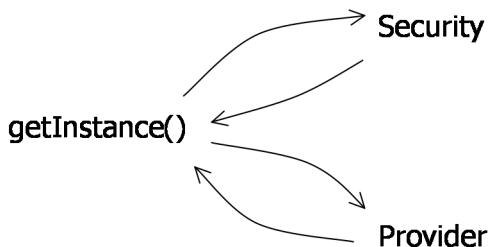


Figura 2.21: Pasos que sigue `getInstance()` para obtener un servicio

8.1 La clase Provider

`Provider` es una clase abstracta de la cual derivan las clases de los distintos proveedores.

En el fichero `java.security` podemos ver cuál es la clase de cada proveedor.

Por ejemplo, `security.provider.1=sun.security.provider.Sun`, significa que `security.provider.Sun` es una derivada de `Provider` con información acerca de este proveedor.

La clase `Provider` dispone de métodos con información sobre el proveedor como:

`String <Provider> getName()`

que devuelve el nombre del proveedor.

`double <Provider> getVersion()`

que devuelve la versión de implementación.

`String <Provider> getInfo()`

con otra información que quiera publicar el proveedor.

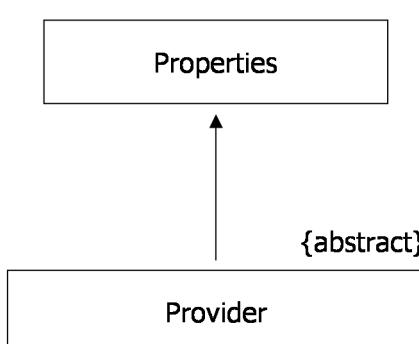


Figura 2.22: Jerarquía de la clase `Provider`

Tal como muestra la Figura 2.22, `Provider` deriva de `Properties` con lo que podemos usar el siguiente método para sacar un par clave-valor:

```
String <Properties> getProperty(String key)
```

A continuación se muestran los pasos que lleva a cabo `getInstance()` para obtener un objeto `MessageDigest` que implemente el algoritmo "MessageDigest.MD5":

```
String clase =
    proveedor.getProperty("MessageDigest.MD5");
MessageDigest md;
if (clase!=null)
{
    Class c = Class.forName(clase);
    Object obj = c.newInstance();
    md = (MessageDigest) obj;
}
```

8.2 Mostrar los proveedores instalados

El Listado 2.2 muestra un programa que primero pregunta a `Security` por los proveedores instalados usando:

```
static Provider[] <Security> getProviders()
```

Y luego saca por pantalla las propiedades de cada objeto `Provider`.

Después de hacer este programa y ejecutarlo, el lector puede probar a añadir al fichero `java.security` la entrada:

```
security.provider.8=org.bouncycastle.jce.provider.BouncyCastle
Provider
```

Y volver a ejecutar el programa `ProveedoresInstalados` para ver si ahora aparece el nuevo proveedor y sus algoritmos.

Nota: Reacuerde que tiene añadir el fichero con los algoritmos criptográficos al `CLASSPATH` (y instalar la librería JCE clean room en caso de usar Java 1.3 o anterior) tal como se explicó en el apartado 7.2

```
/* DESCRIPCION: Muestra los proveedores de seguridad
 * instalados en el sistema
 * AUTOR: Fernando Lopez Hernandez
 */

import java.security.*;
import java.util.*;

public class ProveedoresInstalados
{
    public static void main (String args[]) throws Exception
    {
        Provider[] proveedores = Security.getProviders();
        for (int i=0;i<proveedores.length;i++)
        {
            System.out.println();
            System.out.println(proveedores[i].getName()
                +" Version:"+proveedores[i].getVersion()
                +" "+proveedores[i]. getInfo());
            System.out.println();
            for(Enumeration e=proveedores[i].keys();
                e.hasMoreElements();
                {
                    String clave = (String) e.nextElement();
                    String valor = (String)
                        proveedores[i].getProperty(clave);
                    System.out.println("\t"+clave+"="+valor);
                }
            }
        }
    }
}
```

Listado 2.2: Proveedores de seguridad instalados

9 Tipos opacos y especificaciones

Muchos de los conceptos criptográficos de Java están encerrados bajo **tipos opacos**, que son objetos de los cuales sólo se conoce una interfaz de operaciones, pero no se conocen los valores que almacenan.

Sin embargo, a veces es necesario conocer los valores que ocultan estos tipos opacos, para lo cual existen las **especificaciones** (también conocidas como tipos transparentes).

Por ejemplo, en el caso de las claves, estas se encapsulan en la interfaz `Key`, sin embargo, a veces nos puede interesar saber qué tipo de clave es, y qué partes tiene. Para ello, tal como muestra la Figura 2.23, tenemos la interfaz `KeySpec`, que tiene tantas clases derivadas como tipos de claves define Java.

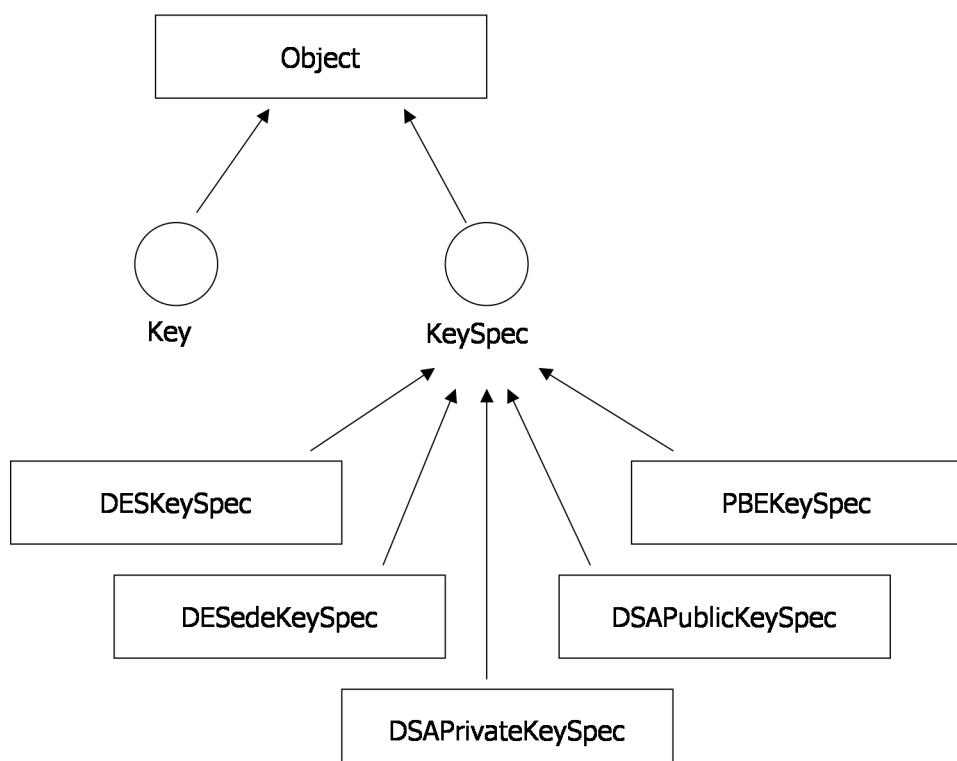


Figura 2.23: Jerarquía de `Key` y `KeySpec`

Otro ejemplo de especificaciones son los parámetros que necesitan los algoritmos para realizar su trabajo. La Figura 2.24 muestra la jerarquía de clases en el caso de la interfaz `AlgorithmParameterSpec`, y las clases que la implementan. Por otro lado `AlgorithmParameters` representa el tipo opaco.

Todas estas especificaciones se almacenan en los paquetes `java.security.spec.*` y `java.crypto.spec.*`

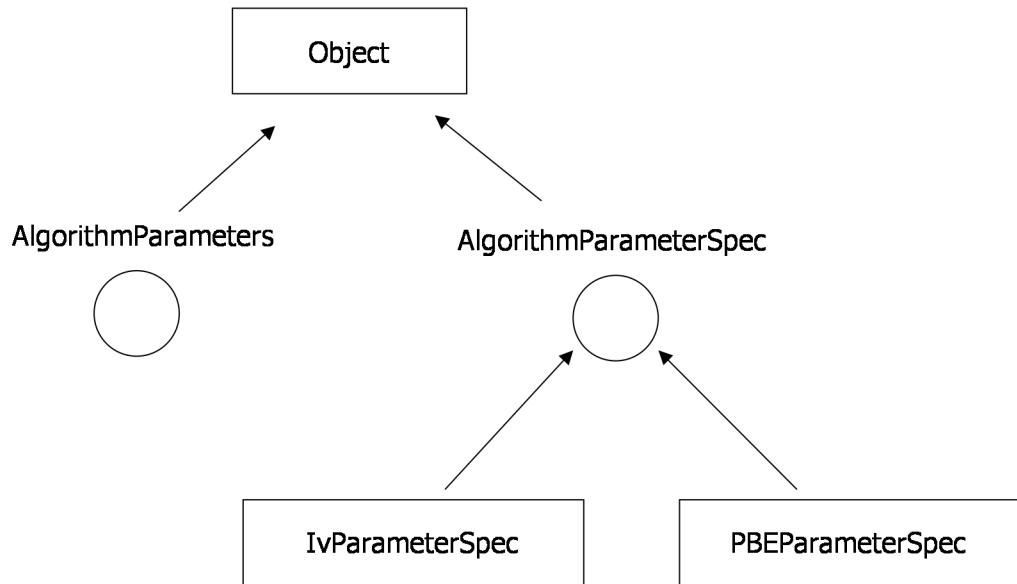


Figura 2.24: Jerarquía de `AlgorithmParameters` y `AlgorithmParameterSpec`

10 Uso de claves desde Java

Para la gestión de claves tenemos clases tanto en la API de la librería JCA (gestión de claves públicas, claves privadas y certificados digitales) como en la librería JCE (gestión de claves secretas).

10.1 La interfaz Key

La interfaz `java.security.Key` es la base de todos los tipos de claves de Java. Implementa `Serializable`, con lo que todas las claves en Java son serializables, y actúa como un tipo opaco, ya que no da información sobre sus atributos.

La interfaz `Key` únicamente tiene tres métodos. En consecuencia todo tipo de clave debe implementar estos métodos:

```
String <Key> getAlgorithm()
```

Devuelve el nombre del algoritmo para el que sirve la clave (p.e. "DES", "Blowfish", "DSA",...).

```
byte[] <Key> getEncoded()
```

Devuelve una ristra de bytes con la clave binaria.

```
String <Key> getFormat()
```

Indica el formato usado para la clave retornada por el método anterior (p.e. "X.509" o "PKCS#8"). Este formato es necesario para poder interpretar la ristra de bytes.

10.2 PublicKey, PrivateKey y SecretKey

Tal como muestra la Figura 2.25, las interfaces `PublicKey`, `PrivateKey` y `SecretKey` representan los distintos tipos de claves. Obsérvese que estas interfaces derivan de `Key` y no de `KeySpec` (véase Figura 2.23), ya que son tipos opacos y no especificaciones.

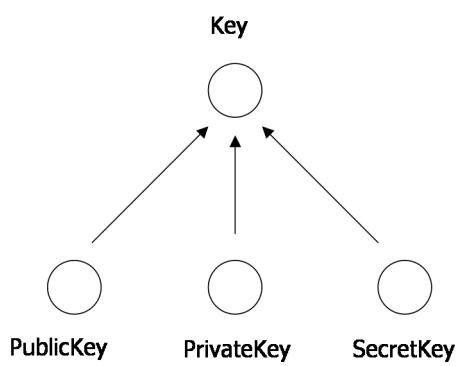


Figura 2.25: `PublicKey`, `PrivateKey` y `SecretKey`

Estas interfaces no añaden más métodos, sino que sólo se crean para clasificar los distintos tipos de claves.

Las interfaces `PublicKey` y `PrivateKey` se utilizan en criptografía asimétrica y vienen en el paquete `java.security.*`

Por el contrario, la interfaz `SecretKey` se utiliza en criptografía simétrica, y está en el paquete `javax.crypto.*`, debido a las restricciones de seguridad que tiene este tipo de librerías.

10.3 KeyPairGenerator y KeyGenerator

Para generar objetos que representen claves se usa una de estas clases:

`java.security.KeyPairGenerator` se usa para generar claves públicas y privadas. Estas siempre se generan en pareja.

`javax.crypto.KeyGenerator` se usa para crear objetos de tipo `SecretKey`, y es la que vamos a estudiar en este tema.

`KeyGenerator` es una clase abstracta que representa un servicio, y cuyos algoritmos están implementados por los proveedores de seguridad.

Como servicio que es, para obtener una instancia usamos el método estático:

```
static KeyGenerator <KeyGenerator>
    getInstance(String algorithm)
```

En el parámetro `algorithm` indicamos el algoritmo criptográfico para el que queremos generar la clave (p.e. "DES").

Después tenemos que inicializar el generador de claves, para lo cual tenemos el método `init()`. Este método está sobrecargado para permitirnos introducir parámetros comunes a todos los tipos de claves como son:

```
void <KeyGenerator> init(int keySize)
void <KeyGenerator> init(SecureRandom sr)
```

Que nos permiten indicar el tamaño de clave que queremos, o un generador de números aleatorios usado para generar la clave. Si no indicamos un `SecureRandom`, el objeto `KeyGenerator` instancia uno para sí sólo, aunque es mejor uno propio si vamos a crear varias instancias de `KeyGenerator`, ya que instanciar un `SecureRandom` consume bastante tiempo.

Luego hay otros parámetros que son propios de algunos algoritmos pero no de todos. Para pasar estos parámetros usamos:

```
void <KeyGenerator> init(AlgorithmParameterSpec param)
```

Reacuerde que, como mostraba la Figura 2.24, había clases que implementaban `AlgorithmParameterSpec`, como por ejemplo la clase `IvParameterSpec`, que sirve para indicar un IV.

Por último llamamos al método:

```
SecretKey <KeyGenerator> generateKey()
```

El cual nos permite generar tantas claves como queramos, una vez inicializado el objeto.

Obsérvese que `KeyGenerator` nos permite generar sólo claves binarias aleatorias. En el siguiente apartado veremos como generar claves PBE a partir de un password.

10.4 KeyFactory y SecretKeyFactory

Los **key factories** se usan para convertir objetos de tipo `Key` (tipos opacos) en especificaciones de esos objetos (tipos transparentes), y viceversa.

Su principal aplicación es permitir exportar e importar claves.

Existen dos formas de representar externamente una clave:

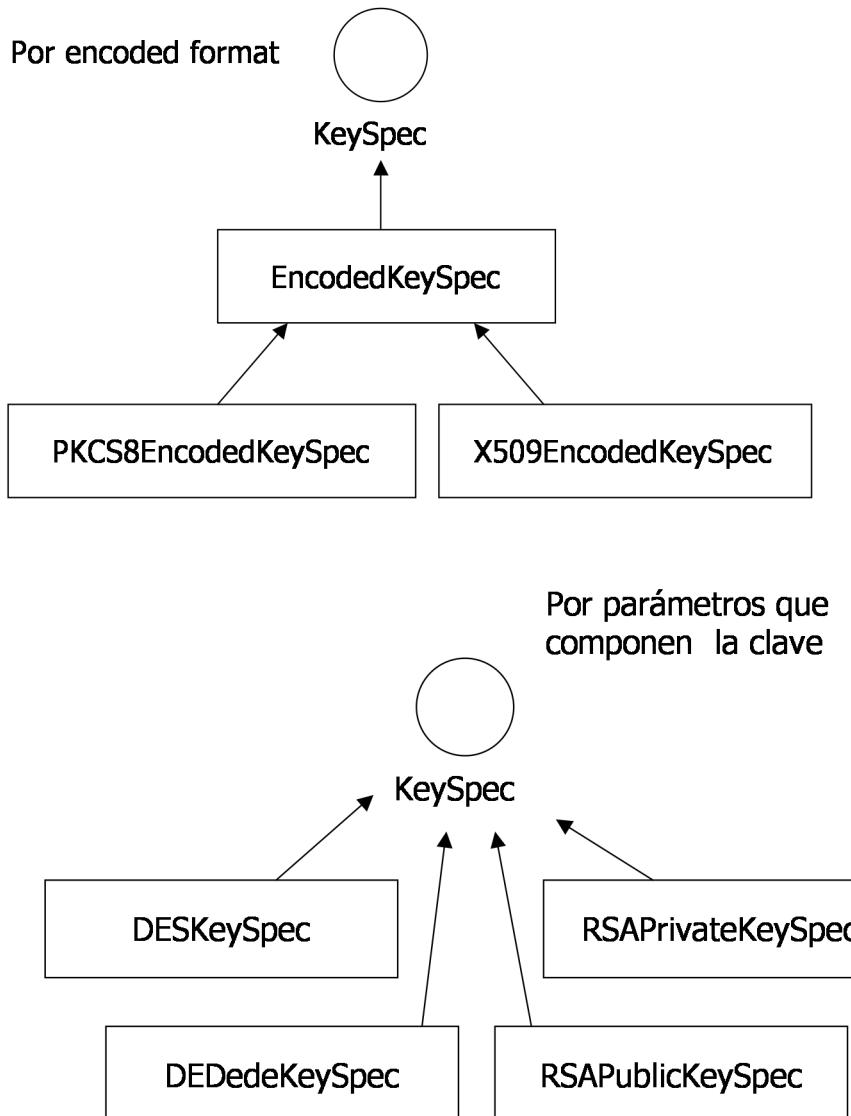
1. Por su encoded format
2. Con los parámetros que componen la clave

Ambas formas están implementadas por clases derivadas de `KeySpec` como muestra la Figura 2.26.

Si tenemos el encoded format devuelto por `getEncoded()`, y el formato devuelto por `getFormat()` (que puede ser "PKCS#8" o "X.509"), podemos crear un objeto derivado de `EncodedKeySpec` usando uno de estos constructores:

```
PKCS8EncodedKeySpec (byte[] encodedKey)  
X509EncodedKeySpec (byte[] encodedKey)
```

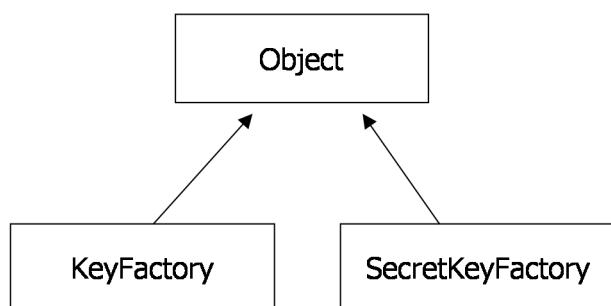
Si de lo que disponemos es de los parámetros que componen una clave, también hay clases derivadas de `KeySpec` que en su constructor reciben estos parámetros.

**Figura 2.26:** Formas de representar externamente una clave

Existen dos clases factory que actúan como key factories, es decir, que sirven para generar claves (véase Figura 2.27):

`KeyFactory` es la clase que sirve para importar y exportar claves asimétricas.

`SecretKeyFactory` es la clase que sirve para importar y exportar claves secretas, y es la que vamos a estudiar en este tema.

**Figura 2.27:** KeyFactory y SecretKeyFactory

`SecretKeyFactory` es una servicio, y como tal, para obtener un algoritmo de un proveedor usamos:

```
static SecretKeyFactory <SecretKeyFactory>
    getInstance(String algorithm)
```

Cada proveedor debe documentar las especificaciones que soporta su clave. Por ejemplo "SunJCE" soporta las especificaciones `DESKeySpec` y `DESEdeKeySpec` para el algoritmo "DES".

Esto significa que una clave "DES" (tipo opaco) de "SunJCE" se puede transformar en los tipos transparentes `DESKeySpec` y `DESEdeKeySpec`, o viceversa.

Para exportar una clave (obtener su tipo transparente) usaremos el método:

```
KeySpec <SecretKeyFactory> getKeySpec(SecretKey key
                                         , Class classSpec)
```

Donde el parámetro `classSpec` debería ser, o bien `DESKeySpec.class`, o bien `DESEdeKeySpec.class`

Por ejemplo, para obtener un `DESKeySpec` de un `SecretKey` aleatorio obtenido de tipo "DES" usando "SunJCE" hacemos:

```
// Creamos una clave como tipo opaco
SecureRandom sr = new SecureRandom();
KeyGenerator kg = KeyGenerator.getInstance("DES", "SunJCE");
kg.init(sr);
kg.init(56);
SecretKey clave = kg.generateKey();

// Obtenemos su correspondiente tipo transparente
SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
DESKeySpec dks = (DESKeySpec) skf.getKeySpec(clave,
                                              DESKEYSPEC.class);
```

Si lo que queremos es importar una clave (obtener el tipo opaco de un tipo transparente), creamos un objeto de algún tipo derivado de `KeySpec`, y se lo pasamos a:

```
SecretKey <SecretKeyFactory> generateSecret(KeySpec ks)
```

Por ejemplo, si tenemos un array de 8 bytes con una clave DES podemos hacer:

```
byte[] buffer; // Array de 64 bits cargado con la clave
DESKeySpec dks = new DESKeySpec(buffer);
SecretKey clave = skf.generateSecret(dks);
```

11 Uso de un cifrador

11.1 La clase Cipher

La clase abstracta `javax.crypto.Cipher` es un servicio que sirve para encriptar y desencriptar información. Como todos los servicios, para obtener una instancia usamos:

```
static Cipher <Cipher> getInstance(String algorithm)
static Cipher <Cipher> getInstance(String algorithm, String provider)
```

En este caso en `algorithm` tenemos que especificar uno de los algoritmos válidos para el proveedor. En la Tabla 2.10 se muestran los algoritmos válidos para el proveedor "SunJCE".

Algoritmo	Descripción
DES	Data Encryption Standard.
DESEde	Triple DES.
PBEWithMD5AndDES	Password Based Encryption definido en PKCS #5. PKCS #5 fue desarrollado por RSA Data Security Inc. Para encriptar claves privadas, aunque se puede usar para encriptar cualquier cosa.
Blowfish	Algoritmo open source desarrollado por Bruce Schneier

Tabla 2.10: Algoritmos de "SunJCE"

Modo	Descripción
ECB	Electronic Code Book. Sólo puede operar en bloques completos por lo que se usa con padding. No requiere vector de inicialización.
CBC	Cipher Block Chaining Mode. Sólo puede operar en bloques completos por lo que se usa con padding. Requiere vector de inicialización.
CFB	Cipher FeedBack Mode. Puede operar en bloques de 8 bits, en cuyo caso no requiere padding. Si lo requiere si ponemos un bloque mayor. Requiere vector de inicialización.
OFB	Output FeedBack Mode. Puede operar en bloques de 8 bits, en cuyo caso no requiere padding. Si lo requiere si ponemos un bloque mayor. Requiere vector de inicialización.
PCBC	Propagating Cipher Block Chaining Mode. Es popular en sistemas como Kerberos versión 4. Kerberos 5 paso a usar CBC. Requiere padding y vector de inicialización.

Tabla 2.11: Modos de "SunJCE"

Padding	Descripción
PKCS5Padding	Asegura que los datos resultantes son múltiplos de 8 bytes
NoPadding	Sin padding.

Tabla 2.12: Padding de "SunJCE"

Por ejemplo, para obtener un algoritmo para DES hacemos:

```
Cipher cifrador = Cipher.getInstance("DES");
```

Aunque también en `algorithm` podemos especificar el modo y el padding usando los valores de la Tabla 2.11 y Tabla 2.12 de la siguiente manera:

```
Cipher cifrador = Cipher.getInstance(
    "Blowfish/CBC/PKCS5Padding");
```

Si no se indica modo y padding cada proveedor tiene unos valores por defecto.

`Cipher` permite cifrar tanto en modos en bloque (p.e. ECB o CBC), como en modos mixtos (p.e. CFB o OFB), y modo en flujo.

Si queremos usar uno de los modos mixtos (que nos permitían cifrar en bloques más pequeños que los modos en bloque) debemos indicar el número de bits a procesar a la vez, añadiendo este número al modo. Por ejemplo, si queremos procesar byte a byte usaríamos:

```
"DES/CFB8/NoPadding"
```

Obsérvese que si vamos a procesar byte a byte no hace falta padding, y que poniendo el tamaño de bloque a 1 podemos conseguir un cifrador de flujo.

Una vez tengamos el objeto `Cipher`, tendremos que inicializarlo con uno de los muchos métodos `init()` sobrecargados de que dispone la clase:

```
void <Cipher> init(int op, Key key)
void <Cipher> init(int op, Key key,
                   AlgorithmParameterSpec aps)
```

El parámetro `op` indica si va a usarse el cifrador para encriptar o desencriptar, usando los valores:

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
```

El parámetro `key` es un objeto de tipo `SecretKey` con la clave a utilizar.

Además, el algoritmo puede requerir otros parámetros, como por ejemplo un IV, en cuyo caso se lo pasamos en un tercer parámetro en un objeto de tipo IvParameterSpec, que como muestra la Figura 2.24 deriva de AlgorithmParameterSpec. En "SunJCE" todos los modos excepto ECB requieren IV.

Hay que tener en cuenta que cuando vayamos a encriptar, si intentamos usar el cifrador sin estar bien inicializado, el cifrador lanza una IllegalStateException para avisarnos del problema.

Para ahora encriptar los datos tenemos dos alternativas:

- Encriptar todos los datos en una sola llamada, en cuyo caso usamos:

```
byte[] <Cipher> doFinal(byte[] input)
```

- Si tenemos muchos datos a encriptar, o los recibimos poco a poco, podemos irlos procesando según los vamos recibiendo llamando a:

```
byte[] <Cipher> update(byte[] input)
```

Y el último array lo pasamos a:

```
byte[] <Cipher> doFinal()  
byte[] <Cipher> doFinal(byte[] input)
```

También conviene tener en cuenta que cuando llamemos a `init()` se vuelve a reiniciar el cifrador.

Por otro lado la clase `Cipher` tiene métodos que nos dan información sobre el tipo de cifrado que realiza el cifrador, como por ejemplo:

```
int <Cipher> getOutputSize(int inputLength)  
int <Cipher> getBlockSize()  
byte[] <Cipher> getIV()
```

Para acabar, vamos a comentar que existe una clase derivada de `Cipher` llamada `javax.crypto.NullCipher` que no realiza ninguna encriptación, y que a diferencia de `Cipher` se puede crear con el constructor por defecto:

```
Cipher cifrador = new NullCipher();
```

Esta clase se puede usar para probar la lógica de nuestro programa sin realizar ninguna encriptación o desencriptación.

11.2 Encriptar y desencriptar de un mensaje

Vamos a realizar un programa que genera una clave secreta aleatoria, y luego la usa para cifrar y descifrar un mensaje pasado como argumento desde la línea de comandos. El Listado 2.3 muestra este programa.

Para cifrar el mensaje vamos a usar el proveedor BouncyCastle "BC", como algoritmo criptográfico usaremos DESede (también llamado TripleDES), como tamaño de clave usaremos una clave de 128 bits, como modo usaremos ECB, y como padding PKCS #5.

```
/*
 * DESCRIPCION: Programa que cifra y descifra un mensaje
 * pasado como argumento usando TripleDES,
 * también llamado DESede
 * Como proveedor usamos "BC" BouncyCastle
 * El tamaño de clave sera 128 bits
 * Como modo vamos a usar ECB
 * Como padding vamos a usar PKCS#5
 */

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

public class CifradoDescifradoClaveBinaria
{
    public static void main (String args[]) throws Exception
    {
        if (args.length!=1)
        {
            System.out.println("Indique texto a cifrar"
                               + " como argumento entre comillas");
            return;
        }

        System.out.print("Generando una clase DESede... ");
        KeyGenerator generadorClaves =
            KeyGenerator.getInstance("DESede", "BC");
        generadorClaves.init(128);
        Key clave = generadorClaves.generateKey();
        System.out.println("\rClave DESede generada");

        // Creamos un cifrador/descifrador
        Cipher cifrador = Cipher.getInstance(
            "DESede/ECB/PKCS5Padding", "BC");
        cifrador.init(Cipher.ENCRYPT_MODE, clave);

        // Encritamos con la clave obtenida
        byte[] texto_plano = args[0].getBytes("UTF8");
        byte[] texto_cifrado = cifrador.doFinal(
            texto_plano);
        System.out.print("Texto plano: ");
```

```
for (int i=0;i<texto_plano.length;i++)
    System.out.print(texto_plano[i]+" ");
System.out.print("\nTexto cifrado: ");
for (int i=0;i<texto_cifrado.length;i++)
    System.out.print(texto_cifrado[i]+" ");
System.out.println();

// Reiniciamos el cifrador al modo de
// desencriptación y desencriptamos
cifrador.init(Cipher.DECRYPT_MODE,clave);
byte[] texto_desencriptado = cifrador.doFinal(
                                texto_cifrado);
System.out.println("Texto desencriptado: "
                    +new String(texto_desencriptado,"UTF8"));
}
```

Listado 2.3: Cifrado y descifrado de un mensaje

Observe que hemos usado una clave binaria aleatoria para encriptar y desencriptar el mensaje. En el próximo apartado veremos como encriptar un mensaje a partir de un password.

12 PBE (Password Based Encryption)

Como explicamos en el apartado 6, es muy común que el usuario en vez de una clave binaria, lo que tenga sea un password, y sea el programa el que genere la clave binaria a partir del password.

Si queremos hacer esto en Java, tal como muestra la Figura 2.28, al cifrador tenemos que pasarle tres parámetros:

1. Password
2. Salt
3. Iteration count

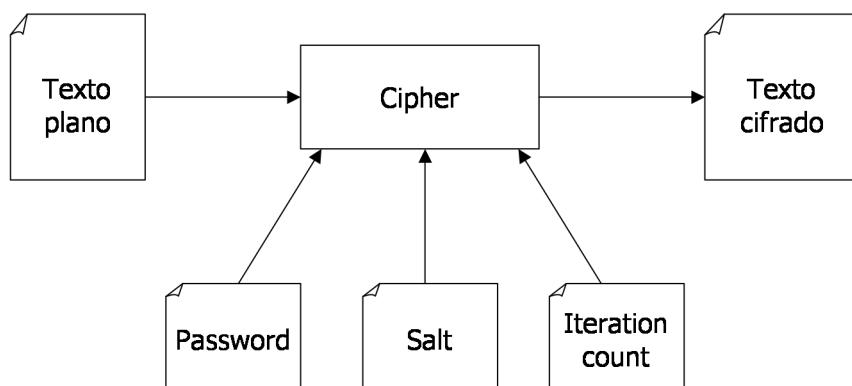


Figura 2.28: Encriptación con PBE

Para que el cifrador funcione en este modo, tenemos que pasar a:

```
static Cipher <Cipher> getInstance(String algorithm)
```

en el parámetro `algorithm` un algoritmo de PBE, que puede ser cualquiera de los de la Tabla 2.13.

Como ejemplo supondremos que vamos a usar "`PBEWithMD5AndDES`". Este algoritmo está descrito en la especificación PKCS #5 de RSA Data Security Inc. Inicialmente fue concebido para encriptar claves privadas, pero puede usarse para encriptar cualquier dato.

Vamos a ver ahora cómo pasar los tres parámetros que necesita el cifrador:

Para pasar el password al cifrador, debemos convertir primero éste a una clave binaria, para lo cual tenemos que empezar creando un objeto de la clase `PBEKeySpec`, la cual tiene el constructor:

```
PBEKeySpec(char[] password)
```

Obsérvese que recibe un array de caracteres, y no un `String`, por seguridad tal como explicamos en el apartado 1.1.2

Proveedor	Algoritmo
Sun	"PBEWithMD5AndDES"
Sun	"PBEWithMD5AndTripleDES"
Bouncy Castle	"PBEWITHMD5ANDDES"
Bouncy Castle	"PBEWITHSHA1ANDDES"
Bouncy Castle	"PBEWITHSHAAND40BITRC2-CBC"
Bouncy Castle	"PBEWITHSHAAND128BITRC2-CBC"
Bouncy Castle	"PBEWITHSHAAND40BITRC4"
Bouncy Castle	"PBEWITHSHAAND128BITRC4"
Bouncy Castle	"PBEWITHSHAAND2-KEYTRIPLEDES-CBC"
Bouncy Castle	"PBEWITHSHAAND3-KEYTRIPLEDES-CBC"
Bouncy Castle	"PBEWITHSHAANDTWOFISH-CBC"
Bouncy Castle	"PBEWITHSHAANDIDEA-CBC"

Tabla 2.13: Algoritmos disponibles para PBE

Ahora podemos crear una clave binaria (un objeto de tipo `SecretKey`) a partir del `PBEKeySpec` siguiendo los siguientes pasos:

1. Creamos un objeto de tipo `SecretKeyFactory` usando su método estático `getInstance()` al que pasamos "PBEwithMD5AndDES" como argumento.
2. Creamos un objeto de tipo `SecretKey` usando:

```
SecretKey <SecretKeyFactory> generateSecret(KeySpec ks)
```

A este método le pasamos el `PBEKeySpec` y nos devuelve la clave binaria en un objeto de tipo `SecretKey`.

Para pasar el salt e iteration count, creamos un objeto de tipo `PBEParameterSpec` que tiene el siguiente constructor:

```
PBEParameterSpec(byte[] salt, int iterationCount)
```

Una vez que tengamos los objetos `SecretKey` y `PBEParameterSpec`, los usamos para inicializar el cifrador con:

```
void <Cipher> init(int op, Key key
, AlgorithmParameterSpec aps)
```

Como muestra la Figura 2.24, `PBEParameterSpec` deriva de `AlgorithmParameterSpec`, y por lo tanto se puede pasar a este método.

Es decir, la forma de crear el cifrador a partir del password, salt e iteration count sería:

```
// 64 bits de salt
byte[] salt = new byte[8];
new Random().nextBytes(salt);

// Creamos los tres parámetros del cifrador
PBEParameterSpec param = new PBEParameterSpec(salt, 1024);
PBEKeySpec pbeks = new PBEKeySpec(password.toCharArray());

// Creamos la clave secreta
SecretKeyFactory skf = SecretKeyFactory.getInstance(
    "PBEwithMD5AndDES");
SecretKey clave = skf.generateSecret(pbeks);

// Creamos e inicializamos el cifrador
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
c.init(Cipher.ENCRYPT_MODE, clave, param);
```

Por último, conviene resaltar que el cifrador, además del password, ha recibido dos parámetros para la encriptación (salt, e iteration count). Estos mismos parámetros los tiene que tener el receptor para poder desencriptar. Recuérdese que el salt e iteration count no tenían porque ser secretos, aunque lógicamente el password sí. Además el cifrador puede usar un tercer parámetro que es el IV, en cuyo caso también lo debe conocer el receptor.

Existe una forma de obtener todos los parámetros que ha usado el cifrador, y así poder pasárselos al cifrador, que consiste en pedírselos el método:

```
AlgorithmParameters <Cipher> getParameters()
```

Donde `AlgorithmParameters`, es un tipo opaco que encapsula los distintos tipos transparentes `AlgorithmParameterSpec` que pasamos al cifrador en `init()`

Podemos volver a recuperar estos tipos transparentes con:

```
AlgorithmParameterSpec <AlgorithmParameters>
    getParameterSpec(Class paramSpec)
```

En `paramSpec` pasamos la clase del parámetro que queremos recuperar, como por ejemplo `IvParamSpec.class` o `PBEParamSpec.class`. Véase la Figura 2.24.

Aunque normalmente no se suele hacer esto, sino que lo que se quiere es enviar los parámetros al receptor, y para ello sacamos el encoded del objeto `AlgorithmParameters` usando el método:

```
byte[] <AlgorithmParameters> getEncoded()
```

Y en recepción se vuelve a reconstruir el objeto `AlgorithmParameters` a partir del encoded de la siguiente forma:

1. Instanciamos un objeto de tipo `AlgorithmParameters`:

```
AlgorithmParameters ap =
    AlgorithmParamter.getInstance("PBEWithMD5AndDES");
```

2. Metemos el encoded en el objeto:

```
ap.init(encoded);
```

Por último pasariamos el objeto `ap` al cifrador del receptor usando el `init()` que existen para tal propósito:

```
void <Cipher> init(int op, Key key, AlgorithmParameter ap)
```

Para acabar de ver los PBE, vamos a hacer un programa que encripta y desencripta un fichero usando un password.

El Listado 2.4 muestra el programa de encriptación el cual encripta un fichero usando el algoritmo "PBEWithMD5AndDES", con un salt de 64 bits y un iteration count de 1024.

El fichero generado consta de las partes que muestra la Figura 2.29:

- Un entero con la longitud de los parámetros encoded
- Los parámetros encoded por `AlgorithmParameters` (salt y iteration count).
- Los datos encriptados (siempre serán múltiplos de 8 bytes que es el tamaño de bloque).

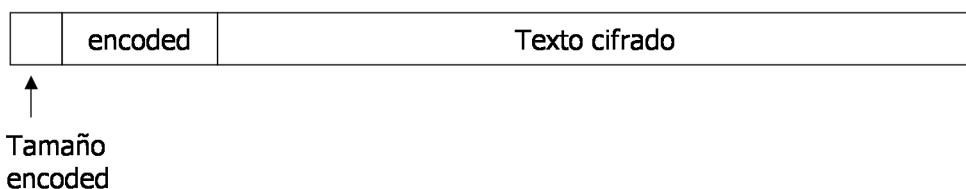


Figura 2.29: Partes del fichero encriptado con PBE

```
/* DESCRIPCION: Programa que encripta un fichero.
 * Encripta el fichero usando el algoritmo basado en
 * PBEWithMD5AndDES, con salt de 64 bits
 * y iteration count do 1024
 * El fichero generado consta de las siguientes partes:
 * - Entero con la longitud de los parametros encoded
 * - Los parametros encoded por AlgorithmParameters
 *   (salt y iteration count)
 * - Los datos encriptados (siempre seran multiplos de
 *   8 bytes, que es el tamano de bloque)
 */

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

public class EncriptaFichero
{
    public static final int ITERACIONES = 1024;
    public static final int TAMANO_SALT_BYTES = 8;
    public static final int TAMANO_BUFFER = 1024;

    public static void main (String args[]) throws Exception
    {
        // Comprobacion de argumentos
        if (args.length<2 || args.length>3)
        {
            System.out.println("Para encriptar indique "
                + "<password> <fichero_plano> "
                + "[<fichero_encriptar>] como argumento");
            return;
        }
        if (args.length>2 && args[2].endsWith(".des"))
        {
            System.out.println("Los ficheros encriptados"
                + " deben tener la extension .des");
            return;
        }

        // Abrimos los ficheros
        FileInputStream fichero_plano =
            new FileInputStream(args[1]);
        DataOutputStream fichero_encriptado;
        if (args.length==2)
            fichero_encriptado = new DataOutputStream(
                new FileOutputStream(args[1]+".des"));
        else
            fichero_encriptado = new DataOutputStream(
                new FileOutputStream(args[2]));

        //Generamos un salt aleatorio
        System.out.print("Generando salt... ");
        SecureRandom sr = new SecureRandom();
```

```
byte[] salt = new byte[8];
sr.nextBytes(salt);

// Generamos una clave secreta a
// partir del password
System.out.print("\rGenerando clave secreta");
PBEKeySpec objeto_password =
    new PBEKeySpec(args[0].toCharArray());
SecretKeyFactory skf =
    SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey clave_secreta =
    skf.generateSecret(objeto_password);

// Generamos los parametros de PBEPParameterSpec
PBEPParameterSpec pbeps =
    new PBEPParameterSpec(salt,ITERACIONES);

// Generamos el cifrador
Cipher cifrador =
    Cipher.getInstance("PBEWithMD5AndDES");
cifrador.init(Cipher.ENCRYPT_MODE
    ,clave_secreta,pbeps);

// Escribimos en el fichero encriptado los
// parametros encoded
System.out.print("\rEscribiendo fichero"
    + " encriptado... ");
AlgorithmParameters ap = cifrador.getParameters();
byte[] encoded = ap.getEncoded();
fichero_encriptado.writeInt(encoded.length);
fichero_encriptado.write(encoded);

// Escribimos en el fichero encriptado los
// datos del fichero plano
byte[] buffer_plano = new byte[TAMANO_BUFFER];
int leidos = fichero_plano.read(buffer_plano);
while(leidos>0)
{
    byte[] buffer_encriptado =
        cifrador.update(buffer_plano,0,leidos);
    fichero_encriptado.write(buffer_encriptado);
    leidos = fichero_plano.read(buffer_plano);
}
fichero_encriptado.write(cifrador.doFinal());

// Cerramos los ficheros
fichero_plano.close();
fichero_encriptado.close();
System.out.println("\rHecho ");
}

}
```

Listado 2.4: Programa que encripta un fichero con PBE

El programa que desencripta está en el Listado 2.5, y recupera del fichero los parámetros encoded metidos por `EncriptaFichero` para usarlos para desencriptar el contenido del resto del fichero. Hemos seguido el convenio de añadir la extensión `.des` a los ficheros encriptados, y eliminar esta extensión para obtener el nombre del fichero desencriptado.

Nota: Cuando se escribió el presente documento no todos los algoritmos de la Tabla 2.13 soportaban el poder leer parámetros encoded. Si quisieramos usar uno de estos algoritmos debemos fijar el salt y iteration count de forma fija en el programa.

```
/* DESCRIPCION: Programa que desencripta un fichero
 *                 encriptado por EncriptaFichero
 */

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

public class DesencriptaFichero
{
    public static final int ITERACIONES = 1024;
    public static final int TAMANO_BUFFER = 1024;

    public static void main (String args[]) throws Exception
    {
        // Comprobacion de argumentos
        if (args.length<2 || args.length>3)
        {
            System.out.println("Indique <password> "
                + " <fichero_encriptado> [<fichero_plano>] "
                + " como argumento");
            return;
        }
        if (!args[1].endsWith(".des"))
        {
            System.out.println("Los ficheros encriptados"
                + " deben tener la extension .des");
            return;
        }

        // Abrimos los ficheros
        System.out.print("Abriendo fichero...");
        DataInputStream fichero_encriptado =
            new DataInputStream(new FileInputStream(args[1]));
        FileOutputStream fichero_plano;
        if (args.length==2)
            fichero_plano = new FileOutputStream(
                args[1].substring(0,args[1].length()-4));
        else
            fichero_plano = new FileOutputStream(args[2]);
    }
}
```

```
// Generamos una clave secreta a partir
// del password
System.out.print("\rGenerando clave secreta");
PBEKeySpec objeto_password =
    new PBEKeySpec(args[0].toCharArray());
SecretKeyFactory skf =
    SecretKeyFactory.getInstance(
        "PBESWithMD5AndDES");
SecretKey clave_secreta =
    skf.generateSecret(objeto_password);

// Leemos los parametros encoded
int longitud_encoded =
    fichero_encriptado.readInt();
byte[] encoded = new byte[longitud_encoded];
fichero_encriptado.read(encoded);
AlgorithmParameters ap =
    AlgorithmParameters.getInstance(
        "PBESWithMD5AndDES");
ap.init(encoded);

// Creamos el cifrador
Cipher cifrador =
    Cipher.getInstance("PBESWithMD5andDES");
cifrador.init(Cipher.DECRYPT_MODE
    ,clave_secreta,ap);

// Desencriptamos el contenido del fichero
// encriptado y lo pasamos al fichero plano
System.out.print("\rDesencriptando fichero...");
byte[] buffer = new byte[TAMANO_BUFFER];
int bytes_leidos =
    fichero_encriptado.read(buffer);
while (bytes_leidos>0)
{
    fichero_plano.write(cifrador.update(
        buffer,0,bytes_leidos));
    bytes_leidos =
        fichero_encriptado.read(buffer);
}
fichero_plano.write(cifrador.doFinal());
// Cerramos los ficheros
fichero_encriptado.close();
fichero_plano.close();
System.out.println("\rHecho
");
}
```

Listado 2.5: Programa que desencripta un fichero encriptado con PBE

13 Los cifradores de streams

La librería JCE introduce el concepto de stream seguro, el cual combina un `InputStream` o un `OutputStream` con un `Cipher`. La jerarquía de estas nuevas clases se muestra en la Figura 2.30.

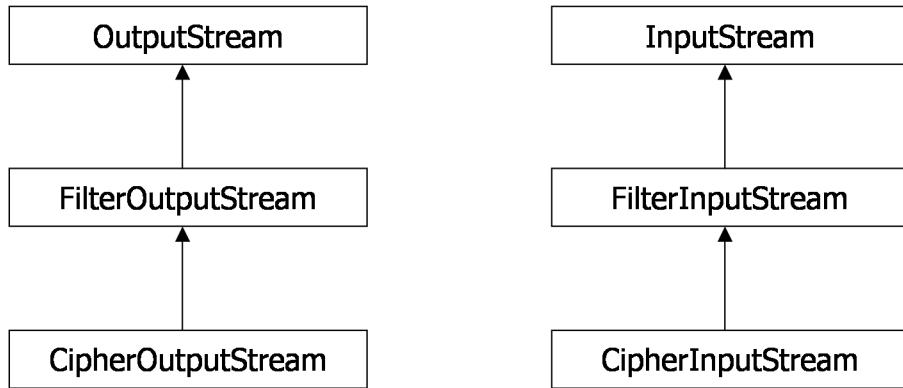


Figura 2.30: Jerarquía de clases de los cifradores de flujos

Como muestra la Figura 2.31 `CipherOutputStream` es una clase stream que encripta o desencripta los datos que pasan por ella dependiendo de si el `Cipher` está en modo de encriptación o modo de desencriptación.

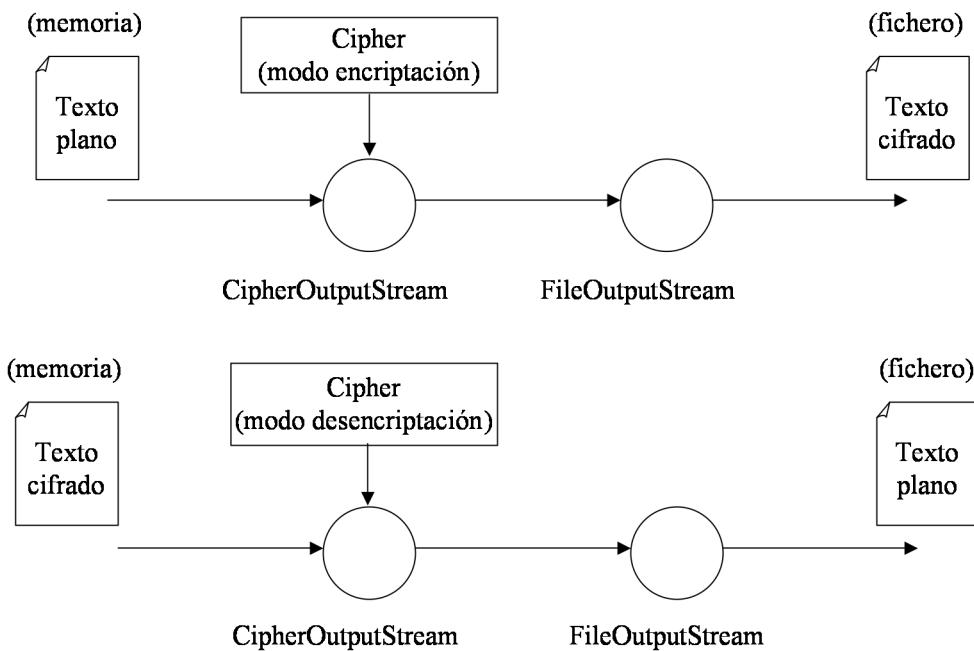


Figura 2.31: Flujo de datos a través de varios streams durante la encriptación

Cuando el objeto `CipherOutputStream` está en modo de encriptación encripta el texto plano para producir un texto cifrado que envía a un objeto

de la clase `OutputStream` o derivado. Cuando el objeto `CipherOutputStream` está en modo de desencriptación desencripta el texto cifrado para producir un texto plano.

Como todas las clases `stream`, la clase `CipherOutputStream` tiene un constructor en el que recibe dos parámetros: un objeto de tipo `OutputStream` o derivado al que engancharla (p.e. `FileOutputStream`), y el `Cipher` a usar para encriptar los datos:

```
CipherOutputStream(OutputStream os, Cipher cipher)
```

Análogamente, `CipherInputStream` es la clase que encripta o desencripta los bytes que pasan por ella. También en su constructor recibe un `InputStream` del que leer (p.e. un `FileInputStream`), y el cifrador a usar para desencriptar:

```
CipherInputStream(InputStream is, Cipher cipher)
```

14 La clase SealedObject

SealedObject es una clase que nos permite encriptar un objeto Java. El único requisito para poder usar esta clase es que el objeto a encriptar sea serializable.

Encriptar un objeto es tan sencillo como envolverlo en una instancia de SealedObject, para ello en el constructor de SealedObject pasamos el objeto a envolver y el cifrador a usar:

```
SealedObject(Serializable object, Cipher c)
```

El objeto SealedObject almacena internamente el algoritmo usado para encriptarlo, con lo que para desencriptarlo sólo hay que dar la clave al método:

```
Object <SealedObject> getObject(Key key)
```

15 Implementar un proveedor de seguridad

Como hemos explicado en el apartado 7.1, las librerías criptográficas de Java son una serie de clases abstractas, llamadas servicios o engine, que representan distintas operaciones criptográficas (p.e. Cipher, MessageDigest, Signature).

El proveedor crea derivadas de éstas, donde implementa las operaciones de la clase abstracta con su algoritmo o algoritmos tal como muestra la Figura 2.20. Cuando usamos `getInstance()` obtenemos referencias a objetos de estas clases derivadas.

Los métodos de la clase abstracta que representa un servicio se pueden dividir en dos grupos:

- API (Application Programming Interface). Son métodos públicos que utiliza el usuario del servicio para pedir servicios al objeto.
- SPI (Service Provider Interface). Son métodos protegidos y abstractos que debe implementar el proveedor.

Por convenio el nombre de los métodos SPI empiezan siempre por `engine`, y para cada método público del API existe su correspondiente método SPI. Por ejemplo para:

```
public byte[] <Cipher> doFinal()
```

Existe:

```
protected abstract byte[] <Cipher> engineDoFinal()
```

En JDK 1.1 ambos métodos estaban en la clase `Cipher`, pero a partir de JDK 1.2 los métodos `engineXXX()` se movieron a otra clase con el mismo nombre que el servicio, pero con el sufijo `Spi` (p.e. `CipherSpi`). La Figura 2.32 muestra la relación jerárquica entre la **clase de un servicio** y la **clase SPI** correspondiente.

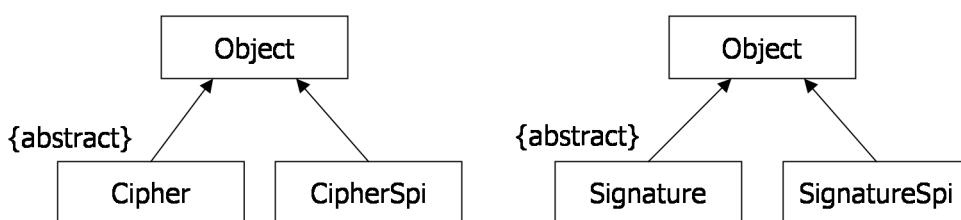


Figura 2.32: Clases abstractas de los servicios y su correspondiente clase SPI

Ahora los servicios lo único que tienen que hacer es delegar en los SPI.

Por ejemplo, a partir de JDK 1.2 si llamamos a:

```
public final byte[] <Cipher> doFinal()
```

Este se limita a llamar a:

```
protected abstract byte[] <CipherSpi> engineDoFinal()
```

El proveedor lo que tiene que hacer es crearse sus propias clases derivadas de la SPI, en las que implementa su algoritmo como muestra la Figura 2.33.

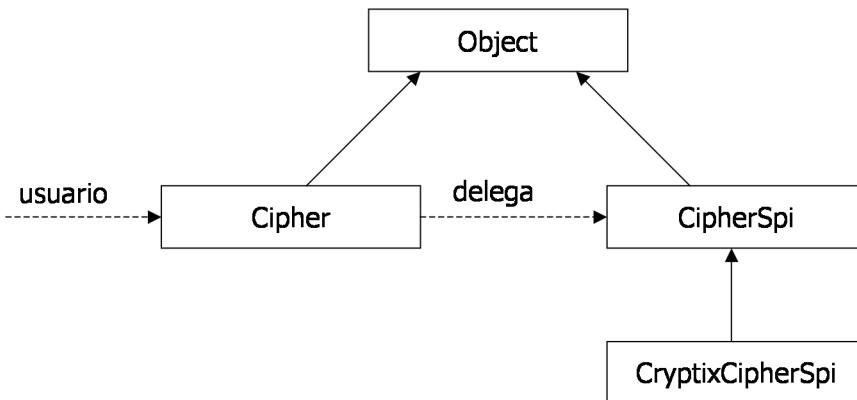


Figura 2.33: Delegación de las operaciones criptográficas en la clase del proveedor

Por otro lado, como vimos en el apartado 8, cuando usábamos `getInstance()` para pedir un servicio (p.e. "DES"), este llamaba a:

```
static Provider[] <Security> getProviders(String service)
```

El cual nos devolvía la lista de proveedores instalados que dan ese servicio.

Después `getInstance()` iba al primero de ellos y le pedía la clase en la que ese proveedor implementaba el servicio, la instanciaba, y nos devolvía el objeto instanciado. Es decir, lo que hace `getInstance()` es más o menos esto:

```

String clase = proveedor.getProperty("Cipher.DES");
if (clase!=null)
{
    Class c = Class.forName(clase);
    Object obj = c.newInstance();
    Cipher c = (Cipher) obj;
}
  
```

Para acabar el tema vamos a hacer un ejemplo en el que crearemos nuestro propio proveedor de seguridad que implemente el algoritmo ROT13. Para crear nuestro proveedor como muestra la Figura 2.34, tendremos que:

1. Crear una derivada de Provider y meterla en el fichero java.security
2. Crearnos un algoritmo derivado de cipherSpi y registrarlo en una derivada de Provider (Rot13Provider en nuestro ejemplo) con:

```
void <Provider>.setProperty(String key, String value)
```

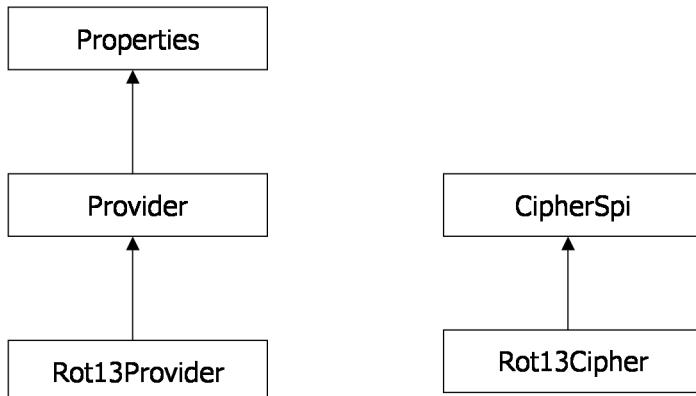


Figura 2.34: Clases necesarias para implementar un proveedor de seguridad

El Listado 2.6 muestra la clase `Rot13Cipher` en la que hemos implementado todos los métodos `engine`.

```
/*
 * DESCRIPCIÓN: Ejemplo de cifrador que usa ROT13 para
 *                 encryptar y desencriptar mensajes
 */

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

public class Rot13Cipher extends CipherSpi
{
    // Atributos
    private int modo = Cipher.ENCRYPT_MODE;

    public Rot13Cipher()
    {
    }

    protected void engineSetMode(String modo)
                    throws NoSuchAlgorithmException
    {
        if (!modo.toUpperCase().equals("ECB"))
            throw new NoSuchAlgorithmException(
                "ROT13 sólo soporta el modo ECB");
    }
}
```

```
protected void engineSetPadding(String padding)
                               throws NoSuchPaddingException
{
    if (!padding.toUpperCase().equals("NOPADDING"))
        throw new NoSuchPaddingException(
            "ROT13 solo soporta la opcion NoPadding");
}

protected AlgorithmParameters engineGetParameters()
{
    return null;
}

protected byte[] engineGetIV()
{
    return null;
}

protected void engineInit(int modo, Key clave
                         , AlgorithmParameterSpec parametros
                         , SecureRandom sr)
{
    this.modo = modo;
}

protected void engineInit(int modo, Key clave
                         , AlgorithmParameters parametros
                         , SecureRandom sr)
{
    this.modo = modo;
}

protected void engineInit(int modo, Key clave
                         , SecureRandom sr)
{
    this.modo = modo;
}

protected byte[] engineUpdate(byte[] entrada
                            , int offset
                            , int longitud)
{
    byte[] salida = new byte[entrada.length];
    for (int i=0;i<longitud;i++)
        salida[i] = rota(entrada[offset+i]);
    return salida;
}

protected int engineUpdate(byte[] entrada,
                         int offset_entrada, int longitud_entrada,
                         byte[] salida, int offset_salida)
                         throws ShortBufferException
{
    if (salida.length-offset_salida
        < longitud_entrada)
```

```
        throw new ShortBufferException(
            "No hay espacio suficiente en el "
            + "buffer de salida");
        for (int i=0;i<longitud_entrada;i++)
            salida[offset_salida+i] =
                rota(entrada[offset_entrada+i]);
        return longitud_entrada;
    }

protected byte[] engineDoFinal(byte[] entrada
                               , int offset, int longitud)
{
    return engineUpdate(entrada,offset,longitud);
}

protected int engineDoFinal(byte[] entrada
                           , int offset_entrada, int longitud_entrada
                           , byte[] salida, int offset_salida)
                           throws ShortBufferException
{
    return engineUpdate(entrada,offset_entrada
                        ,longitud_entrada,salida,offset_salida);
}

protected int engineGetBlockSize()
{
    return 1;
}

protected int engineGetOutputSize(int longitud_entrada)
{
    return longitud_entrada;
}

// Métodos privados
private byte rota(byte b)
{
    if (b>='A' && b<='Z')
    {
        b+=13;
        if (b>'Z')
            b-=26;
    }
    if (b>='a' && b<='z')
    {
        b+=13;
        if (((char)b)>'z')
            b-=26;
    }
    return b;
}
```

Listado 2.6: Implementación del SPI

El Listado 2.7 muestra cómo implementar la clase `Rot13Provider`, la cual sólo tiene un constructor redefinido en el que fijamos el nombre del proveedor, versión e información adicional.

```
/*
 * DESCRIPCIÓN: Ejemplo de Provider que implementa ROT13
 */

import java.security.*;

public class Rot13Provider extends Provider
{
    public Rot13Provider()
    {
        super("macprogramadores", 1.0
              , "Ejemplo de implementacion "
              + "de un Provider en Java");
        put("Cipher.ROT13","Rot13Cipher");
    }
}
```

Listado 2.7: Implementación de la clase del proveedor

Por último, el Listado 2.8 muestra un pequeño programa que usa nuestro proveedor.

```
/*
 * DESCRIPCION: Prueba que el provider funcione
 * correctamente. Para ello hay que crear un .jar con
 * las clases anteriores
 * ponerlo en el $CLASSPATH y anadir a
 * $JAVA_HOME/jre/lib/security/java.security la entrada
 * security.provider.9=Rot13Provider
 */

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

public class Test
{
    public static void listarPropiedadesProveedor()
    {
        Provider p = Security.getProvider(
                      "macprogramadores");
        if (p==null)
        {
            System.out.println(
                "No se encontro el proveedor");
            return;
        }
        System.out.println("INFORMACION DEL PROVEEDOR");
        System.out.println("Nombre proveedor:"
```

```
        +p.getName());
System.out.println("Version:"+p.getVersion());
System.out.println(
    "Informacion del proveedor:"+p.getInfo());
}

public static void probarElCifrador(String mensaje)
{
    try{
        System.out.println(
            "\nCreando un cifrador ROT13...");
        Cipher cifrador = Cipher.getInstance(
            "ROT13","macprogramadores");
        System.out.println(
            "Algoritmo del cifrador:"
            +cifrador.getAlgorithm());
        cifrador.init(Cipher.ENCRYPT_MODE, (Key)null,
                      (SecureRandom)null);
        byte[] mensaje_cifrado = cifrador.doFinal(
            mensaje.getBytes("UTF8"));
        cifrador.init(Cipher.DECRYPT_MODE, (Key)null,
                      (SecureRandom)null);
        byte[] mensaje_descifrado =cifrador.doFinal(
            mensaje_cifrado);
        System.out.println("Texto plano:"+mensaje);
        System.out.println("Texto cifrado:"
            +new String(mensaje_cifrado,"UTF8"));
        System.out.println("Texto descifrado:"
            +new String(mensaje_descifrado,"UTF8"));
    }
    catch(Exception ex){
        ex.printStackTrace();
    }
}

public static void main (String args[])
{
    if (args.length!=1)
    {
        System.out.println(
            "Pase como argumento un texto de prueba");
        return;
    }
    listarPropiedadesProveedor();
    probarElCifrador(args[0]);
}
}
```

Listado 2.8: Uso del proveedor creado

Recuerde crear un .jar con las clases Rot13Cipher y Rot13Provider, para lo cual puede usar el comando:

```
$ jar cf rot13.jar Rot13Provider.class Rot13Cipher.class
```

Así como añadir este .jar al CLASSPATH:

```
$ export CLASSPATH=$CLASSPATH:/Users/fernando/rot13.jar
```

También debe añadir la entrada security.provider.9=Rot13Provider al fichero java.security, para indicar que Rot13Provider es un proveedor de seguridad.

Tema 3

Criptografía de clave pública

Sinopsis:

En este tema vamos a estudiar qué es la criptografía de clave pública, qué ventaja aporta sobre la criptografía de clave secreta, y cómo se usa para enviar datos encriptados de forma segura.

Al acabar el tema el lector conocerá los principios de esta técnica desde un punto de vista teórico, aunque sus principales aplicaciones las vamos a desarrollar en los siguientes temas.

Empezaremos viendo algunos ejemplos de algoritmos de clave pública muy conocidos, para pasar luego a ver qué clases proporciona Java para este tipo de criptografía.

1 Introducción

En el apartado 3.2 del Tema 1, vimos las bases de los protocolos de comunicación mediante clave pública.

El protocolo para estos algoritmos era el siguiente:

1. B (el receptor) genera una clave privada aleatoria, y calcula la clave pública que corresponde a esa clave privada.
2. B envía a A (el emisor) su clave pública.
3. A encripta el mensaje usando la clave pública de B , y se la envía a B .
4. B utiliza su clave privada para desencriptar el mensaje.

Con esto, la criptografía de clave pública evita el problema de ponerse "en secreto" A y B de acuerdo en la clave a utilizar. Operación que en la criptografía de clave secreta tenía que hacerse "fuera de protocolo".

Es importante tener en cuenta que un espía que escucha toda la comunicación por la red obtiene la clave pública de B y el mensaje cifrado, pero no puede descifrarlo porque no tiene la clave privada, ni la puede calcular a partir de la clave pública.

También es muy típico que las claves públicas de los usuarios de un sistema criptográfico se publiquen en una base de datos pública, y cuando alguien quiera enviarles un mensaje, coja la clave pública de la base de datos.

2 Sistemas criptográficos híbridos

Por desgracia, los sistemas criptográficos de clave pública, por sí solos, no pueden usarse para encriptar mensajes, ya que sufren de dos problemas:

El primer problema es que son vulnerables al criptoanálisis de texto plano parcialmente elegido: Recuérdese que, como vimos en el apartado 4 del Tema 1, es aquel en el que el criptoanalista, no sólo tiene acceso a textos cifrados y sus correspondientes textos planos, sino que puede elegir parcialmente los textos planos a encriptar. Un ejemplo de este tipo de criptoanálisis es el que puede realizar un usuario de un servicio de red encriptado, donde el usuario que actúe como criptoanalista puede elegir los textos planos que quiere hacer pasar por el cifrador.

Como ya vimos, este tipo de criptoanálisis era especialmente efectivo si había relativamente pocos mensajes que se podían enviar (como suele ocurrir en los protocolos de comunicación), entonces el criptoanalista podía sacar los mensajes planos que correspondían a cada mensaje encriptado.

En criptografía de clave pública, como la clave de encriptación es pública, el criptoanalista puede calcular el criptograma de cualquier texto plano como:

$$C = E(K_{\text{publica}}, P)$$

Donde P es un valor de los n posibles. Entonces, el criptoanalista sólo tiene que encriptar todos los n posibles textos planos y comparar el mensaje encriptado C con los n mensajes planos P posibles.

Ciertamente el criptoanalista no podría determinar la clave privada, pero podría determinar P (a este tipo de criptoanálisis le denominamos deducción local en el apartado 4 del Tema 1). Además, en algunas formas de ataque basta con conocer que un texto cifrado no corresponde a un texto plano.

Este primer problema se puede solucionar fácilmente añadiendo a cada mensaje un padding de bits aleatorios, con lo que entonces idénticos mensajes planos dan lugar a diferentes textos cifrados.

El segundo problema es que los algoritmos de clave pública son mucho más lentos que los de clave secreta (del orden de 1000 veces más), además de aumentar el tamaño de los mensajes encriptados, lo cual dificulta su transmisión por medios de comunicación relativamente lentos, como pueda ser Internet.

Para solucionar estos problemas, en la práctica se utilizan **sistemas criptográficos híbridos**, que cogen lo mejor de ambos mundos (clave pública, clave secreta).

Estos sistemas se basan en la utilización de la llamada **clave de sesión**, que no es más que una clave binaria que se usa sólo durante lo que dure una comunicación.

El protocolo de los sistemas criptográficos híbridos es el siguiente:

1. B envía a A su clave pública.
2. A genera una clave de sesión aleatoria (binaria), la encripta usando la clave pública de B , y se la envía a B .
3. B desencripta el mensaje enviado por A , usando su clave privada, y recupera la clave de sesión.
4. Ambos se comunican mediante criptografía de clave secreta usando la clave de sesión.

En principio, no hay razón para que A se pueda comunicar con un solo interlocutor B . De hecho, A puede generar mensajes de broadcast encriptados, y sólo los pueden leer los destinatarios legítimos. El protocolo para conseguir que A envíe un mensaje de broadcast a B , C , D sería el siguiente:

1. A genera una clave de sesión aleatoria K_s , y encripta el mensaje M usando K_s : $C = E(K_s, M)$.
2. A encripta K_s con las claves públicas de B , C , D :
 $C_1 = E(K_B, K_s)$ $C_2 = E(K_C, K_s)$ $C_3 = E(K_D, K_s)$
3. A hace un broadcast del mensaje junto con las claves encriptadas, tal como muestra la Figura 3.1.
4. Los destinatarios desencriptan la clave de sesión con su clave privada y leen el mensaje.

$E(K_s, M)$	$E(K_B, K_s)$	$E(K_C, K_s)$	$E(K_D, K_s)$
-------------	---------------	---------------	---------------

Figura 3.1 Broadcast de un mensaje con criptografía de clave pública

3 Algoritmos criptográficos de clave pública más conocidos

En este apartado vamos a comentar cuáles son los algoritmos criptográficos de clave pública históricamente más conocidos. Algunos de ellos ya están rotos, pero se explican aquí porque su simplicidad puede ayudar al lector a entender la base en que se fundamenta esta forma de criptografía.

3.1 Los puzzles de Merkle

Ralph Merkle inventó el primer algoritmo criptográfico de clave pública en 1974 cuando estudiaba en la universidad de Berkeley.

Su técnica se basaba en el uso de puzzles que eran más fáciles de resolver por el emisor y por el receptor, que por cualquier espía.

El protocolo era el siguiente:

1. B genera 2^m mensajes, cada uno de ellos con dos números x, y . Por ejemplo, 2^{20} mensajes, que es aproximadamente un millón de mensajes.

En cada uno de estos mensajes:

- x – Es un índice aleatorio, no repetido
 y – Es una clave binaria aleatoria fuerte (p.e. 128 bits)

2. Usando un algoritmo simétrico encripta cada mensaje (x,y) usando una clave aleatoria débil de k bits. Por ejemplo $k=20$ bits.
3. B envía los 2^m mensajes encriptados a A .
4. A elige un mensaje aleatoriamente y realiza un criptoanálisis por fuerza bruta para recuperar el mensaje. Al ser una clave débil la que encripta el mensaje, el ataque es costoso pero no imposible (p.e. digamos que tarda un minuto).
5. A encripta el mensaje a enviar con la clave fuerte que obtuvo del mensaje (campo y del mensaje), y envía el mensaje encriptado junto con el índice x a B .
6. B a partir del índice x sabe la clave secreta, y la utiliza para recuperar el mensaje encriptado que le manda A .

La seguridad reside en que si el espía quisiera desencriptar el mensaje secreto transmitido por A , tendría primero que desencriptar cada puzzle hasta

encontrar el par (x,y) que le dice que clave se utilizó para encriptar el mensaje.

Luego el trabajo a realizar por el espía es:

1. Recorrer los 2^m puzzles en busca del par (x,y) que tiene la clave correcta, lo cual tiene un coste medio de $2^m/2=2^{m-1}$.
2. Desencriptar cada puzzle, lo cual tiene un coste medio de $2^k/2=2^{k-1}$.

Es decir, si el trabajo de A es de 2^{k-1} (p.e. 1 minuto), el trabajo medio del espía es de $2^{m-1}*2^{k-1}=2^{(m-1)+(k-1)}$.

Es decir, el trabajo del espía es 2^{m-1} veces el de A . P.e. si A tardase 1 minuto, el espía tardaría (para $m=20$) aproximadamente un año, suponiendo que ambos tengan la misma potencia de cálculo.

En este algoritmo se ve que la diferencia entre el coste de A y la del espía no es lo suficientemente grande para un caso real. Además el tamaño de los puzzles es demasiado grande. En consecuencia, este algoritmo no se usa en la práctica, pero es importante porque fue el primer algoritmo de clave pública que se pensó.

3.2 El algoritmo de la mochila

3.2.1 Descripción del problema de la mochila

El problema de la mochila consiste en que tenemos una serie de N elementos e_1, e_2, \dots, e_N (p.e. oro, plata, bronce), cada uno de ellos tiene un peso p_i (p.e. $p_1=5, p_2=4, p_3=4$) y un valor v_i (p.e. $v_1=15, v_2=11, v_3=10$). La Tabla 3.1 resume los valores de ejemplo:

e_i	Oro	Plata	Bronce
v_i	15	11	10
p_i	5	4	4
v_i/p_i	3	2.75	2.5

Tabla 3.1: Ejemplo de pesos y valores para la mochila

Por otro lado disponemos de una mochila que aguanta un peso máximo P (aunque tiene un tamaño infinito).

Se pretende encontrar la combinación de elementos que da un valor máximo sin sobrepasar el peso máximo.

Una primera estrategia consistiría en buscar un máximo global a partir máximos locales, es decir, podemos pensar en ordenar los elementos por

valor relativo v/p_i (valor por kilo), e introducir los elementos por orden en la mochila hasta que introducir uno más rebasase P .

Esta estrategia no siempre da la mejor solución. Por ejemplo, si $P=21$, empezaríamos metiendo sólo oro, que es el que más valor relativo tiene, pero como si metemos oro sólo podemos meter un lingote, al final metemos menos valor en la mochila que si hubiéramos metido plata+bronce que hubieran tenido juntos más valor.

Por desgracia (o por suerte), para encontrar el óptimo de este problema no se han encontrado mejores soluciones que el probar todas las combinaciones. De hecho, la mejor solución que se conoce usa técnicas de programación dinámica, que es una técnica que no vamos a explicar en este libro, basta con decir que es una técnica basada en el uso de recursividad al revés, es decir, en vez de hacer una recursividad en el sentido top-down, que es el normal, la programación dinámica usa recursividad bottom-up.

El algoritmo de la mochila tiene multitud de aplicaciones, entre ellas la de minimizar las monedas que devuelve una máquina al hacer una compra.

3.2.2 El problema de la suma

El problema de la suma es una variante del problema de la mochila, que fue utilizada por Ralph Merkle y Martin Hellman (de la Universidad de Berkeley) para implementar el primer algoritmo de clave pública funcional: El problema de los puzzles de Merkle era demasiado costoso para llevarlo a la práctica.

El problema consiste en que se tiene una colección de N números positivos $L=\{l_1, l_2, \dots, l_N\}$, y un entero $S \geq 0$. Se busca decidir qué suma de elementos (sin repetición) de L vale S , es decir, se busca el subconjunto de $r < N$ elementos $K=\{K_1, K_2, \dots, K_r\}$ tal que $S = l_{k_1} + l_{k_2} + \dots + l_{k_r}$

A este enunciado caben hacerle tres observaciones:

La primera es que a este problema también le se puede escribir como:

$$S = b_1 l_1 + b_2 l_2 + \dots + b_N l_N$$

Donde cada b_i puede valer 0 o 1, indicando que el elemento pertenece o no pertenece al conjunto solución.

Una segunda observación es que el problema de la suma es un tipo de problema de la mochila donde: Por un lado, el valor de cada número es 1 y su peso es l_i (el peso que representa). Y por otro lado, cada número sólo puede aparecer una vez en la mochila.

La tercera observación es que, al igual que la mochila, tiene un coste exponencial, y no hay mejor solución que el probar todas las combinaciones, es decir, se vuelve irresoluble cuando N y S son grandes. En la práctica se usan valores de $N=256$ (se pueden crear 2^{256} combinaciones distintas), y los números a sumar también son grandes (pueden oscilar entre 0 y 2^{128}), con lo que la suma S también estará en este orden.

3.2.3 Uso del problema de la suma para encriptar mensajes

Llamamos **conjunto de unicidad** a un conjunto de números $L=\{l_1, l_2, \dots, l_N\}$ en el que para todo número S , si S se puede escribir como la suma de algunos números de L , sólo existe una forma de escribir S como combinación de los números de L .

El conjunto de unicidad más conocido es el binario: $L=\{128, 64, 32, 16, 8, 4, 2, 1\}$

Ahora el protocolo para un algoritmo criptográfico de clave pública podría ser el siguiente:

1. Como clave pública se elige un conjunto de unicidad L .
2. El emisor descompone el texto plano P en bloques $P=[B_1, B_2, B_3, \dots]$ de N bits cada uno, siendo N el número de elementos de L , es decir: $N=|L|$
3. Cada bloque $B=[b_1, b_2, \dots, b_N]$ se cifra usando los bits del bloque de la forma:

$$C = \sum_{j=1}^n b_j l_j$$

Por ejemplo para encriptar $P=[101000111]$ con el conjunto de unicidad (clave pública) $L=\{5, 2, 1\}$ haríamos:

$$B_1 = [101] = 5+0+1 = 6 = C_1$$

$$B_2 = [000] = 0+0+0 = 0 = C_2$$

$$B_3 = [111] = 5+2+1 = 8 = C_3$$

Con lo que se genera el mensaje cifrado:

$$C=[C_1 \ C_2 \ C_3] = [0110 \ 0000 \ 1000]$$

4. El receptor coge cada bloque cifrado C_j , y resuelve sobre él el problema de la suma, para sacar los 1 y los 0 de bloque original B_j

Como el problema de la suma es un problema de complejidad exponencial respecto a N , el resultado de este protocolo es un método criptográfico tan

seguro, que ni el receptor puede desencriptar el mensaje enviado. En los siguientes apartados veremos cómo afrontar este problema.

3.2.4 Uso de una sucesión supercreciente para encriptar y desencriptar un mensaje

Llamamos **sucesión supercreciente** a una sucesión en la que cada nuevo número de la sucesión es mayor a la suma de todos los anteriores, es decir, una especie de sucesión de Fibonacci donde cada nuevo número se calcula como la suma de todos los anteriores (y algo más para que sea mayor a la suma de todos sus anteriores).

Si usamos como conjunto de unicidad (es decir, como clave pública) una sucesión supercreciente, resolver el problema de la suma tiene un coste lineal, y para resolverlo se resolvería mirando si el número de más peso es mayor a la suma de los anteriores.

Es decir, si `ls[]` es la sucesión supercreciente que actúa como conjunto de unicidad (clave pública), y `s` es el número a desencriptar, entonces podemos hacer una función que nos desencripte `s` usando el conjunto de unicidad `ls[]` como muestra el Listado 3.1.

```
public static boolean[] sumaDe(int s, int[] ls)
{
    int N = ls.length;
    bool[] sol = new Boolean[N];
    for (int i=N-1;i>=0;i++)
    {
        if (S>ls[i])
        {
            sol[i] = true;
            s -= ls[i];
        }
        else
            sol[i] = false;
    }
    return sol;
}
```

Listado 3.1: Solución al problema de la suma con una sucesión supercreciente

Ahora hemos dado la vuelta a la tortilla, y el mensaje es fácil de desencriptar tanto para el receptor como para el espía.

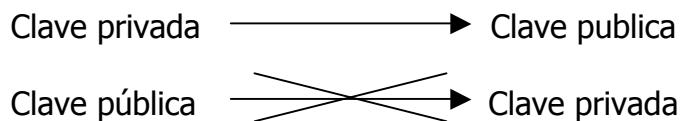
3.2.5 Uso de una sucesión supercreciente camouflada para encriptar y desencriptar un mensaje

Por fin, en este apartado vamos a ver cómo se puede usar el algoritmo de la mochila para realizar criptografía de clave pública.

El truco está en que se descubrió que en aritmética modular una sucesión supercreciente, a la que vamos a llamar **clave privada**, se puede usar para calcular una sucesión normal, a la que vamos a llamar **clave pública**. En concreto, usando aritmética modular la sucesión supercreciente se puede transformar en una sucesión normal que tiene las mismas propiedades que la supercreciente, es decir, que un mensaje encriptado con la sucesión normal se puede desencriptar con la sucesión supercreciente.

Si nosotros sólo damos la sucesión normal (clave pública) cualquiera puede encriptar mensajes, pero sólo los puede desencriptar quien tiene la sucesión supercreciente (clave privada).

Obsérvese que a partir de la sucesión supercreciente se puede obtener la sucesión normal, pero a partir de la sucesión normal no se puede obtener la supercreciente.



De hecho, a partir de la clave privada se pueden sacar infinitas claves públicas, y a partir de la clave pública también se pueden sacar infinitas claves privadas, pero: "a saber cual de ellas es la supercreciente".

El algoritmo para calcular, a partir de la sucesión supercreciente $I_s[]$, otra sucesión $I[]$ que sea equivalente a la primera es el siguiente:

1. Elegimos un modulo M tal que:

$$M > I_s[0] + I_s[1] + \dots + I_s[N-1]$$

2. Elegimos un multiplicador P tal que $\text{mcd}(M, P)=1$. Si M, P son primos relativos entre sí, existe un teorema que dice que: $\exists Q$ tal que $PQ \% M = 1$

3. Calculamos la sucesión pública que cumpla: $\forall i \quad I[i] = (I_s[i]*P)\%M$

Es decir, una sucesión que cumpla esta propiedad se calcularía con el siguiente programa:

```
for (int i=0;i<l.length;i++)
    l[i] = ls[i]*P%M
```

El array $l[]$ sería la clave pública que usaría el emisor para encriptar el mensaje.

Obsérvese que al poder elegir infinitas M y P , hay infinitas $l[]$ para cada $ls[]$, por eso decimos que a partir de la clave privada se pueden sacar infinitas claves públicas.

Ahora, el receptor para desencriptar sigue el siguiente algoritmo:

1. Para cada bloque C calcula su correspondiente C_s así:

$$\forall i \ C_s[i] = (C[i]*Q)\%M$$

2. Resuelve el problema de la suma para cada $C_s[]$ usando la sucesión supercreciente $ls[]$. Este problema es fácil de resolver ya que $ls[]$ es supercreciente.

Ejemplo

Para ilustrar el funcionamiento de este algoritmo vamos a hacer un ejemplo en el que queremos encriptar el mensaje:

$$\begin{aligned} B_1 &= [0 \ 0 \ 0 \ 0] \\ B_2 &= [0 \ 1 \ 0 \ 1] \end{aligned}$$

Usando la clave privada (obsérvese que es una sucesión supercreciente):

$$ls[] = \{21, 9, 5, 2\}$$

Los pasos que vamos a dar en el ejemplo son los siguientes:

- 1 Calcular una clave pública $l[]$ para $ls[]$
- 2 Encriptar
- 3 Calcular la Q
- 4 Desencriptar

1. Calcular una clave pública $l[]$ para $ls[]$

Para ello empezamos cogiendo un M aleatorio que cumpla que:

$$M > ls[0]+ls[1]+\dots+ls[N-1]$$

Por ejemplo, supongamos que elegimos $M=41$.

Ahora elegimos una P aleatoria que sea primo relativo con M , es decir que $mcd(M,P)=1$. Por ejemplo, $P=2$ cumple con la propiedad de que $mcd(41,2)=1$

Ya estamos en condiciones de calcular la clave pública $I[]$ como:

$$\forall i I[i] = (I_s[i]*P)\%M$$

$$\begin{aligned} I[0] &= (21*2)\%41 = 1 \\ I[1] &= (9*2)\%41 = 18 \\ I[2] &= (5*2)\%41 = 10 \\ I[3] &= (2*2)\%41 = 4 \end{aligned}$$

Luego la clave pública es:

$$I[] = \{1,18,10,4\}$$

2. Encriptamos

Para encriptar dijimos que multiplicamos cada bit del bloque por su correspondiente elemento de la clave pública:

$$\begin{aligned} B_1 &= [0 \ 0 \ 0 \ 0] \rightarrow C_1 = 0*1 + 0*18 + 0*10 + 0*4 = 0 \\ B_2 &= [0 \ 1 \ 0 \ 1] \rightarrow C_2 = 0*1 + 1*18 + 0*10 + 1*4 = 22 \end{aligned}$$

3. Calculamos Q

Supongamos que el mensaje ha llegado al receptor (el que tiene la clave privada) y que ahora lo tiene que desencriptar. Para ello además de $I_s[]$ necesita la Q que corresponde a la $I_s[]$ utilizada, y es lo que vamos a calcular.

Dijimos que por un teorema, si M, P eran primos relativos (y así los hemos elegido) entonces: $\exists Q$ tal que $PQ\%M=1$

Para buscar este Q se puede usar el algoritmo de Euclides extendido, pero nosotros lo vamos a buscar a ojo y sale: $Q=21$ ya que:

$$PQ\%M=1 \rightarrow 2*21\%41=1$$

4. Desencriptamos

Para desencriptar dijimos que para cada bloque encriptado $C[]$ teníamos que calcular su correspondiente $C_s[]$ como: $\forall i C_s[i] = (C[i]*Q)\%M$

Luego si: $C=\{0,22\}$

$$\begin{aligned} C_s[0] &= (0*21)\%41 = 0 \\ C_s[1] &= (22*21)\%41 = 462\%41 = 11 \end{aligned}$$

Con lo que: $C_s[] = \{0, 11\}$

Por último tenemos que resolver el problema de la suma para cada $C_s[]$ usando $I_s[]$:

$$\begin{aligned}C_s[] &= \{0, 11\} \\I_s[] &= \{21, 9, 5, 2\}\end{aligned}$$

Y resolviendo este problema con el algoritmo del Listado 3.1 obtenemos:

$$\begin{aligned}B_1 &= [0 \ 0 \ 0 \ 0] = 0*21 + 0*9 + 0*5 + 0*2 = 0 \\B_2 &= [0 \ 1 \ 0 \ 1] = 0*21 + 1*9 + 0*5 + 1*2 = 11\end{aligned}$$

Y vemos que realmente hemos vuelto a desencriptar el mensaje original.

3.2.6 Situación actual del algoritmo de la mochila

Por desgracia, Adi Shamir, uno de los que colaboró en la invención de RSA, rompió el algoritmo de la mochila al descubrir una forma de calcular la sucesión supercreciente (clave privada) que corresponde a la clave pública.

Aun así, el algoritmo ha quedado en la literatura criptográfica como un clásico por su simplicidad.

3.3 RSA

3.3.1 Aritmética

La **aritmética** es la parte de la matemática que trata las operaciones con números enteros positivos (0, 1, 2, 3, ...). Las operaciones que trata son la suma, resta, multiplicación y división, es decir, las que se estudian en la educación primaria. La división se realiza utilizando un cociente y resto enteros.

Antiguamente se llamaba **aritmética superior** al estudio de las partes más complejas de la aritmética, como por ejemplo las propiedades de los números enteros, hoy se prefiere llamarla **teoría de números**.

3.3.2 Los números primos

El tema de estudio preferido de la aritmética superior son los números primos. Un número es **primo** si no puede ser escrito como producto de otros números más pequeños mayores que 1.

Al 1 no se le considera ni primo ni no primo, así se consigue una mayor genericidad, como veremos en breve.

Los primeros números primos son:

2,3,5,7,11,13,17,19,23,29,...

Por el contrario, decimos que un número es compuesto si tiene un divisor mayor a 1. Por ejemplo: $24=4*6$

Los números primos ocuparían un lugar mucho menos importante del que ocupan en la aritmética si no fuera por el siguiente teorema, llamado el **teorema fundamental de la aritmética**:

Cualquier número entero positivo mayor a 1 puede escribirse como un producto de números primos de una única manera.

En este teorema hay dos afirmaciones de suma importancia:

- 1) Los números compuestos pueden descomponerse en productos de números primos:

P.e.: $21 = 7*3$
 $45 = 5*3*3$
 $24 = 3*2*2*2$

De acuerdo a esta observación, los números primos son los átomos a partir de los cuales se pueden sacar todos los demás números compuestos, que actuarían como moléculas. Al igual que en química los átomos de la tabla periódica se agrupan para formar moléculas (p.e. H_2O), en matemáticas los números primos se agrupan para formar los números compuestos.

- 2) Un número compuesto está formado por una única combinación de números primos.

Por ejemplo, si $91= 7*13$, entonces no existe ninguna otra combinación de números primos que al multiplicarlos de 91. Análogamente, en química, cuando un químico descompone H_2O obtiene 2 átomos de hidrógeno y uno de oxígeno, pero nunca obtendrá uno de plomo y 2 de azufre.

Para conseguir esta propiedad hay que excluir al 1 de los números primos, ya que si no un número compuesto se podría descomponer de muchas formas:

$$\begin{aligned}14 &= 2*7 \\14 &= 1*2*7 \\14 &= 1*1*2*7\end{aligned}$$

Por eso al 1 se le considera ni primo ni no primo, sino la **unidad**.

Otro concepto importante es el de números primos relativos. Decimos que dos números a, b son **primos relativos** si no existe ningún divisor común, es decir, no comparten ningún primo en común.

Evidentemente, los números primos son siempre primos relativos con cualquier otro número primo (es decir, son **primos absolutos**), pero va a haber números compuestos que tengan la posibilidad de ser primos relativos con otros números compuestos o primos.

Por ejemplo, los números compuestos 26 y 35 son primos relativos ya que no comparten ningún "átomo" del mismo tipo:

$$26 = 2 \cdot 13$$

$$35 = 5 \cdot 7$$

Y el 26 también es primo relativo con 11, ya que tampoco comparte con él ningún "átomo" del mismo tipo.

Para comprobar si dos números son primos relativos se suele buscar el máximo común divisor entre ellos, y si es 1, es que no tienen ningún divisor común, con lo que son primos relativos:

$$\text{mcd}(a,b)=1$$

Para buscar el máximo común divisor entre dos números se usa el conocido **algoritmo de Euclides**, que no vamos a explicar en aquí.

3.3.3 Aritmética modular

La aritmética modular consiste en realizar cálculos usando sólo un número finito de números enteros.

Un ejemplo de aritmética modular es un programa de ordenador donde cada variable sólo puede contener un rango de números (p.e [0..255], [0..4294967296]).

Otro ejemplo de esta aritmética es la que aprendemos en la escuela al hacer cálculos con el tiempo: Si son las 17h, qué hora será dentro de 11h, $17h+11h=4h$ de la madrugada.

La hora es una aritmética en módulo 24 (o en módulo 12), y se representa así:

$$(17+11) \bmod 24 = 28 \bmod 24 = 4 \bmod 24$$

En este caso se dice que 28 y 4 son **equivalentes** en módulo 24, y se representa así:

$$(28=4) \text{ mod } 24$$

De esta forma existen infinitas formas de representar el 4 en módulo 24, pero sólo una es la **forma normalizada** (el 4), que es la que se suele usar.

La importancia de la aritmética modular viene de que se descubrió que tiene las mismas propiedades que la aritmética normal: conmutativa, asociativa y distributiva:

1) Comutativa

$$\begin{aligned}(a+b) \text{ mod } n &= ((a \text{ mod } n) + (b \text{ mod } n)) \text{ mod } n \\(a-b) \text{ mod } n &= ((a \text{ mod } n) - (b \text{ mod } n)) \text{ mod } n\end{aligned}$$

2) Asociativa

$$((a+b) \text{ mod } n + c \text{ mod } n) \text{ mod } n = ((a \text{ mod } n) + ((b+c) \text{ mod } n)) \text{ mod } n$$

3) Distributiva

$$(a * (b+c)) \text{ mod } n = [(a * b) \text{ mod } n + (a * c) \text{ mod } n] \text{ mod } n$$

Es decir, empezamos a operar, y si el número empieza a crecer y sobrepasa n , simplemente le achicamos restándole n , y a seguir operando como si tal cosa.

En aritmética modular también se pueden realizar exponentiaciones:

$$a^x \text{ mod } n = (a * a * \dots * a) \text{ mod } n$$

Esto nos permite hacer una optimización que es muy buena para los números grandes:

$$a^{16} \text{ mod } n = [((a^2 \text{ mod } n)^2 \text{ mod } n)^2 \text{ mod } n]^2 \text{ mod } n$$

El truco de poder restar n al número cuando el número se hace grande es especialmente útil para poder calcular exponentiaciones gigantescas en un ordenador sin que se nos produzca un desbordamiento.

Otro concepto importante en aritmética modular es el de **inverso modular de un número**, que al igual que en aritmética normal es el valor por el que hay que multiplicar a un número para obtener la unidad.

Por ejemplo, en aritmética normal:

$$4*x = 1 \Rightarrow x=1/4$$

En aritmética modular de módulo 7:

$$(4*x = 1) \text{ mod } 7 \Rightarrow x=2$$

Ya que $(4*2=1) \text{ mod } 7$.

Al inverso modular de un número a se le suele representar normalmente como a^{-1} .

Por ejemplo: $(4^{-1}=2) \text{ mod } 7$

El problema del inverso modular es más difícil de calcular en aritmética modular que en aritmética no modular: En general se sabe que si:

$$(a^{-1}=x) \text{ mod } n$$

Entonces se cumplen estas dos propiedades para el problema de encontrar el inverso modular:

1. Si a, n son primos relativos entonces existe un única solución.
2. Si no es así, a veces puede tener una solución, a veces no, y otras veces hay muchas soluciones.

Como veremos luego (y vimos en el algoritmo de la mochila), para evitarse problemas los matemáticos usan en sus algoritmos siempre números a, n primos relativos entre sí.

Para obtener el inverso modular de un número se utiliza el llamado **algoritmo de Euclides extendido**, que tampoco vamos a explicar en aquí.

3.3.4 RSA

RSA es seguramente el algoritmo de clave pública más conocido, más probado, y más sencillo que se conoce.

Su nombre se debe a los tres inventores: Ron Rivest, Adi Shamir y Leonard Adleman.

Este algoritmo se basa en la dificultad de **factorizar** números grandes, es decir, sacar los primos (átomos) que componen el número. Se conjectura que romper el algoritmo es equivalente a factorizar un número grande. El tamaño de número a factorizar suele ser del orden de $2^{512}, 2^{1024}, 2^{2048} \text{ o } 2^{4096}$.

Como vamos a ver, RSA se puede usar tanto para encriptación como para generar firmas digitales.

Obtención de las claves pública y privada

1) Elegimos dos números primos grandes p, q y calculamos su producto n :

$$n = p * q$$

Para conseguir la máxima seguridad se recomienda que estos números estén entorno al mismo tamaño.

2) Elegimos aleatoriamente una clave de encriptación e , que cumpla que sea primo relativo con $(p-1)*(q-1)$, es decir:

$$\text{mcd}(e, (p-1)*(q-1)) = 1$$

3) Calculamos una clave de desencriptación d para la clave de encriptación e elegida que cumpla la siguiente relación:

$$(e * d = 1) \bmod (p-1)*(q-1)$$

O dicho de otra manera:

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

Es decir, d es el inverso modular de e , que antes vimos que se calculaba usando el algoritmo extendido de Euclides.

4) El par (e, n) forma la clave pública, y d es la clave privada. p, q no se necesitan ya más y se pueden descartar, eso sí nunca se deben desvelar al atacante.

Para encriptar y desencriptar se usan las siguientes fórmulas, que ahora explicaremos:

Encriptar:	$(c_i = m_i^e) \bmod n$
Desencriptar:	$(m_i = c_i^d) \bmod n$

Donde m_i son los textos planos y c_i son los textos cifrados.

Proceso de encriptación

Si queremos encriptar el mensaje m , lo tenemos que empezar dividiendo en bloques de tamaño menor a n .

Normalmente se suele elegir el primer número que sea potencia de 2 menor a n para decidir el número de bits de un bloque. Por ejemplo, si $n=1050$ (en la práctica es un número muchísimo más grande) se elige:

$$(2^{10}=1024) < (n=1050)$$

Luego se decidiría dividir el mensaje en bloques de 10 bits. Lógicamente, también en la práctica los bloques son más grandes porque n es más grande.

Cada uno de estos bloques los llamamos m_i , y darían lugar a bloques encriptados llamados c_i .

Y es aquí donde aplicamos para cada bloque la fórmula de encriptación:

$$(c_i=m_i^e) \text{ mod } n$$

En el apartado 3.3.3 vimos cómo se elevaba un número a una potencia en módulo n , y vimos que era más fácil en aritmética modular que en aritmética normal.

Proceso de desencriptación

En este caso, para cada uno de los bloques encriptados c_i aplicamos la fórmula:

$$(c_i=m_i^d) \text{ mod } n$$

Por último decir que d y e son comutativos, es decir, el algoritmo podría haber sido diseñado de forma que el mensaje se encriptase con d y desencriptase con e . Esta propiedad permite usar RSA para firmar digitalmente, tal como veremos en el Tema 4.

Ejemplo

Para terminar de concretar los detalles vamos a hacer un ejemplo de encriptación y desencriptación del siguiente mensaje:

$$m = 688\ 232\ 687\ 966\ 668\ 003$$

usando los números primos $p=47$ y $q=71$

Empezamos calculando las claves pública y privada:

1) Calculamos $n=p*q$:

$$n = 47*71 = 3337$$

2) Elegimos aleatoriamente una clave de encriptación e que cumpla que sea primo relativo con $(p-1)*(q-1)$, es decir:

$$\text{mcd}(e, (p-1)*(q-1)) = 1$$

Por ejemplo, podemos elegir $e=79$ que tiene esta propiedad. Obsérvese que elegir este número es muy fácil, simplemente hay que elegir alguno que no tenga como factor primo los factores primos de 46 y de 70.

3) Calculamos d de forma que:

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

Esto se calcula usando el algoritmo extendido de Euclides y nos da:

$$(d=e^{-1}) \bmod 3220 = 1019$$

Luego ya tenemos $d=1019$

4) Ahora ($e=79, n=3337$) es la clave pública, $d=1019$ es la clave privada, y p, q se descartan, ya que no se vuelven a usar.

Como ya tenemos las claves públicas y privadas vamos a encriptar el mensaje y después lo volvemos a desencriptar.

Para encriptar, dividimos el mensaje en bloques menores a 3337. Por ejemplo, lo podemos dividir en bloques de 0..999 para lo cual cogemos 3 dígitos del mensaje (por simplicidad estamos trabajando en decimal, aunque un ordenador lo haría en binario):

$$m = 688 \ 232 \ 687 \ 966 \ 668 \ 003$$

$$\begin{array}{lll} m_1=688 & m_3=687 & m_5=668 \\ m_2=232 & m_4=966 & m_6=003 \end{array}$$

$m_1=688$ se encripta como:

$$688^{79} \bmod 3337 = 1570$$

Luego $c_1=1570$

Análogamente encriptamos los demás bloques para obtener:

$$c = 1570 \ 2756 \ 2091 \ 2276 \ 2433 \ 158$$

Por último, para desencriptar usamos la fórmula:

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

Luego para desencriptar el primer bloque $c_1=1570$ haríamos:

$$1570^{1019} \bmod 3337 = 688 = m_1$$

Y análogamente para el resto de los bloques.

Seguridad de RSA

RSA no tiene problemas de seguridad en el algoritmo, pero sí puede tener problemas de seguridad en el protocolo si éste no se sigue correctamente.

El principal problema es el de que:

$$(m^e)^d = (m^d)^e = m$$

Es decir, e , d son intercambiables, con lo que podemos encriptar con e y desencriptar con d , o bien, encriptar con d y desencriptar con e .

Como adelantamos en el Tema 1, al encriptar se encripta con la clave pública y se desencripta con la privada, y al firmar digitalmente se firma con la clave privada y se comprueba la firma con la clave pública. En el Tema 4 veremos con más detalle el protocolo de firma digital.

Si el atacante puede engañar al usuario para que firme un mensaje antes de encriptarlo, puede conseguir acceder al texto plano del mensaje cifrado.

El protocolo que seguiría el atacante sería:

1. A quiere enviar un mensaje a B usando criptografía de clave pública RSA, pero hay un atacante escuchando la comunicación entre A y B .
2. A encripta el mensaje usando la clave pública de B , y se lo envía a B . El atacante guarda el mensaje encriptado.
3. El atacante manda a B un documento, dentro del cual está escrito el mensaje encriptado que captó en el paso anterior, para que B lo firme digitalmente. P.e. supongamos que B es un notario que se encarga de firmar documentos.
4. B manda el mensaje firmado a el atacante, y donde antes estaba el mensaje cifrado ahora está el mensaje plano ya que: $(m^e)^d = m$.

Para evitar este ataque se recomienda utilizar claves privadas distintas para encriptar y para firmar, o bien, usar para firmar otro algoritmo como por ejemplo DSA (Digital Signature Algorithm) que estudiaremos en el Tema 4.

4 Modo y Padding

Cuando vimos los cifradores de clave secreta en el Tema 2, vimos que usaban varios *modos*, de los cuales el más sencillo era ECB (Electronic Code Book), pero tenía el inconveniente de que siempre que encriptábamos un bloque obteníamos el mismo texto cifrado, con lo que era susceptible de dos ataques:

1. **Code book attack**, que consistía en crear un code book del bloque cifrado que correspondía a cada bloque plano.
2. **Relay attack**, que consistía en transmitir el mismo mensaje varias veces (p.e. un ingreso bancario).

Sin embargo, los cifradores de clave asimétrica casi siempre usan ECB, ya que no son susceptibles a estos ataques. Esto se debe a que sólo suelen transmitir un bloque, y este bloque se usa para encriptar una clave de sesión. Si fuéramos a transmitir más bloques usaríamos un algoritmo híbrido, tal como se explicó en el apartado 2.

Respecto al **padding**, en el Tema 2 vimos que los cifradores simétricos solían usar el algoritmo PKCS #5. RSA, que es el único cifrador asimétrico que vamos a estudiar en detalle, usa dos formas de padding distintas:

1. **PKCS #1** Que es la forma estándar de padding para RSA. Desgraciadamente esta forma de padding tiene problemas de seguridad si no se usa para encriptar datos totalmente aleatorios. Es decir, este problema no se presenta cuando encriptamos claves de sesión, pero para encriptar otros tipos de información se usa el siguiente algoritmo de padding.
2. **OAEP (Optimal Asymmetric Encryption Padding)**. Este mecanismo de padding es una mejora sobre PKCS #1 que nos permite encriptar todo tipo de datos aunque sigan patrones muy marcados, como por ejemplo las cabeceras. Desgraciadamente es muy nuevo y no ha sido todavía terminado por RSA Inc. Se espera que con el tiempo acabe suplantando a PKCS #1.

5 Criptografía de clave pública en Java

La mayoría de los aspectos de las librerías criptográficas en criptografía de clave pública son idénticos a los de la criptografía de clave secreta: Básicamente nos limitamos a inicializar un objeto `Cipher` pasándole las claves, y le pedimos que cifre o descifre.

La principal diferencia es que aquí hay dos claves: La clave privada y la clave pública.

Java tiene las clases de la Figura 3.2 para trabajar con estas claves.

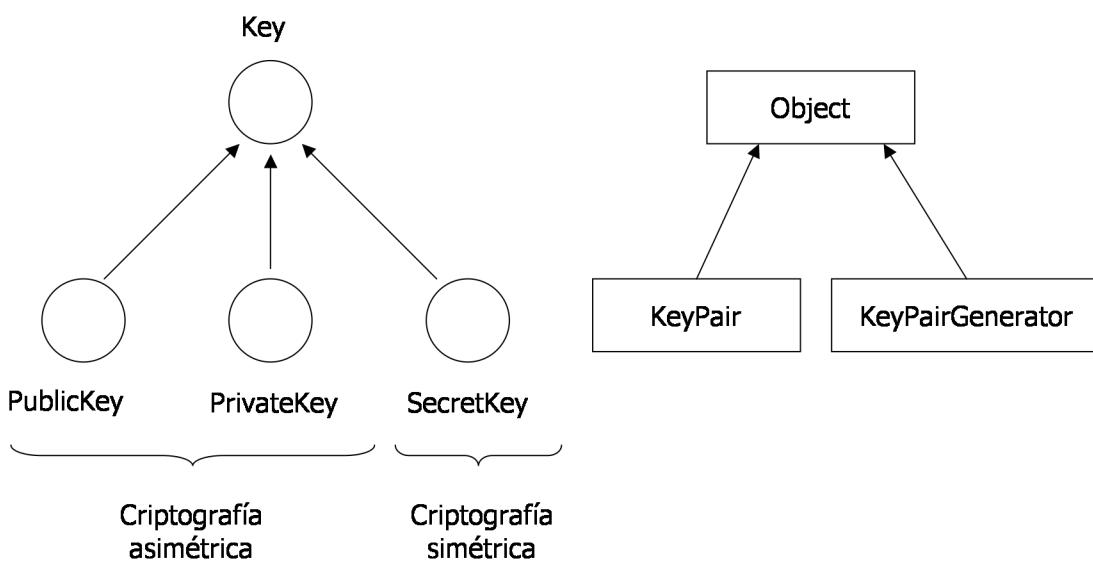


Figura 3.2 Clases de la criptografía asimétrica para representar las claves

`KeyPair` encapsula un grupo formado por una clave privada y una clave pública.

Tiene el constructor:

```
KeyPair(PublicKey publicKey, PrivateKey privateKey)
```

Y los métodos:

```
PublicKey <KeyPair> getPublicKey()
PrivateKey <KeyPair> getPrivateKey()
```

`PublicKey` es una interfaz derivada de `Key` que no añade más métodos y simplemente existe para tipificar.

Como muestra la Figura 3.3, en el paquete `java.security.interfaces.*` está la subinterface `RSAPublicKey` que añade métodos para acceder a información propia de RSA.

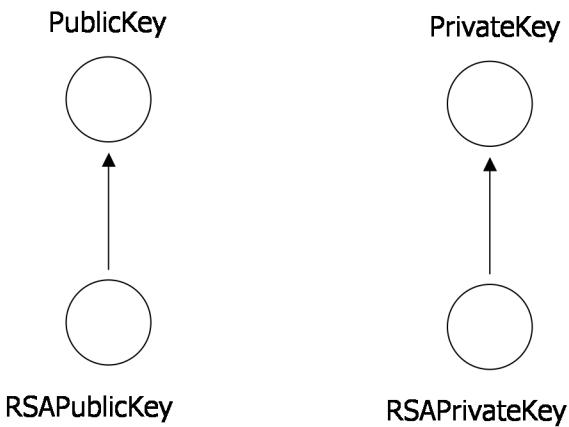


Figura 3.3 Tipos especiales de claves para RSA

`PrivateKey` es idéntica a `PublicKey`, excepto que representa una clave privada.

`java.security.interfaces.*` también contiene la clase `RSAPrivatekey` que tiene métodos para acceder a información propia de RSA.

`KeyPairGenerator` es equivalente a `KeyGenerator` en criptografía simétrica, es decir, nos proporciona una forma de generar una pareja de claves pública / privada.

Para instanciar un objeto de este tipo, como con cualquier otro servicio, usamos el método estático `getInstance()`, que en este caso tiene el prototipo:

```
static KeyPairGenerator <KeyPairGenerator> getInstance(
    String algorithm)
```

Y para generar un par de claves aleatorias usamos:

```
KeyPair <KeyPairGenerator> getKeyPair()
```

5.1 Encriptación de una clave de sesión

Como ya hemos visto en el apartado 2, la principal utilidad de la criptografía asimétrica es encriptar claves de sesión.

En este apartado vamos a comentar los pasos necesarios para encriptar una clave de sesión en Java, y vamos a hacer un programa que encripte una clave de sesión.

El programa se muestra en el Listado 3.2, y da básicamente los siguientes cinco pasos:

1) Empezamos creando una clave simétrica de 128 bits así:

```
KeyGenerator generador_clave_sesion =
    KeyGenerator.getInstance("Blowfish");
generador_clave_sesion.init(128);
Key clave_sesion = generador_clave_sesion.generateKey();
```

2) También tenemos que generar un par de claves RSA así:

```
KeyPairGenerator generador_clave_asimetrica =
    KeyPairGenerator.getInstance("RSA");
generador_clave_asimetrica.initialize(1024);
KeyPair par_claves =
    generador_clave_asimetrica.generateKeyPair();
```

3) Ahora creamos un cifrador RSA pidiendo ECB como modo y PKCS #1 como padding:

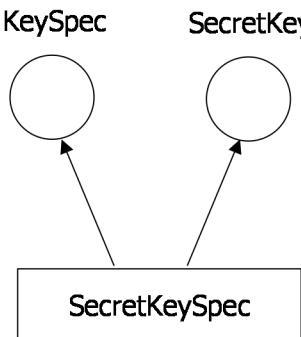
```
Cipher cifrador =
    Cipher.getInstance("RSA/ECB/PKCS1Padding");
cifrador.init(Cipher.ENCRYPT_MODE, par_claves.getPublic());
```

4) Para encriptar la clave simétrica tenemos que convertirla a un array de bytes usando `getEncoded()` así:

```
byte[] bytes_clave_sesion = clave_sesion.getEncoded();
byte[] clave_sesion_cifrada =
    cifrador.doFinal(bytes_clave_sesion);
```

5) Por último, para desencriptar podemos reiniciar el cifrador al modo de desencriptación y para reconstruir la clave de sesión creamos un objeto de tipo `SecretKeySpec`, que como muestra la Figura 3.4, es un tipo transparente que implementa la interfaz `KeySpec`, y que tiene el constructor:

```
SecretKeySpec(byte[] key, String algorithm)
```

**Figura 3.4** Jerarquía de SecretKeySpec

Pero a la vez `SecretKeySpec` implementa `SecretKey`, con lo que puede actuar como tipo opaco sin necesidad de pasar el objeto por un método que lo transforme de tipo opaco a tipo transparente, como pasaba con `PBEKeySpec` en el apartado 12 del Tema 2.

Es decir, la forma de desencriptar la clave de sesión encriptada sería:

```

cifrador.init(Cipher.DECRYPT_MODE, par_claves.getPrivate());
byte[] clave_descifrada =
    cifrador.doFinal(clave_sesion_cifrada);
Key clave_sesion_descifrada =
    new SecretKeySpec(clave_descifrada, "Blowfish");
if (clave_sesion.equals(clave_sesion_descifrada))
    System.out.println("Operacion correcta");
else
    System.out.println("Algo fue mal");
  
```

Por último, es importante destacar que si tenemos una máquina virtual JDK 1.3 o inferior necesitamos tener instalado el JCE de BouncyCastle, ya que SunJCE no soporta la encriptación RSA en estas implementaciones. Si tenemos una máquina virtual JDK 1.4 o superior este problema no se producirá.

```

/*
 * DESCRIPCION: Programa que encripta una clave de sesión
 * y la vuelve a desencriptar para comprobar su correcto
 * funcionamiento.
 */

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

public class EncriptaClaveSesion
{
    public static void main (String args[]) throws Exception
    {
        // Empezamos generando una clave secreta de
        // sesión para encriptar
  
```

```
System.out.println(
    "Generando clave de sesion...");
KeyGenerator generador_clave_sesion =
    KeyGenerator.getInstance("Blowfish");
generador_clave_sesion.init(128);
Key clave_sesion =
    generador_clave_sesion.generateKey();
System.out.println("Clave de sesion generada: "
    + clave_sesion.getAlgorithm());

// Ahora generamos una pareja de claves
// RSA de 1024 bits
KeyPairGenerator generador_clave_asimetrica =
    KeyPairGenerator.getInstance("RSA");
generador_clave_asimetrica.initialize(1024);
KeyPair par_claves =
    generador_clave_asimetrica.generateKeyPair();
System.out.println("Generada clave asimetrica.");

// Creamos el cifrador
Cipher cifrador =
    Cipher.getInstance("RSA/ECB/PKCS1Padding");
cifrador.init(Cipher.ENCRYPT_MODE
    ,par_claves.getPublic());

// Encriptamos la clave secreta
byte[] bytes_clave_sesion =
    clave_sesion.getEncoded();
byte[] clave_sesion_cifrada =
    cifrador.doFinal(bytes_clave_sesion);

// La volvemos a desencriptar
cifrador.init(Cipher.DECRYPT_MODE
    ,par_claves.getPrivate());
byte[] clave_descifrada =
    cifrador.doFinal(clave_sesion_cifrada);
Key clave_sesion_descifrada =
    new SecretKeySpec(clave_descifrada,"Blowfish");

if (clave_sesion.equals(clave_sesion_descifrada))
    System.out.println("Operacion correcta");
else
    System.out.println("Algo fue mal");
}
```

Listado 3.2: Programa que encripta y desencripta una clave se sesión

5.2 Encriptación de un fichero con RSA

Para acabar el tema vamos a hacer un programa que permite al emisor enviar ficheros a un receptor a través de un canal inseguro tal como muestra la Figura 3.5. Para ello el emisor encripta los ficheros antes de enviarlos usando la clave pública del receptor.

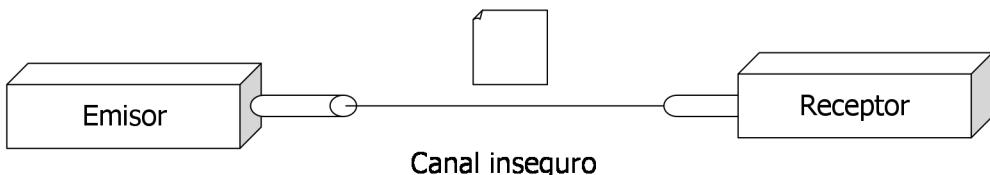


Figura 3.5 Envío de un fichero encriptado a través de un canal inseguro

Lógicamente, este protocolo también se podría usar para enviar mensajes a través de un socket.

El programa se muestra en los Listados 3.3 al 3.6 y consta de tres clases:

`CreaClaves` crea un par de claves privada / pública y guarda cada una en un fichero.

La clave privada se debe guardar siempre encriptada, para ello usaremos un algoritmo PBE. El contenido del fichero con la clave privada encriptada se muestra en la Figura 3.6.

Para poder guardar las claves en fichero, las convertimos en un array de bytes con:

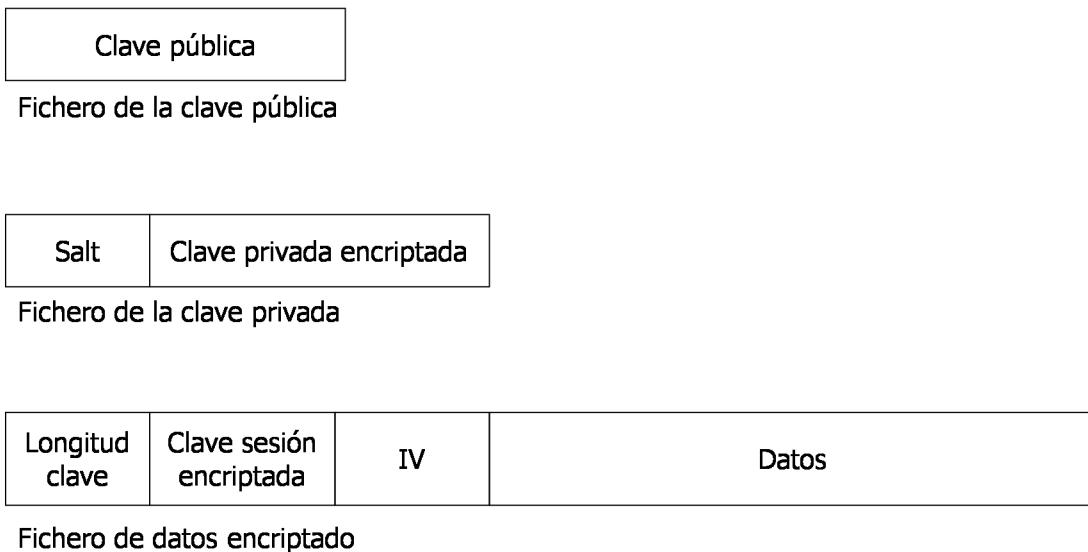
```
byte[] <Key>.getEncoded()
```

Las claves asimétricas tienen un encoded más complejo que las simétricas, aunque nosotros no vamos a estudiar el formato concreto de estos encoded, ya que Java oculta esta complejidad.

Lo que sí que vamos a explicar es que las claves públicas se codifican usando X.509, que es una especificación abierta para representar claves públicas y certificados.

Por el contrario, las claves privadas se codifican usando el estándar PKCS #8.

Igual que en la criptografía simétrica, `getEncoded()` es el método que se encarga de codificar las claves en su correspondiente formato y devolvérselas en un array de bytes.

**Figura 3.6** Campos de los ficheros que usará el programa

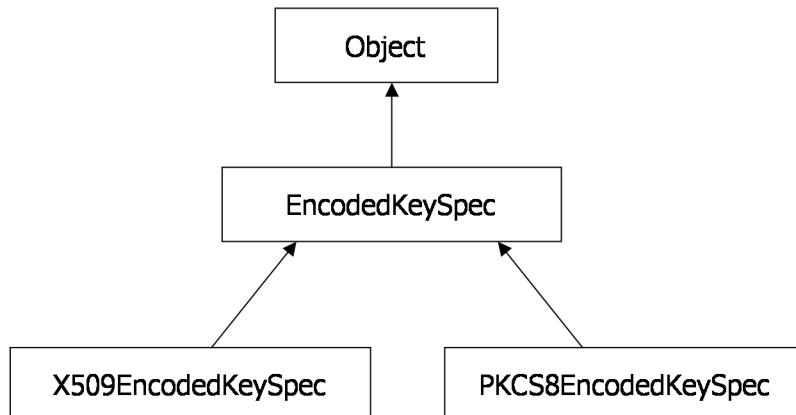
Para decodificar las claves pública y privada necesitamos usar una clase de tipo transparente distinto para cada una, que se muestran en la Figura 3.7

`java.security.spec.X509EncodedKeySpec` es la clase que nos permite descodificar una clave pública o certificado y para crear un objeto de este tipo usamos el constructor:

```
X509EncodedKeySpec(byte[] array)
```

`java.security.spec.PKCS8EncodedKeySpec` nos permite descodificar claves privadas, y para crear el objeto usamos el constructor:

```
PKCS8EncodedKeySpec(byte[] encoded)
```

**Figura 3.7** Clases para descodificar una clave pública y privada

Una vez que tengamos un objeto de uno de estos dos tipos, para volver a reconstruir la `PublicKey` y `PrivateKey`, se lo tenemos que pasar a un `KeyFactory`:

```
X509EncodedKeySpec clave_publica_spec =
    new X509EncodedKeySpec(encoded);
KeyFactory kf = KeyFactory.getInstance("RSA");
PublicKey clave_publica =
    kf.generatePublic(clave_publica_spec);
```

Y análogamente con la clave privada:

```
PKCS8EncodedKeySpec clave_privada_spec =
    new PKCS8EncodedKeySpec(buffer);
KeyFactory kf = KeyFactory.getInstance("RSA");
PrivateKey clave_privada =
    kf.generatePrivate(clave_privada_spec);
```

`EncriptaFichero` es el programa que usa el emisor para encriptar un fichero antes de mandárselo al receptor.

El emisor genera una clave de sesión con la que encripta el fichero y después encripta la clave de sesión con la clave pública. También el formato del fichero es el que se muestra en la Figura 3.6

Es decir, tal como muestra la Figura 3.8, el programa `EncriptaFichero` recibe el fichero de la clave pública y el fichero a encriptar, y devuelve el fichero encriptado.

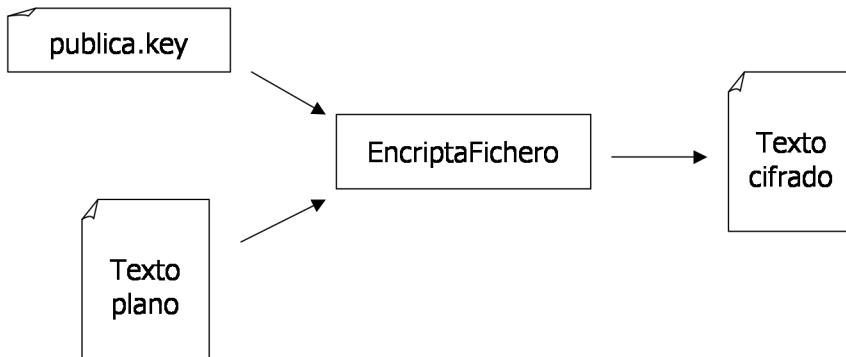
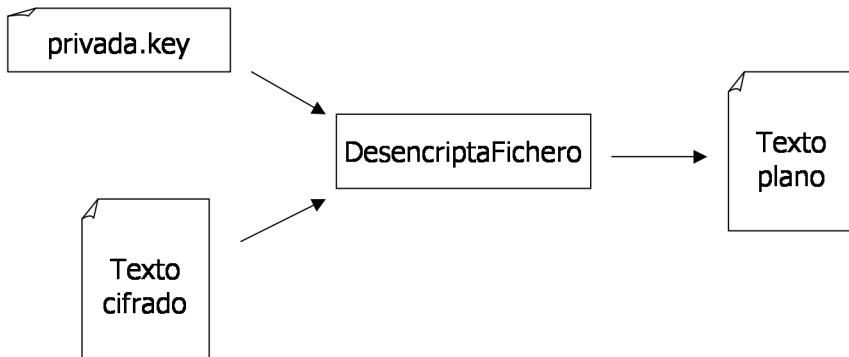


Figura 3.8 Proceso de encriptación realizado por `Encriptafichero`

`DesencriptaFichero` es el programa que usa el receptor para desencriptar el fichero. Tal como muestra la Figura 3.9, el programa recibe el fichero de la clave privada y el fichero a desencriptar, y devuelve un fichero con el texto plano desencriptado.

Reacuerde que la clave privada está protegida con password, luego este programa deberá pedir un password para desencriptarla.

**Figura 3.9** Proceso de desencriptación realizado por Desencriptafichero

```

public interface Constantes
{
    public static final int ITERACIONES_PBE = 1000;
    public static final int TAMANO_CLAVE_RSA = 1024;
    public static final int TAMANO_CLAVE_SESION = 128;
    public static final int TAMANO_SALT_BYTES = 8;
    public static final int TAMANO_IV_BYTES = 8;
}
  
```

Listado 3.3: Constantes del programa

```

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

public class CreaClaves implements Constantes
{
    public static void main(String[] args) throws Exception
    {
        // Generamos las claves publica/privada
        SecureRandom sr = new SecureRandom();
        sr.setSeed(new Date().getTime());
        System.out.println("Generando claves...");
        KeyPairGenerator kpg =
            KeyPairGenerator.getInstance("RSA");
        kpg.initialize(TAMANO_CLAVE_RSA,sr);
        KeyPair par_claves = kpg.generateKeyPair();
        System.out.println("Claves generadas");
        // Generamos el fichero de la clave publica
        System.out.print("Indique fichero para"
                        + " la clave publica:");
        BufferedReader teclado = new BufferedReader(
            new InputStreamReader(System.in));
        String fichero_publica;
        fichero_publica = teclado.readLine();
        FileOutputStream fos =
            new FileOutputStream(fichero_publica);
  
```

```

fos.write(par_claves.getPublic().getEncoded());
fos.close();
System.out.println(
    "Fichero con clave publica generado");
// Generamos el fichero de clave privada
System.out.print(
    "Indique fichero para la clave privada:");
String fichero_privada;
fichero_privada = teclado.readLine();
System.out.print("La clave privada debe estar"
    + " encriptada, indique password con la que"
    + " encriptarla:");
char[] password;
password = teclado.readLine().toCharArray();
// Encriptamos con un PBE
byte[] salt = new byte[TAMANO_SALT_BYTES];
sr.nextBytes(salt);
PBEKeySpec clave_pbe =
    new PBEKeySpec(password);
SecretKey clave_secreta_pbe =
    SecretKeyFactory.getInstance(
        "PBEWithSHAAndTwofish-CBC").generateSecret(
        clave_pbe);
PBEParameterSpec pbe_param =
    new PBEParameterSpec(salt, ITERACIONES_PBE);
Cipher cifrador_pbe = Cipher.getInstance(
    "PBEWithSHAAndTwofish-CBC");
cifrador_pbe.init(Cipher.ENCRYPT_MODE
    ,clave_secreta_pbe,pbe_param);
byte[] clave_privada_cifrada =
    cifrador_pbe.doFinal(
        par_claves.getPrivate().getEncoded());
fos = new FileOutputStream(fichero_privada);
fos.write(salt);
fos.write(clave_privada_cifrada);
fos.close();
System.out.println(
    "Fichero con clave privada generado");
}
}

```

Listado 3.4: Clase que crea las claves pública / privada

```

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

public class EncriptaFichero implements Constantes
{
    public static void main(String[] args) throws Exception
    {

```

```
// Pedimos el fichero a encriptar
// y fichero de clave publica a usar
BufferedReader teclado = new BufferedReader(
    new InputStreamReader(System.in)));
System.out.print(
    "Indique fichero a encriptar:");
String fichero_encriptar = teclado.readLine();
if (!new File(fichero_encriptar).exists())
{
    System.out.println("El fichero "
        +fichero_encriptar+" no existe");
    return;
}
String fichero_encriptado = fichero_encriptar
    + ".crypto";
System.out.print("Indique que fichero tiene la"
    + " clave publica a usar:");
String fichero_publica = teclado.readLine();

// Recuperamos la clave publica
FileInputStream fis =
    new FileInputStream(fichero_publica);
byte[] buffer = new byte[fis.available()];
fis.read(buffer);
X509EncodedKeySpec clave_publica_spec =
    new X509EncodedKeySpec(buffer);
KeyFactory kf = KeyFactory.getInstance("RSA");
PublicKey clave_publica =
    kf.generatePublic(clave_publica_spec);

// Generamos el fichero encriptado
SecureRandom sr = new SecureRandom();
sr.setSeed(new Date().getTime());
fis = new FileInputStream(fichero_encriptar);
DataOutputStream dos = new DataOutputStream(
    new FileOutputStream(fichero_encriptado));
// 1. Generamos una clave de sesion
System.out.println(
    "Generando clave de sesion...");
KeyGenerator kg =
    KeyGenerator.getInstance("Blowfish");
kg.init(TAMANO_CLAVE_SESION,sr);
SecretKey clave_sesion =
    (SecretKey)kg.generateKey();
// 2. Guardamos la clave de sesion
// encriptada en el fichero
System.out.println(
    "Guardando la clave de sesion encriptada...");
Cipher cifrador_rsa = Cipher.getInstance(
    "RSA/ECB/PKCS1Padding");
cifrador_rsa.init(Cipher.ENCRYPT_MODE
    ,clave_publica,sr);
buffer = cifrador_rsa.doFinal(
    clave_sesion.getEncoded());
dos.writeInt(buffer.length);
```

```

dos.write(buffer);
// 3. Generamos un IV aleatorio
byte[] IV = new byte[TAMANO_IV_BYTES];
sr.nextBytes(IV);
IvParameterSpec iv_spec =
    new IvParameterSpec(IV);
dos.write(IV);
// 4. Guardamos los datos encriptados
// en el fichero
System.out.println("Guardando "
    + fichero_encriptar
    + " en el fichero encriptado"
    + fichero_encriptado);
Cipher cifrador_fichero = Cipher.getInstance(
    "Blowfish/CBC/PKCS5Padding");
cifrador_fichero.init(Cipher.ENCRYPT_MODE
    , clave_sesion, iv_spec, sr);
CipherOutputStream cos =
    new CipherOutputStream(dos, cifrador_fichero);
int b = fis.read();
while (b!=-1)
{
    cos.write(b);
    b = fis.read();
}
fis.close();
cos.close();
dos.close();
System.out.println(
    "Fichero encriptado correctamente");
}
}

```

Listado 3.5: Programa que encripta los ficheros

```

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

public class DesencriptaFichero implements Constantes
{
    public static void main(String[] args) throws Exception
    {
        // Pedimos el fichero a desencriptar
        // y fichero de clave privada a usar
        BufferedReader teclado = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.print(
            "Indique fichero a desencriptar:");
        String fichero_encriptado = teclado.readLine();
        if (!new File(fichero_encriptado).exists())
        {

```

```
        System.out.println("El fichero "
                           +fichero_encriptado+" no existe");
        return;
    }
    if (!fichero_encriptado.toLowerCase().endsWith(
        ".crypto"))
    {
        System.out.println("La extension de los"
                           + "ficheros encriptados debe ser .crypto");
        return;
    }
    String fichero_desencriptado =
        fichero_encriptado.substring(0
        ,fichero_encriptado.length()-
        ".crypto".length());
    System.out.print("Indique que fichero tiene la"
                   + " clave privada a usar:");
    String fichero_privada = teclado.readLine();
    System.out.print("Indique password con que se "
                   + "encripto el fichero "+fichero_privada+":");
    char[] password;
    password = teclado.readLine().toCharArray();
    // Recuperamos la clave privada
    System.out.println(
        "Recuperando clave privada...");
    SecureRandom sr = new SecureRandom();
    sr.setSeed(new Date().getTime());
    FileInputStream fis =
        new FileInputStream(fichero_privada);
    byte[] buffer = new byte[TAMANO_SALT_BYTES];
    fis.read(buffer);
    PBEKeySpec clave_pbe_spec =
        new PBEKeySpec(password);
    SecretKey clave_pbe =
        SecretKeyFactory.getInstance(
            "PBEWithSHAAndTwofish-CBC"
        ).generateSecret(clave_pbe_spec);
    PBEParameterSpec param_pbe_spec =
        new PBEParameterSpec(buffer,ITERACIONES_PBE);
    Cipher descifrador_pbe = Cipher.getInstance(
        "PBEWithSHAAndTwofish-CBC");
    descifrador_pbe.init(Cipher.DECRYPT_MODE
        ,clave_pbe,param_pbe_spec,sr);
    buffer = new byte[fis.available()];
    fis.read(buffer);
    buffer = descifrador_pbe.doFinal(buffer);
    PKCS8EncodedKeySpec clave_privada_spec =
        new PKCS8EncodedKeySpec(buffer);
    KeyFactory kf = KeyFactory.getInstance("RSA");
    PrivateKey clave_privada =
        kf.generatePrivate(clave_privada_spec);
    System.out.println("Clave secreta recuperada");
    // Generamos el fichero desencriptado
    DataInputStream dis = new DataInputStream(
        new FileInputStream(fichero_encriptado));
```

```
FileOutputStream fos =
    new FileOutputStream(fichero_desencriptado);
// 1. Recuperamos la clave de sesión
System.out.println(
    "Generando el fichero desencriptado...");
int longitud = dis.readInt();
buffer = new byte[longitud];
dis.read(buffer);
// 2. Desencriptamos la clave de sesión
Cipher descifrador_rsa =
    Cipher.getInstance("RSA/ECB/PKCS1Padding");
descifrador_rsa.init(
    Cipher.DECRYPT_MODE, clave_privada, sr);
buffer = descifrador_rsa.doFinal(buffer);
SecretKeySpec clave_sesion =
    new SecretKeySpec(buffer, "Blowfish");
// 3. recuperamos el IV
byte[] IV = new byte[TAMANO_IV_BYTES];
dis.read(IV);
IvParameterSpec iv_spec =
    new IvParameterSpec(IV);
// 4. Desencriptamos y vamos generando el
// fichero desencriptado
System.out.println("Guardando "
    + fichero_encriptado
    + " en el fichero encriptado "
    + fichero_desencriptado);
Cipher cifrador_fichero =
    Cipher.getInstance(
        "Blowfish/CBC/PKCS5Padding");
cifrador_fichero.init(Cipher.DECRYPT_MODE
    , clave_sesion, iv_spec, sr);
CipherOutputStream cos = new CipherOutputStream(
    fos, cifrador_fichero);
int b = dis.read();
while (b != -1)
{
    cos.write(b);
    b = dis.read();
}
dis.close();
cos.close();
fos.close();
System.out.println(
    "Fichero desencriptado correctamente");
}
```

Listado 3.6: Programa que desencripta los ficheros

Tema 4

Firmas digitales

Sinopsis:

En este tema empezaremos viendo cómo se implementan las funciones hash y MAC (que vimos en el apartado 9 del Tema 1) en Java, para luego pasar a estudiar qué son, para qué sirven, cómo se usan las firmas digitales, y posibles problemas de seguridad con éstas. Acabaremos el tema explicando qué clases proporciona Java para gestionar las firmas digitales.

1 Funciones hash y MAC en Java

1.1 Funciones hash

En el apartado 9 del Tema 1 ya vimos qué eran las funciones hash (también llamadas one-way-functions) y los MAC (Message Authentication Codes). En este apartado vamos a empezar viendo cómo se aplican estas ideas al lenguaje de programación Java.

Uno de los engines o servicios de Java es el que nos permite crear funciones hash, a las que en Java se suelen llamar **message digest**. A este servicio se accede a través de la clase `java.security.MessageDigest`.

Como todos los servicios, `MessageDigest` es una clase abstracta de la cual se obtiene un objeto a través de los métodos estáticos:

```
static MessageDigest <MessageDigest> getInstance(
    String algorithm)
static MessageDigest <MessageDigest> getInstance(
    String algorithm, String provider)
```

El JCA de Sun tiene dos algoritmos que implementan este servicio: "SHA1" y "MD5".

Una vez que tengamos el objeto `MessageDigest`, le podemos pasar el texto a firmar con los métodos:

```
void <MessageDigest> update(byte input)
void <MessageDigest> update(byte[] input)
```

Y cuando queramos calcular el digest del texto usamos:

```
byte[] <MessageDigest> digest()
```

que nos devuelve el digest del texto pasado.

Una vez que el digest se ha calculado, el estado interno del objeto se reinicia para que podamos calcular el digest de otro texto.

Se supone que en recepción se calcularían igualmente los digest, y una vez calculados usaríamos:

```
static boolean <MessageDigest> isEqual(byte[] A, byte[] B)
```

para comprobar si los digest son iguales. Se considera que dos digest son iguales si tienen la misma longitud y los mismos datos.

El método `digest()` devuelve siempre un número fijo de bytes que podemos saber de antemano con el método:

```
int <MessageDigest> getDigestLength()
```

1.2 MAC (Message Authentication Codes)

Como explicamos en el apartado 9.2 del Tema 1, para garantizar la integridad de un mensaje de texto plano podemos usar los MAC.

Para calcular MACs en Java tenemos el engine o servicio `javax.crypto.Mac`. Podemos obtener un objeto de este tipo usando:

```
static Mac <Mac> getInstance(String algorithm)
static Mac <Mac> getInstance(String algorithm,
                           String provider)
```

El JCE de Sun tiene dos algoritmos que implementan este servicio: "HmacSHA1" y "HmacMD5"

Este objeto es muy parecido a `MessageDigest`, sólo que ahora debemos inicializar previamente el objeto usando:

```
void <Mac> init(SecretKey key)
```

Y al final en vez de llamar a `digest()` llamamos a:

```
byte[] <Mac> doFinal()
```

Obsérvese que curiosamente `java.security.MessageDigest` está en el JCA, mientras que `javax.crypto.Mac` está en el JCE (tenía restricciones de exportación), lo cual es muy curioso porque realmente la clase `Mac` no es muy necesaria, ya que podemos crear un MAC a partir de un `MessageDigest` de forma muy fácil, ya que sólo hay que concatenar la clave al mensaje.

1.3 Message Digest Streams

Las llamadas al método `update()` de la clase `MessageDigest` implica obtener los datos como un array de bytes.

En ocasiones es más cómodo utilizar un stream que hayamos obtenido a un fichero o a un socket. En este caso podemos usar las clases:

```
java.security.DigestInputStream
java.security.DigestOutputStream
```

Estas clases, como muestra la Figura 4.1, actúan como filtros sobre datos que están circulando sobre ellas.

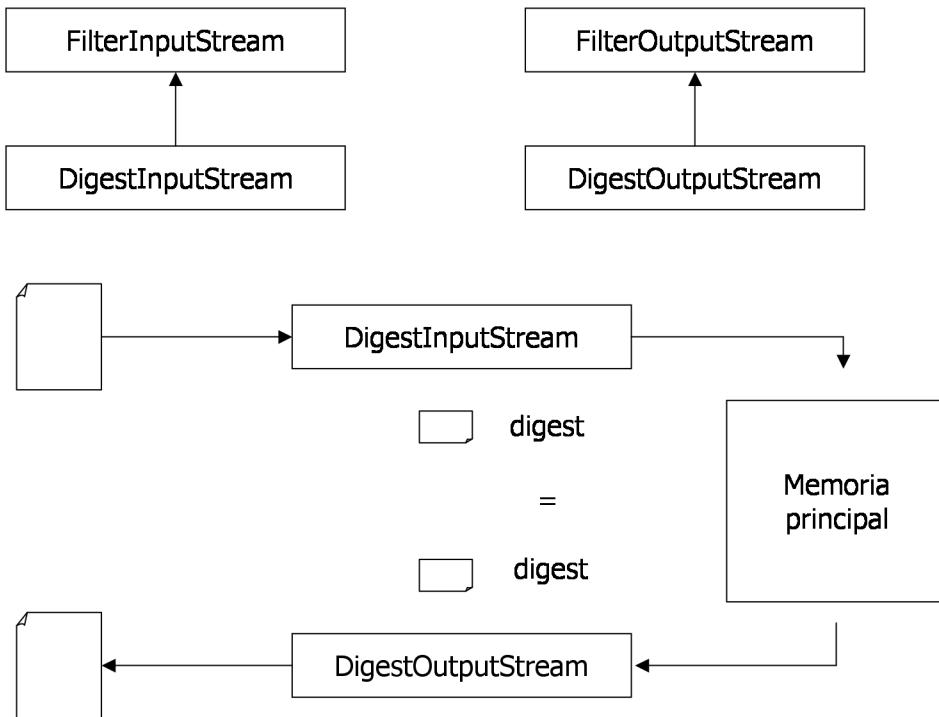


Figura 4.1 Clases filtro para calcular el hash de los datos de un stream

Cada objeto stream se crea a partir de dos objetos:

- Un stream
- Un objeto `MessageDigest`

Luego tienen los constructores:

```
DigestInputStream(InputStream is, MessageDigest md)
DigestOututStream(OutputStream os, MessageDigest md)
```

Según van pasando los datos por el stream se van calculando los digest, y al final podemos obtener el digest calculado con el método:

```
byte[] <MessageDigest> digest()
```

del objeto pasado al filtro. Una vez que llamamos a `digest()` se reinicia el `MessageDigest` igual que en el apartado anterior.

Por ejemplo, `DigestInputStream` se podría usar para ir calculando el digest de un fichero según lo vamos cargando a memoria del programa de forma que al terminar de leerlo ya tenemos calculado su digest, mientras que

`DigestOutputStream` lo usaríamos para ir calculando el digest de un fichero según lo vamos escribiendo en fichero de forma que al acabar de escribirlo ya tenemos calculado su digest.

También podemos habilitar o deshabilitar el cálculo de los digest de los datos que circulan por el filtro llamando a:

```
void <DigestInputStream> on(boolean on)
void <DigestOutputStream> on(boolean on)
```

Para acabar de ver las funciones hash en Java, el Listado 4.1 y el Listado 4.2 muestran un programa que calcula el MAC de un fichero usando la clase `MessageDigest`. Para ello vamos a hacer una clase Java llamada `CalculaMAC` que calcula el MAC de un fichero y lo guarda en otro fichero llamado igual que el original pero añadiéndole la extensión `.mac` tal como muestra la Figura 4.2. El digest se calcula como la suma del fichero más un password que pide por pantalla.

```
import java.io.*;
import java.security.*;

public class CalculaMAC
{
    public static void main(String[] args) throws Exception
    {
        if (args.length!=1)
        {
            System.err.println(
                "Indique el fichero al que calcular su MAC");
            return;
        }
        System.out.print("Indique passphrase:");
        BufferedReader teclado = new BufferedReader(
            new InputStreamReader(System.in));
        String passphrase = teclado.readLine();
        FileInputStream fis =
            new FileInputStream(args[0]);
        FileOutputStream fos =
            new FileOutputStream(args[0]+".mac");
        MessageDigest md = MessageDigest.getInstance("SHA");
        DigestInputStream dis =
            new DigestInputStream(fis,md);
        md.update(passphrase.getBytes());
        while (dis.read() != -1);
        fos.write(md.digest());
        fis.close();
        fos.close();
    }
}
```

Listado 4.1: Programa que calcula el MAC de un fichero

carta.txt

carta.txt.mac

Figura 4.2 Fichero con mensaje y otro con su MAC

Después vamos a hacer otro programa llamado `CompruebaMAC` que le sirve al receptor del fichero para comprobar que éste no fue modificado por ningún atacante.

```
import java.io.*;
import java.security.*;

public class CompruebaMAC
{
    public static void main(String[] args) throws Exception
    {
        if (args.length!=1)
        {
            System.err.println(
                "Indique el fichero al comprobar");
            return;
        }
        File f = new File(args[0]+".mac");
        if (!f.exists())
        {
            System.err.println(
                "No existe el fichero .mac correspondiente");
            return;
        }
        System.out.print("Indique passphrase:");
        BufferedReader teclado = new BufferedReader(
            new InputStreamReader(System.in));
        String passphrase = teclado.readLine();
        FileInputStream fis =
            new FileInputStream(args[0]);
        FileInputStream fismac = new FileInputStream(f);
        MessageDigest md = MessageDigest.getInstance("SHA");
        DigestInputStream dis =
            new DigestInputStream(fis,md);
        md.update(passphrase.getBytes());
        while (dis.read() != -1);
        byte[] mac_almacenado = new byte[fismac.available()];
        fismac.read(mac_almacenado);
        if (MessageDigest.isEqual(md.digest()
            ,mac_almacenado))
            System.out.println("El MAC es correcto");
        else
            System.out.println("El MAC no coincide");
        fis.close();
        fismac.close();
    }
}
```

Listado 4.2: Programa que comprueba el MAC de un fichero

2 Firmas digitales

Las firmas digitales, al igual que las firmas normales, se usan para conseguir básicamente cuatro objetivos:

1. **Integridad.** Una vez que el documento se firma, si luego se modifica se detecta el cambio.
2. **Autenticidad.** La firma convence al receptor del documento de que está firmado por quien aparece como firmante.
3. **Irreutilizabilidad.** Nadie puede copiar la firma de un documento a otro documento, y si lo hace la firma se detecta como inválida.
4. **Irrepudiabilidad.** El firmante no puede luego alegar que él no firmó el documento.

Estos cuatro objetivos también se pueden lograr con los MAC, las firmas digitales añaden el uso de criptografía de clave pública para evitar que, entre emisor y receptor, tengan que compartir una clave secreta.

2.1 Firma de documentos con criptografía de clave secreta y árbitro

En este apartado vamos a ver cómo se puede implementar un sistema de firmas digitales usando sólo criptografía de clave secreta y un árbitro.

Supongamos que A quiere firmar un documento y enviárselo a B . También supongamos que el árbitro comparte una clave secreta K_A con A y otra clave secreta K_B con B .

En este escenario, el protocolo para firmar documentos digitalmente podría ser:

1. A encripta el mensaje que quiere enviar a B firmado usando K_A y se lo envía al árbitro.
2. El árbitro desencripta el mensaje usando K_A y lo guarda en su base de datos.
3. El árbitro encripta el mensaje plano usando K_B y se lo envía a B .
4. B desencripta el mensaje con K_B .

El árbitro sabe que el mensaje proviene de A porque es el único que comparte la clave secreta K_A con él.

B está seguro de que el documento recibido es auténtico ya que se lo ha enviado encriptado el árbitro, que es el único que (además de él mismo) conoce K_B .

Este protocolo cumple con los cuatro objetivos que se consiguen con un protocolo de firma digital:

1. **Integridad.** Si el mensaje se modifica al enviarlo de A al árbitro o del árbitro a B , el proceso de desencriptación falla.
2. **Autenticidad.** Como B confía en el árbitro, y es el único que comparte con el K_B , sabe que el mensaje procede de A .
3. **Irreutilizabilidad.** Si B muestra mensajes firmados que A dice no haber firmado, pueden ir al árbitro que resuelve la disputa consultando su base de datos.
4. **Irrepudiabilidad.** Si A alega no haber firmado un documento que sí ha firmado, B puede ir al árbitro que determina que el documento sí que está en la base de datos.

Si B quiere demostrar a C la autenticidad de un documento firmado por A pueden seguir el siguiente protocolo:

1. B encripta el mensaje con K_B y se lo envía al árbitro indicándole que quiere demostrar a C que el documento fue firmado por A .
2. El árbitro desencripta el mensaje y comprueba en su base de datos que el documento fue firmado por A .
3. El árbitro vuelve a encriptar el mensaje con K_C y se lo envía a C .
4. C desencripta el mensaje.

C confía en el árbitro, y sabe que el mensaje es auténtico porque sólo C y el árbitro conocen la clave secreta K_C .

Como acabamos de ver, este protocolo funciona pero presenta dos inconvenientes:

- Debe existir un árbitro en el cual deben confiar todos.
- El árbitro se convierte en un cuello de botella.

Estos problemas los van a resolver los algoritmos criptográficos de firma con clave pública que vamos a ver en el siguiente apartado.

2.2 Firma de documentos con criptografía de clave pública

La criptografía de clave pública es la técnica que se suele usar para generar firmas digitales. Tiene básicamente dos ventajas respecto a la técnica anterior:

- No necesita un árbitro.
- La clave privada, a diferencia de la clave secreta, no la tenemos que compartir con nadie.

Recuérdese que en el Tema 3 vimos que para la confidencialidad de datos se usaba la clave pública para encriptar y la clave privada para desencriptar. Para firmar digitalmente un documento con técnicas de criptografía asimétrica, tal como pretende mostrar la Tabla 4.1, se utiliza la clave privada para firmar, y la clave pública para comprobar la firma.

	Clave privada	Clave pública
Confidencialidad	Desencriptar	Encriptar
Firma digital	Firmar	Verificar

Tabla 4.1: Uso de las claves públicas/privadas para confidencialidad y firma digital

Realmente firmar es equivalente a encriptar con la clave privada, y verificar es equivalente a desencriptar con la clave pública, con lo que a veces al proceso de firmar se le llama "encriptar con clave privada", y al de verificar "desencriptar con la clave pública".

El protocolo que se sigue para firmar digitalmente un documento es:

1. *A* encripta el documento con su clave privada, dando lugar al documento firmado.
2. *A* envía el documento original y su firma a *B*.
3. *B* desencripta el documento usando la clave pública de *A*, y verifica el documento, para ello el documento firmado desencriptado debe ser igual al documento original. Es decir, si $S(K_{privadaA}, M)$ es la firma (sign) del mensaje M usando la clave privada de *A* ($K_{privadaA}$), y $V(K_{públicaA}, M)$ es la verificación del documento M usando la clave pública de *A*, se cumple que:

$$\begin{aligned} V(K_{públicaA}, S(K_{privadaA}, M)) &= M \\ D(K_{públicaA}, E(K_{privadaA}, M)) &= M \end{aligned}$$

Siendo $E()$ la función de encriptación y $D()$ la función de desencriptación.

Obsérvese que este protocolo, al igual que el anterior, también satisface los cuatro objetivos de las firmas digitales:

1. **Integridad.** Si se modifica el documento, la firma no coincide con la del documento y se detecta el cambio. Es decir, si un atacante modifica el documento tendría que modificar también su firma para evitar que se detecte el cambio, pero el atacante no conoce la clave privada necesaria para recalcular la firma.
2. **Autenticidad.** El receptor sabe que lo ha firmado quien lo ha firmado porque es el único que conoce la clave privada.

3. **Irreutilizabilidad.** Nadie puede copiar la firma de un documento a otro documento, y si lo hace la firma se detecta como inválida.
4. **Irrepudiabilidad.** El firmante no puede luego alegar que él no firmó el documento, porque él es el único que conoce la clave privada.

Respecto a las principales aplicaciones que tienen hoy en día las firmas digitales, vamos a comentar cuatro de ellas:

1. **Firmar e-mails.** Con S/MIME o PGP podemos firmar e-mails usando una clave privada, y luego el receptor del e-mail puede comprobar su autenticidad usando una clave pública.
2. **Firmar un contrato.** Actualmente en España se ha introducido la firma digital para todos los ciudadanos de forma gratuita. La Fabrica Nacional de Moneda y Timbre FNMT es el organismo encargado de emitir certificados digitales para los ciudadanos que lo quieran solicitar. La validez de estos certificados ante un tribunal está avalada en el Real Decreto Ley 14/1999 publicado en el B.O.E. num.224 de 18-09-99.
3. **Crear servidores seguros.** Podemos diseñar servidores que firmen la información que sirven con el fin de que un atacante no la pueda modificar. Por ejemplo, los servidores DNS seguros firman los nombres DNS que resuelven.
4. **Identificación de usuarios frente al servidor.** Una forma más segura que la identificación por password es que el usuario al mandar una petición al servidor la firme digitalmente.

2.3 Algoritmos de firmas digitales más usados

En esta sección vamos a comentar cuáles son los algoritmos de firmas digitales más usados. En concreto nos vamos a centrar en las diferencias entre estos dos:

1. RSA, que además de para criptografía de clave pública se puede usar para firmar digitalmente.
2. DSA (Digital Signature Algorithm), que es un algoritmo propuesto en 1991 por el NIST (National Institute for Standards and Technology).

DSA se diseñó con el fin de que sólo se pudiera usar para firmar, pero no para encriptar, lo cual solucionaba dos problemas:

- Las restricciones de exportación de algoritmos criptográficos de los EEUU, ya que éstas no afectan a las firmas digitales.
- Evitar tener que pagar las tasas de la patente de RSA, que hasta entonces era el algoritmo que se estaba usando.

Estos problemas actualmente han dejado de serlo ya que EEUU ha levantado las restricciones de exportación de software criptográfico, y la patente de RSA también ha terminado.

También sería recomendable que quedara clara la diferencia entre dos términos muchas veces confundidos:

DSS (Digital Signature Standard) es el estándar propuesto por el NIST para firmar digitalmente, dentro del cual se propone usar un algoritmo llamado DSA (Digital Signature Algorithm). Es decir, el algoritmo es parte del estándar, aunque normalmente se habla de DSA y no de DSS.

Una diferencia que hay entre RSA y DSA está en lo que respecta a la velocidad: DSA es más rápido para generar la firma que RSA, pero RSA es más rápido validando la firma. Como una firma se valida más veces que se crea, de acuerdo a este criterio sería mejor RSA.

2.4 Firmar documentos con timestamp

Las firmas digitales son susceptibles al replay attack. Para ver esto imaginemos que *A* firma digitalmente un cheque a *B*. Ahora *B* va con el documento al banco a cobrar el cheque, el banco comprueba la firma y paga el cheque.

Pero *B* se ha hecho una copia del documento, y una semana después va al banco y vuelve a cobrar el cheque.

Este problema no se produce con los cheques normales porque son difíciles de duplicar, pero sí con los documentos digitales, que por su naturaleza son fácilmente copiables.

Para evitarlo, *A* puede poner un timestamp en el cheque de forma que el banco apunta el timestamp del cheque pagado, y así evita pagarla varias veces.

2.5 Firmar el hash de un documento

En la práctica, la criptografía de clave pública es muy lenta para firmar documentos largos, con lo que lo que se suele hacer para ahorrar tiempo es firmar el hash de un documento, que lógicamente es mucho más pequeño.

El protocolo que se sigue es el siguiente:

1. *A* calcula el hash del documento a firmar.

2. A encripta (firma) el hash del documento (firma) usando su clave privada.
3. A envía el documento y el hash firmado a B .
4. B calcula el hash del documento.
5. B usando la clave pública de A desencripta (verifica) el hash.
6. B verifica el documento comprobando que el hash del documento coincide con la desencriptación del hash.

2.6 Múltiples firmas

Un documento puede firmarse por varios firmantes, para ello cada firmante firma el hash del documento usando su clave privada, y después el receptor simplemente comprueba cada uno de los hash firmados.

2.7 Firmas digitales y encriptación

Las firmas digitales se pueden combinar con la encriptación para conseguir tanto confidencialidad como autenticidad. En este caso el protocolo que se suele seguir es que primero se firma el mensaje y luego se encripta. Nada nos impide hacerlo al revés, pero es más natural hacerlo de esta forma, ya que en el mundo real también actuamos así: primero firmamos una carta y luego la metemos en un sobre cerrado.

El protocolo sería el siguiente:

1. A firma el mensaje con su clave privada $S(K_{privadaA}, M)$.
2. A encripta el mensaje firmado con la clave pública de B y se lo envía:

$$E(K_{públicaB}, S(K_{privadaA}, M))$$

3. B desencripta el mensaje con su clave privada:

$$D(K_{privadaB}, E(K_{públicaB}, S(K_{privadaA}, M))) = S(K_{privadaA}, M)$$

4. B verifica el mensaje usando la clave pública de A y recupera el mensaje:

$$V(K_{públicaA}, S(K_{privadaA}, M)) = M$$

2.8 Ataques contra las firmas digitales RSA

El ataque más conocido contra las firmas digitales RSA se basa en el hecho de que los algoritmos de encriptación y firma son los mismos, y los procesos de

desencriptación y verificación también usan el mismo algoritmo. Lo que los diferencia es que aunque el algoritmo es el mismo, la clave usada para cada fin es distinta, tal como muestra la Tabla 4.2.

	Modo del algoritmo	Clave
Encriptación	Encriptación	Clave pública del receptor
Firma	Encriptación	Clave privada del emisor
Desencriptación	Desencriptación	Clave privada del receptor
Verificación	Desencriptación	Clave pública del emisor

Tabla 4.2: Claves usadas para los procesos de encriptación y firma

Basándose en este hecho, un ataque muy conocido es **el ataque de los recibos**, que permite a un atacante desencriptar un mensaje M enviado de A a B , usando encriptación y firmas digitales RSA.

El protocolo que vamos a atacar es el siguiente:

1. A firma un mensaje con su clave privada, lo encripta con la clave pública de B , y se lo envía a B :

$$E(K_{públicaB}, S(K_{privadaA}, M))$$

2. B desencripta el mensaje con su clave privada, y verifica la firma con la clave pública de A , con lo que comprueba que el mensaje enviado por A es auténtico.

$$V(K_{públicaA}, D(K_{privadaB}, E(K_{públicaB}, S(K_{privadaA}, M)))) = M$$

3. B ahora tiene que enviar un recibo a A , luego firma el mensaje con su clave privada, lo encripta con la clave pública de A , y se lo envía a A .

$$E(K_{públicaA}, S(K_{privadaB}, M))$$

4. A desencripta el mensaje con su clave privada, y verifica la firma con la clave pública de B . Si el mensaje resultante es el mismo que envío a B , sabe que B recibió el mensaje correctamente.

$$V(K_{públicaB}, D(K_{privadaA}, E(K_{públicaA}, S(K_{privadaB}, M))))$$

No es raro imaginar que muchos sistemas de mensajería se podrían implementar así.

Ahora veamos cómo H , el atacante que es un usuario legítimo del sistema (con sus claves pública/privada), se las apaña para leer el mensaje encriptado que envía A a B :

1. H captura el mensaje encriptado que A envía a B en el primer paso.
2. Después H envía el mensaje a B indicando que el autor es él.
3. B piensa que es un mensaje legítimo de H , con lo que lo desencripta con su clave privada, y después verifica la firma de H , que lógicamente no será correcta porque lo firmó A . Lo que tenemos en este momento es:

$$\begin{aligned} V(K_{públicaH}, D(K_{privadaB}, E(K_{públicaB}, S(K_{privadaA}, M)))) = \\ V(K_{públicaH}, S(K_{privadaA}, M)) \end{aligned}$$

Obsérvese que B ha quitado la encriptación al mensaje, luego ha roto la seguridad.

4. B no obtiene más que basura y una firma incorrecta, pero supongamos que el sistema procede a enviar el recibo. Entonces B firma el mensaje con su clave privada y lo encripta con la clave pública de H . Luego tenemos:

$$\begin{aligned} E(K_{públicaH}, S(K_{privadaB}, V(K_{públicaH}, S(K_{privadaA}, M)))) = \\ E(K_{públicaH}, E(K_{privadaB}, D(K_{públicaH}, E(K_{privadaA}, M)))) \end{aligned}$$

Ya que $E()=S()$ y $D()=V()$.

Ahora el criptoanalista H no tiene más que quitar los envoltorios que protegen al mensaje. Obsérvese que en el mensaje que tenemos en este momento H puede quitarle todas las operaciones que estén hechas con una clave privada, ya que sólo tiene que aplicar la operación opuesta con la clave pública correspondiente (que todo el mundo conoce). Es decir, para quitar $E(K_{privadaB}, X)$ usaría $D(K_{públicaB}, X)$, y para quitar $E(K_{privadaA}, X)$ usaría $D(K_{públicaA}, X)$. Quitar una operación hecha con clave pública en principio es más difícil, ya que hay que tener la correspondiente clave privada, pero en el caso de $D(K_{públicaH}, X)$ sí que puede quitarla H porque conoce su propia clave privada. Luego H acaba quitando todos los envoltorios al mensaje de la siguiente forma:

5. H desencripta el mensaje con su clave privada y obtiene:

$$\begin{aligned} D(K_{privadaH}, E(K_{privadaH}, E(K_{privadaB}, D(K_{privadaH}, E(K_{privadaA}, M))))) = \\ E(K_{privadaB}, D(K_{privadaH}, E(K_{privadaA}, M))) \end{aligned}$$

6. H desencripta el mensaje con la clave pública de B :

$$\begin{aligned} D(K_{públicaB}, E(K_{privadaB}, D(K_{privadaH}, E(K_{privadaA}, M)))) = \\ D(K_{privadaH}, E(K_{privadaA}, M)) \end{aligned}$$

7. H encripta el mensaje con su clave privada:

$$E(K_{privadaH}, D(K_{privadaH}, E(K_{privadaA}, M))) = E(K_{privadaA}, M)$$

8. Por último H desencripta el mensaje con la clave pública de A :

$$D(K_{públicaA}, E(K_{privadaA}, M)) = M$$

Y ya tiene el mensaje original

Para evitar este ataque, si estamos trabajando con RSA, simplemente debemos usar pares de clave privada / clave pública distintos para encriptar y firmar. DSA no tiene este problema ya que sólo sirve para firmar, y no es simétrico como RSA.

3 Firmas digitales en Java

La creación y verificación de firmas digitales es otro servicio o engine de Java.

Al no estar las firmas digitales sujetas a restricciones de exportación se distribuyen dentro del JCA.

3.1 La clase Signature

Los objetos de esta clase sirven para firmar un documento, y para crear un objeto de este tipo, al igual que siempre, usamos un método de factoría:

```
static Signature <Signature> getInstance(String algorithm)  
static Signature <Signature> getInstance(String algorithm,  
                                         String provider)
```

La clase `Signature` hace primero un message digest de los datos a firmar, con lo que no hace falta que hagamos nosotros un message digest del documento antes de firmarlo.

Como algoritmo se puede usar tanto DSA como RSA. En la implementación de Sun se pueden pedir las siguientes firmas: "SHA1withDSA", "MD2withRSA", "MD5withRSA", "SHA1withRSA".

Una vez tengamos el objeto `Signature` debemos inicializarlo con uno de estos métodos:

```
void <Signature> initSign(PrivateKey k)  
void <Signature> initVerify(PublicKey k)
```

Dependiendo de si vamos a usar el objeto para firmar o para verificar.

Después ya podemos pasar el documento a firmar por el objeto `Signature` usando:

```
void <Signature> update(byte[] b)
```

Y finalmente lo firmamos o verificamos usando los métodos:

```
byte[] <Signature> sign()
```

que retorna la firma del documento.

```
boolean <Signature> verify(byte[] signature)
```

al que le pasamos la firma (no el documento ya que éste ya lo pasamos con `update()`), y nos devuelve si la firma es válida.

Una vez que llamamos a los métodos `sign()` y `verify()` el objeto `Signature` se reinicia para que podamos firmar o verificar otro documento. No hace falta llamar a `initSign()` o `initVerify()` ya que la inicialización permanece válida a no ser que la cambiemos.

Ejemplo

Para acabar este apartado vamos a hacer un programa que sirva para firmar ficheros. El Listado 4.3 muestra el programa que firma digitalmente un fichero, y el Listado 4.4 contiene otro programa que comprueba la firma del fichero firmado.

Para crear el par clave pública / clave privada vamos a usar el programa `Constantes.java` y `CreaClaves.java` que hicimos en el apartado 5.2 del Tema 3, y que encontramos en el Listado 3.3 y el Listado 3.4.

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

public class FirmaFichero implements Constantes
{
    public static void main(String[] args) throws Exception
    {
        // Pedimos el fichero a firmar
        // y fichero de clave privada a usar
        BufferedReader teclado = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.print("Indique fichero a firmar:");
        String fichero_firmar = teclado.readLine();
        if (!new File(fichero_firmar).exists())
        {
            System.out.println("El fichero "
                +fichero_firmar+" no existe");
            return;
        }
        String fichero_firma = fichero_firmar+".sign";
        // Recuperamos la clave privada
        System.out.print("Indique que fichero tiene la"
            + " clave privada a usar:");
        String fichero_privada = teclado.readLine();
        System.out.print(
            "Indique password con que se encripto el fichero "
            +fichero_privada+":");
        String password = teclado.readLine();
        System.out.println(
```

```
        "Recuperando clave privada...");  
SecureRandom sr = new SecureRandom();  
sr.setSeed(new Date().getTime());  
FileInputStream fis =  
        new FileInputStream(fichero_privada);  
byte[] buffer = new byte[TAMANO_SALT_BYTES];  
fis.read(buffer);  
PBEKeySpec clave_pbe_spec =  
        new PBEKeySpec(password.toCharArray());  
SecretKey clave_pbe =  
        SecretKeyFactory.getInstance(  
        "PBEWithSHAAndTwofish-CBC").generateSecret(  
        clave_pbe_spec);  
PBEParameterSpec param_pbe_spec =  
        new PBEParameterSpec(buffer, ITERACIONES_PBE);  
Cipher descifrador_pbe = Cipher.getInstance(  
        "PBEWithSHAAndTwofish-CBC");  
descifrador_pbe.init(Cipher.DECRYPT_MODE,  
        clave_pbe, param_pbe_spec, sr);  
buffer = new byte[fis.available()];  
fis.read(buffer);  
fis.close();  
buffer = descifrador_pbe.doFinal(buffer);  
PKCS8EncodedKeySpec clave_privada_spec =  
        new PKCS8EncodedKeySpec(buffer);  
KeyFactory kf = KeyFactory.getInstance("RSA");  
PrivateKey clave_privada =  
        kf.generatePrivate(clave_privada_spec);  
System.out.println("Clave secreta recuperada");  
// Firmamos el fichero  
fis = new FileInputStream(fichero_firmar);  
Signature s = Signature.getInstance("SHA1withRSA");  
s.initSign(clave_privada);  
buffer = new byte[1024];  
while (fis.available()>=1024)  
{  
    fis.read(buffer);  
    s.update(buffer);  
}  
int quedan = fis.available();  
fis.read(buffer,0,quedan);  
s.update(buffer,0,quedan);  
buffer = s.sign();  
fis.close();  
FileOutputStream fos =  
        new FileOutputStream(fichero_firma);  
fos.write(buffer);  
fos.close();  
System.out.println(  
        "Fichero firmado, la firma se deposito en "  
        +fichero_firma);  
}  
}
```

Listado 4.3: Programa que firma digitalmente un fichero

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

public class VerificaFirma implements Constantes
{
    public static void main(String[] args) throws Exception
    {
        // Pedimos el fichero a verificar
        // y fichero de clave publica a usar
        BufferedReader teclado =
            new BufferedReader(new InputStreamReader(
                System.in));
        System.out.print("Indique fichero a verificar:");
        String fichero_verificar = teclado.readLine();
        if (!new File(fichero_verificar).exists())
        {
            System.out.println("El fichero "
                + fichero_verificar + " no existe");
            return;
        }
        String fichero_firma = fichero_verificar+".sign";
        if (!new File(fichero_firma).exists())
        {
            System.out.println("El fichero "
                + fichero_verificar + " no existe");
            return;
        }

        // Recuperamos la clave publica
        System.out.print("Indique que fichero tiene la"
            + " clave publica a usar:");
        String fichero_publica = teclado.readLine();
        FileInputStream fis =
            new FileInputStream(fichero_publica);
        byte[] buffer = new byte[fis.available()];
        fis.read(buffer);
        fis.close();
        X509EncodedKeySpec clave_publica_spec =
            new X509EncodedKeySpec(buffer);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PublicKey clave_publica =
            kf.generatePublic(clave_publica_spec);
        System.out.println("Carta publica recuperada");

        // Comprobamos la firma
        fis = new FileInputStream(fichero_firma);
        byte[] firma = new byte[fis.available()];
        fis.read(firma);
        fis.close();
        fis = new FileInputStream(fichero_verificar);
```

```

Signature s = Signature.getInstance("SHA1withRSA");
s.initVerify(clave_publica);
buffer = new byte[1024];
while(fis.available()>=1024)
{
    fis.read(buffer);
    s.update(buffer);
}
int quedan = fis.available();
fis.read(buffer,0,quedan);
s.update(buffer,0,quedan);
fis.close();
if (s.verify(firma))
    System.out.println(
        "El fichero firmado es correcto");
else
    System.out.println("La firma no es correcta");
}
}

```

Listado 4.4: Programa que comprueba la firma de un fichero

3.2 La clase SignedObject

La clase `java.security.SignedObject` nos permite encapsular a un objeto y su firma. El objeto se crea con el constructor:

```
SignedObject(Serializable object, PrivateKey signingKey
            , Signature signingEngine)
```

`SignedObject` hace una copia del objeto pasado serializado, con lo que si luego modificamos el objeto, no se modifica el objeto que tiene dentro firmado `SignedObject`.

Obsérvese que para que se pueda firmar un objeto, éste debe ser serializable, es decir, debe implementar la interfaz `Serializable`.

Podemos preguntar por el objeto firmado o por su firma con:

```
Object <SignedObject> getContent()
byte[] <SignedObject> getSignature()
```

La aplicación más típica de los `SignedObject` es firmar objetos remotos que vamos a enviar por una red insegura. El receptor luego los puede verificar con:

```
boolean <SignedObject> verify(PublicKey verificationKey,
                               Signature verificationEngine)
```

Tema 5

Gestión de claves

Sinopsis:

En este tema profundizaremos en el estudio de la gestión de claves analizando las distintas técnicas usadas para crear, almacenar, intercambiar, y destruir claves de forma segura.

También estudiaremos técnicas de distribución de claves mediante servidores de claves como Kerberos, u otras técnicas de gestión de claves más distribuidas como PGP y GPG.

Después estudiaremos distintas técnicas de identificación y autorización utilizadas en muchos protocolos de seguridad, como pueda ser SSH.

Acabaremos el tema estudiando distintas formas de configurar una VPN (Virtual Private Network).

1 Gestión de claves (Key management)

La **gestión de claves** son un conjunto de técnicas destinadas a generar, intercambiar, almacenar y destruir claves.

En el mundo real la gestión de claves es la parte más complicada de la criptografía: diseñar un algoritmo criptográfico seguro es relativamente fácil, y existen multitud de ellos actualmente. Mantener una clave en secreto es más difícil: esto es así porque es más fácil encontrar puntos débiles en las personas que en los algoritmos criptográficos. ¿De qué vale montar un sistema criptográfico muy avanzado en nuestra organización, si luego hay empleados fáciles de corromper?.

En este apartado vamos a empezar comentando una serie de aspectos fundamentales relacionados con la gestión de claves.

1.1 Generación de claves

La seguridad de todo el sistema depende de la clave, si un espía descubre la clave, de nada sirven todos los demás esfuerzos. Sin embargo, hay una serie de factores que influyen negativamente en la seguridad de la clave:

El primer problema es la **pobre elección de las claves**. Normalmente los usuarios tienden a elegir las claves fáciles de recordar, con lo que los ataques con diccionario suelen sacar gran parte de las claves de los sistemas que criptoanalizan.

Para evitar estos ataques, una técnica muy usada es recomendar a los usuarios utilizar **passphrases** en lugar de passwords, que es una frase a la que se hace un hash para obtener la clave.

Si la frase es lo suficientemente larga, la clave se vuelve realmente aleatoria. Pero ¿cuánto larga debería ser la frase?: Según la teoría de la información cada letra tiene aproximadamente 1.3 bits de información. Si queremos generar una clave realmente aleatoria de 64 bits necesitaremos 49 letras, o lo que es lo mismo, unas 10 palabras. Como regla general podemos decir que necesitamos unas 5 palabras por cada 4 bytes que tenga la clave.

Un segundo problema que vamos a comentar son los **espacios de claves reducidos**. Muchas veces sólo se aceptan subconjuntos del juego de caracteres como password.

Por ejemplo, Norton Discreet para MS-DOS hasta la versión 8.0 sólo aceptaba caracteres ASCII bajos, obligando a que el bit más significativo del carácter

fueran siempre cero. Además no distinguía entre mayúsculas y minúsculas. Si este programa usaba DES que en principio tiene una clave de 56 bits con 2^{56} combinaciones, al aplicar esta reducción se quedaba en tan sólo 2^{40} combinaciones, lo cual es fácil de romper por fuerza bruta hoy en día.

La mayoría de los sistemas criptográficos que implementan un ataque por fuerza bruta disponen de tres opciones:

1. Probar todas las combinaciones de letras minúsculas para claves entre 1 y n caracteres.
2. Probar todas las combinaciones de letras minúsculas, mayúsculas, y números para claves de entre 1 y n caracteres.
3. Probar todos los caracteres ASCII para claves entre 1 y n caracteres.

Experimentalmente se sabe que más de la mitad de las claves se encuentran con la primera opción.

El tercer factor negativo que vamos a comentar son los **defectos en la generación de claves aleatorias**. Cuando se usan claves binarias (p.e. clave de sesión, o par de claves pública/privada), hay que tener cuidado de que el algoritmo que usemos realmente genere claves aleatorias. Muchos sistemas se han atacado aprovechando que la clave que generaba el sistema era predecible. Por ejemplo, las primeras implementaciones de SSL de Netscape generaban una clave de sesión en función de la hora, un espía podía predecir la clave de sesión que iba a generar el programa si sabía la hora del reloj del host donde se estaba ejecutando Netscape. Esta hora además la mandaba Netscape en sus cabeceras, y aunque no se supiera la hora exacta bastaba con saber la hora aproximada para hacer un ataque por fuerza bruta.

La generación de claves realmente aleatorias es un proceso realmente complicado ya que los ordenadores son máquinas predecibles.

Un estándar para generación de claves muy conocido es ANSI X9.17. El algoritmo usa una semilla S_0 y un timestamp T_i de la siguiente forma:

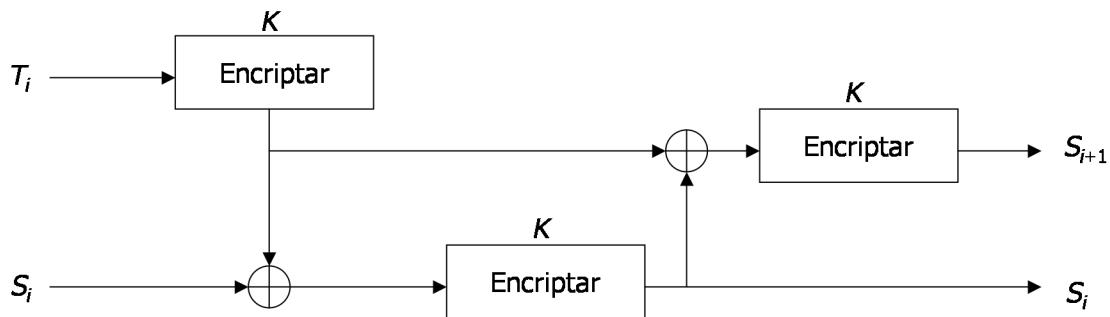
Sea $E(K, X)$ la encriptación TripleDES de X con la clave K .

1. Calculamos un número aleatorio:

$$R_i = E(K, E(K, T_i) \oplus S_i)$$
2. Para calcular la siguiente semilla usamos:

$$S_{i+1} = E(K, E(K, T_i) \oplus R_i)$$

La Figura 5.1 muestra gráficamente este proceso. Obsérvese que la seguridad de este algoritmo de generación de números aleatorios reside en que el criptoanalista no conoce la clave K , ya que el timestamp T y la semilla S son siempre predecibles en un ordenador.

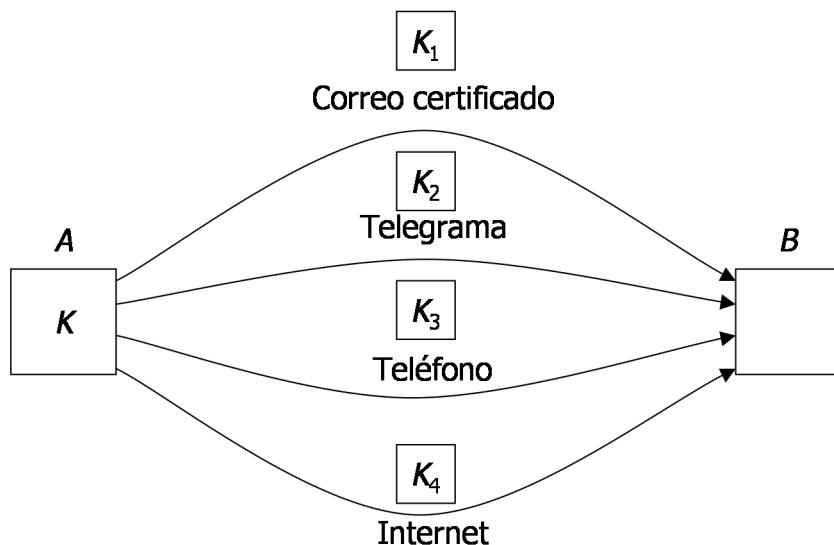
**Figura 5.1** Algoritmo de generación de números aleatorios

1.2 Transferencia de claves

Otro problema es el de cómo se ponen de acuerdo A y B en la clave a usar.

Una solución es que *A* y *B* se reúnan en un lugar físico y acuerden la clave que van a utilizar, pero esto no siempre es posible.

Otra posible solución es que partan la clave en trozos y envíen cada trozo por un canal distinto, como muestra la Figura 5.2, así el supuesto espía tendría que espiar todos los canales.

**Figura 5.2** Enviar los trozos de una clave por distintos canales

La técnica más usada para el intercambio seguro de claves se basa en la criptografía de clave pública y la veremos en el apartado 2.2.

1.3 Almacenamiento de claves

Las claves binarias (p.e. una clave privada) no las puede memorizar una persona, sino que hay que almacenarlas en disco, pero su almacenamiento en disco es peligroso porque alguien podría acceder a ellas. Para evitarlo las claves se almacenan encriptadas con un password o passphrase.

Ahora la seguridad de la clave depende de dos factores:

1. Las posibilidades que tiene el espía de acceder al fichero (p.e en una máquina UNIX, si un usuario desactiva el permiso `r` de un fichero, los demás usuarios no pueden ver sus ficheros, pero el superusuario de la máquina sí que puede acceder a él).
2. La fuerza del password o passphrase que se usó para encriptar el fichero.

Otra alternativa consiste en almacenar las claves en tarjetas USB destinadas a este fin, de forma que las claves nunca permanecen en disco y si el espía quiere nuestra clave no tiene más remedio que quitarnos la tarjeta. Esta solución tiene la ventaja de que podemos estar seguros de si alguien tiene acceso al sistema o no, ya que siempre podemos saber si tenemos la tarjeta en el bolsillo, o no.

Recuérdese que, como vimos en el apartado 1.3 del Tema 2, un posible ataque que puede lanzar el espía para acceder a nuestra clave consiste en estudiar el fichero de memoria virtual del sistema. Este ataque es posible porque los sistemas operativos modernos paginan los programas de RAM a disco sin avisar, y nosotros nunca sabemos si la clave no se ha paginado a disco en mitad de la ejecución del programa.

1.4 Previsión de pérdida de claves

Imaginemos que *A* trabaja en una empresa que le obliga a encriptar todos sus ficheros. Y supongamos que *A* muere, entonces todos sus datos se pierden para siempre.

Para evitarlo podemos usar el algoritmo de compartición de secreto que explicamos en el apartado 8.5 del Tema 1: *A* reparte su clave entre otros empleados de forma que un empleado, por sí solo, no puede acceder a los datos de *A*, pero si es necesario se pueden reunir todos los empleados y reconstruir la clave de *A*.

1.5 El ciclo de vida de una clave

Normalmente, no se recomienda mantener una clave durante mucho tiempo, ya que esto aumenta la probabilidad de que alguien la descubra, y si la descubre mayor será la cantidad de información a la que podrá acceder. Además la tentación que tiene el atacante por descubrirla será mayor si la clave no se cambia muy a menudo que si se cambia diariamente.

Por el contrario, los usuarios suelen ser reacios a cambiar regularmente sus claves, con lo que una solución de compromiso podría ser la **actualización automática de claves**, que consiste en que el sistema cambia periódicamente las claves como un hash de la clave anterior. Realmente esta solución no hace más segura la clave ya que si el espía descubre la clave también puede hacer un hash de esta. Lo que si evita es el efecto de ataques que explicamos en el apartado 4 del Tema 1, como son la deducción global, donde el criptoanalista encuentra un algoritmo alternativo para calcular $D(K, M)$ sin conocer K , o la deducción local donde el criptoanalista descubre un texto plano y su correspondiente texto cifrado.

2 Intercambio seguro de claves

Como hemos visto, cuando A y B se quieren comunicar de forma segura primero acuerdan usar una clave de sesión. Debido a que muchas veces A y B son usuarios que no viven en el mismo sitio físico, el **intercambio seguro de claves**, también llamado **key exchange** o **key agreement** permite a estos usuarios ponerse de acuerdo de forma segura en la clave de sesión a usar (p.e. Internet), sin que un atacante pasivo o activo pueda conocer la clave de sesión acordada.

2.1 Intercambio seguro de claves con criptografía simétrica

En el protocolo que vamos a estudiar en este apartado supondremos que A y B comparten una clave secreta con un árbitro. Esta clave debe estar fijada antes de que comience el protocolo, e ignoramos el problema de cómo se distribuyen estas claves secretas, sólo supondremos que están fijadas y que el espía no tiene ni idea de cuáles son.

En este escenario, el protocolo de criptografía simétrica que podemos usar es el siguiente:

1. A se pone en contacto con el árbitro y le pide una clave de sesión para comunicarse con B .
2. El árbitro genera una clave de sesión aleatoria, encripta una copia con la clave secreta de A y otra con la clave secreta de B , y envía ambas a A .
3. A desencripta la copia.
4. A se conecta a B y le envía su clave de sesión encriptada.
5. B desencripta su clave de sesión.
6. A y B se comunican con esa clave de sesión.

Este protocolo funciona, pero tiene dos inconvenientes: hay que confiar en la integridad del árbitro, y el árbitro se convierte en un cuello de botella.

2.2 Intercambio seguro de claves con criptografía asimétrica

Este mecanismo ya le estudiamos en el apartado 2 Tema 3 cuando estudiamos los sistemas híbridos. Básicamente consiste en que A y B usan criptografía asimétrica para acordar una clave de sesión que luego usan con su algoritmo simétrico para comunicarse.

Recuérdese que el protocolo era el siguiente:

1. B envía a A su clave pública.
2. A genera una clave de sesión aleatoria (binaria), la encripta usando la clave pública de B , y se la envía a B .
3. B desencripta el mensaje enviado por A , usando su clave privada, y recupera la clave de sesión.
4. Ambos se comunican encriptando sus mensajes con la clave de sesión.

2.3 El man in the middle attack

En el algoritmo de intercambio de claves con criptografía de clave pública del apartado anterior, un espía (también llamado atacante pasivo) no puede hacer nada mejor que intentar romper el algoritmo de clave pública por fuerza bruta. Sin embargo, un atacante activo puede hacer muchas más cosas, ya que el atacante activo puede borrar mensajes o generar nuevos mensajes, así como encarnar a A cuando este habla con B .

En concreto, el ataque que vamos a ver en este apartado se llama **man in the middle attack** y es el siguiente:

1. A envía a B su clave pública, el atacante intercepta el mensaje y envía a B su propia clave pública.
2. B envía a A su clave pública, el atacante la intercepta y envía a A su propia clave pública.
3. Cuando A envía a B un mensaje encriptado con la clave pública de B (que realmente es la del atacante) el atacante la intercepta, la desencripta con su clave privada, la vuelve a encriptar con la clave pública de B y se lo envía a B .
4. Cuando B envía un mensaje a A encriptado con la clave pública de A (que realmente es la del atacante), el atacante lo intercepta, desencripta con su clave privada, lo vuelve a encriptar con la clave pública de A , y se lo envía a A .

Este ataque funciona porque A y B no tienen forma de saber si realmente están hablando el uno con el otro.

2.4 Evitar el man in the middle attack

Ron Rivest y Adi Shamir inventaron un protocolo que evita el man in the middle attack al que llamaron **interlock protocol**.

El protocolo es el siguiente:

1. A envía a B su clave pública.
2. B envía a A su clave pública.
3. A encripta su mensaje usando la clave pública de B , y envía la mitad del mensaje encriptado a B .
4. B encripta su mensaje usando la clave pública de A , y envía la mitad del mensaje encriptado a A .
5. A envía la otra mitad del mensaje a B .
6. B junta las dos partes del mensaje de A , y desencripta el mensaje con su clave privada, y comprueba que sea correcto.
7. B envía la otra mitad de su mensaje a A .
8. A junta las dos partes del mensaje de B y desencripta el mensaje con su clave privada, y comprueba que sea correcto.

Este protocolo además exige que no se pueda desencriptar medio mensaje sin tener el otro medio. Para ello tenemos varias opciones: Si usamos un algoritmo de cifrado por bloques podemos enviar medio bloque en cada parte, también podemos usar un IV que se envía en la segunda parte, o la primera parte del mensaje puede contener un hash del mensaje completo.

Veamos qué problema tendría ahora el atacante activo: El atacante todavía puede cambiar las claves públicas en los pasos 1 y 2, pero en el paso 3 cuando A envía medio mensaje no puede desencriptarlo (sin tener el otro medio) y enviárselo a B (encriptado con la clave pública de B). Luego no le queda más remedio que inventarse un mensaje totalmente distinto y B detectará el problema en el paso 6.

2.5 Intercambio seguro de claves con firmas digitales

Podemos usar firmas digitales para evitar el man in the middle attack. Para ello necesitamos un árbitro que firme las claves públicas de A y B .

Una vez que A y B tienen sus claves públicas firmadas por el árbitro, el atacante no puede cambiar las claves de A y B , ya que no sabe firmarlas con la firma del árbitro.

Ahora, lo más que puede hacer el atacante activo es impedir que A y B se comuniquen, pero no puede escuchar la comunicación.

3 Diffie-Hellman

Diffie-Hellman es el primer algoritmo de intercambio de claves seguro inventado en 1976.

Su seguridad se basa en la dificultad de calcular el logaritmo discreto de un número en aritmética modular, comparado con la dificultad de calcular la exponenciación de un número en aritmética modular.

El algoritmo se puede usar para intercambiar claves secretas, pero no puede usarse para encriptar o desencriptar mensajes.

3.1 Matemáticas previas

Antes de explicar el algoritmo tenemos que explicar que significa que un número g sea primitivo respecto a otro número p : si p es primo y $g < p$, decimos que g **es primitivo respecto a p** si para todo b desde 1 hasta $p-1$ existe un a tal que $(g^a \equiv b) \text{ mod } p$, es decir:

$$\forall b \in [1..p-1] \exists a / (g^a \equiv b) \text{ mod } p$$

Por ejemplo, si $p=11$, $g=2$ es primitivo respecto a p ya que:

$$\begin{aligned} (2^{10} &\equiv 1) \text{ mod } 11 \\ (2^1 &\equiv 2) \text{ mod } 11 \\ (2^8 &\equiv 3) \text{ mod } 11 \\ (2^2 &\equiv 4) \text{ mod } 11 \\ (2^4 &\equiv 5) \text{ mod } 11 \\ (2^9 &\equiv 6) \text{ mod } 11 \\ (2^7 &\equiv 7) \text{ mod } 11 \\ (2^3 &\equiv 8) \text{ mod } 11 \\ (2^6 &\equiv 9) \text{ mod } 11 \\ (2^5 &\equiv 10) \text{ mod } 11 \end{aligned}$$

De hecho, en el caso de $p=11$ tenemos que $g=2$, $g=6$, $g=7$, $g=8$ son primitivos respecto a p . Otros números como $g=3$ no son primitivos respecto a p ya que no hay solución para:

$$(3^a \equiv 2) \text{ mod } 11$$

En general, comprobar si un número g es primitivo respecto a p es un problema costoso, pero se vuelve fácil si conocemos la factorización de $p-1 = q_1 * q_2 * ... * q_r$, ya que en este caso para comprobar si g es primitivo respecto a p calculamos:

$(g^{(p-1)/q}) \bmod p$ para todo $q=q_1, q_2, \dots, q_r$.

Si en algún caso el cálculo de $(g^{(p-1)/q}) \bmod p$ nos da 1, entonces g no es primitivo respecto a p . Si el cálculo no nos da 1 en ningún caso, entonces g es primitivo respecto a p .

Por ejemplo, si queremos ver si $g=2$ es primitivo respecto a $p=11$:

Primero calculamos los factores primos de $p-1=10$ que son: 2,5.

Después calculamos:

$$\begin{aligned} (2^{(11-1)/2} \equiv 10) &\bmod 11 \\ (2^{(11-1)/5} \equiv 4) &\bmod 11 \end{aligned}$$

Luego, como ninguno da 1, $g=2$ es primitivo respecto a $p=11$.

3.2 Algoritmo de Diffie-Hellman

En este apartado vamos a ver el algoritmo de Diffie-Hellman.

Si A y B quieren intercambiar una clave de forma segura, primero se ponen de acuerdo en un número primo p , y un g que sea primitivo respecto a p .

Estos dos números no tienen porqué ser secretos, incluso pueden usarse siempre los mismos números. De hecho, hay estándares como SKIP (Simple Key management for Internet Protocols) que fijan estos números en el estándar. SKIP es un protocolo usado para la comunicación segura entre equipos de una LAN muy usado en las VPN.

Lo único de lo que hay que tener cuidado es de que si estos números se envían por la red, un atacante activo no los pueda modificar.

El protocolo es el siguiente:

1. A elige un número entero grande x aleatoriamente (que no debe mostrar) y calcula:

$$(X=g^x) \bmod p$$

y envía X a B .

2. B elige un número entero grande y aleatoriamente (que no debe mostrar) y calcula:

$$(Y=g^y) \text{ mod } p$$

y envía Y a A .

3. A calcula:

$$(K=Y^x) \text{ mod } p$$

4. B calcula:

$$(K'=X^y) \text{ mod } p$$

Y ahora resulta que se cumple que:

$$(g^{xy} = X^y = Y^x = K = K') \text{ mod } p$$

Con lo que $(K=K') \text{ mod } p$ es la clave de sesión que pueden usar A , B para comunicarse.

Obsérvese que K nunca ha viajado por la red, con lo que un posible espía no puede conocer la clave de sesión, ya que como el espía no conoce x ni y , si quiere conocer K tiene que resolver el sistema de ecuaciones:

$$\begin{cases} (g^{xy}=X^y) \text{ mod } p \\ (g^{xy}=Y^x) \text{ mod } p \end{cases} \Rightarrow \begin{cases} (xy \log(g) = y \log(X)) \text{ mod } p \\ (xy \log(g) = x \log(Y)) \text{ mod } p \end{cases}$$

Con x , y , como incógnitas y X , Y , g como valores conocidos.

El problema que tiene aquí el espía es que calcular el logaritmo discreto en aritmética modular es muy complejo, y aunque $\log(g)$ se puede conocer (por ser g siempre la misma), $\log(X)$ y $\log(Y)$ dependen de las x , y que eligieron los interlocutores.

Para que el algoritmo sea seguro p debe ser un número muy grande. Sin embargo, la seguridad del algoritmo no depende del tamaño de g , con lo que g se elige como el número más pequeño que sea primitivo respecto a p . De hecho, normalmente g en las implementaciones reales tiene un solo dígito.

3.3 Diffie-Hellman con tres o más participantes

El algoritmo de intercambio seguro de claves de Diffie-Hellman se puede extender fácilmente a redes de tres o más participantes.

Vemos cuál sería el protocolo para A , B , C :

1. *A* elige un número entero grande x aleatoriamente (que no debe mostrar) y calcula:

$$(X=g^x) \text{ mod } p$$

y envía X a *B*.

2. *B* elige un número entero grande y aleatoriamente (que no debe mostrar) y calcula:

$$(Y=g^y) \text{ mod } p$$

y envía Y a *C*.

3. *C* elige un número entero grande z aleatoriamente (que no debe mostrar) y calcula:

$$(Z=g^z) \text{ mod } p$$

y envía Z a *A*.

4. *A* calcula:

$$(Z'=Z^x) \text{ mod } p$$

y envía Z' a *B*.

5. *B* calcula:

$$(X'=X^y) \text{ mod } p$$

y envía X' a *C*.

6. *C* calcula:

$$(Y'=Y^z) \text{ mod } p$$

y envía Y' a *A*.

7. *A* calcula:

$$(K=Y'^x) \text{ mod } p$$

8. *B* calcula:

$$(K=Z'^y) \text{ mod } p$$

9. *C* calcula:

$$(K=g^{xz}) \bmod p$$

Ahora todos tienen la clave de sesión ($K=g^{xz} \bmod p$), y ningún espía puede deducir la clave escuchando en el canal.

3.4 Diffie-Hellman para una clave decidida por A

Existe una variante de Diffie-Hellman que permite a A enviar una clave concreta K a B de forma segura. De esta forma A puede enviar a B una clave que ya fue decidida previamente. Esto también permite que A deje encriptado un mensaje y se lo envíe de forma segura a varios interlocutores.

El protocolo es el siguiente:

1. A elige un número entero grande aleatorio x , y calcula:

$$(K=g^x) \bmod p$$

Ahora A puede usar K para dejar encriptado un mensaje.

2. Cuando B quiere obtener la clave con la que lo encriptó A , elige un número entero grande aleatorio y , que tenga inverso multiplicativo $z=y^{-1}$, calcula:

$$(Y=g^y) \bmod p$$

y envía Y a A .

3. A envía a B :

$$(X=Y^x) \bmod p$$

4. B calcula la clave como:

$$(K=X^z) \bmod p$$

Con lo que B ya tiene la clave K con la que A había encriptado el mensaje. El espía no puede desencriptar la clave secreta porque no conoce $y=z^{-1}$.

4 Intercambio seguro de claves en Java

Para el intercambio seguro de claves en Java se usa el servicio o engine `javax.crypto.KeyAgreement` que encapsula un algoritmo de intercambio de claves seguro.

El proveedor "SunJCE" proporciona Diffie-Hellman. En este proveedor la X y la x se representan con una clave pública y una clave privada respectivamente, luego lo primero que tenemos que hacer es obtener este par de claves usando el engine `KeyPairGenerator` al que le pedimos como algoritmo "DH":

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
KeyPair claves = kpg.generateKeyPair();
```

Al generar este par también podemos pedir usar una g y una p determinadas, lo cual es útil si vamos a conectarnos a un servidor que sigue el protocolo SKIP. Si vamos a pasar estos parámetros, se los tenemos que pasar al método:

```
void <KeyPairGenerator> initialize(
    AlgorithmParameterSpec param)
```

antes de llamar a:

```
KeyPair <KeyPairGenerator> generateKeyPair()
```

Para pasárselos al método `initialize()`, se los pasamos en un objeto del tipo `DHParameterSpec`, que como muestra la Figura 5.3, es una derivada de la clase `AlgorithmParameterSpec` que tiene el constructor:

```
DHParameterSpec(
    BigDecimal p
    , BigDecimal g)
```

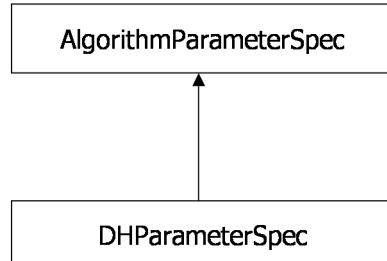


Figura 5.3 Clase para los parámetros de Diffie-Hellman

Ahora enviamos nuestra clave pública al interlocutor (la X), y él nos devuelve su clave pública (la Y).

Después para calcular la clave de sesión a partir de nuestra clave privada y la clave pública del interlocutor tenemos que usar un objeto de tipo `KeyAgreement`, el cual obtenemos pasando como parámetro "DH" a:

```
static KeyAgreement <KeyAgreement> getInstance(
    String algorithm)
```

Y después lo inicializamos con:

```
void <KeyAgreement> init(Key key)
```

Al que le pasamos como parámetro nuestra clave privada, y después ejecutamos:

```
Key <KeyAgreement> doPhase (Key key, boolean lastPhase)
```

Si estamos realizando un intercambio de claves entre dos participantes, este método deberá ejecutarse sólo una vez, y en `lastPhase` pasamos `true`. En este caso el retorno será `null`.

Si estamos realizando un intercambio de claves entre tres o más participantes el método retorna la clave pública de la siguiente fase. En este caso en `lastPhase` ponemos `false` hasta llegar a la última fase.

Por último debemos obtener la clave secreta de sesión que ha generado el algoritmo, que no es el retorno de la última llamada a `doPhase()`, ya que como dijimos retornaba `null` en la última fase. Para ello llamamos a:

```
byte[] <KeyAgreement> generateSecret()
```

y ya tenemos la clave de sesión.

Para acabar este apartado vamos a hacer un ejemplo de intercambio de claves seguro entre un cliente y un servidor usando SKIP de 1024 bits.

En la clase `ParametrosSkip` del Listado 5.1 se encuentran los parámetros usados por el algoritmo.

Las clases `ClienteSkip` y `ServidorSkip` del Listado 5.2 y Listado 5.3 implementan el cliente y servidor que dialogan para acabar obteniendo el mismo número en la clave secreta.

```
/* Fichero con los parametros de SKIP para 1024 bits
 * Parametros sacados de
 * http://www.skip-vpn.org/spec/numbers.html
 */

import java.math.BigInteger;
import javax.crypto.spec.DHParameterSpec;

public class ParametrosSkip
{
    private static final String strModulo1024 =
```

```

    "F488FD584E49DBCD20B49DE49107366B336C380D451D"
    + "0F7C88B31C7C5B2D8EF6F3C923C043F0A55B188D8EBB"
    + "558CB85D38D334FD7C175743A31D186CDE33212CB52A"
    + "FF3CE1B1294018118D7C84A70A72D686C40319C80729"
    + "7ACA950CD9969FABD00A509B0246D3083D66A45D419F"
    + "9C7CBD894B221926BAABA25EC355E92F78C7";

    // Modulo (n)
    private static final BigInteger modulo1024 =
        new BigInteger(strModulo1024, 16);

    // Base (g)
    private static final BigInteger base1024 =
        BigInteger.valueOf(2);

    // Parametros DH
    public static final DHParameterSpec parametrosDH =
        new DHParameterSpec(modulo1024, base1024);
}

```

Listado 5.1: Parámetros de SKIP

```

import java.io.*;
import java.net.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import java.math.BigInteger;

public class ClienteSkip
{
    public static void main(String[] args) throws Exception
    {
        // Abrimos la conexión al servidor
        if (args.length!=1)
        {
            System.err.println(
                "Indique puerto como argumento");
            return;
        }
        Socket s = new Socket(InetAddress.getLocalHost(),
            Integer.parseInt(args[0]));
        DataInputStream dis =
            new DataInputStream(s.getInputStream());
        DataOutputStream dos =
            new DataOutputStream(s.getOutputStream());

        // Creamos el par de clave privada/publica
        // Diffie-Hellman
        KeyPairGenerator kpg =
            KeyPairGenerator.getInstance("DH");
        kpg.initialize(parametrosSkip.parametrosDH);
        KeyPair claves = kpg.generateKeyPair();

        // Enviamos nuestra clave publica

```

```

byte[] buffer = claves.getPublic().getEncoded();
dos.writeInt(buffer.length);
dos.write(buffer);

// Recibimos la clave publica
buffer = new byte[dis.readInt()];
dis.read(buffer);
KeyFactory kf = KeyFactory.getInstance("DH");
X509EncodedKeySpec x509_spec =
    new X509EncodedKeySpec(buffer);
PublicKey su_clave_publica =
    kf.generatePublic(x509_spec);

// Terminamos el protocolo de intercambio de
// clave seguro
KeyAgreement ka = KeyAgreement.getInstance("DH");
ka.init(claves.getPrivate());
ka.doPhase(su_clave_publica,true);
buffer = ka.generateSecret();
System.out.println(
    "Clave de sesion obtenida con exito:"
    + new BigInteger(buffer));

// Cerramos la conexion
s.close();
}
}

```

Listado 5.2: Cliente SKIP

```

import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;
import java.security.spec.*;
import java.math.BigInteger;

public class ServidorSkip
{
    public static void main(String[] args) throws Exception
    {
        // Ponemos al servidor a ceptar conexiones
        if (args.length!=1)
        {
            System.err.println(
                "Indique puerto como argumento");
            return;
        }
        ServerSocket ss =
            new ServerSocket(Integer.parseInt(args[0]));
        System.out.println("Esperando conexiones...");
        Socket s = ss.accept();
        DataInputStream dis =
            new DataInputStream(s.getInputStream());
        DataOutputStream dos =

```

```
        new DataOutputStream(s.getOutputStream());\n\n        // Creamos el par de clave privada/publica\n        // Diffie-Hellman\n        KeyPairGenerator kpg =\n            KeyPairGenerator.getInstance("DH");\n        kpg.initialize(ParametrosSkip.parametrosDH);\n        KeyPair claves = kpg.generateKeyPair();\n\n        // Recibimos la clave publica\n        byte[] buffer = new byte[dis.readInt()];\n        dis.read(buffer);\n        KeyFactory kf = KeyFactory.getInstance("DH");\n        X509EncodedKeySpec x509_spec =\n            new X509EncodedKeySpec(buffer);\n        PublicKey su_clave_publica =\n            kf.generatePublic(x509_spec);\n\n        // Enviamos nuestra clave publica\n        buffer = claves.getPublic().getEncoded();\n        dos.writeInt(buffer.length);\n        dos.write(buffer);\n\n        // Terminamos el protocolo de intercambio\n        // de clave seguro\n        KeyAgreement ka = KeyAgreement.getInstance("DH");\n        ka.init(claves.getPrivate());\n        System.out.println(\n            ka.doPhase(su_clave_publica,true));\n        buffer = ka.generateSecret();\n        System.out.println(\n            "Clave de sesion obtenida con exito:"\n            + new BigInteger(buffer));\n\n        // Cerramos la conexión\n        s.close();\n        ss.close();\n    }\n}
```

Listado 5.3: Servidor SKIP

5 Los certificados digitales

El punto más débil que tiene un sistema criptográfico es el intercambio de claves públicas que realizan los interlocutores al principio del protocolo, ya que aquí se puede producir el llamado man in the middle attack que explicamos en el apartado 2.3.

Para evitar este ataque podemos usar los llamados **certificados digitales**, que no son más que la clave pública de alguien firmada por una autoridad. Normalmente el certificado digital, además de la clave pública contiene otros datos sobre la persona a quien corresponde la clave pública, como por ejemplo su nombre o su dirección. Al organismo que se encarga de expedir los certificados digitales se le llama **CA (Certification Authority)**.

Ahora si *A* y *B* tienen un certificado de sus claves públicas, pueden intercambiar entre sí sus certificados sin temor al man in the middle attack, y cuando reciben el certificado del otro, primero comprueban la firma de la autoridad.

El atacante activo todavía puede seguir cambiando el certificado en su transmisión, pero no puede firmar un certificado falso, o el receptor detectaría el intento de fraude.

Obsérvese que para que el sistema de certificados funcione, los interlocutores todavía tienen que fiarse de la autenticidad de la clave pública del árbitro. Como veremos en el Tema 6, normalmente los programas lo que hacen es que llevan embebida la clave pública de uno o más árbitros, pero siempre tendremos que tener cuidado de que nadie modifique la clave pública del árbitro que tiene embebida nuestro programa.

6 Los Key Distribution Center (KDC)

En una red grande en la que se van a comunicar muchos usuarios, el problema de intercambiar entre sí los certificados digitales de sus claves públicas se vuelve muy costoso, ya que para n usuarios se necesitan intercambiar $n(n-1)$ certificados digitales, es decir el crecimiento es cuadrático respecto al número de usuarios.

Una solución a este problema es usar los KDC, que son servidores que se encargan de distribuir los certificados digitales entre los usuarios. Ya que lo que reparte el KDC son certificados digitales y no claves públicas, el atacante no puede modificar el certificado, lo más que puede hacer es impedir que se acceda al servidor.

El uso de KDC permite además que un usuario pueda acceder a la clave pública de otro sin necesidad de que el otro esté conectado, ya que de donde coge el certificado es del KDC.

Dos ejemplos de KDC actuales son:

- **Kerberos**, que es uno de los KDC más antiguos, fue diseñado por el MIT y utiliza sólo criptografía de clave secreta para el intercambio de claves.
- **VeriSign y Thawte**, las empresas que firman la mayoría de los certificados digitales que usan las páginas web seguras. Estos KDC los estudiaremos con más detalle en el Tema 6.

7 Kerberos

En este apartado vamos a comentar cómo funciona Kerberos. Como ya hemos comentado en el apartado anterior, Kerberos es uno de los KDC más antiguos, fue diseñado por el MIT, y utiliza sólo criptografía de clave secreta para el intercambio de claves.

El nombre de Kerberos viene del perro con tres cabezas de la mitología griega que protegía la entrada al reino de los muertos. La Figura 5.4 muestra un screenshot de la herramienta que proporciona Mac OS X para gestión de Tickets Kerberos, la cual podemos encontrar en el directorio `/System/Library/CoreServices`.

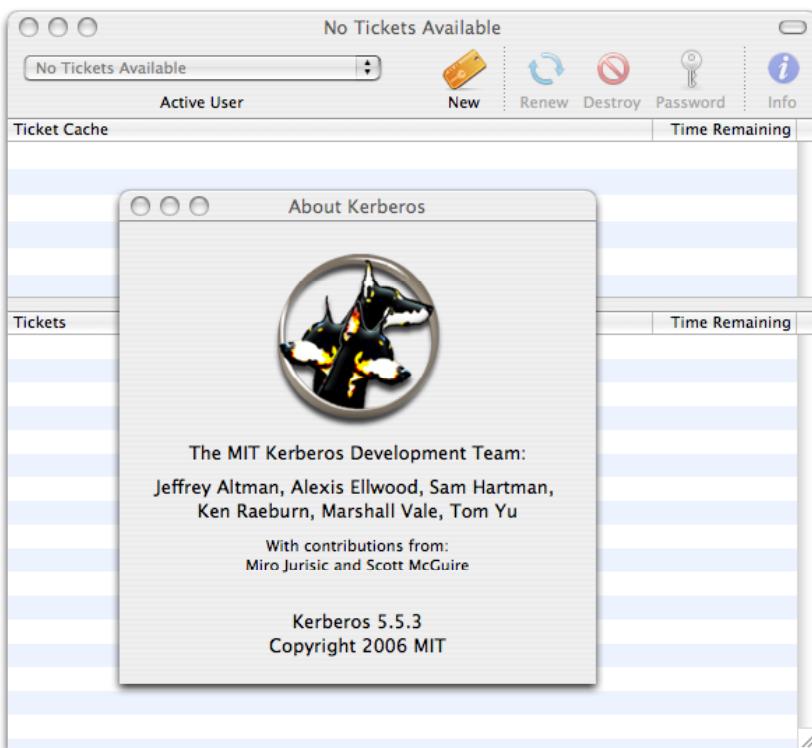


Figura 5.4 Logotipo de Kerberos

Kerberos se distribuye gratuitamente como código fuente, y como ejecutable para la mayoría de los sistemas UNIX en la Web del MIT [KERBEROS]. Actualmente van por la versión 5. Las versiones 1,2 y 3 fueron experimentales y no se publicaron, y la versión 4 está desestimada por tener un agujero de seguridad. Kerberos forma parte de un proyecto más grande llamado Athena.

7.1 Motivación

Vamos a empezar comentando qué problema es el que resuelve Kerberos.

En un sistema UNIX donde todos los usuarios se conectan a una misma máquina donde están todos los recursos, podemos montar un sistema de identificación seguro con sólo disponer de un administrador (`root`), y varios usuarios a cada uno de los cuales se les da unos permisos distintos en función de sus necesidades, con lo que en este sistema no nos haría falta Kerberos.

Si tenemos una red de ordenadores UNIX con el sistema de identificación centralizado, el administrador se encarga de configurar y administrar todas las workstations de la red, siendo él el único que puede entrar en las workstation como `root` y dejando que los demás usuarios entren como usuarios normales con permisos restringidos. En consecuencia el sistema también es seguro.

Otra forma de tener una red de ordenadores con un sistema de identificación centralizado es usar NIS (Network Information System). En una red de ordenadores con NIS los usuarios pueden sentarse en cualquier puesto, y dan su user y password para logarse. Una vez que el usuario entra, éste tendrá acceso a su directorio `home`, independientemente de la máquina en la que se siente.

Usar NIS implica que todas las máquinas de la organización permanezcan en esa organización, ya que si los usuarios tienen la posibilidad de instalar y configurar sus propias máquinas este sistema de seguridad ya no funciona, ya que los usuarios pueden entrar en la workstation (que ellos mismos han configurado) como `root`, y desde ahí pueden entrar como `root` en las demás máquinas que usen un sistema de usuarios compartidos como NIS. Además en este caso los programas UNIX clásicos como `rlogin`, `rsh`, o `rcp` pierden toda su seguridad, ya que un usuario `root` de una máquina podría entrar como `root` en otra máquina sin tener que dar un user y un password.

El sistema de identificación Kerberos resulta útil cuando algunas máquinas pueden pertenecer a sus usuarios (p.e. portátiles), pero otras máquinas permanecen en la organización (p.e. servidores). En este escenario los dueños de sus máquinas querrán acceder a los servicios que les ofrecen las máquinas de la organización. Para ello necesitan que la organización les proporcione un sistema de identificación como el que implementa Kerberos.

7.2 Qué solución aporta Kerberos

Kerberos mantiene una BD centralizada de usuarios de forma que cuando un usuario quiere acceder a cualquier servidor de la organización, este usuario tiene que identificarse primero como un usuario legítimo.

Para ello en la red se instala un servidor Kerberos que actúa como un KDC en el que todos los usuarios de la red confían. Este servidor mantiene una clave secreta con cada uno de los usuarios de la red.

Kerberos propone modificar los comandos de UNIX (`telnet`, `ftp`, `rlogin`, `rsh`, `rcp`,...) para que usen Kerberos como el sistema de identificación en vez de tener que escribir el `user` y el `password` cada vez que ejecutamos el comando. A este proceso se le llama **kerberización**. Actualmente la mayoría de las distribuciones UNIX tienen sus comandos kerberizados¹, con lo que basta con configurar el sistema cliente de identificación Kerberos para poder identificarse por Kerberos, en vez de por la técnica tradicional de proporcionar el `usuario` y `password`. Los servidores también se deben kerberizar. Algunas distribuciones UNIX tienen sus servidores ya kerberizados, y otras veces necesitan ser recompilados con una opción que habilite la identificación mediante Kerberos. Es común que a los servidores kerberizados se les ponga nombres que empiezan por `k`, por ejemplo: `klogind`, `kftpd`, `ktelnetd`, `krshd`,...

Cuando un cliente quiere acceder a cualquier servidor de la red, pide al servidor de Kerberos un **ticket**, que es un mensaje encriptado con la clave secreta del servidor al que queremos acceder. En el ticket el servidor Kerberos dice al otro servidor que el usuario es quien dice ser, y que le da servicio. Ahora el cliente va al servidor y le entrega el ticket, el servidor comprueba que el ticket sea válido y le deja entrar. Todo esto ocurre de forma transparente al usuario, es decir, sin que el programa cliente tenga que pedir al usuario un `password`.

Un posible problema que podría haber aquí es que los comandos de UNIX son programas individuales, con lo que cada vez que el usuario fuera a ejecutar un programa kerberizado el programa debería de pedir el `password` al cliente para obtener la clave secreta que comparte con Kerberos. Para evitarlo se han creado dos soluciones:

1. Cuando el cliente ejecuta un comando kerberizado el `password` se almacena en un fichero del directorio `/tmp` al cual sólo tiene acceso ese usuario. Las demás veces que el usuario ejecuta un comando kerberizado se usa este fichero para obtener el `password`. El usuario también puede usar el comando `kinit` para inicializar su clave secreta.
2. El comando `login` que es el que nos da acceso a un terminal UNIX también se ha kerberizado de forma que si el `user` y `password` de la workstation coincide con el `user` y `password` de Kerberos, a la vez que el usuario se loga en su máquina se genera la clave secreta de Kerberos.

¹ Muchas aplicaciones de interfaz gráfica también están kerberizadas. Por ejemplo en Mac OS X los programas Safari o Mail están kerberizados.

7.3 Componentes de Kerberos

Los principales componentes de Kerberos son:

1. **Kerberos Application Library**, es una librería que utilizan los clientes y servidores para identificarse y comprobar autenticidades a través del servidor Kerberos. Originariamente se usaba DES como algoritmo criptográfico de Kerberos, pero el algoritmo criptográfico es un módulo reemplazable que actualmente se ha reemplazado por otros algoritmos más seguros.
2. **Users Commands**. Son una serie de comandos que puede usar el usuario para administrar sus opciones de Kerberos. Entre ellos tenemos:

`kinit` que permite al usuario cargar su clave secreta de Kerberos. Para ello se usa PBE, es decir, el usuario proporciona un password y se genera una clave secreta binaria.

`kdestroy` que libera la clave secreta cargada.

`klist` que permite al usuario ver los tickets de los que dispone, y para qué servicio son cada uno.

`kpasswd` que permite al usuario cambiar su password en el servidor Kerberos.

En este grupo además se incluirían todos los comandos kerberizados.

3. **Kerberos Database Library**, es un conjunto de funciones de librería que permiten administrar la base de datos de usuarios de Kerberos que tenemos en el servidor Kerberos.
4. **Database Administration Commands**, son un conjunto de comandos que permiten al administrador gestionar la BD de usuarios del servidor Kerberos.
5. **El servidor Kerberos**, que es el que se encarga de dispensar los tickets a los usuarios de la red para que puedan acceder a los servidores kerberizados de la red.

El servidor Kerberos a su vez está formado por dos servidores: El **servidor de identificación** que se instala en el puerto 88 y es el que se encarga de dispensar tickets a los clientes, y el **servidor de administración** que se instala el puerto 749 y es el que usa el administrador para administrar el servidor Kerberos. Ambos aparecen en el fichero `/etc/services`

6. **Ficheros de configuración** que indican a Kerberos cómo funcionar, y normalmente son estos dos:

/etc/krb5.conf es un fichero que se instala en los clientes y servidores indicando donde está el servidor Kerberos.

/etc/kdc.conf que es un fichero con las opciones de configuración del servidor Kerberos.

7. **DB propagation software**, es el software usado cuando hay varios servidores Kerberos en la red (un master y uno o más slaves). El master pasa la BD de usuarios actualizada a los slaves periódicamente a través de este servicio.

7.4 El sistema de nombres de Kerberos

Cuando el administrador configura un servidor Kerberos, debe especificar un **reino (realm)**, que es el conjunto de máquinas que confían en el servidor Kerberos. Aunque el nombre del reino puede ser cualquiera, el MIT recomienda usar el mismo nombre que el del dominio DNS en mayúsculas. Por ejemplo, macprogramadores.org usaría el nombre de reino MACPROGRAMADORES.ORG

Un servidor Kerberos puede servir tickets de varios reinos, los cuales en principio no tienen porqué confiar los unos en los otros, aunque luego el administrador puede crear relaciones entre los reinos de forma que los miembros de un reino puedan confiar en los miembros de otro reino. Estas relaciones de confianza pueden ser unidireccionales (los miembros de A confían en los de B, pero los de B no en los de A), o bidireccionales.

En Kerberos, tanto a los usuarios como a los servidores se les da un nombre, y de cara a Kerberos es indiferente si se trata de un usuario o de un servidor, para Kerberos a cada nombre se le asocia una clave secreta. Este nombre se le llama **principal** y se escribe de la forma:

primary/instance@REALM

primary en el caso de los usuarios es el nombre del usuario (p.e. fernando), y en el caso de los servidores es el nombre del servicio (p.e. rlogind).

instance es un campo opcional. En el caso de los usuarios se usa sólo si un usuario puede tener varios roles, por ejemplo, fernando como usuario normal y fernando/admin como administrador. Para Kerberos ambos nombres son totalmente distintos. En el caso de los servidores indica el host donde está (p.e. rlogind/pc21).

REALM es el nombre del reino.

Luego posibles ejemplos de nombres de principal de Kerberos serían:

`fernando/admin@MACPROGRAMADORES.ORG`
`telnetd/servidor1@MACPROGRAMADORES.ORG`

7.5 Protocolo de Kerberos

El protocolo que usa Kerberos es el siguiente:

1. El árbitro (servidor Kerberos) comparte una clave secreta con cada uno de los usuarios, la cual habrá sido fijada con anterioridad.
2. Cuando A (p.e. el cliente) se quiere comunicar con B (p.e. el servidor), A envía un mensaje al árbitro con las identidades de A y B .
3. El árbitro genera dos mensajes:

Uno encriptado con K_A , la clave secreta de A , que contiene un timestamp (T), un lifetime (L), una clave de sesión (K_S), y una identidad B :

$$E(K_A, (T, L, K_S, B))$$

Este mensaje se puede interpretar como: "Con este ticket puedes hablar con B ".

Y otro encriptado con K_B , la clave secreta de B , con un timestamp (T), lifetime (L), la misma clave de sesión que antes (K_S) y la identidad de A .

$$E(K_B, (T, L, K_S, A))$$

Este mensaje se puede interpretar como: "Atienda al cliente A ".

Y envía ambos mensajes a A .

4. A desencripta el primer mensaje, y genera un mensaje con su identidad y un timestamp (T_S) encriptado con la clave de sesión (K_S).

$$E(K_S, (T_S, A))$$

Este mensaje permite a A demostrar a B que es quien dice ser. En el siguiente apartado veremos que este mensaje es el authenticator.

Y envía a B el mensaje anterior que le dio el árbitro para B , junto con este nuevo mensaje:

$$\begin{aligned} &E(K_B, (T, L, K_S, A)) \\ &E(K_S, (T_S, A)) \end{aligned}$$

5. B desencripta el primer mensaje usando su K_B . Una vez desencriptado obtiene la clave de sesión K_S , la cual usa para desencriptar el segundo mensaje, con lo que comprueba que el mensaje realmente procede de A .
6. B crea un mensaje consistente en el timestamp recibido en el segundo mensaje T_S sumándole uno, lo encripta con la clave de sesión, y se lo envía a A :

$$E(K_S, (T_S+1, B))$$

7. A desencripta el mensaje y comprueba que realmente procede de B .

En este momento ambos tienen la certeza de que están hablando con el interlocutor correcto, y comienzan una comunicación para que A pueda pedir un servicio a B .

Obsérvese que este protocolo usa timestamps T y T_S para evitar replay attacks, lo cual obliga a que las máquinas implicadas tengan sus relojes más o menos sincronizados. Cada interlocutor debe mantener cacheados los mensajes recibidos durante un lifetime L para detectar un posible replay attack.

También es importante resaltar que un atacante activo no puede suplantar a A o a B , ya que si suplanta a A no puede generar correctamente el mensaje en el paso 4, y si suplanta a B no puede generar correctamente el mensaje en el paso 6.

7.6 Credenciales

Kerberos utiliza dos tipos de credenciales: Tickets y authenticators.

Un **ticket** lo expide un servidor Kerberos y se lo entrega a un cliente para que pueda entrar en un servidor. El ticket contiene información que permite al servidor estar seguro de que quien está usando el ticket es el mismo que al que se le ha expedido.

En concreto, un ticket que deja a A entrar en el servicio B , T_{AB} tiene la siguiente información:

$$T_{AB} = E(K_B, (A, IP, T, L, K_S))$$

Donde:

- A - La identidad de A
- IP - La IP de A
- T - Timestamp
- L - Lifetime (determina la caducidad del ticket, normalmente 10 horas)
- K_s - Clave de sesión

Todo ello encriptado con la clave del servidor (K_B).

Una vez que el cliente obtiene un ticket (del árbitro) puede usarlo para entrar muchas veces en el servidor (hasta que caduque).

Conviene resaltar que el cliente no puede desencriptar el ticket (no conoce K_B), pero puede presentar el ticket encriptado al servidor.

Obsérvese que el ticket demuestra al servidor que el cliente es quien dice ser, pero si un espía captura el ticket podría usarlo para entrar en el servidor. Para evitarlo existe el authenticator. El **authenticator** es un mensaje que genera el cliente cada vez que quiere entrar en el servidor encriptado con la clave de sesión. Este mensaje no se puede reutilizar, pero esta restricción no incrementa el trabajo del árbitro ya que el cliente lo puede generar sin ayuda del árbitro.

En concreto el authenticator contiene los siguientes campos:

$$\text{Authenticator} = E(K_s, (A, T_s))$$

Es decir, contiene la identidad de A y un timestamp T_s (que no se debe repetir), encriptados con la clave de sesión.

El servidor comprueba que el authenticator sea correcto y lo cachea para evitar un posible replay attack. Ahora un espía que captura tanto el ticket como el authenticator no puede usarlo para entrar en el servidor.

7.7 Obtener el ticket inicial

Antes de que el cliente pueda pedir tickets al servidor Kerberos, debe obtener un ticket llamado **TGT (Ticket Granting Ticket)**. El TGT es un ticket que sirve al cliente para pedir servicios al servidor Kerberos. Es decir, el servidor Kerberos es un servidor más, para el cual el cliente también puede obtener un ticket.

Como el cliente no debe enviar su password por la red, lo que envía es una petición al servidor Kerberos para que le devuelva un TGT para él. El servidor

le devuelve un TGT encriptado con la clave secreta de A , con lo que sólo A puede desencriptarlo:

$$TGT_A = E(K_A, (T, L, K_S, S))$$

Donde S es el identificador del servidor Kerberos (por si hay varios servidores Kerberos).

Si A es un impostor no podrá desencriptar el TGT_A , ya que no conoce K_A . Si no, A ya tiene una clave de sesión K_S que usará para pedir el resto de los tickets al servidor Kerberos¹.

7.8 Obtener tickets de servidor

Ahora el cliente querrá acceder a otros servidores, para lo cual deberá obtener un ticket para cada servidor al que quiera acceder. Esto lo hará utilizando la clave de sesión K_S que obtuvo en el TGT_A para pedir al servidor Kerberos tickets para otro servidor.

Para evitar un replay attack el cliente también debe crear un authenticator, que conviene recordar que debía ser único para cada petición. Una vez que obtenga un ticket para un servicio puede presentar ese ticket junto con su correspondiente authenticator para acceder a ese servicio.

7.9 Servidor maestro y esclavo

En una red con muchos usuarios es importante que el servidor Kerberos siempre esté disponible. Para ello podemos montar varios servidores Kerberos que sirvan uno o más dominios. Ahora los clientes configuran en su fichero `/etc krb5.conf` todos los servidores Kerberos en que confíen.

De todos los servidores Kerberos, a uno se le designará como master, y a los demás como slaves. El master es el único servidor por el que podremos actualizar los usuarios y sus claves (a través del servicio de administración del puerto 749). Todos los demás servidores actúan como esclavos y se actualizan periódicamente a partir del master.

¹ Obsérvese que K_S es una clave de sesión entre A y el servidor Kerberos. En los apartados anteriores a esta clave la denotamos con K_A .

8 Certificación distribuida de claves

La gestión de claves mediante un KDC suele implicar montar un servidor o contratar el servicio de certificación, cosa a la que no están dispuestos muchos usuarios.

Una solución alternativa al uso de una CA para que nos firme la clave (véase apartado 5) es la que usan programas como GPG (Gnu Privacy Guard) o PGP (Pretty Good Privacy). Esta aproximación se basa en el uso de **introductiones**, que son otros usuarios del sistema que firman las claves de sus conocidos, es decir, presentan usuarios a otros usuarios.

El protocolo se basa en la siguiente idea: Supongamos que *A* conoce a *B*, y *B* conoce a *C*, pero *A* y *C* no se conocen (véase la Figura 5.5 (a)). En este caso *B* podría firmar la clave pública de *C* para que *A* pudiera confiar en la clave pública de *C*, y viceversa.

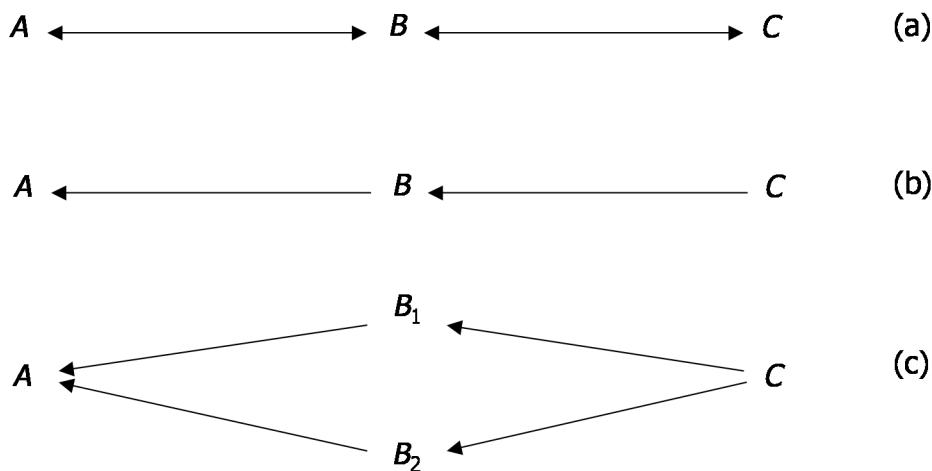


Figura 5.5 Certificación distribuida de claves

Para que este sistema funcione es importante definir las **confianzas (trust)** que tienen unos usuarios en las firmas de otros. Por ejemplo, *A* puede confiar en la clave pública de *C* si:

1. *A* confía totalmente en la firma de *B*, y *B* confía totalmente en la firma de *C* (véase Figura 5.5 (b)).
2. *A* no confía en las firmas de *B1, *B2 individualmente, pero sí confía en sus firmas si los dos firman una clave (véase Figura 5.5 (c)).**

En estos sistemas los usuarios son los que definen las relaciones de confianza que quieren tener con los demás.

9 GPG (Gnu Privacy Guard)

9.1 Introducción

En este apartado vamos a explicar cómo funciona la herramienta GPG (Gnu Privacy Guard) una herramienta GNU que podemos conseguir en:

<http://www.gnupg.org>

Existen implementaciones de GPG para muchos sistemas operativos.

La versión para Mac OS X la tenemos en:

<http://macgpg.sourceforge.net>

Esta herramienta es la implementación de GNU de OpenPGP, la especificación pública de PGP documentada en la RFC2240

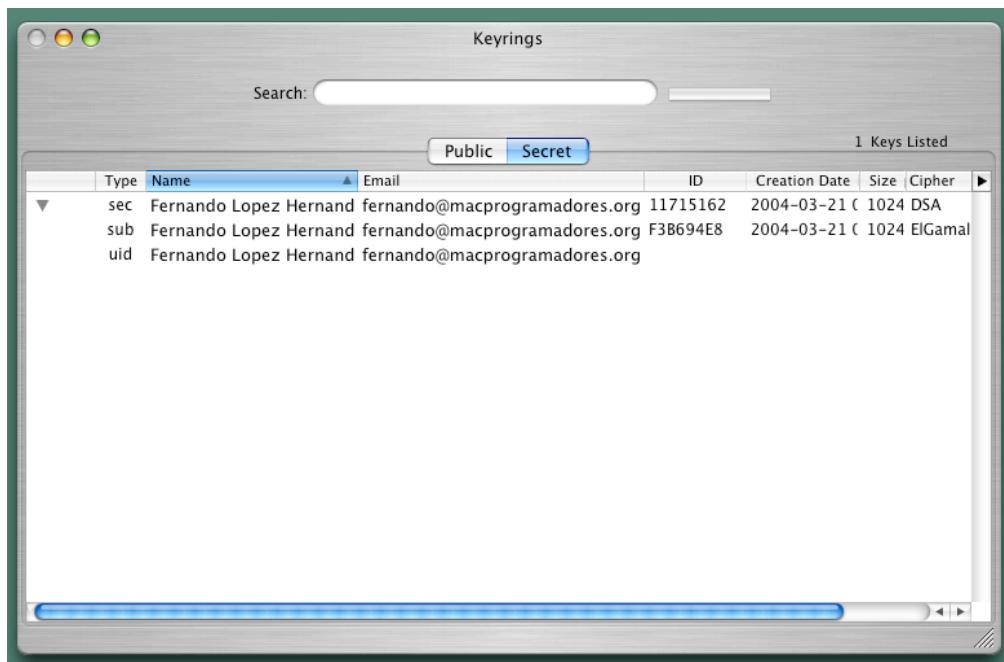
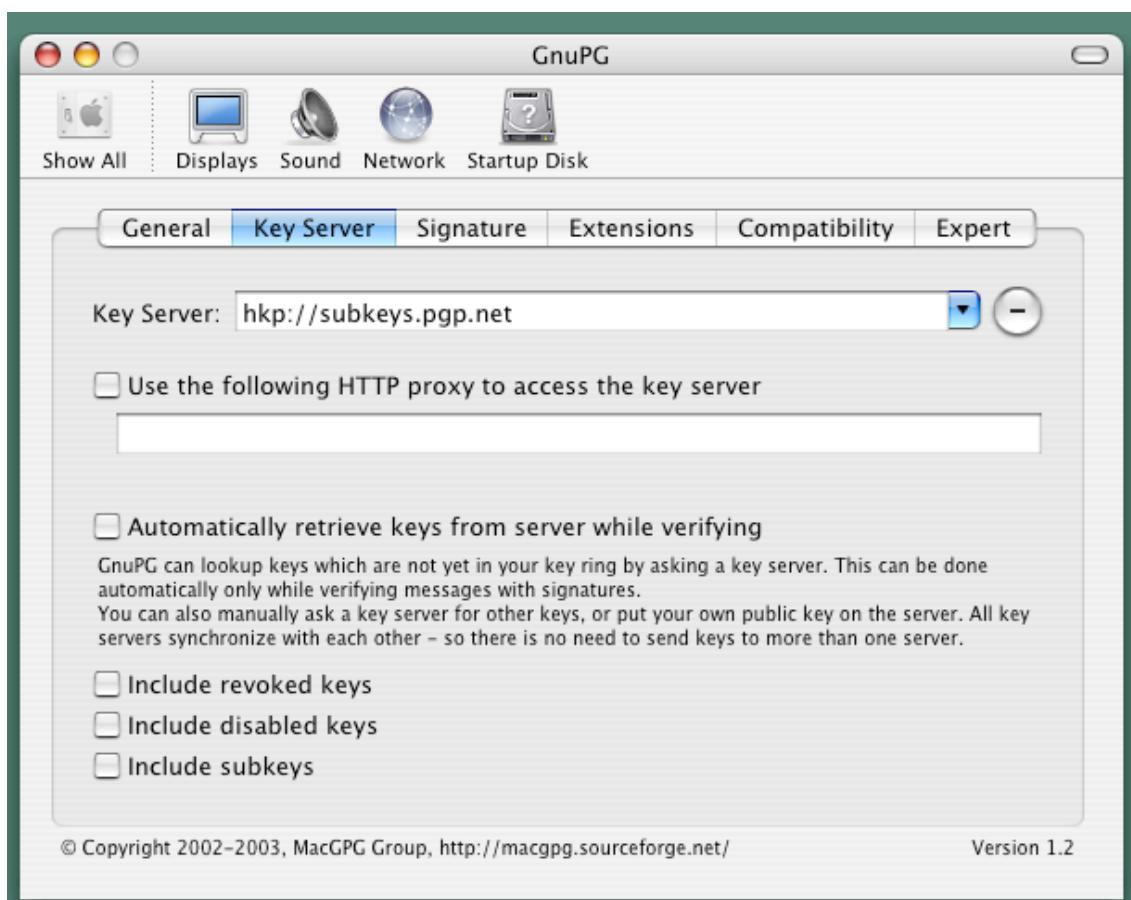
La implementación más conocida de OpenPGP es PGP que podemos encontrar en:

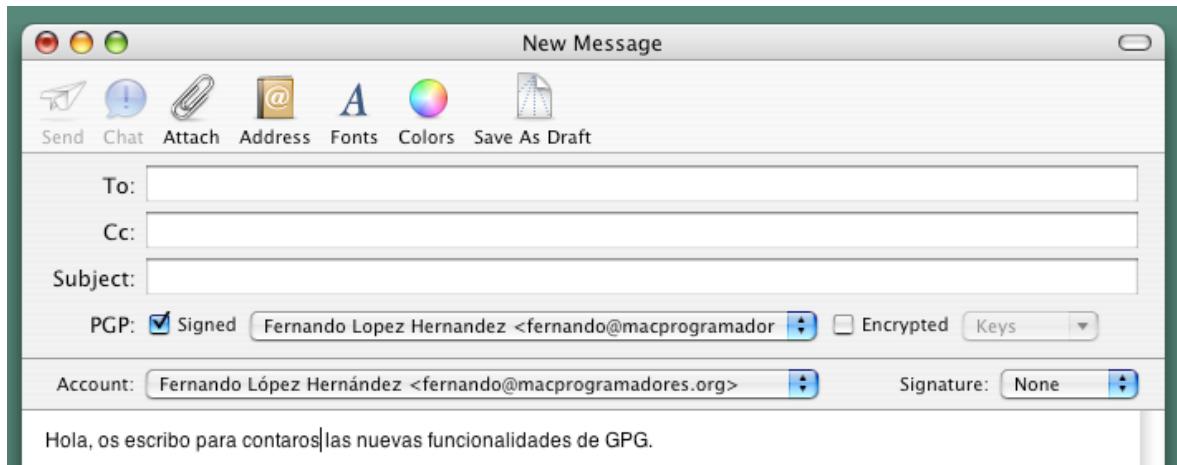
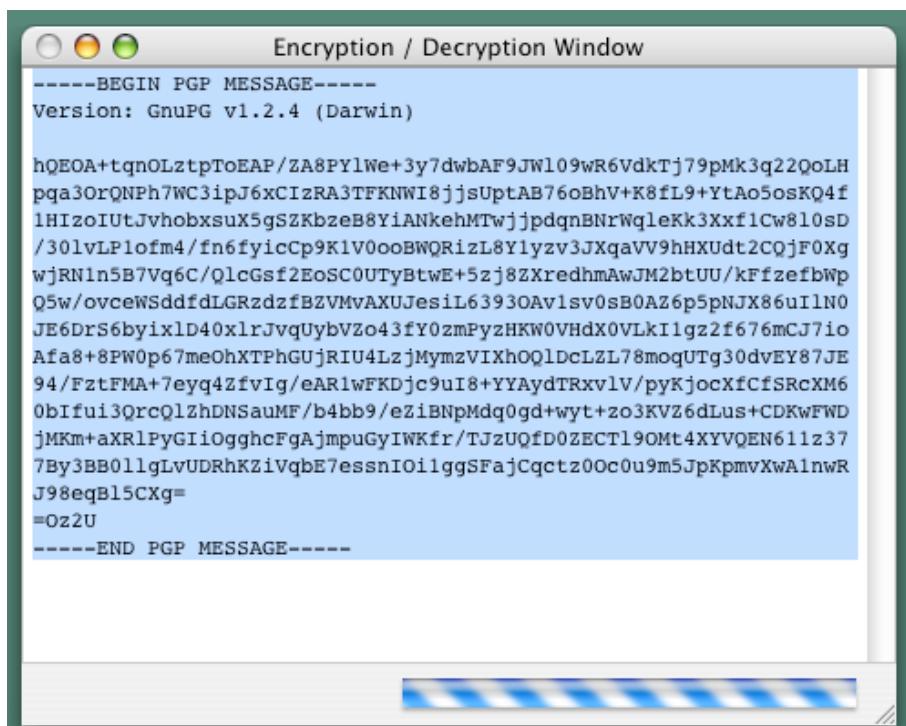
<http://www.pgpi.org>

GPG en principio es un comando de consola llamado `gpg`, pero existen interfaces gráficas que se montan encima de este comando para facilitar su uso.

Entre ellas destacan:

1. GPG Keyrings. Una herramienta visual que nos permite administrar las claves de nuestro llavero. Véase Figura 5.7 (a).
2. GPG Preferences. Un cuadro de diálogo que nos permite personalizar las opciones de GPG. Una vez instalado accedemos a él desde las preferencias del sistema de Mac OS X. Véase Figura 5.7 (b).
3. GPG Mail. Un Add-On para AppleMail que nos permite encriptar / firmar / desencriptar / verificar los mensajes de correo que vamos a enviar. Véase Figura 5.7 (c).
4. GPG DropThings. Una herramienta que nos permite encriptar y firmar documentos fácilmente. Véase Figura 5.7 (d).

**Figura 5.7 (a):** GPG Keyrings**Figura 5.7 (b):** GPG Preferences

**Figura 5.7 (c): GPG Mail****Figura 5.7 (d): GPG Drop Things**

9.2 Instalación y puesta a punto de GPG

GPG se distribuye empaquetado como un fichero `.pkg` que ejecuta un wizard el cual realiza la instalación.

Una vez acaba la instalación, lo primero que debemos hacer es crearnos un **llavero** (keyring), que es un contenedor de pares de claves pública/privada. Aquí tendremos tanto nuestras claves pública y privada, como las claves públicas de otros usuarios.

En el llavero siempre habrá un **par de claves principales**, que es una clave privada y su correspondiente clave pública. Además puede haber varios **pares de claves subordinadas**, que son pares de claves que también nos representan, pero que están firmadas por nuestra clave principal. La razón de poder crear nuevos pares de claves subordinados es que si tenemos que invalidar una clave subordinada, no tengamos que invalidar la clave principal. P.e. podemos crear un par de claves subordinados que caduquen transcurrido un periodo, o crear un par de claves que nos represente como parte de una determinada empresa. Si después cambiamos de trabajo podemos revocar esta clave subordinada sin revocar la clave principal.

La única condición que se pone a la clave principal es que sea capaz de firmar, no es necesario que pueda encriptar (p.e. puede ser DSA que sólo sirve para firmar), ya que la clave principal se usa para firmar las claves subordinadas, las cuales ya sí podrían encriptar.

Lo primero que debemos hacer cuando vamos a empezar a trabajar con GPG es pedirle que queremos crear un par de claves principal ejecutando el comando `gpg` con el comando `--gen-key`

```
fernando$ gpg --gen-key
Please select what kind of key you want:
 (1) DSA and Elgamal (default)
 (2) DSA (sign only)
 (5) RSA (sign only)
Your selection? 2
```

ElGamal es un algoritmo que sirve tanto para firmar como para encriptar, mientras que DSA sólo sirve para firmar.

En mi caso la clave principal va a ser de tipo 2 ya que sólo la voy a usar para firmar otras claves. El lector puede elegir crear una clave de tipo 1 si quiere que su clave principal además de para firmar las claves subordinadas le pueda servir para encriptar documentos.

Después el programa nos pedirá un tamaño de clave. En general 1024 bits es un tamaño suficientemente seguro, pero el usuario puede elegir un tamaño

de clave mayor si lo desea, lo cual aumenta la dificultad de desencriptar sus mensajes por fuerza bruta, aunque aumentará también el consumo de CPU del programa.

Hay que tener en cuenta que una vez que elijamos el tamaño de clave, éste no se podrá cambiar nunca más.

```
About to generate a new DSA keypair.
      minimum keysize is 768 bits
      default keysize is 1024 bits
      highest suggested keysize is 2048 bits
What keysize do you want? (1024) 1024
```

Por último debemos indicar la fecha de caducidad de la clave principal. Para la mayoría de los usuarios una clave que no caduca nunca es lo más adecuado, pero a veces nos interesa que la clave caduque transcurrido un tiempo. P.e. cuando vamos a estar desempeñando un cargo sólo durante un tiempo limitado y que conocemos de antemano.

```
Please specify how long the key should be valid.
      0 = key does not expire
      <n> = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 0
```

Cuando una clave caduca, todavía la podemos usar para desencriptar documentos que nos envíen, pero no la podemos usar para firmar nuevos documentos. Además los documentos que hayamos firmado aparecerán como caducados a los demás usuarios.

Por último el sistema nos pedirá que le demos un **user-id** para esa clave que acabamos de generar. El objetivo del user-id es identificar de forma única al propietario de esa clave. Normalmente el user-id consta de tres campos:

- El nombre del usuario
- Un comentario
- Una cuenta de correo

```
You need a User-ID to identify your key; the software
constructs the user id from Real Name, Comment and Email
Address in this form: "Heinrich Heine (Der Dichter)
<heinrichh@duesseldorf.de>"
```

```
Real name: Fernando Lopez Hernandez
Email address: fernando@macprogramadores.org
Comment: Alias ActiveMan
You selected this USER-ID:
  "Fernando Lopez Hernandez (Alias ActiveMan)
  <fernando@macprogramadores.org>"
```

Por último GPG nos pide una passphrase con la que proteger nuestro llavero. Esta passphrase se usa para encriptar el fichero de clave privada. Los ficheros de clave pública se guardan sin encriptar, con lo que para acceder a ellos no se nos pedirá clave alguna.

```
You need a Passphrase to protect your secret key.
```

```
Enter passphrase:*****
```

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

```
.++++++...++++++...++++++...++++++...++++++...++++++...+++++.  
++++.++++...++++++...++++...++++++...++++...++++...++++...++++  
++++...++++...++++...++++>++++...++++...>++++...++++...++++
```

public and secret key created and signed.

Note that this key cannot be used for encryption. You may want to use the command "`--edit-key`" to generate a secondary key for this purpose.

Elegir una buena passphrase es crucial para una buena seguridad, ya que si alguien accede a nuestros ficheros, la seguridad entera del sistema dependerá de que el atacante no descubra esta passphrase, y de que no pueda romper la seguridad del fichero con un ataque de diccionario.

Después de ejecutar el programa se nos habrá creado un subdirectorio en nuestro directorio de usuario llamado `~/.gnupg` dentro del cual podremos encontrar los ficheros de la Tabla 5.1.

Fichero	Descripción
<code>gpg.conf</code>	Fichero de configuración de GPG. Es un fichero de texto.
<code>trustdb.gpg</code>	Fichero con las relaciones de confianza que tenemos con los demás usuarios.
<code>pubring.gpg</code>	Fichero destinado a almacenar nuestras claves públicas y las claves públicas de los demás usuarios.
<code>secring.gpg</code>	Fichero destinado a almacenar nuestras claves privadas. Se encuentra encriptado con la passphrase para una mayor seguridad.
<code>randomseed</code>	Fichero usado por GPG para generar números aleatorios.

Tabla 5.1: Ficheros de GPG

9.3 Intercambiar la clave pública

Para comunicarnos con otros usuarios debemos intercambiar primero nuestras claves públicas con ellos. Podemos empezar consultando qué claves tenemos en el llavero con el comando:

```
fernando$ gpg --list-keys
pub 1024D/8774470F 2006-11-18
uid Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Vemos que de momento sólo hay una clave donde:

- `pub` indica que es una clave pública.
- `1024D` indica que tenemos una clave DSA de 1024 bits. Es decir, sólo sirve para firmar.
- `8774470F` es un identificador único para nuestro user-id.
- `2006-11-18` es la fecha de creación de la clave.
- Por último aparece el user-id (nombre, comentario, e-mail) al que está asociada la clave.

9.3.1 Exportar nuestras claves públicas

Para enviar la clave pública a un amigo, primero debemos exportarla con el comando `--export [<user-id>]`, además debemos indicar el fichero donde exportamos la clave pública con la opción¹ `--output <fichero>` de la siguiente forma:

```
fernando$ gpg --output fernando.export --export Fernando
```

Si no proporcionamos el parámetro opcional `<user-id>` se exportarán todas las claves, si lo proporcionamos `<user-id>` puede contener cualquier parte del nombre, comentario o e-mail del user-id cuya clave pública queramos exportar.

Ahora se habrá creado el fichero `fernando.export` con nuestras claves públicas (de momento sólo tenemos una), que enviaremos a nuestros amigos.

¹ GPG diferencia entre dos tipos de argumentos: **comandos**, que indican acciones a realizar como p.e. `--export` y **opciones**, que son modificadores sobre el comportamiento del comando como p.e. `--output`. En caso de usar comandos y opciones, las opciones deben aparecer antes que los comandos. Además, mientras que comandos debemos indicar uno y sólo uno, opciones podemos indicar varias, una, o ninguna.

El fichero generado es en formato binario, podemos pedir a GPG que lo genere en formato ASCII con el fin de facilitar su legibilidad, o el poder hacer un copy/paste de él en un e-mail usando la opción `--armor` así:

```
fernando$ gpg --output fernando.export --armor --export
Fernando
```

El fichero generado ahora sería:

```
fernando$ cat fernando.export
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.5 (Darwin)

mQGiBDyLcfkRBADb1PaMYWlafcBh7+kIycZjXiBJ3kqBqgVM1bLiK/qhTpIech
K6HqIk0tbWYSGcpGFhf3Tbj8UMgZm3Vk7fcFFPRPI9aIgJQm2IXob0FEcoFdbI
7d/Es7g2OSI03ky94hzS0+vNY+b1c11md9vD5YLa51ea8nKwbpDM6uEuz5ZNxw
Cg1SxqjTuqvm/+gLa9KmgievWPf4kD/0RULFE7vxCNv32/mh7q0BmiKYMkvk/x
GUNGGkLLprPU7Zar4wTP7GodrL9KDXsO8XMzoxkFTZ87C9F1cZium4ymqClvZn
BfzLkdUWZpykXAgFgWB5q/a1XYxNp0EqPkTfcPg7bRakdg3nT3WV1CrMk8I2hz
BSnNPcWWcPekC5AA/9tg4hHPKV1N/AnrVzJnCMqg5RF9YOQuw6e+NPGg2bBXn
sWtzUzs/YRfSaml+L7zkdAVZz1GOow38IDMGvx50+Mt0ZtHZUB25LcfWBcoT2J
UyhvQ7Qw4HSxYH3DhzonFYe4ZQihsrB1QEeAb9CzdjtGAiPma4bay30uCvMMCY
XGVLRKrmVybmFuZG8gTG9wZXogSGVybmfuZGV6IChbBGlhcyBBY3RpdmVNYW4p
IDxmZXJuYW5kb0BtYWNwcm9ncmFtYWRvcvcmVzLm9yZz6IVwQTEQIAFWUCPITx+Q
ULBwoDBAMVAwIDFgIBAheAAAoJEKhCIWYDjmI9NOEAnjAn0Ofa7pnqKqoVxDX
vm088TOgAJ9n8x0WaqvimFOAmruoAK25MgyZ0LkBDQQ8i4SUEAQAOtsr4dA60W
naFpG1FLIZKmVPCD5M0VtNbi/9F1DBxCU3+ZwkJ9w1sCR5bxTtobQGx4EIAv0a
ptE7/MiydR8a7iJprzUpEMILxsctOcnJnLsKPrvMFv8nwaxJVxffsQjLVbZb4u
FJb7DY6GYIUEfsNKVV2zCH9T3LZ+v2TwgtTtcAAwcD/3iV0SK6GLtez6rTWSEn
xVJXAMwMF4ShwOfSv+cv4xjTGuBzqlIqogG4C/eKW971AXdIRsPQnDdJYLje8k
onJ5 SLt
-----END PGP PUBLIC KEY BLOCK-----
```

9.3.2 Importar las claves públicas de otros usuarios

Supongamos que Carolina recibe el fichero `fernando.export` y quiere importarlo en su llavero. Podría usar el comando `--import <fichero>` así:

```
carolina$ gpg --import fernando.export
gpg: key 8774470F: public key "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>" imported
gpg: Total number processed: 1
gpg:                      imported: 1
```

Ahora si Carolina lista sus claves verá que, además de sus claves, tiene la clave pública de Fernando en su llavero:

```
carolina$ gpg --list-keys
pub    1024D/24A736EA 2006-11-18
uid    Carolina Fernandez Arias <carolina@macprogramadores.org>
```

```
pub 1024D/8774470F 2006-11-18
uid Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

9.3.3 Validar las claves importadas

Uno de los principales puntos débiles que tiene GPG es el man in the middle attack que se puede producir en la exportación/importación de claves. Si Fernando manda una clave pública a Carolina, y el atacante cambia esta clave en el camino, Carolina recibirá la clave pública del atacante creyendo que es la de Fernando, con lo que creerá que documentos firmados por el atacante son documentos firmados por Fernando, y cuando encripte ficheros que quiera enviar a Fernando con la clave pública, los encriptará realmente con la clave del atacante.

Para evitar este ataque, Carolina debe validar la clave pública recibida. Esta validación debe hacerse en dos pasos:

1. Comprobar el fingerprint de la clave pública

Carolina calcula un hash de la clave pública recibida, al que llaman fingerprint y comprueba con Fernando que coinciden los fingerprint. Esto puede hacerse en persona o por teléfono.

Para saber el fingerprint de la clave pública Fernando ejecuta el comando --fingerprint

```
fernando$ gpg --fingerprint
pub 1024D/8774470F 2006-11-18
      Key fingerprint = BA10 1898 E3BD 1215 3300 559E 7B29
1A6C 8774 470F
uid Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Carolina realiza la misma operación:

```
carolina$ gpg --fingerprint
pub 1024D/24A736EA 2006-11-18
      Key fingerprint = CCD9 BF04 C171 B292 235F A830 23A9
4B7B 24A7 36EA
uid Carolina Fernandez Arias <carolina@macprogramadores.org>

pub 1024D/8774470F 2006-11-18
      Key fingerprint = BA10 1898 E3BD 1215 3300 559E 7B29
1A6C 8774 470F
uid Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Y ahora ya pueden comprobar que los fingerprint coincidan. En este caso el fingerprint de la clave pública principal de Fernando es BA10 1898 E3BD 1215 3300 559E 7B29 1A6C 8774 470F que realmente coincide a Carolina y Fernando. Luego ahora Carolina sabe que la clave pública que ha recibido es la correcta.

2. Carolina firma la clave pública recibida

La clave pública que ha importado Carolina es inválida hasta que Carolina la valide. Al validarla Carolina indica que confía en la autenticidad de la clave pública recibida. Para ello Carolina debe firmarla con su clave principal. Como Carolina ya sabe que la clave pública que tiene en su llavero es auténtica (lo comprobó con el fingerprint), la firma utilizando el comando `--sign-key`:

```
carolina$ gpg --sign-key Fernando

pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC          trust: unknown      validity: unknown

Primary key fingerprint: BA10 1898 E3BD 1215 3300 559E 7B29
1A6C 8774 470F

Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>

Are you sure that you want to sign this key with your
key "Carolina Fernandez Arias <carolina@macprogramadores.org>"?
(24A736EA)

Really sign? (y/N) y

You need a passphrase to unlock the secret key for
user: "Carolina Fernandez Arias
<carolina@macprogramadores.org>"
1024-bit DSA key, ID 24A736EA, created 2006-11-18
Enter passphrase:*****
```

En cualquier momento Carolina puede ver qué claves públicas tiene firmadas usando el comando `--list-sigs`:

```
carolina$ gpg --list-sigs
pub 1024D/24A736EA 2006-11-18
uid Carolina Fernandez Arias <carolina@macprogramadores.org>
sig 3 24A736EA 2006-11-18 Carolina Fernandez Arias
<carolina@macprogramadores.org>

pub 1024D/8774470F 2006-11-18
uid Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
sig 3 8774470F 2006-11-18 Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>
sig 24A736EA 2006-11-18 Carolina Fernandez Arias
<carolina@macprogramadores.org>
```

Aquí se indica que la clave pública de Carolina está firmada por ella misma, y que la clave pública de Fernando está firmada por él mismo, y por Carolina.

9.4 Firmar un documento con la clave principal

Para firmar un documento se usa el comando `--sign <file>`. Por ejemplo, si queremos firmar el fichero `carta.txt` haríamos:

```
fernando$ gpg --sign carta.txt
You need a passphrase to unlock the secret key for
user: "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
1024-bit DSA key, ID 8774470F, created 2006-11-18
```

Enter passphrase:*****

El programa nos genera un fichero llamado `carta.txt.gpg` con la carta firmada y comprimida en formato binario.

Ahora Carolina podría verificar la carta usando el comando `--verify` así:

```
carolina$ gpg --verify carta.txt.gpg
gpg: Signature made Sat Nov 18 23:51:43 2006 CET using DSA key
ID 8774470F
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
```

Aunque con el comando `--verify` sólo puede saber que el documento es auténtico, pero no puede leer su contenido para poder leer su contenido debe usar el comando `--decrypt <fichero>` así:

```
carolina$ gpg --decrypt carta.txt.gpg
gpg: Signature made Sat Nov 18 23:51:43 2006 CET using DSA key
ID 8774470F
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
```

Hola Carolina

El comando manda el fichero desencriptado a la salida estándar. Si queremos mandarlo a un fichero podemos usar la opción `--output <file>`

También Fernando puede enviar la carta firmada en texto legible usando la opción `--clearsign` con la que el fichero generado tiene formato ASCII legible:

```
fernando$ gpg --clearsign carta.txt
You need a passphrase to unlock the secret key for
```

```
user: "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
1024-bit DSA key, ID 8774470F, created 2006-11-18
Enter passphrase:*****
```

El comando genera el fichero `carta.txt.asc` con el documento firmado pero en formato ASCII legible:

```
fernando$ cat carta.txt.asc
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Hola que tal
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.5 (Darwin)

iD8DBQFFX49SeykabId0Rw8RAox7AJ9Z1dzotvAe6331XdGESTcqEftIWwCg40
NR
bw4IE18BaXIqVr7beuBSybg=
=TBMA
-----END PGP SIGNATURE-----
```

Ahora Carolina puede leer tranquilamente el fichero, y si quiere comprobar su autenticidad usa el comando:

```
carolina$ gpg --verify carta.txt.asc
gpg: Signature made Sat Nov 18 23:55:14 2006 CET using DSA key
ID 8774470F
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
```

Por último, a veces es recomendable depositar la firma de un fichero en otro fichero aparte. P.e. en muchos programas ejecutables que se distribuyen en Internet (p.e. el propio GPG), muchas veces la firma se pone aparte, para que el usuario no se vea obligado a usar GPG para desencriptar el fichero.

Para crear un fichero de firma aparte usamos el comando `--detach-sig <fichero>` así:

```
fernando$ gpg --detach-sig miejecutable
You need a passphrase to unlock the secret key for user:
"Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
1024-bit DSA key, ID 8774470F, created 2006-11-18
Enter passphrase: *****
```

Ahora se habrá generado el fichero `miejecutable.sig` con la firma de Fernando. Carolina podrá comprobar la firma así:

```
carolina$ gpg --verify miejecutable.sig miejecutable
gpg: Signature made Sun Nov 19 00:02:31 2006 CET using DSA key
ID 8774470F
```

```
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
```

9.5 Crear claves subordinadas

La clave que ha estado usando Fernando hasta ahora es una clave DSA que sirve sólo para firmar otras claves y documentos. Si queremos enviar y recibir ficheros encriptados necesitamos crear una clave subordinada. Antes de crear la clave subordinada podemos empezar consultando qué claves tenemos en el llavero con el comando:

```
fernando$ $gpg --list-keys
pub 1024D/8774470F 2006-11-18
uid Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Vemos que de momento sólo tenemos creada la clave principal. Vamos a crear ahora la clave subordinada que vamos a usar para encriptar. Para ello usamos el comando **--edit-key** seguido del user-id de la clave. Como es habitual, no hace falta escribir todo el user-id, basta con escribir parte de éste:

```
fernando$ gpg --edit-key Fernando
Secret key is available.
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
Command>
```

Ahora para crear la clave subordinada usamos el comando **addkey** así:

```
Command> addkey
Key is protected.
You need a passphrase to unlock the secret key for
user: "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
1024-bit DSA key, ID 8774470F, created 2006-11-18
Enter passphrase:*****
```

El comando nos pide la passphase porque la clave subordinada va a firmarse con la clave principal, y hay que desencriptarla.

Después se piden los demás datos necesarios para crear la clave subordinada. Empieza preguntándonos qué tipo de clave queremos, y vamos a indicar que la queremos del tipo ElGamal, porque la vamos a usar sólo para encriptar (para firmar ya tenemos la clave principal).

```
Please select what kind of key you want:
```

```

(2) DSA (sign only)
(4) Elgamal (encrypt only)
(5) RSA (sign only)
(6) RSA (encrypt only)
Your selection? 4
ELG-E keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 2048
Requested keysize is 2048 bits
Please specify how long the key should be valid.
    0 = key does not expire
    <n> = key expires in n days
    <n>w = key expires in n weeks
    <n>m = key expires in n months
    <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
Really create? (y/N) y
We need to generate a lot of random bytes. It is a good idea
to perform some other action (type on the keyboard, move the
mouse, utilize the disks) during the prime generation; this
gives the random number generator a better chance to gain
enough entropy.
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++.+++++
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC          trust: ultimate    validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>

```

Ya podemos abandonar el modo de edición con el comando `quit`

```

Command> quit
Save changes? (y/N) y

```

Ahora la clave subordinada está creada. Podemos volver a consultar por las claves del llavero usando el comando:

```

fernando$ gpg --list-keys
pub    1024D/8774470F 2006-11-18
uid    Fernando Lopez Hernandez (Alias ActiveMan)
      <fernando@macprogramadores.org>
sub    2048g/E31AD51A 2006-11-19

```

Vemos que ahora también aparece la clave subordinada marcada con `sub`. `2048g` significa que es una clave ElGamal de 2048 bits.

9.6 Encriptar con la clave subordinada

Podemos encriptar un documento usando la clave subordinada. En concreto usamos la clave pública para encriptar y la clave privada para desencriptar.

Para encriptar un documento usamos el comando `--encrypt <fichero>` junto con la opción `--recipient <user-id>`. P.e. si queremos encriptar la carta que escribimos antes usamos:

```
fernando$ gpg --recipient Fernando --encrypt carta.txt
```

El comando nos genera el fichero `carta.txt.gpg` con la carta encriptada con la clave pública subordinada de Fernando.

Ahora para desencriptar el fichero usamos el comando `--decrypt <fichero>` así:

```
fernando$ gpg --decrypt carta.txt.gpg
You need a passphrase to unlock the secret key for user:
  "Fernando Lopez Hernandez (Alias ActiveMan)
  <fernando@macprogramadores.org>"
2048-bit ELG-E key, ID E31AD51A, created 2006-11-19
(main key ID 8774470F)
Enter passphrase:*****
```

gpg: encrypted with 2048-bit ELG-E key, ID E31AD51A, created
2006-11-19 "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
Hola Carolina

Si queremos enviar la salida a otro fichero podemos usar la opción `--output <fichero>`.

9.7 Exportar la clave subordinada

La criptografía de clave pública se suele usar para que otra persona, p.e. Carolina, pueda enviarnos ficheros encriptados de forma segura.

Para que Carolina pueda encriptar ficheros con la clave pública de Fernando, primero Carolina debe disponer de la clave pública de Fernando, con lo que Fernando debe exportar su clave subordinada (y no sólo su clave principal).

Para ello, como vimos en el apartado 9.3, Fernando usaría el comando:

```
fernando$ gpg --output fernando.export --export Fernando
```

Ahora Carolina debe importar el fichero `fernando.export` igual que explicamos antes:

```
carolina$ $gpg --import fernando.export
gpg: key 8774470F: "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>" 1 new subkey
gpg: Total number processed: 1
gpg:                 new subkeys: 1
```

Si ahora consultamos las claves de Carolina, vemos que Carolina ya tiene la clave subordinada de Fernando E31AD51A:

```
carolina$ gpg --list-keys Fernando
pub 1024D/24A736EA 2006-11-18
uid Carolina Fernandez Arias <carolina@macprogramadores.org>

pub 1024D/8774470F 2006-11-18
uid Fernando Lopez Hernandez (Alias ActiveMan)
    <fernando@macprogramadores.org>
sub 2048g/E31AD51A 2006-11-19
```

En este caso Carolina ya no tiene que validar la clave subordinada, igual que pasó con la clave principal, porque la clave subordinada está firmada por la clave principal de Fernando, y Carolina confía en la clave principal de Fernando.

Ahora Carolina puede encriptar ficheros que quiera enviar a Fernando con el comando:

```
carolina$ gpg --recipient Fernando --encrypt carta.txt
```

Y Fernando lo podrá desencriptar con su clave privada igual que antes.

9.8 Encriptar ficheros con clave secreta

También podemos encriptar un fichero con criptografía de clave secreta. Para ello en vez de usar el comando `--encrypt <file>` usamos el comando `--symmetric <file>`

```
fernando$ gpg --symmetric carta.txt
Enter passphrase: *****
```

El programa genera el fichero `carta.txt.gpg` encriptado con esa clave secreta. Es importante destacar que el comando nos ha pedido una passphrase con la que encriptar el fichero. Esta passphrase no tiene porqué coincidir con la passphrase usada para encriptar la clave privada, ya que no pide la passphrase para desencriptar la clave privada, sino para hacer un PBE.

Para desencriptar el fichero usamos el comando `--decrypt <file>` igual que en la criptografía de clave asimétrica:

```
fernando$ gpg --output carta.txt --decrypt carta.txt.gpg
Enter passphrase: *****
```

9.9 Añadir más identidades al llavero

Imaginemos ahora que Fernando está trabajando como profesor para www.uam.com, y tiene una cuenta llamada `f.lopez@uam.com`. En este caso Fernando puede añadir esta nueva identidad a su llavero. Para ello tiene que pasar al modo de edición, y añadir un nuevo user-id con el comando `adduid`

```
fernando$ gpg --edit-key Fernando
Secret key is available.

pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>

Command> adduid
Real name: Fernando Lopez Hernandez
Email address: f.lopez@uam.es
Comment: UAM
You selected this USER-ID:
  "Fernando Lopez Hernandez (UAM) <f.lopez@uam.es>"
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a passphrase to unlock the secret key for user:
  "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
```

1024-bit DSA key, ID 8774470F, created 2006-11-18
 Enter passphrase:*****
 pub 1024D/8774470F created: 2006-11-18 expires: never
 usage: SC trust: ultimate validity: ultimate
 sub 2048g/E31AD51A created: 2006-11-19 expires: never
 usage: E
 [ultimate] (1) Fernando Lopez Hernandez (Alias ActiveMan)
 <fernando@macprogramadores.org>
 [unknown] (2). Fernando Lopez Hernandez (UAM)
 <f.lopez@uam.es>

Ahora podemos abandonar con:

```
Command> quit
Save changes? (y/N) y
```

Y vemos que Fernando tiene dos identidades:

```
fernando$ gpg --list-keys
gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust
model
```

```

gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m,
0f, 1u
pub  1024D/8774470F 2006-11-18
uid  Fernando Lopez Hernandez (UAM) <f.lopez@uam.es>
uid  Fernando Lopez Hernandez (Alias ActiveMan)
      <fernando@macprogramadores.org>
sub  2048g/E31AD51A 2006-11-19

```

La primera que aparece es la **identidad primaria**, es decir, la primera identidad que aparecerá al receptor del mensaje. Es decir, cuando alguien reciba un mensaje firmado y lo verifique, obtendrá una respuesta de la forma:

```

fernando$ gpg --verify carta.txt.asc
gpg: Signature made Sun Nov 19 17:46:09 2006 CET using DSA key
ID 8774470F
gpg: Good signature from "Fernando Lopez Hernandez (UAM)
<f.lopez@uam.es>"
gpg: aka "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
```

Lógicamente, si Carolina todavía no ha exportado el pseudónimo, obtendrá:

```

carolina$ gpg --verify carta.txt.asc
gpg: Signature made Sun Nov 19 17:46:09 2006 CET using DSA key
ID 8774470F
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
```

Para cambiar el user-id primario podemos pasar de nuevo a modo de edición, seleccionar una identidad con el comando `uid`, y fijarla como primaria con el comando `primary`. Por ejemplo, para que la identidad de `fernando@macprogramadores.org` sea la primaria haríamos:

```

fernando$ gpg --edit-key Fernando
Secret key is available.
pub 1024D/8774470F  created: 2006-11-18  expires: never
usage: SC
                  trust: ultimate    validity: ultimate
sub  2048g/E31AD51A  created: 2006-11-19  expires: never
usage: E
[ultimate] (1). Fernando Lopez Hernandez (UAM)
<f.lopez@uam.es>
[ultimate] (2)  Fernando Lopez Hernandez (Alias ActiveMan)
                <fernando@macprogramadores.org>
```

Command> **uid 2**

```

pub 1024D/8774470F  created: 2006-11-18  expires: never
usage: SC          trust: ultimate    validity: ultimate
sub* 2048g/E31AD51A created: 2006-11-19  expires: never
usage: E
[ultimate] (1). Fernando Lopez Hernandez (UAM)
<f.lopez@uam.es>
```

```
[ultimate] (2)* Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

El punto detrás de (1) indica cual es la identidad primaria, el asterisco indica cual es la identidad seleccionada con `key`. El comando `primary` hace que la identidad seleccionada pase a ser la primaria:

```
Command> primary
You need a passphrase to unlock the secret key for
user: "Fernando Lopez Hernandez (UAM) <f.lopez@uam.es>"  

1024-bit DSA key, ID 8774470F, created 2006-11-18  

Enter passphrase:*****  

pub 1024D/8774470F created: 2006-11-18 expires: never  

usage: SC trust: ultimate validity: ultimate  

sub* 2048g/E31AD51A created: 2006-11-19 expires: never  

usage: E  

[ultimate] (1) Fernando Lopez Hernandez (UAM)  

<f.lopez@uam.es>  

[ultimate] (2)* Fernando Lopez Hernandez (Alias ActiveMan)  

<fernando@macprogramadores.org>
```

Ahora podemos abandonar con:

```
Command> quit  
Save changes? (y/N) y
```

9.10 Añadir más claves al llavero

Una ocasión en la que resulta útil añadir más claves subordinadas al llavero es cuando una clave con la que vamos a firmar mensajes tiene fecha de caducidad. Imaginemos que la cuenta `f.lopez@uam.com` caducará cuando acabe el curso. Fernando podría querer crear una clave subordinada que le represente sólo hasta que acabe su trabajo. Para ello debe pasar a modo de edición y crear una nueva clave con el comando `addkey`:

```
fernando$ gpg --edit-key Fernando
Secret key is available.  

ub 1024D/8774470F created: 2006-11-18 expires: never  

usage: SC trust: ultimate validity: ultimate  

sub 2048g/E31AD51A created: 2006-11-19 expires: never  

usage: E  

[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)  

<fernando@macprogramadores.org>  

[ultimate] (2) Fernando Lopez Hernandez (UAM)  

<f.lopez@uam.es>  

Command> addkey  

Key is protected.  

You need a passphrase to unlock the secret key for user:  

"Fernando Lopez Hernandez (Alias ActiveMan)  

<fernando@macprogramadores.org>"  

1024-bit DSA key, ID 8774470F, created 2006-11-18
```

Enter passphrase:*****

Como la clave la necesitamos sólo para firmar, elegimos el tipo 2, e indicamos el plazo de caducidad:

```
Please select what kind of key you want:  
(2) DSA (sign only)  
(4) Elgamal (encrypt only)  
(5) RSA (sign only)  
(6) RSA (encrypt only)  
Your selection? 2  
DSA keypair will have 1024 bits.  
Please specify how long the key should be valid.  
    0 = key does not expire  
<n>  = key expires in n days  
<n>w = key expires in n weeks  
<n>m = key expires in n months  
<n>y = key expires in n years  
Key is valid for? (0) 4m  
Key expires at Mon Mar 19 18:00:04 2007 CET  
Is this correct? (y/N) y  
Really create? (y/N) y  
We need to generate a lot of random bytes. It is a good idea  
to perform some other action (type on the keyboard, move the  
mouse, utilize the disks) during the prime generation; this  
gives the random number generator a better chance to gain  
enough entropy.  
+++++  
++.+++++++.+++++++.+++++.+++++++.+++++  
+++++++.+++++++.+++++++.+++++.+++++++.+++++  
.....  
....  
pub 1024D/8774470F created: 2006-11-18 expires: never  
usage: SC trust: ultimate validity: ultimate  
sub 2048g/E31AD51A created: 2006-11-19 expires: never  
usage: E  
sub 1024D/4493F34C created: 2006-11-19 expires: 2007-03-19  
usage: S  
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)  
<fernando@macprogramadores.org>  
[ultimate] (2) Fernando Lopez Hernandez (UAM)  
<f.lopez@uam.es>
```

La nueva clave subordinada queda firmada por la clave principal. La primera clave 1024D/8774470F indica que es una clave DSA (`D`) de 1024 bits, y el campo `usage` indica que se puede usar para firmar (`S`) y para certificar (`C`) las otras claves. Esto se debe a que es la clave primaria. La segunda clave subordinada (`sub`) es del tipo El Gammar (`G`), de 2048 bits, y sólo se puede usar para encriptar. La tercera clave también es subordinada (`sub`), de tipo DSA (`D`), de 1024 bits, y sólo se puede usar para firmar (`S`).

9.11 Encriptar y firmar con una determinada clave subordinada

Supongamos que ahora Fernando quiere responder a una duda de la alumna Carolina por correo y quiere firmar el mensaje. Fernando debe especificar que la clave que quiere usar es la clave subordinada con fecha de caducidad, y no la clave principal, para ello puede usar la opción `--default-key <key>` así:

```
fernando$ $gpg --default-key 4493F34C --clearsign carta.txt
You need a passphrase to unlock the secret key for user:
"Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
1024-bit DSA key, ID 4493F34C, created 2006-11-19 (main key ID
8774470F)
Enter passphrase:*****
```

El comando genera el fichero `carta.txt.asc` con la carta firmada.

Ahora Carolina, suponiendo que Carolina ha exportado la clave subordinada, comprobaría la firma así:

```
carolina$ gpg --verify carta.txt.asc
gpg: Signature made Sun Nov 19 18:21:44 2006 CET using DSA key
ID 4493F34C
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
gpg: aka "Fernando Lopez Hernandez (UAM) <f.lopez@uam.es>"
```

Pero si Carolina intentara comprobar la firma cuando ésta ya hubiera caducado obtendría un mensaje avisándola de que la clave es buena, pero está caducada:

```
carolina$ gpg --verify carta.txt.asc
gpg: Signature made Sun Nov 19 18:21:44 2006 CET using DSA key
ID 4493F34C
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
gpg: aka "Fernando Lopez Hernandez (UAM) <f.lopez@uam.es>"
```

gpg: Note: This key has expired!

```
Primary key fingerprint: BA10 1898 E3BD 1215 3300 559E 7B29
1A6C 8774 470F
Subkey fingerprint: 3686 5337 34B8 A0E0 A4B7 4E79 6545
0573 4493 F34C
```

9.12 Borrar claves del llavero

Imaginemos que Fernando ha acabado ya los cursos que estaba dando en la UAM. Ahora querrá borrar su user-id y clave del llavero, para ello debe pasar a modo de edición, seleccionar el user-id a borrar con `uid`, y borrarlo con `deluid`:

```
fernando$ gpg --edit-key Fernando
Secret key is available.

pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
sub 1024D/4493F34C created: 2006-11-19 expires: 2007-03-19
usage: S
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
[ultimate] (2) Fernando Lopez Hernandez (UAM)
<f.lopez@uam.es>
Command> uid 2
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
sub 1024D/4493F34C created: 2006-11-19 expires: 2007-03-19
usage: S
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
[ultimate] (2)* Fernando Lopez Hernandez (UAM)
<f.lopez@uam.es>
Command> deluid
Really remove this user ID? (y/N) y
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
sub 1024D/4493F34C created: 2006-11-19 expires: 2007-03-19
usage: S
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Análogamente para borrar la clave subordinada usaremos el comando `key <n>` para seleccionarla, y `delkey` para borrarla:

```
Command> key 2
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
sub* 1024D/4493F34C created: 2006-11-19 expires: 2007-03-19
usage: S
```

```
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
Command> delkey
Do you really want to delete this key? (y/N) y

pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: ultimate validity: ultimate
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
[ultimate] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Si por el contrario Fernando hubiera sido contratado un año más por CSM, Fernando podría cambiar la fecha de expiración de su clave subordinada usando el comando `expire`.

9.13 Revocar claves

Imaginemos ahora que a Fernando le descubren la passphrase con la que protegía sus claves privadas. Fernando debe revocar sus claves lo antes posible, para que los demás usuarios sepan que él no es quien está generando firmas comprometedoras que están empezando a aparecer en Internet.

La mejor manera de hacerlo es generar un certificado de revocación y enviárselo a todos sus conocidos. Un **certificado de revocación** es un documento firmado con la clave principal de Fernando donde Fernando dice a los demás usuarios que su clave ya no es válida.

Para generararlo tenemos el comando:

```
fernando$ gpg --output fernando.revoke --gen-revoke Fernando
sec 1024D/8774470F 2006-11-18 Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>
Create a revocation certificate for this key? (y/N) y
```

El programa nos pregunta cuál es la razón por la que generamos el certificado de revocación: `compromised` (Alguien a descubierto la clave), `superseded` (alguien está suplantándonos), `no longer used` (no vamos a usar más esa clave).

Please select the reason for the revocation:

- 0 = No reason specified
- 1 = Key has been compromised
- 2 = Key is superseded
- 3 = Key is no longer used
- Q** = Cancel

(Probably you want to select 1 here)

Your decision? **2**

Enter an optional description; end it with an empty line:

```
> Os mando mi nuevas claves, las anteriores me las ha crakeado
un hijo de ...
>
Reason for revocation: Key is superseded
Os mando mi nuevas claves, las anteriores me las ha crakeado
un hijo de ...
Is this okay? (y/N) y
You need a passphrase to unlock the secret key for
user: "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>"
1024-bit DSA key, ID 8774470F, created 2006-11-18
Enter passphrase:*****
ASCII armored output forced.
Revocation certificate created.
Please move it to a medium which you can hide away; if Mallory
gets access to this certificate he can use it to make your key
unusable. It is smart to print this certificate and store it
away, just in case your media become unreadable. But have
some caution: The print system of your machine might store
the data and make it available to others!
```

Como indica el mensaje generado por GPG, si generamos un certificado de revocación para hacer pruebas, debemos asegurarnos de que un atacante no se apodere de él, o podría usarlo para revocar nuestra clave aunque él no la haya crackeado realmente.

Ahora enviamos el certificado de revocación a Carolina que lo mete en su llavero importándolo con el comando `--import <file>`

```
carolina$ $gpg --import fernando.revoke
gpg: key 8774470F: "Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>" revocation certificate
imported
gpg: Total number processed: 1
gpg: new key revocations: 1
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust
model
gpg: depth:0 valid:1 signed:0 trust:0-, 0q, 0n, 0m, 0f, 1u
```

Si ahora Carolina intenta comprobar la carta que le envió Fernando obtiene el siguiente mensaje:

```
carolina$ gpg --verify carta.txt.asc
gpg: Signature made Sun Nov 19 18:21:44 2006 CET using DSA key
ID 4493F34C
gpg: Good signature from "Fernando Lopez Hernandez (Alias
ActiveMan) <fernando@macprogramadores.org>"
gpg: aka "Fernando Lopez Hernandez (UAM) <f.lopez@uam.es>"
gpg: WARNING: This key has been revoked by its owner!
gpg: This could mean that the signature is forged.
gpg: reason for revocation: Key is superseded
gpg: revocation comment: Os mando mi nuevas claves, las
anteriores me las ha crakeado un hijo de ...
```

```
gpg: WARNING: This key is not certified with a trusted
signature!
gpg: There is no indication that the signature belongs to the
owner.
Primary key fingerprint: BA10 1898 E3BD 1215 3300 559E 7B29
1A6C 8774 470F
Subkey fingerprint: 3686 5337 34B8 A0E0 A4B7 4E79 6545 0573
4493 F34C
```

Otra forma de revocar claves es usando el comando de edición `revkey`, que revoca una o mas claves firmándolas como revocadas.

Cuando Carolina importa nuestras claves importa como revocadas las que hayamos marcado como revocadas.

Las claves de nuestro llavero también están firmadas por la clave principal, incluida la propia clave principal, y las claves de los demás usuarios. Podemos revocar la firma de una clave usando el comando `revsig`, que no revoca la clave, sólo revoca que nosotros firmemos esa clave. Esto es útil si p.e. Carolina había firmado una clave de Fernando, y más tarde descubre que esa clave realmente no pertenece a Fernando, sino a un atacante. En este caso Carolina revoca la firma que ella hizo sobre esa clave.

9.14 Usar un servidor de claves

Hasta ahora para intercambiar claves, los usuarios tenían que enviarse el fichero de claves y importarlo manualmente.

Existen en Internet servidores de claves PGP destinados a recoger y distribuir claves públicas. Cuando el servidor recibe una clave, éste la añade a su lista de claves, o la fusiona con la clave anterior, si ésta ya existía. Estos servidores usan o bien el protocolo HTTP o bien el protocolo LDAP, y entre otros servidores de claves tenemos estos:

```
ldap://keyserver.pgp.com
http://pgpkeys.mit.edu:11371
ldap://europe.keys.pgp.com:11370
ldap://certserver.pgp.com
http://keys.gnupg.net
```

Da lo mismo el servidor que usemos ya que entre ellos se intercambian las claves, con lo que poco después de enviar nuestra claves públicas a uno, la clave se encuentra en todos los demás.

Para publicar nuestras claves tenemos el comando `--send-key <user-id>` junto con la opción `--keyserver <servidor>` que indica el servidor al que conectarnos.

Luego Fernando puede publicar sus claves públicas con el comando:

```
fernando$ gpg --keyserver keyserver.pgp.com --send-key  
Fernando  
gpg: success sending to `keyserver.pgp.com' (status=200)
```

Ahora Carolina puede importar las claves de Fernando con el comando --recv-key <clave> así:

```
carolina$ gpg --keyserver keyserver.pgp.com --recv-key  
038E623D  
gpg: requesting key 8774470F from keyserver.pgp.com ...  
gpg: key 8774470F: public key imported  
gpg: Total number processed: 1  
gpg:                      imported: 1
```

Obsérvese que Carolina no puede dar el user-id de Fernando, sino que tiene que dar el número de la clave principal que quiere bajarse del servidor de claves.

Esto no sólo le baja la clave principal, sino también las claves subordinadas de Fernando:

```
carolina$ gpg --list-keys  
pub 1024D/24A736EA 2006-11-18  
uid Carolina Fernandez Arias <carolina@macprogramadores.org>  
  
pub 1024D/8774470F 2006-11-18  
uid Fernando Lopez Hernandez (Alias ActiveMan)  
<fernando@macprogramadores.org>  
sub 2048g/E31AD51A 2006-11-19
```

Fernando podrá volver a enviar sus claves actualizadas al servidor y Carolina también podrá usar el comando anterior para actualizar las claves de Fernando siempre que quiera.

Una gran carencia que tiene actualmente GPG respecto a PGP, es que PGP también permite buscar claves en el servidor usando el user-id (p.e. la dirección de correo), lo cual es normalmente más útil y fácil de hacer.

9.15 Editar las claves privadas del llavero

Hemos dicho que en el llavero se almacenaban pares de claves (pública/privada), esto es así cuando se trata de nuestras claves, pero no cuando se trata de las claves de otro usuario, del cual sólo solemos tener la clave pública.

Podemos ver las claves privadas que tenemos en el llavero usando el comando de edición `toggle` que cambia de mostrar las claves públicas a mostrar las claves privadas:

```
carolina$ gpg --edit-key Carolina
Secret key is available.

pub 1024D/24A736EA created: 2006-11-18 expires: never
usage: SC          trust: ultimate    validity: ultimate
[ultimate] (1). Carolina Fernandez Arias
<carolina@macprogramadores.org>

Command> toggle

sec 1024D/24A736EA created: 2006-11-18 expires: never
(1) Carolina Fernandez Arias <carolina@macprogramadores.org>
```

Vemos que Carolina tiene sus claves públicas y privadas, pero si edita las claves de Fernando, sólo va a tener sus claves públicas:

```
carolina$ gpg --edit-key Fernando
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC          trust: unknown    validity: full
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
[ full ] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
Command> toggle
Need the secret key to do this.
```

Alternativamente podemos ver las claves secretas que tiene Carolina con el comando `--list-secret-keys`

```
carolina$ gpg --list-secret-keys
-----
sec 1024D/24A736EA 2006-11-18
uid Carolina Fernandez Arias <carolina@macprogramadores.org>
```

Que nos muestra las claves secretas de que dispone Carolina (tanto las suyas como las de otros usuarios).

9.16 Web of trust

El punto más débil de GPG es sin duda la posibilidad que tiene el atacante de darnos una clave pública suya fingiendo ser la clave pública de nuestro interlocutor. Nosotros debemos ser conscientes de este hecho e intentar evitarlo.

En principio la mejor forma de evitar este ataque es comprobar el fingerprint de una clave pública contrastándolo con nuestro interlocutor.

Cuando se firma una clave sólo se valida la clave principal, porque las subordinadas se firman por la clave principal. Para ello tenemos los comandos `--fingerprint`, `--sign-key <clave>` y `--list-sigs` que explicamos en el apartado 9.3.3. Alternativamente desde el modo de edición podemos usar los comandos de la Tabla 5.2.

Comando	Descripción
fpr	Para mostrar los fingerprint de la clave principal editada
sign	Para firmar la clave principal que estamos editando
check	Para mostrar las firmas que tiene la clave principal que estamos editando.

Tabla 5.2: Comandos para validar una clave

Decimos que una clave pública es **válida** cuando sabemos que pertenece a quien dice pertenecer. Para validar una clave lo que hacemos es firmarla.

Aunque el proceso de validación por comprobación del fingerprint es bastante seguro, también es bastante engorroso, para intercambiar un gran número de claves o para comunicarnos con personas que no conocemos personalmente.

GPG soluciona este problema con un mecanismo llamado **web of trust**. En este modelo, la responsabilidad de validar las claves públicas se delega en la gente en la que confiamos.

Además del concepto de validez, en GPG existe el concepto de **confianza (trust)**, donde decimos que confiamos en una persona si confiamos en esa persona para que valide claves públicas. Para que esa persona valide una clave pública, lo que tiene que hacer es firmarla.

En la práctica, la confianza es subjetiva, con lo que GPG tiene 5 niveles de confianza para representar los distintos grados de confianza que podemos tener en una persona. Estos niveles de confianza se muestran en la Tabla 5.3.

Nivel de confianza	Letra	Descripción
never	n	Sabemos que esa persona firma claves malintencionadamente.
unknown	q	No sabemos nada sobre esa persona. Es el valor inicial por defecto que tomas las claves públicas cuando las metemos en el llavero.
marginal	m	Esta persona nos inspira confianza, pero no la suficiente como para confiar ciegamente en ella.
full	f	Confiamos totalmente en las firmas de esta persona como si fuera la nuestra propia.
ultimate	u	Confiamos en nosotros mismos.

Tabla 5.3: Niveles de confianza (y de validez) de GPG

El *nivel de confianza* es un valor que cada usuario asigna a los dueños de sus claves, y se considera información privada de ese usuario, con lo que (para evitar malos rollos) no se empaqueta junto con las claves cuando las exportamos, de hecho, el nivel de confianza se almacena en ficheros distintos del directorio `~/.gnupg`, en concreto se almacena en el fichero `trustdb.gpg`.

El *nivel de validez* es un valor que se asigna a las claves (no a sus dueños), y también puede tomar los valores de la Tabla 5.3: `never` la clave es falsa. `unknown` no está fijado el nivel de confianza en esa clave. `marginal` confiamos moderadamente en la clave. Esto ocurre cuando hemos recibido la clave de menos de tres personas en las que tenemos confianza `marginal`. `full` confiamos totalmente en la clave. Esto ocurre cuando, o bien hemos firmado la clave (tras comprobar su fingerprint), o cuando hemos recibido esa clave de tres o más personas en las que tenemos confianza `marginal`. `ultimate` confiamos en la clave porque es nuestra.

En resumen, para decidir si una clave se considera válida, GPG sigue las siguientes reglas:

1. Está firmada por nosotros mismos
2. Está firmada por alguien en quien confiamos
3. Esta firmada por tres personas en las que tenemos confianza marginal

Podemos ver los niveles de validez y confianza en una clave desde el modo de edición. Por ejemplo, Carolina puede ver el nivel de confianza y validez de las claves que tiene de Fernando con el comando:

```
carolina$ gpg --edit-key Fernando
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: unknown validity: full
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
[ full ] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

En este caso vemos que Carolina considera la clave principal de Fernando totalmente válida (`full`), porque ella la firmó en el apartado 9.3.3. Por contra, el nivel de confianza de Carolina en Fernando es `unknown` porque no ha indicado nunca a GPG este valor.

Para cambiar el nivel de confianza que Carolina tiene en Fernando puede ejecutar el comando `trust`:

```
Command> trust
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC trust: unknown validity: full
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
```

```
[ full ] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

```
Please decide how far you trust this user to correctly verify
other users' keys (by looking at passports, checking
fingerprints from different sources, etc.)
```

```
1 = I don't know or won't say
2 = I do NOT trust
3 = I trust marginally
4 = I trust fully
5 = I trust ultimately
m = back to the main menu
```

Your decision? **3**

```
pub 1024D/8774470F created: 2006-11-18 expires: never
usage: SC          trust: marginal      validity: full
sub 2048g/E31AD51A created: 2006-11-19 expires: never
usage: E
[ full ] (1). Fernando Lopez Hernandez (Alias ActiveMan)
<fernando@macprogramadores.org>
```

Es importante tener en cuenta que para que GPG nos deje cambiar el nivel de confianza en un usuario, primero su clave tiene que ser válida.

Por último decir que una clave se puede firmar localmente con el comando --lsign <UserID> (o bien, si estamos en modo de edición, con el comando lsigin), lo cual hace que nosotros confiemos en esa firma, pero no exportemos la firma a otros usuarios a los que exportamos nuestro llavero.

10 Identificación, autentificación y autorización

10.1 Terminología

Existen tres términos que muchas veces se mezclan aunque conviene destacar qué significa cada uno de ellos más detenidamente: La **identificación** es el hecho de reconocer a una persona u organización. La **autentificación** es el hecho de comprobar que alguien (o algo) es quien (o lo que) dice ser, y la **autorización** consiste en dejar a alguien realizar una determinada operación.

Estos tres conceptos no tienen por qué ir unidos, por ejemplo un servidor que deja a una persona realizar determinadas operaciones, la deja porque esta persona se ha *identificado* dando un user y password válidos con lo cual tiene *autorización* para realizar estas tareas. Pero esta persona podría no estar *autenticada*, por ejemplo si un usuario ha dado su user y el password a un amigo, el amigo es el que puede estar realizando estas tareas.

Normalmente los ordenadores no *autentifican* a sus usuarios, sólo los *identifican* (comprobando que conocen el user y el password) y los *autorizan* a realizar determinadas operaciones.

A veces los ordenadores también *autentifican* a sus usuarios. P.e. los sistemas de reconocimiento biométrico que reconocen al usuario en base a la forma de la retina ocular.

También se pueden *identificar* otros objetos que no sean personas, por ejemplo una máquina o una firma digital.

En la literatura, y en la práctica, es muy común encontrar el término *identificación* no sólo para referirse a la identificación, sino para referirse también a la consiguiente *autorización*. El término *autentificación* suele usarse con más precisión, aunque a veces se llama *autentificación* a lo que no es más que una *identificación*.

10.2 Login por password handshake

Para conectarnos a un servidor la forma más simple de ceder autorización a un usuario es en base una identificación mediante un user y password. Pero esta forma de identificación tiene el inconveniente de que un espía podría lograr leer este user y el password, y podría usarlo para entrar a realizar tareas sobre las que no tiene autorización.

Para evitar que un espía pudiera detectar el user y password de un usuario legítimo, muchos sistemas de identificación utilizan una técnica llamada **login por password handshake**. La Figura 5.6 resume el funcionamiento de esta técnica. Básicamente el login por password handshake consiste en que el servidor envía al cliente un mensaje aleatorio llamado **challenge** (desafío), y el cliente tiene que calcular un hash de su password más el challenge, y enviar el resultado de la suma al servidor. Sólo si el hash que recibe el servidor del cliente coincide con el que él ha calculado, daría acceso al cliente.

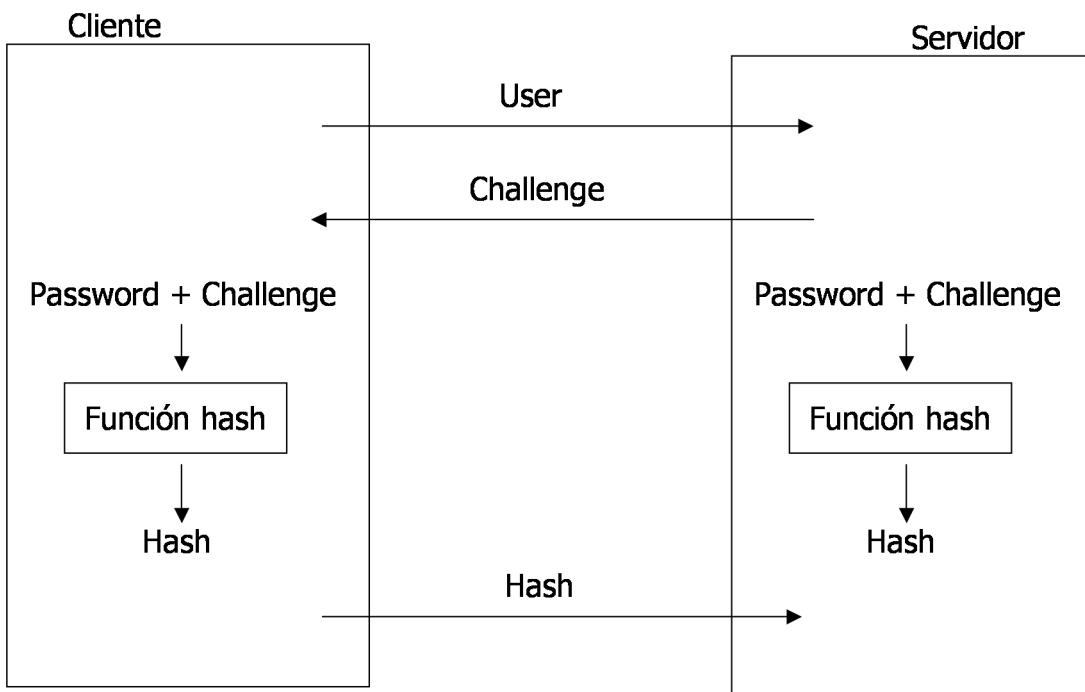


Figura 5.6 Identificación por password handshake

La ventaja de este sistema de identificación es que el password nunca viaja por la red. La seguridad reside en que el servidor cada vez manda al cliente un challenge distinto, porque en caso contrario un atacante activo podría capturar el hash del password+challenge y usarlo para entrar en el sistema.

Sin embargo la identificación por password handshake no es segura ante un ataque de diccionario. La Universidad de Stanford está diseñando un protocolo de identificación por password que sí que es seguro a un ataque por diccionario llamado SRP (Secure Remote Password), publicado en [SRP].

10.3 Login por clave pública

El login por password handshake tiene principalmente dos inconvenientes: El primero es que para que el password sea seguro debe ser largo y aleatorio, si

no el servidor puede sufrir un ataque de diccionario. El segundo es que la mayoría de los sistemas operativos soportan sólo un password por cuenta. En el caso de las cuentas compartidas (p.e. la cuenta de superusuario) esto puede ser un problema, ya que un cambio en el password debe ser comunicado a todos los usuarios de esa cuenta. Para solucionarlo lo que se suele hacer son grupos (p.e. el grupo `admin`).

Una forma más segura de identificar es el login por clave pública. En este tipo de login se genera un par de claves pública/privada. La clave pública se guarda en el servidor y la clave privada en el cliente.

Vamos a ver qué protocolo seguiría el cliente para conectarse al servidor usando identificación por clave pública:

1. El cliente se conecta al servidor indicando su user.
2. El servidor manda un mensaje aleatorio al cliente también llamado **handshake**, para que lo firme con su clave privada
3. El cliente lo firma generando un **authenticator** y se lo envía al servidor.
4. El servidor comprueba la firma con la clave pública y si es correcta da paso al cliente.

En el apartado 13.4 se explica cómo configurar la identificación por clave pública en SSH.

11 Sistemas de identificación OTP

11.1 Introducción

OTP (One Time Password) es un sistema de identificación en el que a un usuario se le autentifica en base a una clave de uso único o one-time password.

El problema que intenta solucionar los sistemas de identificación OTP es el problema de transmitir la clave de forma segura por un canal inseguro, por ejemplo, es útil para servicios como Telnet o FTP donde la clave se transmite sin cifrar.

Lo que hacen los sistemas de identificación OTP es que cuando nos pide el password, en vez de enviar el texto plano del password, enviamos una frase única o one-time password (OTP), que sólo es útil durante una sesión, con lo que si un espía intenta luego usar esa misma password para entrar, el sistema no le dejara.

Para saber el OTP a usar en cada sesión, debe existir un programa que genere un OTP para cada sesión usando una función hash (típicamente MD4 o MD5) a partir de:

- Una clave secreta
- Una semilla. Normalmente consiste en dos letras y 4 dígitos
- Un iteration count (número de veces que se aplica la función hash)

Aunque la clave secreta y semilla son siempre las mismas, el iteration count va a ser distinto cada vez. Típicamente empieza valiendo 100 y va bajando de uno en uno. Cuando llega a 1 hay que volver a fijar otra semilla distinta, ya que el espía puede haber ido guardando los OTP de cada iteration count.

OPIE (One-time Password in Everything) es un estándar creado por el IETF y descrito en las RFC1938 y RFC2243 del cual existe una implementación para sistemas UNIX en [OPIE].

OPIE se creó como un producto de código fuente abierto equivalente a S/Key, un sistema de identificación OTP diseñado por los laboratorios Bell, ya que sus creadores han reclamado el copyright de este producto.

11.2 Tipos de password

Durante el estudio de los sistemas de identificación OTP vamos a ver tres tipos diferentes de password:

El primero de ellos es el **password UNIX**, que es el password que usamos para entrar normalmente en un sistema operativo. Como este método de identificación se ideó para sistemas UNIX, se le llamó password UNIX.

El segundo es el **password OTP**, es el one-time password que tenemos que pasar al servidor con el que nos estamos autenticando. Tanto OPIE como S/Key generan un one-time password consistente en 6 palabras en inglés que se obtiene como consecuencia de ejecutar los comandos `opiekey` o `skey` (o `key` en algunos sistemas) como veremos.

Por último, la **clave secreta** es una clave que nosotros usamos para que los programas `opiekey` o `skey` generen el password OTP, que es el que necesitamos enviar al sistema de identificación. Esta clave es una clave que nunca debemos transmitir por un canal inseguro, ya que si la interceptase un espía la podría usar para calcular los password OTP.

11.3 Instalación

Muchos sistemas UNIX ya traen S/Key o OPIE instalados, con lo que antes de proceder a instalarlo debemos comprobar si ya lo tenemos instalado.

Para ello podemos mirar a ver si existen los comandos:

Para ver si tenemos instalado OPIE podemos buscar:

```
$ whereis opieinfo
```

Y para ver si tenemos instalado S/Key podemos hacer:

```
$ whereis skeyinfo
```

En algunos sistemas como FreeBSD el comando se llama `keyinfo`, luego podemos hacer:

```
$ whereis keyinfo
```

Si no los encontramos podemos bajarlos de [OPIE] o de [SKEY], y los instalamos siguiendo las instrucciones de instalación, que básicamente son ejecutar los comandos:

```
$ ./configure
$ make
$ make install
```

El programa de instalación reemplaza los ficheros:

/usr/sbin:

```
---x----- 1 root      bin      68248 Sep 21 22:06 in.ftpd
```

/bin:

```
---s--x--x  1 root      bin      36668 Sep 21 22:06 su
---s--x--x  1 root      bin      43338 Sep 21 22:06 login
```

Estos son los comandos que usan los sistemas UNIX para identificar a los usuarios cuando vamos a hacer un login, telnet o FTP con lo que tienen que ser reemplazados para que cuando nos vayamos a identificar, además de poder dar el password UNIX podamos dar el OTP.

La instalación de OPIE deposita en el disco los ejecutables:

/usr/local/bin:

```
-r-xr-xr-x  1 root      bin      7779 Sep 21 22:06 opieinfo
-r-x--x--x  1 root      bin     29778 Sep 21 22:06 opiekey
-r-s--x--x  1 root      bin     38387 Sep 21 22:06 opiepasswd
lrwxrwxrwx  1 root      root      22 Sep 21 22:06 otp-md4 ->
/usr/local/bin/opiekey
lrwxrwxrwx  1 root      root      22 Sep 21 22:06 otp-md5 ->
/usr/local/bin/opiekey
```

que son los comandos que vamos a utilizar para gestionar OPIE.

Mientras que la de S/Key crea los ejecutables:

/usr/local/bin:

```
-r-sr-sr-x  1 root      staff   43016 Apr 27 18:47 skey
-r-sr-sr-x  1 root      staff   56604 Apr 27 18:47 skeyaudit
-r-sr-sr-x  1 root      staff   52036 Apr 27 18:47 skeyinfo
-r-sr-sr-x  1 root      staff   56272 Apr 27 18:47 skeyinit
-r-sr-sr-x  1 root      staff    2593 Apr 27 18:47 skeyprune
```

11.4 Registrar un usuario OTP en el sistema

Antes de poder empezar a logarnos en un sistema que soporte OTP debemos registrar al usuario que quiera usar este sistema de identificación.

Para registrar un usuario en OPIE el propio usuario debe usar el comando:

```
$ opiepasswd -c
Adding fernando:
Only use this method from the console; NEVER from remote. If
you are using telnet, xterm, or a dial-in, type ^C now or exit
with no password.
Then run opiepasswd without the -c parameter.
Using MD5 to compute responses.
Enter new secret pass phrase:*****
Again new secret pass phrase:*****
```

```
ID fernando OTP key is 499 lo3435
TO OATH HAIL COLA GUM HURT
```

El comando crea una entrada en el fichero /etc/opiekeys

```
# cat /etc/opiekeys
fernando 0499 lo3435 4118d649b4e1793a May 02, 2002 17:14:01
```

Obsérvese que en el fichero se guarda el usuario, iteration count, semilla a utilizar y clave secreta. Como la clave secreta está guardada en este fichero, cualquiera que pudiera leer este fichero podría descubrir la clave secreta de los usuarios. Para evitarlo este fichero tiene permisos que sólo permiten que root (o un proceso ejecutando con el uid de root pueda acceder a él).

```
# ls -ls /etc/opiekeys
2 -rw----- 1 root wheel 70 Jan 1 17:01 /etc/opiekeys
```

Esa es la razón por la que los programas que instalamos tenían el permiso s

Para registrarse un usuario de S/Key debe ejecutar el comando:

```
$ skeyinit
Adding fernando:
Reminder - Only use this method if you are directly connected.
If you are using telnet or rlogin exit with no password and
use keyinit -s.
Enter secret password:*****
Again secret password:*****
```

```
ID fernando s/key is 99 pc00892
LICK BEAR ANEW SEEN GAS WORK
```

El cual crea una entrada en el fichero que ahora se llama /etc/skeykeys:

```
# cat /etc/skeykeys
fernando 0099 pc00892 ae0a9933ec6163fb Jan 01,2002 17:01:47
```

que tiene el mismo formato que el fichero de OPIE.

Estos son los ficheros que consultan `login` y `su` para dar acceso, o no, a un usuario.

11.5 Identificación en un sistema OTP

Obsérvese que en el apartado anterior ambos programas nos han devuelto 6 palabras en mayúsculas que son los OTP que usaremos la primera vez que nos queramos conectar.

Es decir al irnos a conectar haremos:

```
$ telnet pcbox
Trying pcbox...
Connected to pcbox
Escape character is '^]'.

FreeBSD/i386 (pcbox.macprogramadores.org) (tttyp0)

login: fernando
opt-md5 499 lo3435
Password: *****
```

En el password debemos indicar el OTP que nos devolvió antes `opiepasswd -c` al registrarnos. Un problema aquí está en que es muy fácil equivocarse al escribir el OTP, para evitarlo existe el truco de pulsar intro cuando se nos pide el password, y este se nos va a volver a preguntar pero dejándonos ver lo que escribimos.

Análogamente lo haríamos si estamos usando S/Key:

```
$ telnet pcbox
Trying pcbox...
Connected to pcbox
Escape character is '^]'.

FreeBSD/i386 (pcbox.macprogramadores.org) (tttyp0)

login: fernando
s/key 99 pc00892
Password: *****
```

11.6 Generar nuevos OTP

La principal característica de los algoritmos de identificación OTP es que una vez usamos para identificarnos un password OTP, ya no podemos volver a usarlo. Pero lo que sí podemos es generar nuevos password OTP para volver a identificarnos.

La próxima vez que vayamos a entrar en el sistema obtenemos:

```
$ telnet pcbox
Trying 172.26.0.2...
Connected to pcbox.
Escape character is '^]'.
login: fernando
otp-md5 498 pc3017 ext
Password:
```

Obsérvese que ahora se nos indica que el algoritmo de identificación es OPIE, que el iteration count es 498 y que la semilla vale pc3017. Ahora tendremos que calcular el password OTP para un iteration count de 498, para lo cual nos vamos a otra consola y ejecutamos el comando:

```
$ opiekey 498 pc3017
Using the MD5 algorithm to compute response.
Reminder: Don't use opiekey from telnet or dial-in sessions.
Enter secret pass phrase:*****
LORE DOES BLUM AIDS BILE MUST
```

Ahora podemos introducir el OTP a Telnet. Recuérdese el truco de pulsar intro una vez situados en el password para poder ver lo que escribimos:

```
$ telnet pcbox
Trying 172.26.0.2...
Connected to pcbox.
Escape character is '^]'.
login: fernando
otp-md5 496 pc3017 ext
Password: (Intro)
otp-md5 496 pc3017 ext
Password: LORE DOES BLUM AIDS BILE MUST
Last login: Tue Jan  1 12:22:27 from localhost
Linux 2.2.19.
```

Análogamente pasa con S/Key, en cuyo caso vemos que el sistema de identificación nos indica que está usando s/key con un iteration count de 98 y una semilla pc00892:

```
$ telnet pcbox
Trying 172.26.0.2...
Connected to pcbox.
Escape character is '^]'.
```

FreeBSD/i386 (pcbox.macprogramadores.org) (ttyp0)

```
login: fernando
s/key 98 pc00892
Password:
```

Para calcular el OTP, desde otra consola, usamos el comando:

```
$ skey 98 pc00892
Reminder - Do not use this program while logged in via telnet
or rlogin.
Enter secret password:*****
BUSY CHAW CAM LIT CHUB MERT
```

Que nos devuelve el OTP que usamos para identificarnos:

```
$ telnet pcbox
Trying 172.26.0.2...
Connected to pcbox.
Escape character is '^]'.
```

FreeBSD/i386 (pcbox.macprogramadores.org) (ttyp0)

```
login: fernando
s/key 98 pc00892
Password: (Intro)
s/key 98 pc00892
Password [echo on]: BUSY CHAW CAM LIT CHUB MERT
```

Como el tener que calcular muchas OTP es un poco tedioso, una solución que podemos adoptar es generar varias de las OTP siguientes que vayamos a necesitar usando el argumento **-n** de la forma:

```
$ opiekey -n 5 497 pc3017
Using the MD5 algorithm to compute response.
Reminder: Don't use opiekey from telnet or dial-in sessions.
Enter secret pass phrase:*****
493: MIT SHIN POW CASK RIO SHOE
494: LUNG ALL BELT BUT IOTA GRAY
495: NELL DUCK BAWD DOG EDIT NOVA
496: LORE DOES BLUM AIDS BILE MUST
497: RUTH SPY MANY BREW PIN MOOT
```

La opción **-n** también funciona en S/Key:

```
$ skey -n 5 97 pc00892
Reminder - Do not use this program while logged in via telnet
or rlogin.
Enter secret password:*****
93: JANE BOLO HUNK ODE LEER YANK
94: CURB COY SLAB LIE GRIM NAB
95: SILL STAN JUJU LIME NEW DELL
```

```
96: BULL DUAL BOND SMOG HARM ETC  
97: SALK HOC SOB DOUG HEAR SWAN
```

Otro comando importante que podemos usar solamente cuando estamos logados en el sistema es el comando **opieinfo** o **skeyinfo** que nos indican cuál es el siguiente iteration count y semilla que nos pedirá el sistema de identificación:

```
$ opieinfo  
497 pc3017
```

```
$ skeyinfo  
97 local14666
```

Los comandos se limitan a sacarlo del fichero de configuración correspondiente.

Siempre podemos usar la salida de estos comandos en combinación con **opiekey** o **skey** para sacar la siguiente password OTP a usar así:

```
$ opiekey `opieinfo`  
Using the MD5 algorithm to compute response.  
Reminder: Don't use opiekey from telnet or dial-in sessions.  
Enter secret pass phrase:*****  
ITEM WANT BREW VEAL NAIR RACY
```

```
$ skey `skeyinfo`  
Reminder - Do not use this program while logged in via telnet  
or rlogin.  
Enter secret password:*****  
RUE LATE HOC VAIL MUG LIME
```

12 Los keychain

Los **keychain** (llaveros) son una forma de solucionar el problema de la gestión de múltiples claves de un usuario.

Un keychain puede almacenar principalmente tres tipos de elementos: **Passwords** usados por el usuario para acceder a cuentas webs, de correo, FTP, Samba, etc... **Claves binarias** usadas para encriptar ficheros y otros recursos. Y **certificados digitales**, que pueden ser, o bien certificados del propio usuario (p.e. para encriptar correos S/MIME o acceder a webs seguras que requieren identificación por certificado), o bien certificados de otros usuarios con lo que nos queremos comunicar.

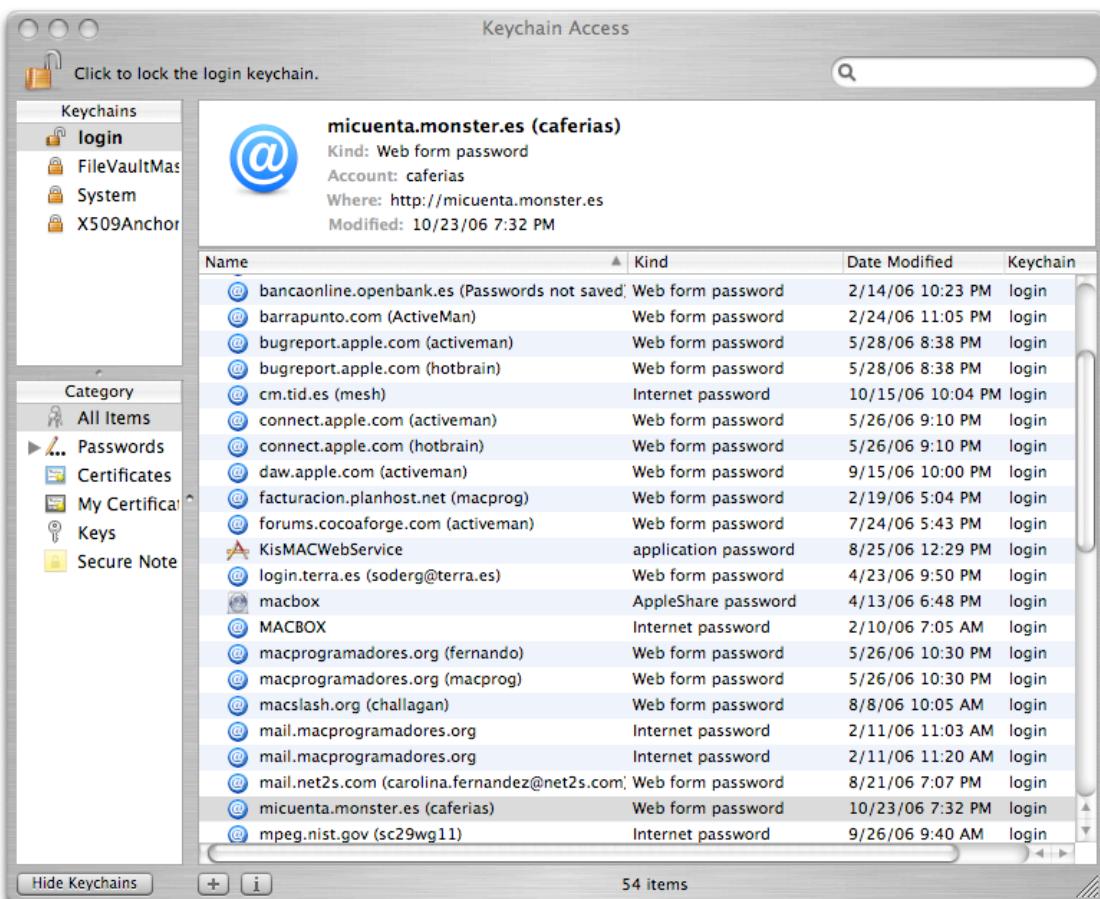


Figura 5.7: Keychain de Mac OS X

El keychain suele abrirse cuando el usuario se loga, momento en el cual se usa el password de login para desencriptar el keychain, y permanece abierto hasta que el usuario cierra la sesión. Las aplicaciones disponen de APIs que permiten consultar y almacenar información en el keychain de forma transparente al usuario. Esto permite que el usuario pueda usar distintos passwords y claves binarias para cada cuenta sin necesidad de recordarlos todos, o apuntarlos en un cuaderno. Además los keychains ayudan a evitar la

mala costumbre de los usuarios de tender a usar siempre el mismo password para todas sus cuentas.

Un buen ejemplo de keychain es el keychain de Mac OS X que se muestra en la Figura 5.7. Este keychain permite que las distintas aplicaciones (Safari, Mail, Finder,...) almacenen los passwords, claves binarias y certificados digitales del usuario. Los valores que aparecen entre paréntesis son los nombres de usuario (no los passwords).

Los keychains también se suelen poder personalizar para que las entrada extremadamente sensibles se protejan con un segundo password que la aplicación pedirá al usuario antes de poder obtener esa entrada del keychain.

13 SSH (Secure SHell)

13.1 ¿Qué es SSH?

SSH es un protocolo que permite establecer una conexión segura entre un cliente y un servidor a través de una red insegura como es la red IP. Aunque inicialmente se diseñó para ser un sustituto seguro a telnet, actualmente se le han añadido muchas más funciones.

Como muestra la Figura 5.8, SSH consta de dos componentes principales:

- El **cliente de SSH** que permite al usuario conectarse al servidor de forma segura.
- El **servidor de SSH** (SSH Server) que actúa como un superservidor en el sentido de que una vez que autentifica al usuario crea un proceso por cada cliente que atiende a sus peticiones.

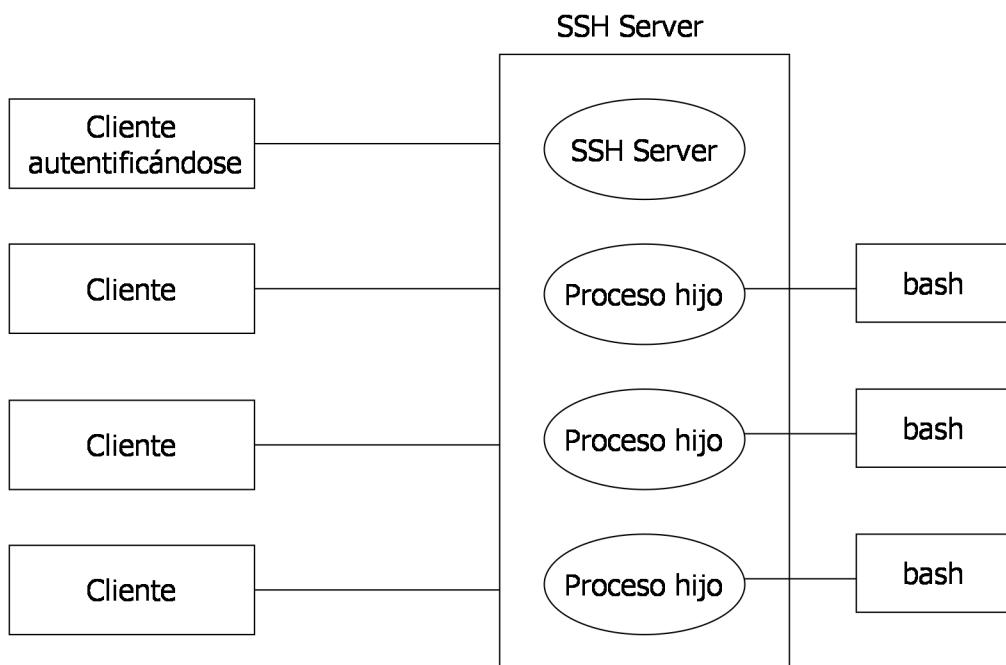


Figura 5.8 Componentes principales de SSH

El proceso hijo lanza una shell (p.e. bash o tcsh) en el servidor que atiende al cliente. Es importante tener claro que SSH lo único que proporciona es un canal de comunicación seguro, pero no es un shell en sí mismo, como puedan ser bash o tcsh, sino que el proceso hijo simplemente ejecuta una shell estándar, y actúa como un proxy entre el cliente y la shell que se está ejecutando en el servidor.

13.2 Servicios y aplicaciones de SSH

El protocolo SSH proporciona tres servicios básicos:

- **Identificación** la cual permite comprobar que un usuario es quien dice ser.
- **Encriptación** para lo cual SSH utiliza los algoritmos TripleDES, IDEA y Blowfish para garantizar la privacidad de las comunicaciones.
- **Integridad** con la que SSH garantiza que los datos que llegan a su destino llegan sin alterar. O bien detecta el cambio.

La identificación no es un servicio propio del protocolo SSH, sino que es una tarea del sistema operativo. Es decir, cuando se está ejecutando una shell en el servidor a través de SSH, es el sistema operativo quien comprueba si el usuario tiene permiso para realizar tareas que requieran autorización, como puedan ser leer o escribir un fichero, o bien abrir un socket.

Respecto a las principales aplicaciones del protocolo SSH tenemos:

1. **Shells remotos.** SSH permite acceder a un shell (UNIX o MS-DOS) de forma segura, y se encarga previamente de realizar la identificación del usuario. En este sentido SSH es un sustituto seguro de telnet.

Por ejemplo para conectarnos desde el host `macbox` al host `pcbox` usaríamos el comando:

```
macbox$ ssh pcbox
```

2. **Transferencia segura de ficheros.** Otro servicio que nos proporciona SSH es el de transferir de forma segura ficheros. Esta transferencia se puede realizar de dos formas:

- **Transferencia interactiva.** Existe una herramienta llamada `sftp` que es el sustituto seguro de la herramienta `ftp`. Por ejemplo, para conectarnos con `sftp` desde el host `pcbox` al host `macbox` usaríamos el comando:

```
pcbox$ sftp macbox
Connecting to macbox...
fernando@macbox's password:*****
sftp>
```

- **Transferencia puntual.** La herramienta `scp` es el sustituto seguro a la herramienta `rcp`, y nos permite copiar los ficheros que indiquemos en la línea de comandos. Por ejemplo, para copiar desde el host `macbox` el fichero `carta.doc` al host `pcbox` en la carpeta `textos` haríamos:

```
macbox$ scp carta.doc pcbox:~/textos
```

3. **Tunneling.** Normalmente llamado en SSH **port forwarding**, tiene dos aplicaciones principales:

Por un lado permiten crear canales seguros por los que enviar información normalmente insegura tal como muestra la Figura 5.9. El canal que va desde que entramos por el puerto 1110 del cliente de SSH hasta que salimos del servidor de SSH situado en el servidor es un canal seguro. El ejemplo de configuración de la Figura 5.9 permitiría acceder a un servidor de correo POP3 situado en el puerto 110 (con los que la comunicación no está encriptada) de forma segura. Los puertos marcados como anónimo son puertos cuyo valor cambia en cada conexión.

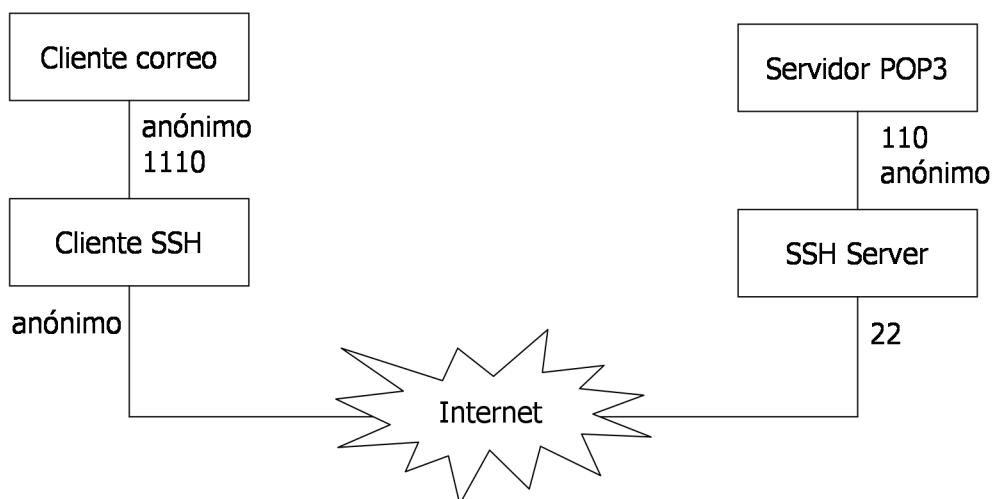
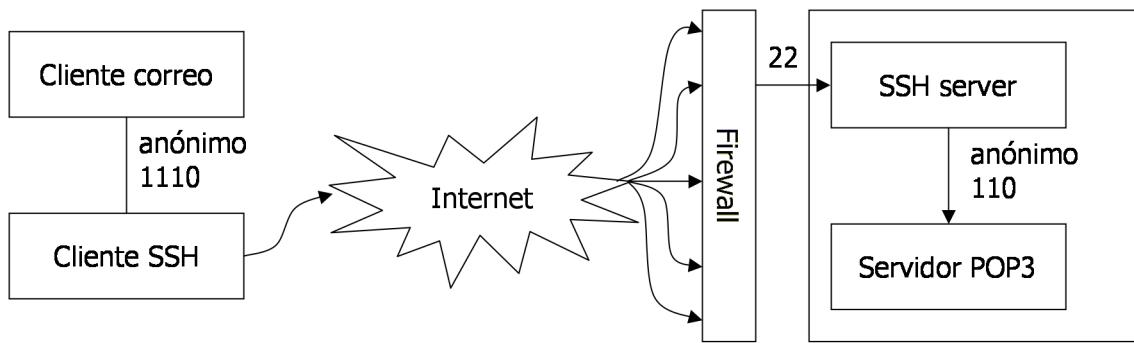
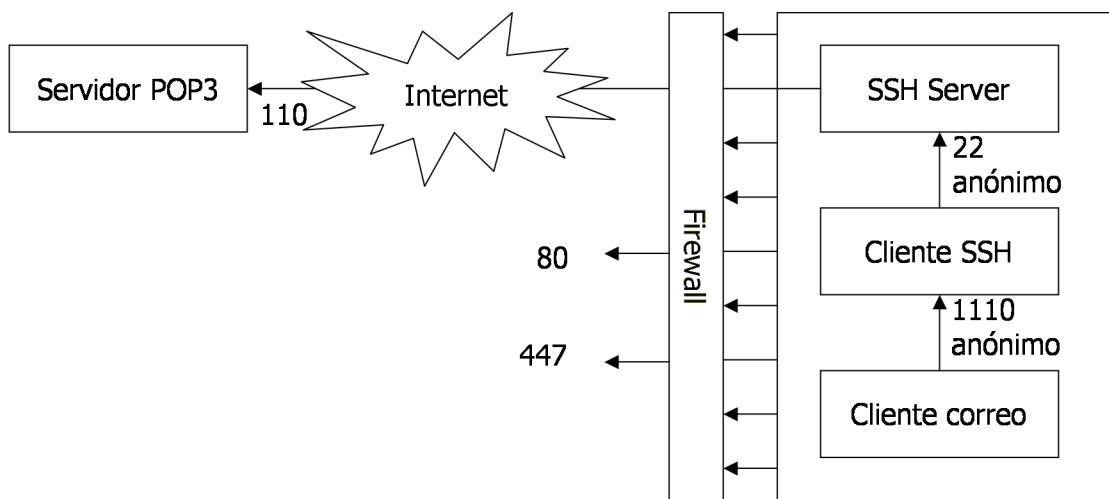


Figura 5.9 Tunneling en SSH

La otra aplicación del tunneling, tal como muestra la Figura 5.10, es crear un **bastión**, que es una red que está protegida por un firewall. Los bastiones sólo tienen abierto a la entrada determinados puertos (p.e. el puerto 22), entonces podemos crear un canal seguro que nos permita llegar a un supuesto servidor de correo que en principio sólo era accesible desde dentro del bastión.

**Figura 5.10** Bastión accedido desde el exterior

Además los bastiones normalmente sólo tienen abierta la salida a determinados puertos. Por ejemplo, en los bastiones es muy típico permitir sólo a sus usuarios acceder a servidores web externos (puerto 80 y 447), pero el servidor de SSH sí que tiene acceso al exterior. Podemos usar SSH para llegar a un servidor que está fuera del bastión, y que trabaja en otro puerto distinto a los que están abiertos (p.e. un servidor de POP3 que suelen estar situados en el puerto 110), a través del servidor de SSH (véase Figura 5.11).

**Figura 5.11** Acceso a un servidor externo desde el bastión

A veces los bastiones son tan restrictivos que no proporcionan SSH, y además el acceso web al exterior lo dan a través de un proxy web (p.e. puerto 8080). En estos casos todavía podemos usar SSH para escapar del bastión usando el puerto de salida 447, el cual, como detecta el man in the middle attack, no se puede cachear en el proxy web, y normalmente está abierto. Para ello debemos instalar un servidor de SSH en una máquina externa al bastión y abrir una conexión de port forwarding a ella desde un cliente interno al bastión.

La Figura 5.12 muestra cómo, usando port forwarding, un usuario astuto puede acceder a un servidor de correo externo desde un bastión fuertemente cerrado gracias a que tiene instalado un servidor de SSH en el puerto 447 del ordenador de su casa.

Aunque SSH da esta posibilidad, debe saber el lector que si está en un bastión protegido de esta forma probablemente será porque el administrador de la red no quiere que el usuario salga de allí, con lo que el lector debería comprobar las reglas de la organización antes de hacerlo.

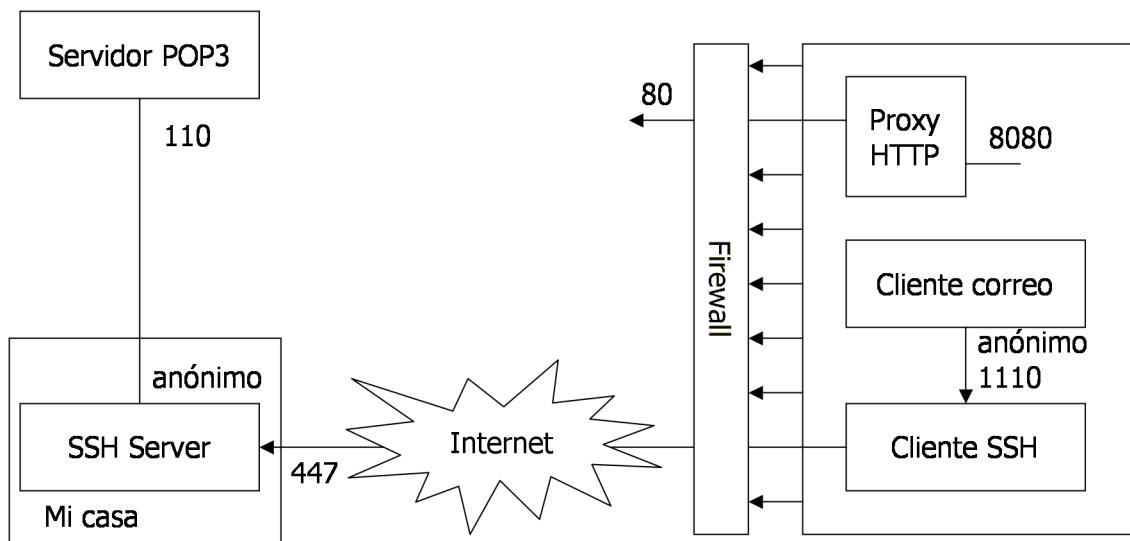


Figura 5.12 Acceso al exterior desde un bastión fuertemente cerrado

13.3 Evolución histórica de SSH

13.3.1 Orígenes

SSH fue desarrollado en 1995 por Tatu Ylönen, un investigador de la Universidad de Helsinki (Finlandia) después de haber sufrido un ataque a su máquina por parte de alguien que le había cogido el password de una de sus sesiones de telnet.

Tatu Ylönen desarrollo SSH-1 y publicó el código fuente de forma gratuita en Internet.

Después, al observar su gran popularidad decidió crear otra versión SSH-2 y venderla en una empresa que creó [SSH].

En la licencia decidió hacer este producto gratuito para investigadores y usuarios de Linux, FreeBSD, OpenBSD y NetBSD (no para Windows), y lo llevó a la IETF para que lo estandarizaran.

Al estar el producto estandarizado un grupo de programadores del sistema operativo OpenBSD [OPENSSH] implementaron una versión de licencia BSD llamada OpenSSH. En la actualidad OpenBSD es la implementación de SSH más utilizada.

13.3.2 Protocolos, productos, clientes y servidores

Es importante diferenciar entre el **protocolo** SSH, del cual existen dos versiones: SSH-1 (también llamado SSH-1.5) y SSH-2. Estas versiones son incompatibles entre sí. SSH-2 se creó porque a SSH-1 se le encontraron algunas vulnerabilidades.

Cuando utilicemos el término **producto** nos estamos refiriendo al fabricante del programa. Los dos fabricantes más conocidos de SSH son: la empresa de Tatu Ylönen, y OpenSSH, aunque existen otros muchos fabricantes del producto para Windows como F-Secure.

Por último, el **cliente** de SSH es el programa que usamos para conectarnos a SSH que normalmente es el comando `ssh` en minúsculas. El **servidor** de SSH es otro comando que se suele llamar `sshd`, también en minúsculas.

13.4 Mecanismos de identificación

13.4.1 Identificación del servidor

En principio, en SSH existe lo que se llama una **desconfianza mutua**, es decir, ni cliente ni servidor se fían de que la otra parte sea quien dice ser, con lo que se tiene que producirse una identificación bidireccional.

El servidor siempre utiliza el siguiente mecanismo de identificación por clave pública para identificarse ante un cliente:

1. La primera vez que el cliente se conecta al servidor, el servidor entrega al cliente su **clave pública de servidor**. Entonces el cliente de SSH muestra al usuario un hash de la clave pública de servidor, y pide al usuario que coteje el hash mostrado en pantalla con el administrador del servidor. Una vez que el usuario acepta la clave pública de servidor, ésta se guarda en el disco del usuario para identificar al servidor en futuras conexiones.
2. La próxima vez que el cliente se conecta al servidor, éste le manda un número aleatorio para que se lo firme.
3. El servidor firma el número aleatorio con su **clave privada de servidor**.

4. El cliente comprueba la firma del servidor con la clave pública de servidor que tiene almacenada.
5. Si la firma es correcta acepta al servidor, y si no imprime un mensaje de alerta en la pantalla del usuario indicando que el servidor ha cambiado.

13.4.2 Identificación del cliente

El servidor de SSH siempre utiliza el mismo mecanismo de identificación basado en clave pública, sin embargo el cliente dispone de cuatro mecanismos de identificación que vamos a comentar a continuación:

1. Identificación por password

Consiste en que el usuario proporciona un password al cliente de SSH, el cual transporta de forma segura hasta el servidor de SSH, y éste utiliza el mecanismo de identificación del servidor (p.e. /etc/passwd o PAM) para comprobar la identidad del usuario.

La ventaja de este mecanismo de identificación es que el usuario no tiene que configurar nada, el password con el que entraba en el host le sirve para entrar de forma remota.

En principio el usuario está libre de enviar el password a un falso servidor, ya que previamente el servidor se habrá identificado mediante el mecanismo de identificación de clave pública que hemos visto en el apartado anterior. Sin embargo, si el servidor ha sido comprometido por un atacante, éste podría capturar el password de los usuarios. Para evitarlo el cliente no envía el password sino un hash de éste, tal como se hace en la identificación tradicional UNIX que explicamos en el apartado 6 del Tema 2.

2. Identificación mediante clave pública

Aquí el cliente utiliza el mismo algoritmo de identificación por clave pública que el servidor.

Antes de iniciar el protocolo es necesario que el cliente genere un par clave privada / clave pública. La clave privada se guarda en el directorio `home` del usuario con permiso de sólo lectura para ese usuario. Por el contrario el usuario tiene que llevar el fichero con su clave pública a su directorio `home` en el servidor. Obsérvese que la clave pública no se tiene que transportar de forma segura, y de hecho se puede dejar que otros usuarios la vean, lo que no se puede dejar es que otros usuarios la modifiquen.

El protocolo de identificación por clave pública sería el siguiente:

1. El servidor envía un número aleatorio al cliente para que lo firme.
2. El cliente firma el número con su clave privada y se lo envía al servidor.
3. El servidor comprueba la firma con la clave pública del usuario.

La identificación mediante clave pública presenta básicamente dos ventajas: La primera es que una vez instalada la clave pública en el servidor no es necesario que el usuario proporcione su password cada vez que se loga en el servidor. La segunda es que es mucho más difícil atacar por fuerza bruta una clave pública que un password, ya que a veces los passwords de los usuarios son simples y se pueden atacar con diccionario.

Este método de identificación también presenta algunos inconvenientes: Por un lado está el problema de que el usuario tiene que saber configurar SSH para instalar su clave pública en el servidor. Y por otro lado está el problema de que, al no proporcionar el usuario password, otro usuario podría aprovechar que el cliente ha dejado su cuenta abierta para robarle la clave privada. Para evitarlo la clave privada se suele encriptar con password, de forma que sí que se pide password al usuario, aunque no sea para logarse en el servidor sino para desencriptar la clave privada.

3. Identificación por host

Este mecanismo de identificación consiste en que el servidor de SSH tiene una lista de hosts en los que confía. Además el servidor de SSH tiene la clave pública de estos host (los cuales también tienen que tener instalado un servidor de SSH para poder identificarse).

Ahora cuando un usuario en la máquina *A* quiere logarse en la máquina *B*:

1. Pide al servidor de SSH de *B* que se identifique mediante clave pública.
2. El servidor de SSH de *B* pide al servidor de SSH de *A* que se identifique mediante clave pública.

Este método de identificación es especialmente útil cuando tenemos una red con varias máquinas UNIX que comparten la misma base de datos de usuarios, ya que los usuarios y passwords de todas las máquinas UNIX son los mismos.

Por el contrario, este método no se recomienda para identificar usuarios que se conectan desde una workstation (UNIX o Windows), ya que el usuario puede crear cuentas con cualquier otro nombre que no sea el suyo y logarse en otra máquina fingiendo ser otro usuario, lo cual se conoce como **spoofing** (parodia).

4. Identificación Kerberos

Tanto SSH como OpenSSH proporcionan la opción de identificar a los usuarios usando Kerberos.

Para poder usar esta forma de identificación necesitamos compilar SSH con una opción para que use Kerberos, es decir kerberizar la aplicación (véase el apartado 7.2).

14 VPN

14.1 Qué es una VPN

Una **VPN** (Virtual Private Network) es una forma de simular una red privada, también llamada LAN (Local Area Network), a través de una red pública (p.e. Internet). A estas redes se las llama *virtuales* porque, como muestra la Figura 5.13, se crea una conexión segura a través de Internet entre dos máquinas de LANs distintas. Esto permite conectar en una misma VPN máquinas situadas en posiciones geográficas distintas.



Figura 5.13 Red virtual en una VPN

La conexión virtual puede ser entre dos LAN, o una LAN y un terminal (p.e. el portátil de un comercial), en cualquier caso la impresión que tienen los usuarios es la de estar conectados a una red local (p.e. red Samba de Windows).

Se llama **intranet** a una red que da los servicios de Internet sobre una LAN. Por ejemplo, podemos tener un servidor web que da acceso a la lógica de negocio de la empresa a la que pueden acceder los miembros de la LAN. Los otros servicios típicos de una LAN son el correo, las news y el chat.

Se llama **extranet** a una red que en vez de abarcar sólo una LAN abarca toda una VPN. Los servicios de extranet son accesibles sólo a los miembros de la VPN a pesar de que la VPN pase por Internet.

Tanto en las intranet como en las extranet es muy común limitar el acceso de los usuarios, además de por password, por dirección IP, de forma que los servidores no aceptan conexiones desde direcciones IP que no pertenezcan a la LAN.

14.2 Los firewalls

El **firewall** es el componente de una LAN que impide que los usuarios no deseados entren en la LAN, mientras que permite a los usuarios legítimos entrar en la LAN.

Los firewalls desempeñan principalmente tres funciones:

1. Controlan que máquinas del exterior puedan acceder a servicios del interior.
2. Controlan que máquinas de la LAN pueden acceder a que servicios externos.
3. Permiten que se creen conexiones seguras entre dos LANs de la VPN.

Para ello existen tres tipos de firewalls que vamos a describir más en detalle:

1. Firewalls de filtrado de paquetes

Son firewalls los cuales tienen configuradas una serie de reglas en las que se indican, en función de la dirección IP y puertos origen y destino de los paquetes, cuales pueden pasar por el router y cuales no.

Este tipo de firewall nunca mira los datos (payload) de los paquetes, sólo las cabeceras.

La principal ventaja de este tipo de firewalls es que son rápidos (ya que sólo procesan las cabeceras) y fáciles de entender y configurar.

Respecto a sus inconvenientes, hay que tener en cuenta que para crear una VPN en la que intervengan varias LAN tenemos que configurar las tablas de routing para indicar que aceptamos los paquetes procedentes de otras LAN de la VPN, tal como muestra la Figura 5.14:

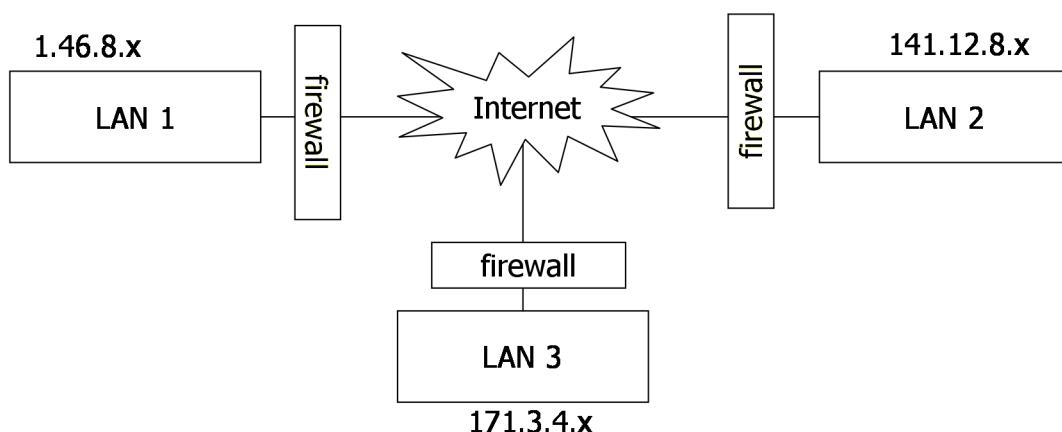


Figura 5.14 Configuración de un firewall de filtrado de paquetes para una VPN

Un inconveniente importante está en que si algún miembro de la VPN cambia su IP tenemos que modificar las tablas de routing de los demás firewalls, lo cual aumenta el coste de administración.

Otro inconveniente es que estos firewalls son susceptibles a un ataque por spoofing de IP, ya que no disponen de ningún mecanismo de identificación.

2. Bastiones

Los bastiones son LANs protegidas por dos elementos:

1. Un firewall tradicional que filtra en función de la dirección IP y puertos de los paquetes.
2. Un **bastión** que es un host que se encarga de controlar todas las conexiones de tráfico entrante y saliente.

La ventaja que aporta esta solución es que al haber un único punto de entrada y salida, éste es el único punto que tiene que controlar el administrador.

Como muestra la Figura 5.15, en el bastión se pueden instalar servicios como SSH, correo, web o FTP. Aunque muchas veces el bastión no realiza estos servicios, sino que los delega en otras máquinas para descargarse de trabajo. El bastión y el firewall pueden estar instalados en la misma o en distinta máquina.

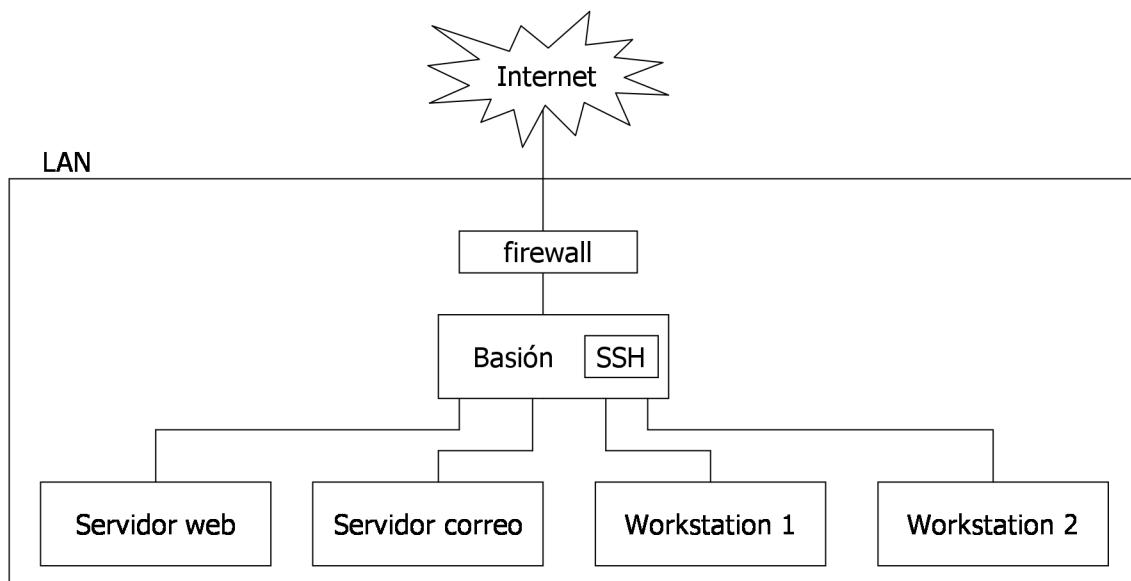


Figura 5.15 Configuración de un bastión

3. Red de zona perimetral

Una **red de zona perimetral**, también llamada DMZ (DeliMited Zone), tal como muestra la Figura 5.16, es una LAN en la que el bastión está protegido por dos firewalls, uno que le protege de los ataques del exterior y otro que le protege de los ataques del interior. Esto evita posibles ataques por parte de los miembros de la organización.

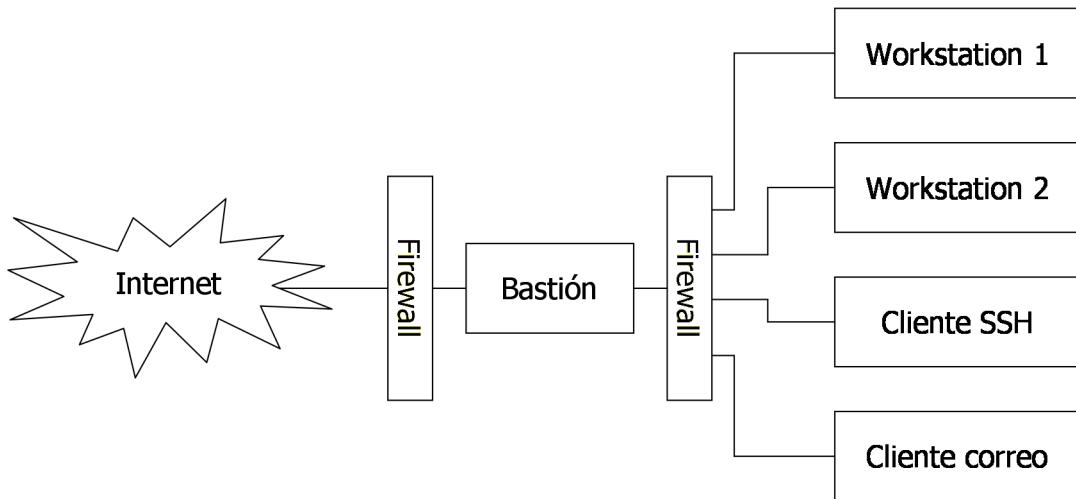


Figura 5.16 Red de zona perimetral

14.3 Tuneling

14.3.1 Qué es el tuneling

El **tuneling** permite transportar paquetes que no sean IP (p.e. NetBEUI, IPX) de forma segura sobre una red IP.

Esta técnica, como muestra la Figura 5.17, se utiliza para poder conectar varias LAN que no sean IP (p.e. red Windows tradicional) enviando sus paquetes sobre una red IP (Internet) a otras LAN de la VPN.

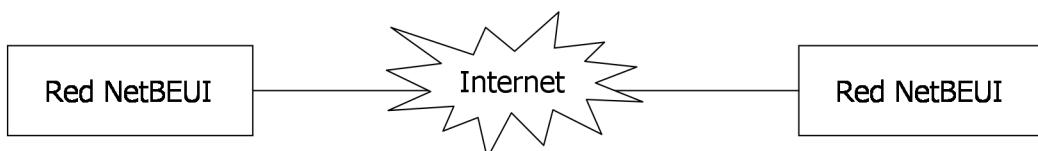


Figura 5.17 Tuneling entre dos redes NetBEUI

Otra aplicación del tuneling es enviar paquetes IP no enrutables sobre una red IP (Internet) que usa paquetes IP enrutables.

14.3.2 RAS (Remote Access Service)

RAS es una forma muy usada para permitir que, usuarios conectados a través de un modem, pasen a formar parte de la LAN. Para ello en la LAN debemos tener un **rack de modems** que acepten llamadas de los clientes.

Para que los clientes se conecten a través de modem se han usado principalmente dos protocolos que vamos a explicar:

1. SLIP (Serial Line IP)

SLIP es un protocolo muy antiguo del mundo de UNIX, que sólo sirve para transportar paquetes IP, y que no dispone de mecanismo de identificación de usuarios, sino que delega la identificación a otros niveles.

2. PPP (Point to Point Protocol)

Es el protocolo que más se utiliza actualmente. A diferencia de SLIP, es multiprotocolo (NetBEUI, IPX, IP), funciona sobre cualquier medio (modem, RDSI, X.25, fibra óptica), y permite identificar usuarios. Fue creado por la IETF y está documentado en la RFC 1547. Los paquetes que viajan sobre este protocolo no están encriptados, sino que delegan la encriptación a niveles superiores.

PPP dispone de tres mecanismos de identificación de usuarios:

1. PAP (Password Authentication Protocol). Este mecanismo es el más inseguro ya que el usuario envía el user y password sin cifrar por el cable.
2. CHAP (Challenge Handshake Authentication Protocol). Es un mecanismo de identificación seguro en el que, como muestra la Figura 5.18, no se envía el password a través de la línea, sino que se hace un password handshake (véase apartado 10.2). Además, este protocolo se caracteriza porque el servidor puede pedir varias veces a lo largo de la conexión al cliente que se identifique (y no sólo durante el establecimiento de la conexión). Esto evita un posible ataque llamado **session hijacking** (secuestro de sesión).

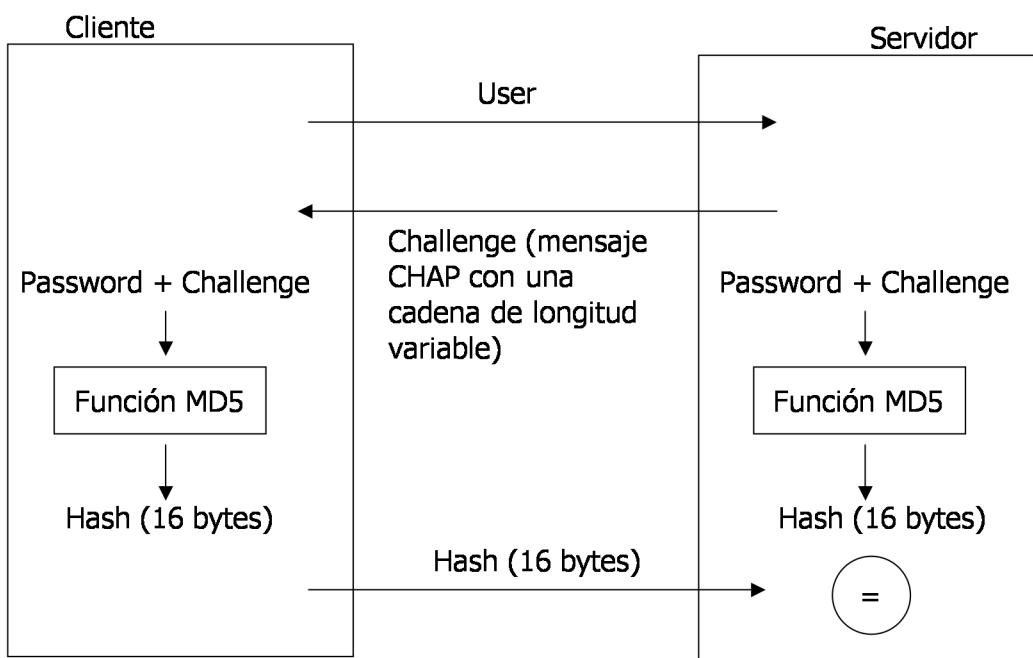


Figura 5.18 Mecanismo de identificación en CHAP

3. MS-CHAP. Es un mecanismo de identificación similar a CHAP, excepto que usa MD4 en vez de MD5 para calcular el hash. No se recomienda su uso ya que sólo funciona en sistemas Windows.

14.3.3 PPTP (Point To Point Tunneling Protocol)

14.3.3.1 Qué es PPTP

PPTP es un protocolo que permite transportar tramas de nivel 2 (PPP, Ethernet, ATM) sobre una conexión IP (socket TCP) de forma segura. Esto tiene la ventaja de que podemos transportar paquetes de distintos tipos (IP no enruteables, NetBEUI, IPX) sobre una conexión IP, en vez de usar un modem o una red Ethernet, que es lo que se utiliza tradicionalmente. Con esto conseguimos por ejemplo que los usuarios de una red Windows accedan a todos sus servicios de red local sobre una conexión TCP/IP.

PPTP utiliza dos algoritmos de encriptación: Uno débil de 40 bits (por cuestiones de exportación) y otro fuerte de 128 bits. Recuérdese que, como vimos en el apartado anterior, PPP no tenía ningún mecanismo de encriptación, sino que delegaba la encriptación a los niveles superiores que transporta.

Normalmente PPTP se utiliza para transportar PPP (obsérvese que aunque PPP es un protocolo de nivel 2, es transportado sobre PPTP para viajar sobre una red IP en vez de sobre un modem). En el apartado anterior vimos que PPP tenía tres mecanismos de identificación: PAP, CHAP y MS-CHAP.

14.3.3.2 Configuraciones para usar PPP

Vamos a suponer que tenemos una LAN con un **RAS Server** (Remote Access Service Server) que da acceso a una red LAN en Madrid, y un comercial en Barcelona que quiere acceder a su LAN Windows de la oficina de Madrid para copiar unos ficheros que tiene en su ordenador de la oficina.

Para ello necesitará tener instalado un RAS Server (p.e. Microsoft Windows NT RAS Server), configurado en el puerto 1723, y que el firewall de la empresa permita tráfico entrante y saliente por ese puerto. Obsérvese que en este caso el RAS Server actúa como un bastión.

En este escenario el comercial puede acceder de tres posibles formas:

1. Usando un modem que le conecta a un ISP con soporte para PPTP

Para ello el ISP debe disponer de un modem con soporte para PPTP (p.e. el Ascend MAX 4000 de US Robotics). Si el ISP no tiene un modem de este tipo (que es lo más normal) no podemos usar esta opción.

Además es necesario que a el ISP se le haya dicho la dirección IP del RAS al que debe conectarse el nombre de usuario de nuestro comercial.

Los pasos para establecer la conexión son:

1. El comercial se conecta a su ISP dando su user y password
2. EL ISP detecta que el usuario es un usuario de RAS y abre una conexión PPTP al servidor de RAS de la empresa.
3. El RAS de la empresa autentifica al usuario y, si es un usuario correcto, establece una conexión PPP con éste como si el usuario se hubiera conectado directamente al modem del RAS.

La Figura 5.19 ilustra gráficamente este proceso.

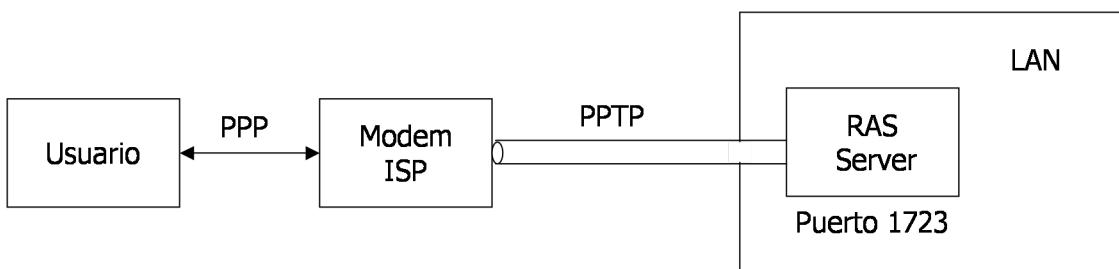


Figura 5.19 Conexión por modem a un ISP con soporte para PPTP

2. Usando un modem que le conecta a un ISP sin soporte para PPTP

En este caso el usuario tiene que abrir dos conexiones:

- Una al ISP que le conecte a una red TCP/IP
- Una conexión PPTP que le conecte al RAS Server de su empresa

El protocolo sería el siguiente:

1. El usuario abre una conexión por modem al su ISP dando el user y password de su ISP
2. El usuario abre una conexión al RAS Server de su empresa dando el user y password de su RAS

Como muestra la Figura 5.20, en este caso la conexión PPTP se establece para todo el camino (y no sólo para el camino que va desde el ISP al RAS), con lo que la conexión es más segura, ya que antes el tramo del PPP hasta el ISP no estaba encriptado.

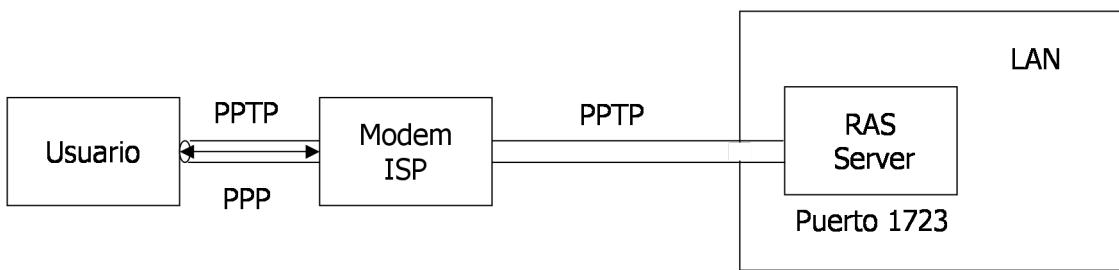


Figura 5.20 Conexión por modem a un RAS Server

El único inconveniente está en que el usuario tiene que abrir dos conexiones.

3. Usando una conexión TCP/IP que le conecta al RAS Server de la empresa

Si el comercial está en un sitio con acceso a red IP, sólo necesita establecer una conexión PPTP a su RAS Server, tal como muestra la Figura 5.21

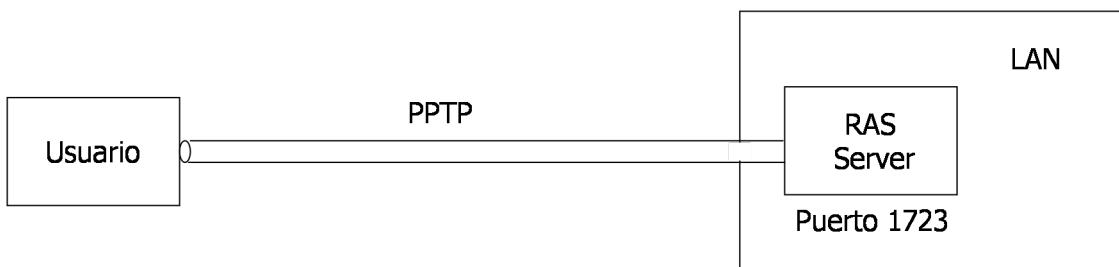


Figura 5.21 Conexión a un RAS Server desde una red IP

La ventaja que aporta aquí utilizar PPTP en vez de una conexión TCP/IP normal es que se pueden transmitir todo tipo de paquetes (NetBEUI, IPX, TCP/IP no enrutables).

14.3.3.3 Formato de los paquetes PPTP

El protocolo PPTP está basado en otro protocolo de Internet llamado **General Routing Encapsulation (GRE)** descrito en la RFC 1701 y RFC 1702. La versión de GRE para PPTP se llama GREv2 y añade algunas extensiones para poder usarlo en PPTP, como el Call ID y el control de la velocidad de la conexión.

Los paquetes que envía PPTP encapsulados (payload) se envuelven en dos cabeceras tal como muestra la Figura 5.22:

- Una cabecera IP que permite al paquete viajar por la red IP

- Una cabecera GRE con información del tipo de payload que está transportando

En nuestro caso el payload será una trama PPP que envuelve paquetes IP, IPX o NetBEUI.

En caso de que el ISP dé soporte para PPTP, el procedimiento de transporte es el siguiente:

1. El usuario envía tramas PPP a través de su modem al modem del ISP
2. El modem del ISP envuelve las tramas PPP en paquetes PPTP (con cabecera GREv2 y cabecera IP) y se los envía al RAS
3. El RAS quita las cabeceras y recupera la tramas PPP, las cuales primero autentifica (PAP, CHAP), y luego las enruta en la LAN, habiendo quitado primero la cabecera PPP.

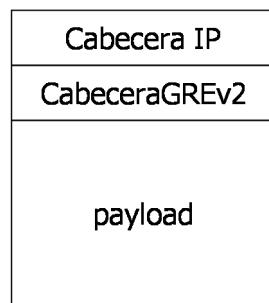


Figura 5.22 Paquete PPTP

Si el usuario establece una conexión PPTP directa al RAS el procedimiento es análogo, excepto que quien envuelve las tramas PPP en paquetes PPTP no es el modem del ISP, sino el equipo del usuario.

Tema 6

PKI

Sinopsis:

En este tema vamos a aprender qué son los certificados digitales, y cómo se crean, usan y validan, así como sus principales formatos. Una vez introduzcamos los certificados digitales a nivel conceptual, veremos cómo se utilizan estos certificados desde Java.

En la segunda parte del tema veremos qué se esconde detrás de las siglas SSL y TLS, para qué vale, y algunos ejemplos de aplicación de esta tecnología. Acabaremos esta parte del tema viendo cómo se usa esta tecnología desde Java.

En la tercera parte del tema vamos a ver una serie de herramientas relacionadas con los certificados y las conexiones SSL, como son OpenSSL, S/MIME, o IPSec.

Acabaremos el tema explicando cómo funciona el protocolo HTTPS, y cómo se puede hacer un programa Java cliente o servidor que se comunique a través de este protocolo.

1 PKI

1.1 Introducción

PKI (Public Key Infrastructure) son un conjunto de herramientas que hacen posible el uso de un sistema criptográfico de forma segura.

En el apartado 2.3 del Tema 5 vimos que los sistemas de comunicación basados en clave pública eran seguros ante un ataque pasivo, pero no ante un ataque activo, en concreto ante el man in the middle attack, ya que no había forma de saber si la clave pública con la que se nos pedía trabajar era realmente la del interlocutor, o si por el contrario nos "había dado el cambazo" un atacante activo.

Para solucionar este problema PKI proporciona cuatro **herramientas** que vamos a comentar en este apartado. Estas cuatro herramientas son:

1. Los certificados digitales
2. Las listas de revocación de certificados
3. Las competencias de los certificados
4. La validación del camino de certificación

1.1.1 Certificados digitales

El principal problema de la criptografía de clave pública es determinar a quién corresponde realmente una clave pública. Para solucionar este problema se utilizan los **certificados digitales**, o simplemente **certificados**, los cuales, tal como muestra la Figura 6.1, actúan como un carnet de identidad que asocia una clave pública a los datos de un usuario.

Un certificado digital debería de tener al menos las siguientes características:

Número de serie:38034107 Usuario: Fernando López E-mail:fernando@macprogramadores.org Competencias: firmar, encriptar Expedido por: Verisign Inc. Fecha activación: 18/3/2004 Fecha caducidad: 18/3/2009 Clave pública:3F74A34B42E53A54F63
Firma: Verisign Inc. 0B3426DE423F233A42C232

Figura 6.1 Ejemplo de certificado digital

- Debería de mostrar los datos personales del usuario, unas fechas de alta y caducidad, y la clave pública del usuario.
- Debería estar firmado por una autoridad distinta al propio usuario.
- Debería de ser fácil comprobar que el certificado es auténtico, y que no ha sido comprometido.

- Debería de informar de las aplicaciones para las que es competente el dueño del certificado (firmar digitalmente, encriptar documentos, servir páginas web, comprar por Internet, vender por Internet,...).

1.1.2 Listas de revocación de certificados

Un problema que tienen los certificados es que una vez expedido, si su seguridad es comprometida (alguien se apodera de la clave privada, el empleado cambia de trabajo, etc.) es muy difícil anular el certificado hasta que éste expire.

Para solucionar este problema se usan las **CRL (Certificate Revocation List)**, que son listas públicas en las que la autoridad que firma los certificados indica números de serie de los certificados no caducados que han sido revocados.

1.1.3 Las competencias de los certificados

Las **competencias de los certificados (certificate policy)** son las funciones para las que se puede usar el certificado: firma de documentos, firmar otros certificados, encriptar datos, etc.

Esto permite que, por ejemplo, un usuario no pueda firmar otros certificados, pero sí que pueda firmar sus mensajes de correo.

1.1.4 Validación del camino de certificación

Para conseguir escalabilidad es necesario que haya varios organismos encargados de firmar certificados. Estos organismos suelen organizarse de forma jerárquica tal como muestra la Figura 6.2.

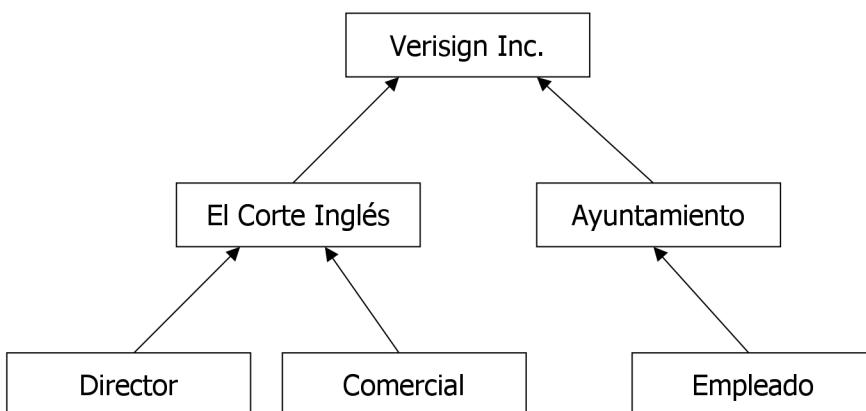


Figura 6.2 Organización jerárquica de los organismos de certificación

En el ejemplo de la Figura 6.2, cuando un empleado del ayuntamiento quiere comprar un producto a un comercial del Corte Inglés, éste necesita primero saber que el certificado del comercial del Corte Inglés es válido. Como el empleado del ayuntamiento no conoce la firma del Corte Inglés, éste necesita subir en la jerarquía hasta Verisign Inc (del cual sí que conoce la firma) para asegurarse de que el certificado del comercial es correcto.

Para realizar esta comprobación necesita crearse un camino de certificación, tal como muestra la Figura 6.3.

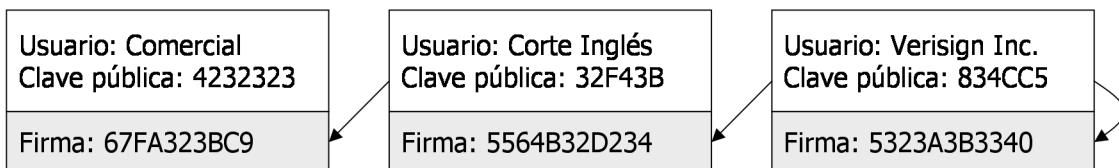


Figura 6.3 Camino de certificación

Obsérvese que cada clave pública de un certificado es firmada por un usuario del nivel superior, excepto el certificado de nivel superior que se firma a sí mismo.

1.2 Roles

Hasta ahora hemos supuesto que PKI consta de dos tipos de participantes: La CA (Certification Authority), que es la encargada de emitir certificados digitales, y en la que los demás participantes confían, y el usuario que es el que dispone de un certificado firmado. En este apartado vamos a detallar cuáles son los roles que pueden desempeñar cada uno de los participantes.

1.2.1 Roles de las CA

Vamos a empezar viendo los cuatro roles que corresponden a las cuatro funciones de la CA:

Expedir certificados. Consiste en firmar certificados digitales a otros usuarios o CAs, y desarrollar las medidas de seguridad necesarias para evitar que su clave privada sea comprometida.

Comprobar la veracidad de la información de los certificados. Es importante que la CA, antes de firmar un certificado, se cerciore de que la información que el certificado contiene (nombre, dirección postal, etc.) sea correcta.

Mantener la información de los certificados actualizada. Es obligación de la CA informar de si la seguridad de un certificado ha sido comprometida, lo cual suele hacerlo a través de las CRL (Certificate Revocation List).

Mantener información histórica. La CA debe conservar un histórico de los certificados emitidos por si en el futuro surgen disputas (p.e. un contrato firmado hace 20 años).

La principal función de una CA es expedir certificados. Las demás funciones las puede delegar, parcial o totalmente, a otros componentes del PKI. Estos componentes son:

Registration Authority (RA). Es la entidad que se encarga de recoger y verificar la información de los usuarios a los que vamos a emitir un certificado. Esta entidad también se puede encargar de recoger las CRL para publicarlas en el repositorio. Para realizar este rol es necesario tener una oficina física donde puedan acudir los usuarios a acreditar su identidad.

Repositorio. Se encarga de publicar los certificados y CRL de la CA. Como veremos luego, este repositorio es un servidor colocado en una determinada dirección IP y al que se accede a través de un protocolo.

Archivo. Es la entidad que se encarga de mantener una copia de los certificados emitidos durante un largo periodo de tiempo. Esto evita posibles disputas futuras sobre la veracidad de un documento digital.

1.2.2 Roles del usuario

Los usuarios también los podemos dividir en dos tipos:

Certificate Holder. Son los usuarios que poseen un certificado a su nombre.

Relying party. Son los usuarios que confían en los certificados emitidos por una CA.

En la práctica los usuarios pueden desempeñar ambos roles, por un lado pueden poseer un certificado, y por otro pueden confiar en los usuarios que poseen certificados firmados por la CA.

1.3 Topología

Los sistemas PKI se están utilizando, principalmente, con dos fines distintos que vamos a detallar a continuación:

Por un lado se utilizan en Internet, donde hay organismos como Verisign Inc. o Thawte que se encargan de firmar digitalmente certificados a sitios web. Los principales browsers vienen con los certificados de estos organismos preinstalados, con lo que cualquier certificado emitido por una de estas

empresas es automáticamente reconocido como válido por el browser, y usado para establecer conexiones seguras TLS.

Por otro lado hay muchas organizaciones que han decidido usar un sistema de PKI para controlar el acceso a los recursos de la organización. En este caso la empresa tiene un servidor que actúa como CA, y el cual emite certificados a los trabajadores de la empresa.

Los trabajadores pueden usar estos certificados en programas de gestión propios de la empresa (p.e. hechos en Java), o instalar estos certificados en sus navegadores web para identificarse ante los servidores web de la empresa donde se realiza la gestión. En cualquier caso, un usuario necesita elegir a una (o más) CA como su trusting point. Los **trusting point** son las CA en las que confía un usuario para comprobar la validez de un certificado digital.

En las organizaciones pequeñas es posible la existencia de una sola CA que actúe como trusting point de todos los miembros de esa organización, pero en organizaciones grandes es necesaria la existencia de varias CA, debido a problemas de escalabilidad.

En los siguientes subapartado vamos a clasificar las distintas topologías y relaciones de confianza que pueden existir entre las CA.

1.3.1 Topologías simples

En las topologías de CA más simples no existen relaciones de confianza entre los CA, sino que cada CA sólo emite certificados a sus usuarios finales.

Podemos describir básicamente dos topologías de este tipo:

1. Una única CA

Como muestra la Figura 6.4, en esta topología hay una única CA en la que confían todos los usuarios.

Esta solución es la más sencilla pero no escala bien para sistemas con muchos usuarios.

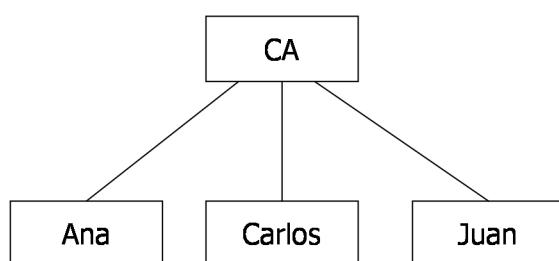


Figura 6.4 Una única CA

2. Trusting list

En esta solución existen varias CAs, donde cada CA firma los certificados de sus usuarios tal como muestra la Figura 6.5.

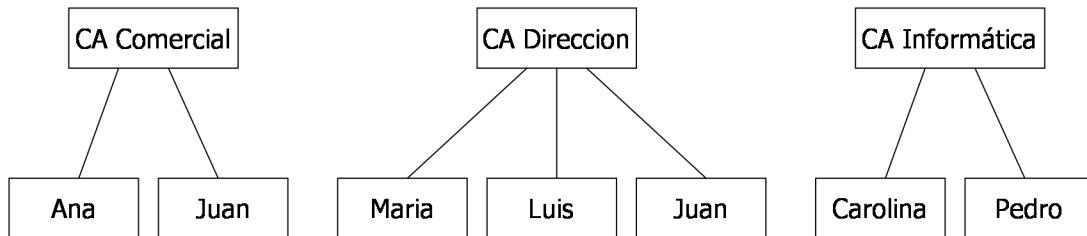


Figura 6.5 Trusting list

En esta forma de organización no existen relaciones de confianza entre las CA, sino que cada usuario debe elegir como trusting point a cada una de las CA de las que cuelgan usuarios con los que se quiere comunicar.

En el ejemplo de la Figura 6.5, si Juan se quiere comunicar con Maria, necesita añadir a su trusting list la CA de Maria.

El principal inconveniente que tiene esta forma de organización es que delegamos en los usuarios el decidir si es realmente auténtica la CA que añaden a su trusting list, cada vez que se quieren comunicar con otro usuario.

1.3.2 Topologías escalables

En estas topologías las CA establecen relaciones de confianza entre ellas, con lo que los usuarios no van a tener que gestionar estas relaciones, y de hecho cada usuario va a tener un solo trusting point.

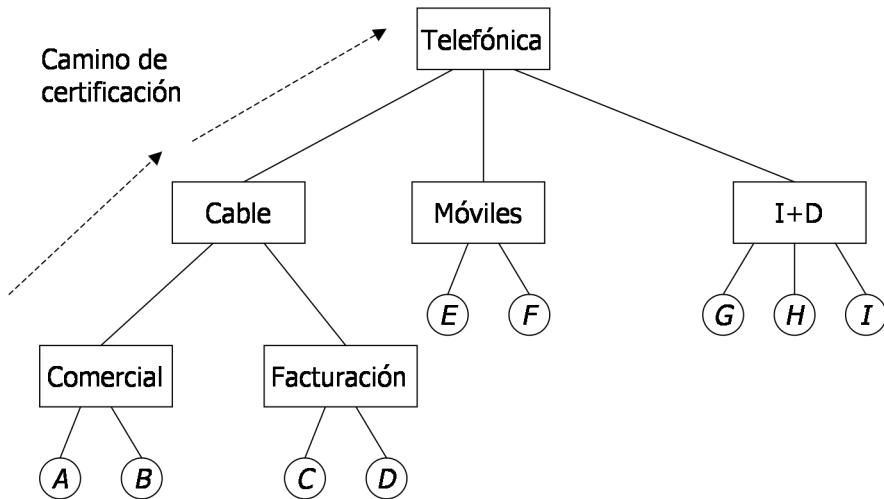
Vamos a estudiar dos tipos de topologías escalables:

1. Topología jerárquica

Esta es la topología tradicional en la que cada usuario tiene una única CA que firma su certificado, y las CA pueden firmar los certificados tanto de usuarios como de otras CA formando una jerarquía tal como muestra la Figura 6.6.

Como pasa en los certificados X.509, que se explican más adelante, cada CA puede firmar certificados con derechos a expedir otros certificados (certificados de CA), o sólo con derecho a usar el certificado (certificados de usuario).

Independientemente de la CA que firma un certificado, los usuarios tienen un único trusting point, que es la CA raíz, de forma que como muestra la Figura 6.6, cuando *F* se quiere comunicar con *A*, primero crea un camino de certificación que le permita comprobar que el certificado de *A* es válido.

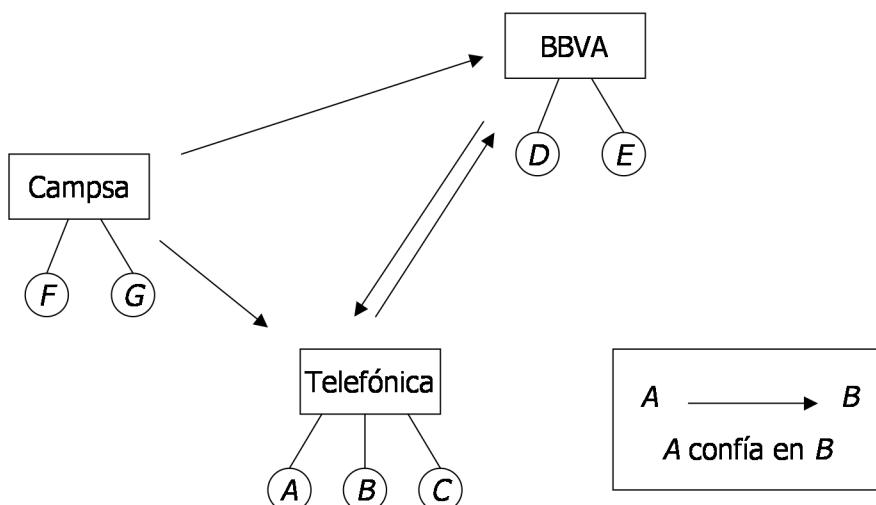
**Figura 6.6** Topología jerárquica

En la topología jerárquica, si una CA es comprometida, su CA superior revoca su certificado, con lo que el certificado de la CA y todos los certificados que ha emitido dejan de ser válidos, y se deben sustituir por otros certificados.

2. Topología enmarañada

Muchas veces la organización jerárquica no es posible porque no está clara la relación jerárquica entre organizaciones (p.e. qué país actúa como CA raíz, o qué empresa actúa como CA raíz de otras empresas).

En este caso la solución pasa por crear una **web of trust**, en la que cada CA firma los certificados de sus usuarios, y las CA establecen con otras CA relaciones peer to peer, donde, como muestra la Figura 6.7, las relaciones de confianza pueden ser bidireccionales o unidireccionales.

**Figura 6.7** Topología enmarañada

A diferencia del modelo jerárquico, ahora la CA de cada usuario es la misma que su trusting point, y desde el punto de vista de los usuarios su CA actúa como la CA raíz para crear el camino de certificación.

1.3.3 Topologías híbridas

En el mundo real es muy común encontrarse con topologías mixtas como las que vamos a comentar en este apartado.

1. Extended trusting list

Esta topología es parecida a la trusting list, sólo que ahora, como muestra la Figura 6.8, cada CA de la trusting list es una CA que puede actuar como CA raíz de una topología jerárquica, o como CA de una topología enmarañada.

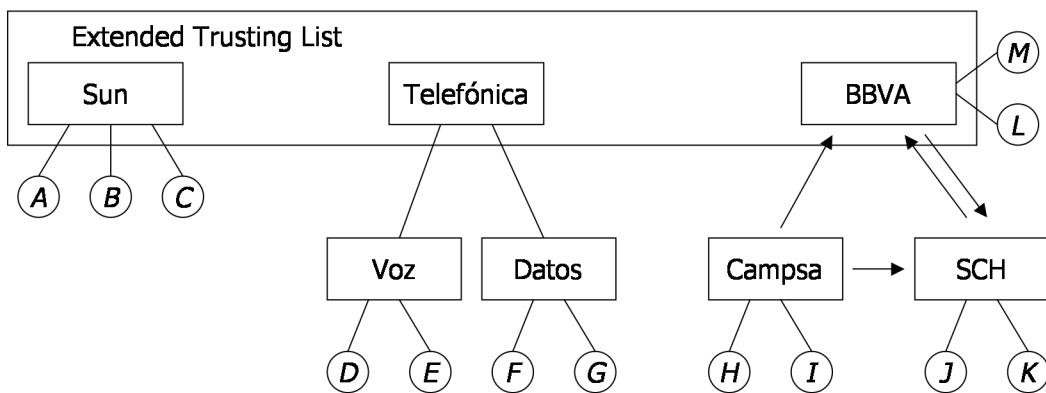


Figura 6.8 Extended Trusting list

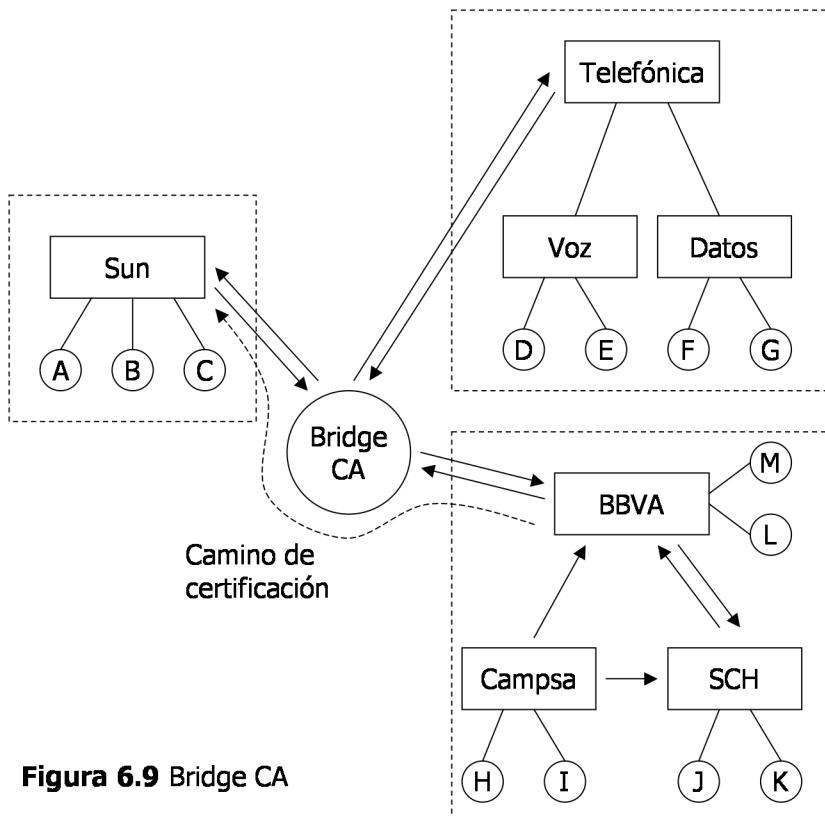
2. Bridge CA

Esta topología se diseñó con el fin de solucionar los problemas que presenta la extended trust list en la que delegamos en los usuarios la gestión de sus CA, y la topología enmarañada en la que el administrador de un CA debe establecer y mantener relaciones peer to peer con otros CA.

Como muestra la Figura 6.9, aquí la **bridge CA** actúa como una especie de árbitro, pero a diferencia de la arquitectura jerárquica, la bridge CA no emite certificados ni actúa como trusting point de los usuarios, sino que actúa como un mero intermediario.

Si la bridge CA se comunica con una topología jerárquica se conecta a su CA raíz, y si se conecta a una topología enmarañada se conecta sólo a una de las CA.

Por ejemplo, en la Figura 6.9, si *A* se quiere comunicar con *I*, en la figura se muestra cuál sería el camino de certificación hasta el trusting point de *A*.

**Figura 6.9** Bridge CA

1.3.4 Topología de Internet

El modelo que usan los browsers en Internet es un modelo extended trusting list con los trusting point instalados en el browser, y donde cada trusting point tiene una organización jerárquica. Además al browser se le pueden añadir nuevos trusting point que no vengan instalados (p.e. el de la FNMT).

1.4 Los certificados X.509

1.4.1 Formato

Aunque hay otros formatos para certificados digitales (p.e. el formato propuesto por PGP), el formato más ampliamente aceptado es el X.509 que vamos a estudiar en este apartado.

Este formato ha pasado por tres versiones:

X.509 v.1 Fue desarrollado por la ITU para los certificados de directorios X.500.

X.509 v.2 Fue una revisión de la ITU para X.509 que permite poner identificadores únicos al issuer y subject del certificado.

X.509 v.3 Es una revisión de la IETF publicado en la RFC 2459 que permite poner extensiones al certificado con información adicional no soportada por los campos básicos del certificado.

Actualmente todas las implementaciones de PKI soportan X.509 v.3

El contenido de los campos de un certificado se definen en ASN.1 (Abstract Syntax Notation) que es un metalenguaje (como pueda serlo XML) definido por el OSI para su capa de aplicación.

En este metalenguaje se han definido algunos tipos de datos propios de X.509 como son:

Object Identifier (OID). Son una secuencia de números (p.e. 1.2.8.9.3) que identifican el contenido de un campo, y que se usan para representar algoritmos criptográficos (p.e. RSA o DSA), competencias de los certificados, y extensiones.

Distinguished Names (DN). Permiten nombrar de forma jerárquica y única un elemento mediante pares clave-valor de la forma:

```
c=ES o=MacProgramadores ou=Formación cn=Fernando López
```

Aunque en teoría se puede usar cualquier par clave-valor, en la práctica se suelen usar los que aparecen en el ejemplo.

La Figura 6.10 muestra el contenido de un certificado, el cual tiene los siguientes campos:

- **Version.** Suele ser v.3.
- **SerialNumber.** Un número distinto para cada issuer.
- **Signature.** Una copia de `SignatureAlgorithm` protegida bajo la firma.
- **Issuer.** El DN de la CA que firma el certificado.
- **Validity.** Un periodo de validez con campos `NotBefore` y `NotAfter` que indican desde cuándo hasta cuándo es válido el certificado.
- **Subject.** El DN del usuario que posee la clave pública
- **PublicKey.** contiene dos campos, uno con el tipo de clave pública, y otro con la clave pública en sí.
- **IssuerUniqueID, SubjectUniqueID.** Campos introducidos en X.509 v.2 para dar un número único a cada issuer y subject, pero en la práctica no se están usando
- **Extensions.** Campo opcional metido en X.509 v.3 con información adicional que no aparece en ninguno de los campos predefinidos. Cada extensión tiene tres atributos: (1) **OID** que indica el tipo de la extensión. (2) **Flag crítico** que indica si la extensión es crítica. Si el flag crítico aparece en la extensión de un certificado, el programa que

no sepa procesar esta extensión debe considerar al certificado como inválido. Si la extensión no tiene el flag crítico activado, el programa puede ignorar esta extensión. Por último (3) estaría el **valor de la extensión**, cuyo valor depende del tipo de extensión. Ejemplos típicos de extensiones se muestran en la Tabla 6.1.

- `SignatureAlgorithm`. Algoritmo usado por el issuer para firmar el certificado
- `SignatureValue`. Valor de la firma

Version	← v.3
Serial number	← 48
Signature	← DSA with SHA-1
Issuer	← C=US o=Verisign cn=Verisign Inc.
Validity	← 3/10/2004 - 3/10/2009
Subject	← C=SP o=MacProgramadores cn=Fernando López
PublicKey	← RSA: 03 5A 34 32 B3 43 D3 43 F3 2A 43 E3 25
IssuerUniqueID	← (Normalmente vacío)
SubjectUniqueID	← (Normalmente vacío)
Extensions	
SignatureAlgorithm	← DSA with SHA-1
SignatureValue	← 53 93 A6 4F 8B 28 78 32 AB 46 92 08 C5 2A 50

Figura 6.10 Certificado X.509

1.4.2 Tipos de certificados

Si atendemos a las extensiones del certificado, básicamente existen cuatro tipos de certificados X.509:

- **Certificados de usuario**, que se expediten a los usuarios finales.
- **Certificados de servidor**, que se instalan en un servidor web o de otro tipo, y que tienen una extensión que indica que el servidor acepta conexiones SSL y TLS.
- **Certificados de CA**, expedidos a una CA, y que en las competencias del certificado (extensión `CertificatePolicies`) indica que el certificado puede firmar otros certificados.
- **Certificados self-signed**, son certificados que se firman a sí mismos y que poseen los trusting point.

Extensión	Descripción
BasicConstraints	Tipo de certificado que puede ser <code>EndEntity</code> (usuarios y servidores) o <code>CA</code>
KeyUsage	Posibles usos que puede tener el certificado: <code>CertSign</code> , <code>CRLSign</code> , <code>DigitalSignature</code> , <code>KeyAgreement</code> , <code>KeyEncryptment</code> , <code>DataEncryptment</code> .
ExtendedKeyUsage	Indica aplicaciones para el certificado. Cada aplicación tiene un OID. Por ejemplo, <code>ServerAuthentication</code> con OID <code>(1.3.6.1.5.5.7.3.1)</code> indica que se puede usar la clave pública para un servidor TLS.
CertificatePolicies	Tiene información sobre la política de seguridad del servidor. Suele incluir una URL donde se detalla este documento
CRLDistributionPoints	Indica una o más URL donde se pueden encontrar las CRL del este certificado

Tabla 6.1: Extensiones X.509

1.4.3 Cómo solicitar un certificado

Para solicitar un certificado hay que generar un CSR (Certificate Signature Request) que es un documento con la extensión `.csr` con la información que queremos que se firme. Despu  s este documento se env  a a una CA para que lo firme.

En Java, la herramienta `keytool`, que veremos en el apartado 2.2.1, es una herramienta que nos permite entre otras cosas generar CSR. `openssl`, otra herramienta que veremos en el apartado 6, tambi  n nos permite generarlos. Normalmente, cada servidor web dispone de una herramienta que permite, entre otras cosas, generar un CSR.

1.5 Las CRL X.509

Cuando un certificado necesita ser revocado la CA pone su n  mero de serie en una CRL.

Los certificados suelen llevar la extensi  n `CRLDistributionPoints` en la que se indica las URLs donde encontrar su CRL. Las CA suelen agrupar las CRL de un gran conjunto de usuarios en un determinado fichero puesto en una URL (p.e. agrupando a sus usuarios por pa  ses). Estos CRL tampoco suelen crecer mucho ya que afortunadamente pocas veces los certificados son revocados.

El formato de una CRL aparece en la Figura 6.11, y contiene los siguientes campos:

- **Version.** Suele ser la v.3
- **Signature.** Una copia de `SignatureAlgorithm` que aparece firmada
- **Issuer.** CN del organismo que firma la CRL
- **ThisUpdate.** Fecha en que se ha expedido la CRL
- **NextUpdate.** Primera fecha de salida de la próxima CRL
- **RevokedCertificates.** Lista de números de serie de certificados revocados
- **CrlExtensions.** Extensiones del CRL. En la práctica poco usadas
- **SignatureAlgorithm.** Algoritmo con que se firma la CRL
- **SignatureValue.** La firma en sí

Version	← v.3
Signature	← DSA with SHA-1
Issuer	← C=US o=Verisign cn=Verisign Inc.
ThisUpdate	← 3/10/2004
NextUpdate	← 17/10/2004
Revoked Certificates	
CrlExtensions	← DSA with SHA-1
SignatureAlgorithm	← 53 93 A6 4F 8B 28 78 32 AB 46 92 08 C5 2A 50
SignatureValue	

Figura 6.11 Formato de una CRL

1.6 Los repositorios

La RFC 2585 define un protocolo genérico para acceder a los certificados y CRL de un repositorio.

De acuerdo a esta especificación, a estos datos podemos acceder básicamente mediante tres protocolos:

- 1) **HTTP.** En este caso las URL tienen la forma¹:

`http://cert.verisign.com/pki/id81.cer` (certificados)
`http://crl.verisign.com/pki/id81.crl` (CRLs)

- 2) **FTP anónimo.** En cuyo caso accedemos a direcciones como:

`ftp://cert.verisign.com/pki/id81.cer` (certificados)
`ftp://crl.verisign.com/pki/id81.crl` (CRLs)

- 3) **LDAP.** En el caso de HTTP o FTP estos nos devuelven directamente un fichero con un certificado o una CRL con la lista de certificados revocados. En el caso de LDAP debemos preguntar por una determinada entrada, y después acceder a los atributos de esta entrada tal como detalla la RFC 2587. Además este protocolo nos permite realizar búsqueda de certificados y CRL por tipo.

1.7 Validar el camino de certificación

Un **camino de certificación**, tal como muestra la Figura 6.12, es una cadena de certificados donde el subject del primer certificado es el dueño del certificado que queremos validar, y el issuer del último certificado es el trusting point.

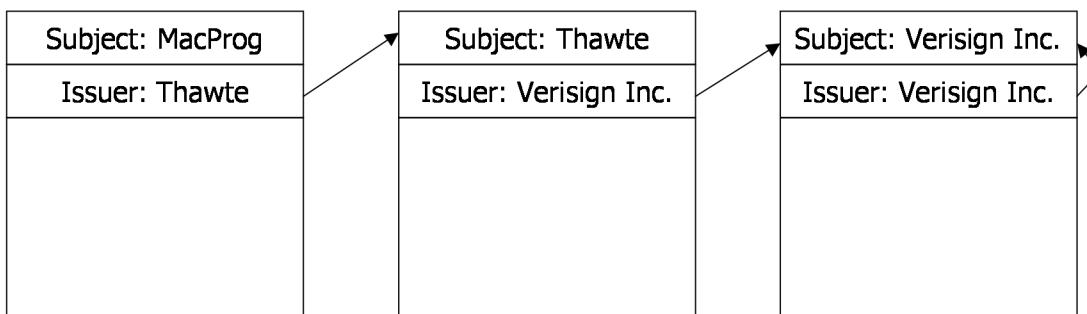


Figura 6.12 Camino de certificación

La comprobación de un certificado se suele hacer en dos pasos:

1. **Construcción del camino de certificación**, donde se construye un camino desde el certificado que queremos validar hasta un trusting point, tal como muestra la Figura 6.12.
2. **Validación del certificado**, donde se van validando los certificados desde el certificado que queremos validar hasta un trusting point, comprobando que para ningún certificado intermedio haya un CRL.

¹ Actualmente los certificados pueden tener la extensión .crt o .cer

Por eficacia estos dos pasos se suelen simultanear de acuerdo al siguiente algoritmo recursivo:

```
CompruebaCertificado(c)
{
    crl <- ObtenerCRL(c)
    IF (c ∈ crl) THEN
        return FALSE
    IF (c es un trusting point) THEN
        return TRUE
    c <- Issuer(c)
    return CompruebaCertificado(c)
}
```

Actualmente la mayoría de los programas no comprueban el camino de certificación por el overhead que esto conlleva. En el caso de los browsers, ningún browser conocido comprueba el camino de certificación, aunque sí que podemos ir a [CRLVERISIGN] y descargarnos e instalarnos los certificados ahí publicados en nuestro browser.

1.8 Formatos estándar

En este apartado vamos a comentar cuáles son los formatos más conocidos para el intercambio de certificados a través de un protocolo de comunicación. Básicamente vamos a comentar estos cuatro:

1. PKCS#10

Este es el formato más sencillo para solicitar un certificado en una CSR, y el más usado. En el documento se indican los campos que aparecen en la Figura 6.13. Vemos que empieza con tres campos estándares, y todos los demás campos se meten en [Attributes]. Por último, el solicitante firma el documento para demostrar que es el dueño del mismo.

Version
DN
PublicKey
[Attributes]
SignatureAlgorithm
Signature

Figura 6.13 Campos de un documento PKCS#10

PKCS#10 es sólo el formato de una solicitud, para transportarlo necesitamos usar un protocolo de comunicación que suele ser, o bien SSL, donde el documento se envía por SSL al servidor, o bien PKCS#7 que es un estándar para transportar un documento encriptado. El documento encriptado se puede enviar a la CA a través de HTTP, o a través de S/MIME.

2. PCKS#12

Es un formato para transportar claves privadas, certificados y datos encriptados usado por Netscape, Mozilla y FireFox.

3. CMP (Certificate Management Protocol)

Es un protocolo muy completo diseñado por la IETF en las RFC 2510 y RFC 2511 para solicitar certificados, CRLs, y solicitudes de revocación. Actualmente no se ha implantado mucho debido a su complejidad.

4. SCEP (Simple Certified Enrollment Protocol)

Protocolo diseñado por Cisco System para distribuir de forma segura certificados a través de sus routers, y comprobar CRLs.

2 Certificados digitales en Java

2.1 Clases para gestión de certificados

La clase `java.security.cert.Certificate`¹ es una clase abstracta que sirve para representar un certificado genérico.

La clase tiene los métodos:

```
abstract void <Certificate> verify(PublicKey key)
```

Que comprueba que el certificado haya sido firmado por la clave privada del CA que corresponde a la clave pública de la CA pasada en el parámetro `key`.

```
abstract PublicKey <Certificate> getPublicKey()
```

Que devuelve la clave pública del certificado.

```
abstract byte[] <Certificate> getEncoded()
```

Que devuelve un encoded del certificado.

Hay que tener cuidado de no confundir la clave pública que debemos pasar a `verify()`, que es la clave pública del CA que firma el certificado, con la clave pública que devuelve `getPublicKey()` que es la clave pública del usuario dueño del certificado.

Los certificados son normalmente inmutables, luego la clase carece de métodos `set`.

Como muestra la Figura 6.14, la clase `X509Certificate` deriva de la clase abstracta `Certificate` y representa un certificado X.509.

La clase añade nuevos métodos como `getIssuerDN()`, `getSubject()`, `getNotAfter()`, ... que nos permiten acceder a los distintos campos de un certificado X.509.

La clase `CertificateFactory` es una clase que nos permite cargar un certificado de fichero, y generar CRL para un certificado.

¹ En JDK 1.1 existía la clase `java.security.Certificate`, pero en el JDK 1.2 se cambió para que todas las clases relacionadas con certificados digitales se metieran en el paquete `java.security.cert`, con lo que la clase se cambio a `java.security.cert.Certificate` y se deprecó la anterior, con lo que hay que tener cuidado con los `import` para importar la nueva y no la antigua.

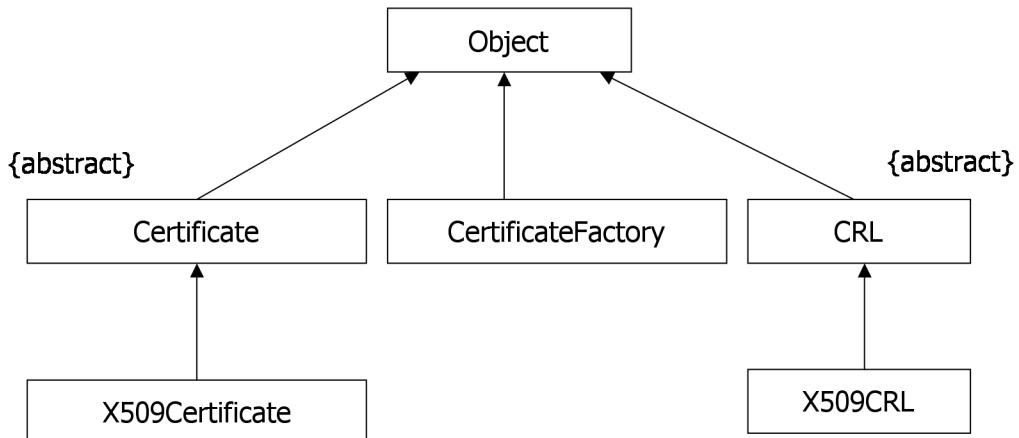


Figura 6.14 Clases para trabajar con certificados

Como todos los servicios criptográficos, para crear un certificado usamos:

```
static CertificateFactory <CertificateFactory> getInstance(
    String type)
```

En el parámetro `type` debemos pasarle "X.509"

Después, para cargar un certificado de fichero usamos:

```
Certificate <CertificateFactory> generateCertificate(
    InputStream inStream)
```

O bien, si tenemos un fichero PKCS #7 con varios certificados, los podemos cargar todos usando:

```
Collection <CertificateFactory> generateCertificates(
    InputStream inStream)
```

También podemos generar una CRL para un certificado usando:

```
CRL <CertificateFactory> generateCRL(InputStream inStream)
```

El método nos devuelve un objeto de tipo `X509CRL` que, como muestra la Figura 6.14, es una derivada de `CRL` que representa las CRL para X.509.

Como ejemplo el Listado 6.1 muestra un pequeño programa que nos permite imprimir el contenido de un certificado que tendremos en un fichero.

Los certificados se pueden descargar usando un browser de cualquier servidor seguro, para lo cual debemos pedir al browser que nos muestre el certificado, e indicarle en qué fichero nos lo debe guardar.

```

import java.io.*;
import java.security.cert.*;

public class ImprimeCertificado
{
    public static void main(String[] args) throws Exception
    {
        if (args.length==0)
        {
            System.out.println("Indique fichero del"
                + " certificado como argumento");
            return;
        }
        FileInputStream fis =
            new FileInputStream(args[0]);
        CertificateFactory cf =
            CertificateFactory.getInstance("X.509");
        X509Certificate c = (X509Certificate)
            cf.generateCertificate(fis);
        System.out.println(c);
        fis.close();
    }
}

```

Listado 6.1: Programa que imprime el contenido de un certificado

La salida del programa es la siguiente:

```

$ java ImprimeCertificado www.uno-e.com.cer
[
[
    Version: V3
    Subject: CN=www.uno-e.com, OU="Member, VeriSign Trust Network",
    OU=Authenticated by VeriSign, OU=Terms of use at www.verisign.ch/rpa (c) 04,
    OU=Departamento Informatica, O=UNO-E, L=Madrid, ST=Madrid, C=ES
    Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

    Key: Sun RSA public key, 1024 bits
    modulus:
1380505526238048446631004122307444037295473254050232951249087699472756919322
5314914018021360336533253990752245812761653175876133563849615896052971551600
9491839554058673788328402553952968748262674006235138247798124277377568227721
0296422609785697992939226752561052471119462473350393459465499065698978259686
78581
    public exponent: 65537
    Validity: [From: Wed Apr 26 02:00:00 CEST 2006,
               To: Thu May 08 01:59:59 CEST 2008]
    Issuer: OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97
    VeriSign, OU=VeriSign International Server CA - Class 3, OU="VeriSign,
    Inc.", O=VeriSign Trust Network
    SerialNumber: [ 12396bd8 407b98fe 581c65a8 f8e2b7a4]

    Certificate Extensions: 4
    [1]: ObjectId: 2.16.840.1.113730.1.1 Criticality=false
    NetscapeCertType [
        SSL server
    ]

    [2]: ObjectId: 2.5.29.3 Criticality=false
    Extension unknown: DER encoded OCTET string =

```

```

0000: 04 82 02 16 30 82 02 12 30 82 02 0E 30 82 02 0A ....0...0...0...
0010: 06 0B 60 86 48 01 86 F8 45 01 07 01 01 30 82 01 ..`H...E....0..
0020: F9 16 82 01 A7 54 68 69 73 20 63 65 72 74 69 66 .....This certif
0030: 69 63 61 74 65 20 69 6E 63 6F 72 70 6F 72 61 74 icate incorporat
0040: 65 73 20 62 79 20 72 65 66 65 72 65 6E 63 65 2C es by reference,
0050: 20 61 6E 64 20 69 74 73 20 75 73 65 20 69 73 20 and its use is
0060: 73 74 72 69 63 74 6C 79 20 73 75 62 6A 65 63 74 strictly subject
0070: 20 74 6F 2C 20 74 68 65 20 56 65 72 69 53 69 67 to, the VeriSig
0080: 6E 20 43 65 72 74 69 66 69 63 61 74 69 6F 6E 20 n Certification
0090: 50 72 61 63 74 69 63 65 20 53 74 61 74 65 6D 65 Practice Stateme
00A0: 6E 74 20 28 43 50 53 29 2C 20 61 76 61 69 6C 61 nt (CPS), availa
00B0: 62 6C 65 20 61 74 3A 20 68 74 74 70 73 3A 2F 2F ble at: https://
00C0: 77 77 77 2E 76 65 72 69 73 69 67 6E 2E 63 6F 6D www.verisign.com
00D0: 2F 43 50 53 3B 20 62 79 20 45 2D 6D 61 69 6C 20 /CPS; by E-mail
00E0: 61 74 20 43 50 53 2D 72 65 71 75 65 73 74 73 40 at CPS-requests@
00F0: 76 65 72 69 73 69 67 6E 2E 63 6F 6D 3B 20 6F 72 verisign.com; or
0100: 20 62 79 20 6D 61 69 6C 20 61 74 20 56 65 72 69 by mail at Veri
0110: 53 69 67 6E 2C 20 49 6E 63 2E 2C 20 32 35 39 33 Sign, Inc., 2593
0120: 20 43 6F 61 73 74 20 41 76 65 2E 2C 20 4D 6F 75 Coast Ave., Mou
0130: 6E 74 61 69 6E 20 56 69 65 77 2C 20 43 41 20 39 ntain View, CA 9
0140: 34 30 34 33 20 55 53 41 20 54 65 6C 2E 20 2B 31 4043 USA Tel. +1
0150: 20 28 34 31 35 29 20 39 36 31 2D 38 38 33 30 20 (415) 961-8830
0160: 43 6F 70 79 72 69 67 68 74 20 28 63 29 20 31 39 Copyright (c) 19
0170: 39 36 20 56 65 72 69 53 69 67 6E 2C 20 49 6E 63 96 VeriSign, Inc
0180: 2E 20 20 41 6C 6C 20 52 69 67 68 74 73 20 52 65 . All Rights Re
0190: 73 65 72 76 65 64 2E 20 43 45 52 54 41 49 4E 20 served. CERTAIN
01A0: 57 41 52 52 41 4E 54 49 45 53 20 44 49 53 43 4C WARRANTIES DISCL
01B0: 41 49 4D 45 44 20 61 6E 64 20 4C 49 41 42 49 4C AIMED and LIABIL
01C0: 49 54 59 20 4C 49 4D 49 54 45 44 2E A0 0E 06 0C ITY LIMITED.....
01D0: 60 86 48 01 86 F8 45 01 07 01 01 01 A1 0E 06 0C `H...E.....
01E0: 60 86 48 01 86 F8 45 01 07 01 01 02 30 2C 30 2A `H...E....0,0*
01F0: 16 28 68 74 74 70 73 3A 2F 2F 77 77 77 2E 76 65 .(https://www.ve
0200: 72 69 73 69 67 6E 2E 63 6F 6D 2F 72 65 70 6F 73 risign.com/repos
0210: 69 74 6F 72 79 2F 43 50 itory/CPS
53 20

```

[3]: ObjectId: 2.5.29.37 Criticality=false
 ExtendedKeyUsages [
 [2.16.840.1.113730.4.11]]

```

[4]: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[  

CA:false  

PathLen: undefined  

]  

]  

Algorithm: [SHA1withRSA]  

Signature:  

0000: 52 0C AD BF 7B C4 F6 71 4A 08 0B 93 C6 81 21 7C R.....qJ.....!.  

0010: BD 38 8C 9C 81 4A 8F 4D FE 59 6C 1C CE 0B E1 39 .8...J.M.Yl....9  

0020: 06 E9 81 B5 9C 37 D9 A9 AB FB 32 02 2F 4B 08 2A .....7....2./K.*  

0030: 0F 20 38 0F FA 5D 65 8D AA 25 B1 D0 E2 E7 8A E3 . 8..]e..%.....  

0040: 89 EA 11 0F 2B BE E2 B5 0D 18 2E 2B AD 00 AD C7 .....+.....+....  

0050: 01 04 BC DA 71 1C BE CD 0F B5 34 F8 26 C6 05 5F .....q.....4.&...  

0060: 99 A1 AC 1A FC 85 AE F5 50 B6 FC FF 61 BA 23 97 .....P...a.#.  

0070: CD 69 4C 80 31 8C 59 58 7D 80 86 53 AC C7 DE 11 .iL.1.YX...S....

```

2.2 El Keystore de Java

Java dispone de un **keystore** que nos permite almacenar tres tipos de claves:

1. **Claves secretas binarias**, protegidas por un password.
2. **Certificados**, son pares de clave pública / clave privada, los cuales se almacenan junto con una cadena de certificados que llevan hasta un trusting point que avala la cadena. El primer certificado siempre será el certificado que contiene la clave pública y el último es el trusting point. La clave privada se protege con un password.
3. **Trusting Points**, que son certificados de los cuales sólo conocemos la clave pública, y que queremos usar como trusting points para nuestro sistema PKI.

2.2.1 La herramienta keytool

Para gestionar los keystores Java dispone del comando `keytool`, el cual si lo ejecutamos sin parámetros obtenemos:

```
$ keytool
keytool usage:

-certreq      [-v] [-protected]
               [-alias <alias>] [-sigalg <sigalg>]
               [-file <csr_file>] [-keypass <keypass>]
               [-keystore <keystore>] [-storepass <storepass>]
               [-storetype <storetype>] [-providerName <name>]
               [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-delete       [-v] [-protected] -alias <alias>
               [-keystore <keystore>] [-storepass <storepass>]
               [-storetype <storetype>] [-providerName <name>]
               [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-export       [-v] [-rfc] [-protected]
               [-alias <alias>] [-file <cert_file>]
               [-keystore <keystore>] [-storepass <storepass>]
               [-storetype <storetype>] [-providerName <name>]
               [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-genkey      [-v] [-protected]
               [-alias <alias>]
               [-keyalg <keyalg>] [-keysize <keysize>]
               [-sigalg <sigalg>] [-dname <dname>]
               [-validity <valDays>] [-keypass <keypass>]
               [-keystore <keystore>] [-storepass <storepass>]
               [-storetype <storetype>] [-providerName <name>]
               [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-help
-identitydb [-v] [-protected]
              [-file <idb_file>]
              [-keystore <keystore>] [-storepass <storepass>]
              [-storetype <storetype>] [-providerName <name>]
```

```

[-providerClass <provider_class_name> [-providerArg <arg>]] ...

-import      [-v] [-noprompt] [-trustcacerts] [-protected]
             [-alias <alias>]
             [-file <cert_file>] [-keypass <keypass>]
             [-keystore <keystore>] [-storepass <storepass>]
             [-storetype <storetype>] [-providerName <name>]
             [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-keyclone    [-v] [-protected]
             [-alias <alias>] -dest <dest_alias>
             [-keypass <keypass>] [-new <new_keypass>]
             [-keystore <keystore>] [-storepass <storepass>]
             [-storetype <storetype>] [-providerName <name>]
             [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-keypasswd   [-v] [-alias <alias>]
             [-keypass <old_keypass>] [-new <new_keypass>]
             [-keystore <keystore>] [-storepass <storepass>]
             [-storetype <storetype>] [-providerName <name>]
             [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-list        [-v | -rfc] [-protected]
             [-alias <alias>]
             [-keystore <keystore>] [-storepass <storepass>]
             [-storetype <storetype>] [-providerName <name>]
             [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-printcert  [-v] [-file <cert_file>]

-selfcert    [-v] [-protected]
             [-alias <alias>]
             [-dname <dname>] [-validity <valDays>]
             [-keypass <keypass>] [-sigalg <sigalg>]
             [-keystore <keystore>] [-storepass <storepass>]
             [-storetype <storetype>] [-providerName <name>]
             [-providerClass <provider_class_name> [-providerArg <arg>]] ...

-storepasswd [-v] [-new <new_storepass>]
             [-keystore <keystore>] [-storepass <storepass>]
             [-storetype <storetype>] [-providerName <name>]
             [-providerClass <provider_class_name> [-providerArg <arg>]] ...

```

Si al ejecutar keytool no especificamos como argumento un fichero, con la opción `-keystore`, el programa usa por defecto el fichero `$HOME/.keystore`

La Tabla 6.2 resume brevemente la utilidad de cada una de estos argumentos:

Operación	Descripción
<code>-certreq</code>	Permite crear un CSR.
<code>-delete</code>	Borra una entrada del keystore.
<code>-export</code>	Permite exportar certificados del keystore.
<code>-genkey</code>	Permite crear un certificado self-signed en el que se almacena el par clave pública / clave privada.
<code>-help</code>	Muestra un mensaje de ayuda con las distintas opciones de keytool.
<code>-identitydb</code>	Convierte un keystore del JDK 1.1 al JDK 1.2.
<code>-import</code>	Permite importar un certificado de un trusting point.
<code>-keyclone</code>	Permite hacer una copia de un certificado con otro alias.

<code>-keypasswd</code>	Permite indicar el password de una determinada entrada.
<code>-list</code>	Muestra el contenido del keystore.
<code>-printcert</code>	Permite imprimir el contenido de un certificado pasado como argumento con la opción <code>-file</code> .
<code>-selfcert</code>	Permite crear un certificado self-signed a partir de un certificado creado con <code>-genkey</code> (que también son self-signed). Esta opción es útil sólo para cambiar el nombre del dueño del certificado.
<code>-storepasswd</code>	Permite indicar el password de todo el keystore.

Tabla 6.2: Operaciones del comando `keytool`

Trusting points de Sun

Sun dispone de una lista de trusting points en el fichero `$JRE_HOME/lib/security/cacerts` que podemos consultar con el comando:

```
$ keytool -keystore $JRE_HOME/lib/security/cacerts -list
```

Para acceder a este keystore nos pide un password que por defecto es "changeit".

JKS, JCEKS y PCKS12

Como veremos en el siguiente apartado, el keystore es un servicio del API de Java, y como tal puede tener varios algoritmos y proveedores. Aunque actualmente sólo "SUN" es proveedor.

Sun proporciona tres algoritmos para el keystore que son:

- "JKS". Es el algoritmo por defecto. Este algoritmo es capaz de leer y almacenar certificados y trusting points, pero no puede almacenar claves asimétricas. Además las claves privadas almacenadas no se protegen con un algoritmo PBE, sino que la clave privada se almacena en texto plano, y lo único que se guarda es un hash del password que usa el algoritmo a la hora de recuperar el password. Esto se hizo así por las restricciones de exportación de los EEUU.
- "JCEKS". Este algoritmo sí que permite almacenar claves simétricas encriptadas con PBE, y las claves privadas también las encripta con PBE.
- "PKCS12". No es un keystore totalmente funcional, sino que sólo permite leer los documentos PKCS #12 de Netscape, aunque no permite escribirlos.

Podemos indicar a `keytool` que queremos usar otro algoritmo con el argumento `-provider`.

También podemos indicar que queremos usar por defecto otro algoritmo modificando la entrada:

```
keystore.type=jks
```

del fichero `$JAVA_HOME/lib/security/java.security`

Opciones globales

Antes de ver cómo se realizan cada una de las operaciones básicas de gestión de certificados con el comando `keytool`, vamos a comentar una serie de opciones globales que tiene la herramienta:

`-alias alias`

Los distintos tipos de claves se almacenan en el keystore asignándoles un alias que indicamos con esta opción (p.e `-alias sdo`).

`-dname distinguishedname`

Permite especificar un DN. Por ejemplo:

```
-dname "CN=Fernando Lopez, O=macprogramadores, C=ES"
```

`-keypass password`

Permite indicar el password que protege una determinada entrada.

`-storepass password`

Permite indicar el password que protege todo el keystore.

`-keystore keystore`

Permite indicar el path del fichero con el keystore al que queremos acceder.

`-storetype type`

Permite indicar el tipo del keystore que puede ser `JKS`, `JCEKS` o `PKCS12`.

Muchas opciones de keystore tienen un valor por defecto cuando no se indica valor. La Tabla 6.3 muestra estos valores.

Opción	Valor por defecto
<code>-alias</code>	<code>mykey</code>
<code>-keyalg</code>	<code>DSA</code>
<code>-keysize</code>	<code>1024</code>
<code>-sigalg</code>	<code>DSA/SHA-1</code>
<code>-validity</code>	<code>90</code>
<code>-keystore</code>	<code>\$HOME/.keystore</code>
<code>-file</code>	Entrada y salida estándar

Tabla 6.3 : Valores por defecto de las opciones del `keytool`

Importar un certificado

Cuando vamos a importar un certificado debemos indicar el alias donde guardarlo, con la opción `-alias`, así como el nombre del fichero del que leer el certificado, con la opción `-file`.

```
$ keytool -import -alias uno -file www.uno-e.com.cer
Enter keystore password: ****
Owner: CN=www.uno-e.com, OU="Member, VeriSign Trust Network",
OU=Authenticated by VeriSign, OU=Terms of use at www.verisign.ch/rpa (c) 04,
OU=Departamento Informatica, O=UNO-E, L=Madrid, ST=Madrid, C=ES
Issuer: OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign,
OU=VeriSign International Server CA - Class 3, OU="VeriSign, Inc.",
O=VeriSign Trust Network
Serial number: 12396bd8407b98fe581c65a8f8e2b7a4
Valid from: Wed Apr 26 02:00:00 CEST 2006 until: Thu May 08 01:59:59 CEST
2008
Certificate fingerprints:
    MD5: EC:67:9F:98:72:18:58:CC:C8:9B:49:EE:22:3D:B6:DC
    SHA1: CC:2A:4F:B3:65:77:24:19:0E:7B:EF:78:6A:EF:6F:E1:27:5C:DC:91
Trust this certificate? [no]: yes
Certificate was added to keystore
```

El programa primero nos pide un password para el keystore que va a crear, y luego nos muestra el fingerprint del certificado tanto en MD5 como en SHA-1, y nos pide que comprobemos este fingerprint antes de añadir el certificado al keystore.

Ahora podemos comprobar que el certificado haya sido añadido correctamente con la opción `-list`:

```
$ keytool -list
Enter keystore password: ****
Keystore type: jks
Keystore provider: SUN
Your keystore contains 1 entry
uno, Dec 24, 2006, trustedCertEntry,
Certificate fingerprint (MD5):
EC:67:9F:98:72:18:58:CC:C8:9B:49:EE:22:3D:B6:DC
```

Exportar un certificado

Análogamente podemos exportar un certificado que tengamos en el keystore con la opción `-export`:

```
$ keytool -export -alias uno -file uno-e2.cer
Enter keystore password: ****
Certificate stored in file <uno-e2.cer>
```

Crear un certificado

El certificado importado previamente era un trusting point del cual sólo tenemos la clave pública. También podemos pedir a `keytool` que cree un

certificado propio del que tendremos el par clave privada / clave pública con el argumento `-genkey`:

```
$ keytool -genkey -alias fernando
Enter keystore password: ****
What is your first and last name?
[Unknown]: Fernando Lopez Hernandez
What is the name of your organizational unit?
[Unknown]: Formacion
What is the name of your organization?
[Unknown]: MacProgramadores
What is the name of your City or Locality?
[Unknown]: Madrid
What is the name of your State or Province?
[Unknown]: Madrid
What is the two-letter country code for this unit?
[Unknown]: ES
Is CN=Fernando Lopez Hernandez, OU=formacion, O=macprogramadores, L=Madrid,
ST=Madrid, C=ES correct?
[no]: yes
Enter key password for <fernando>
      (RETURN if same as keystore password):
```

Este certificado es un certificado self-signed que luego se puede exportar e instalar en un browser que quiera establecer una conexión segura con nosotros, como veremos más adelante.

Crear un CSR

Podemos crear un CSR (Certificate Signature Request) que luego enviaremos a una CA para que nos lo firme usando la opción `-certreq`:

Para ello tenemos que tener ya un certificado creado y puesto un alias en el keystore. Por ejemplo si tenemos un certificado con el alias `fernando`, podemos generar un CSR con el comando:

```
$ keytool -certreq -alias fernando -file fernando.csr
Enter keystore password: *****
```

Cuando el CA nos devuelva el certificado firmado lo podremos instalar con `-import`.

Borrar un certificado instalado

Por último comentar que podemos borrar un certificado que tengamos instalado con la opción `-delete`:

```
$ keytool -delete -alias fernando
Enter keystore password: *****
```

2.2.2 La clase KeyStore

Podemos acceder programáticamente a un keystore a través de la clase `java.security.KeyStore`. Como servicio que es, para crear un objeto de este tipo usamos el método:

```
static KeyStore <KeyStore> getInstance(String type)
```

Al cual le pasamos en `type` el tipo de keystore a crear: "`JKS`", "`JCEKS`" o "`PKCS12`". Una vez tengamos el objeto, tenemos que cargarlo con datos de un fichero usando:

```
void <KeyStore> load(InputStream stream, char[] password)
```

Después podemos obtener un certificado almacenado en el keystore usando:

```
Certificate <KeyStore> getCertificate(String alias)
```

En caso de tratarse de un trusting point nos devuelve el certificado del trusting point, o si se trata de un certificado normal nos devuelve el primer certificado de la cadena.

O bien una clave que tengamos en un alias con:

```
Key <KeyStore> getKey(String alias, char[] password)
```

En caso de ser una clave secreta lo que tengamos en `alias` nos devuelve la clave secreta en un objeto de tipo `SecretKey`, y si se trata de un certificado nos devuelve la clave privada del certificado en un objeto de tipo `PrivateKey`.

Para almacenar claves secretas dejamos `chain` a `null` en el método:

```
void <KeyStore> setKeyEntry(String alias, Key key
                           , char[] password, Certificate[] chain)
```

Si por el contrario queremos almacenar un certificado, en `key` debemos pasar la clave privada, y en `chain` el certificado o certificados que forman el camino de certificación. Para almacenar un trusting point usamos el método:

```
void <KeyStore> setCertificateEntry(String alias
                                    , Certificate cert)
```

Por último, para almacenar el contenido del `KeyStore` a disco usamos:

```
void <KeyStore> store(OutputStream stream, char[] password)
```

3 S/MIME

S/MIME es un protocolo desarrollado conjuntamente por RSA Security Inc, Microsoft y Netscape para la firma y encriptación de mensajes. El protocolo ha sido estandarizado por la IETF en la RFC 2633.

A diferencia de PGP, este protocolo exige disponer de un certificado de usuario firmado por una CA. Actualmente Thawte está expidiendo gratuitamente certificados digitales para correo electrónico en su web.

Una vez disponemos del certificado digital expedido a nuestro nombre, y en el que se incluye nuestra dirección de correo con una extensión X.509, podemos firmar documentos y enviárselos a otras personas.

Los mensajes encriptados con S/MIME se transportan como mensajes de correo MIME con los tipos `multipart/signed` y `multipart/encrypted`.

Además el mensaje lleva un objeto de tipo `application/pkcs7-mime` en el que se transporta el certificado digital del usuario.

4 IPSec

4.1 Visión general

IPSec es una solución criptográfica que pretende introducir la seguridad, no a nivel de aplicación o de transporte, sino a nivel IP. Esto presenta la ventaja de que proporciona seguridad a todas las aplicaciones existentes sin necesidad de modificarlas. En principio el inconveniente de IPSec sería que añade un overhead al tener que encriptar incluso comunicaciones que no hace falta encriptar. Para evitar este inconveniente IPSec propone que cuando no se requiera seguridad se utilice un null cipher.

IPSec es un trabajo de la IETF que está descrito en varias RFC, entre ellas están las RFC 2401, RFC 2402, RFC 2406 y RFC 2410.

Los principales servicios que proporciona IPSec son cuatro:

- Confidencialidad
- Integridad de datos
- Identificación de la fuente y destino de datos
- Protección frente a un replay attack

Además IPSec es independiente del algoritmo criptográfico subyacente, con el fin de que si a un algoritmo se le encuentra un problema de seguridad, éste se pueda reemplazar fácilmente.

Uno de los aspectos curiosos de IPSec es que aunque está en la capa IP, es orientado a conexión, lo cual no debe sorprendernos ya que para enviar datos encriptados necesitamos establecer una clave de sesión.

A las conexiones en IPSec se las llama **SA (Security Association)** y son conexiones unidireccionales entre dos puntos. Luego si queremos establecer una conexión bidireccional necesitamos dos SA.

El identificador de cada conexión se llama **SPI (Security Parameter Index)** y se transporta en cada paquete IP, como veremos en el apartado 4.3.

IPSec proporciona varias formas de proteger la comunicación: Proteger una sola conexión TCP, proteger todo el tráfico entre dos host, o proteger todo el tráfico entre un par de routers (útil para crear una VPN).

Además, para proteger estas comunicaciones IPSec tiene dos **modos de operación**:

- **Transport mode**, donde la cabecera IPSec se inserta justo después de la cabecera IP. En este caso el campo `protocol` de la cabecera IP se cambia al valor 51 para indicar que después de la cabecera IP va una cabecera IPSec. La cabecera IPSec contiene básicamente un SPI, un número de secuencia, información de integridad, y otros campos que comentaremos luego.
- **Tunnel mode**. En este modo cada paquete IP a proteger y la información de seguridad de este paquete se envuelve dentro de otro paquete IP. Este modo es especialmente útil cuando la dirección del paquete es distinta al destino final, como pasa en los routers que usaremos para crear una VPN, ya que así las máquinas de la organización no tienen que tener implementado IPSec, sólo el router.

El tunnel mode tiene una ventaja respecto al transport mode: el criptoanalista no puede saber quién está enviando paquetes a quién. A veces esta información es útil para el criptoanalista. Por ejemplo, si durante una crisis militar el criptoanalista detecta una bajada en el tráfico entre el pentágono y la casa blanca, y un aumento similar del tráfico entre el pentágono y una base militar en Colorado, esta información puede dar ideas al criptoanalista sobre lo que está ocurriendo.

El inconveniente del tunnel mode es que aumenta el tamaño de los datos transmitidos al añadir otra cabecera IP al protocolo.

Una comunicación IPSec se realiza en dos pasos:

1. Establecimiento de una clave de sesión y SPI entre las partes, lo cual se realiza utilizando el protocolo IKE.
2. Envío de datos protegidos, para lo cual existen dos protocolos alternativos: AH y ESP

En los siguientes apartados vamos a detallar estos protocolos.

4.2 IKE (Internet Key Exchange)

IKE es el protocolo que define cómo crear un SPI entre dos ordenadores e intercambiar una clave de sesión de forma segura. Este protocolo es muy flexible y permite usar varios algoritmos de intercambio de claves, así como comprobar la identidad del interlocutor usando certificados digitales.

4.3 AH (Authentication Header)

AH es un formato de cabecera IPSec que proporciona integridad y protección ante un replay attack, pero no tiene encriptación.

Como muestran la Figura 6.15 y Figura 6.16, se debe añadir un padding al payload para poder calcular el MAC.

El significado de los campos de la cabecera es el siguiente:

- `NextHeader` indica el valor anterior que estaba en el campo `protocol`, lo cual permite usar IPSec para trasportar todo tipo de paquetes (TCP, UDP, etc).
- `PayLength` indica el número de unidades de 32 bits que ocupan los campos `TCPHeader+Payload+Padding`
- `SPI` lo usa el receptor para identificar la SA del paquete.
- `SequenceNumber` numera todos los paquetes enviados sobre una SA. Cada paquete tiene un número único, incluidas las retransmisiones, lo cual permite detectar un replay attack.
- `MAC` es un MAC (véase apartado 9.2 del Tema 1) del `Payload+Padding` que garantiza su integridad. Para generarlo se usa la clave de sesión de la SA. Para calcular la MAC se incluyen también los campos fijos de la cabecera IP (p.e. `TimeToLive` no se incluye), lo cual evita que se modifiquen por un atacante activo (p.e. modificar la dirección IP origen).

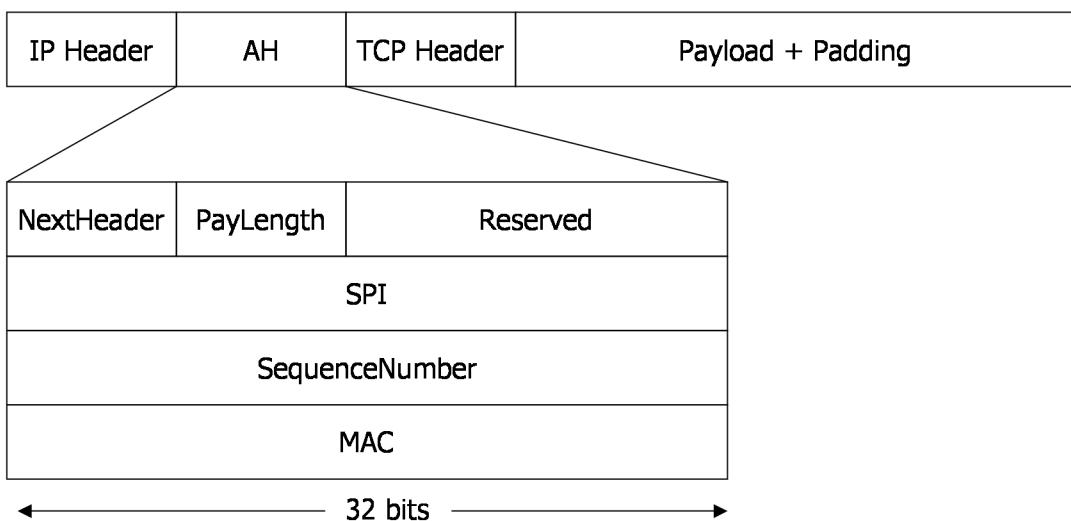


Figura 6.15 Formato de una cabecera AH de IPSec

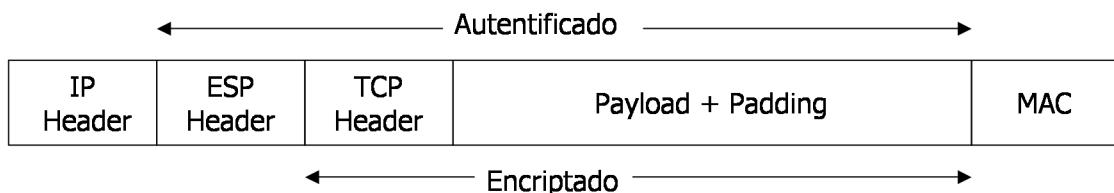
4.4 ESP (Encapsulating Security Payload)

Este modo añade a AH la encriptación (aunque puede usarse un null cipher) y pretende sustituir al anterior.

Como muestra la Figura 6.16 se puede usar tanto en tunnel mode como en transport mode.

La cabecera ESP es parecida a la cabecera AH, pero añade un campo con un IV usado para encriptación, y el MAC se mueve al final del paquete ya que es más fácil de implementar por hardware la generación del MAC si se pone al final.

Transport mode



Tunnel mode

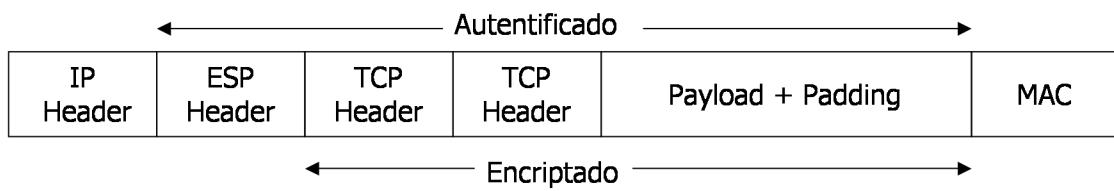


Figura 6.16 Formato de una cabecera ESP de IPSec

5 SSL y TLS

5.1 Introducción

SSL (Socket Secure Layer) es un protocolo que nos proporciona un canal (socket) seguro entre un cliente y un servidor. Tiene básicamente tres características: Es una capa transparente a la aplicación, permite comprobar la autenticidad del servidor (y opcionalmente la del cliente), y todos los datos circulan encriptados.

Para conseguir esta transparencia existen librerías para varios lenguajes y entornos (p.e. OpenSSL) con API similares a las de los sockets.

SSL se puede aplicar a cualquiera de los protocolos de aplicación ya existentes usando uno de estos dos enfoques:

- **Puertos separados.** consiste en elegir dos puertos separados para las conexiones seguras y no seguras, p.e. http en el 80 y https en el 443. También existen versiones seguras para otros protocolos como son ftps, nntps, etc...
- **Negociación de la conexión.** La otra opción es que un servidor instalado en el mismo puerto pueda negociar entre usar el protocolo tradicional sin encriptación, o encriptarlo con SSL. Por ejemplo, SMTP y POP3 usan esta estrategia.

Esta segunda estrategia presenta el inconveniente de que hay que modificar el servidor, y que no todos los clientes pueden tener implementado el soporte para esta modificación.

5.2 Evolución histórica

La Figura 6.17 muestra la evolución histórica de SSL.

SSL v.2 fue la primera versión pública que sacó Netscape. Desafortunadamente tenía problemas de seguridad en la generación de números aleatorios, para lo que usaba la hora del reloj.

PCT (Private Communication Technology) es una alternativa a SSL de Microsoft que incluye soporte para encriptar comunicaciones UDP y realizar identificación sin encriptación.

SSL v.3 es una SSL de Netscape mejorado, a la que le quitó los problemas de seguridad y le añadió la posibilidad de usar varios algoritmos. En concreto, le

añadió la posibilidad de usar DH y DSA en lugar de RSA, ya que en aquel tiempo RSA todavía conservaba la patente.

STLP (Secure Transport Layer Protocol) es una propuesta de mejora a SSL v.3 de Microsoft que no fue muy aceptada.

TLS (Transport Layer Security) es una mejora a SSL v.3 de la IETF (RFC 2246) que ha sido aceptada por Microsoft y Netscape.

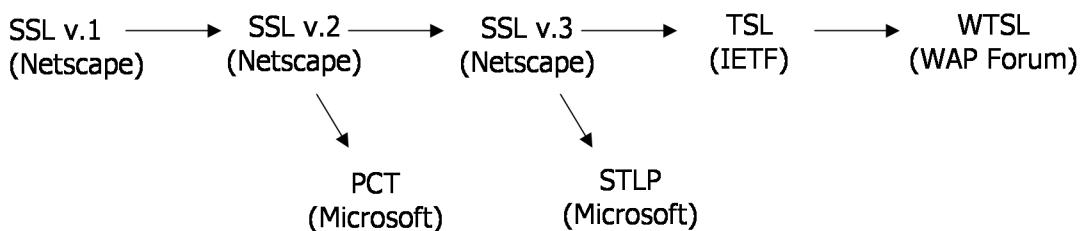


Figura 6.17 Evolución histórica de SSL

5.3 Establecimiento de una conexión

Vamos a empezar viendo cómo se establece una conexión típica en la que sólo se autentifica el servidor.

La conexión tiene básicamente tres etapas:

1. **Handshake**, donde se autentifica al servidor, y se realiza un key agreement entre cliente y servidor.
2. **Transmisión de datos segura**, donde se transmiten los datos encriptados de forma segura.
3. **Cierre de la conexión** de forma segura

A continuación vamos a detallar cada una de estas etapas.

5.3.1 Handshake

El handshake tiene tres propósitos:

- Acordar los algoritmos criptográficos a usar durante la conexión.
- Acordar las claves criptográficas de sesión.
- Identificar al servidor (y opcionalmente al cliente).

El protocolo de handshake sería el siguiente:

1. El cliente envía al servidor una lista con los algoritmos criptográficos que soporta, junto con un número aleatorio que usará para el key agreement.
2. El servidor elige un algoritmo criptográfico de la lista y se lo envía al cliente junto con su certificado digital y el número aleatorio que había enviado el cliente firmado con su clave privada.
3. El cliente comprueba la validez del certificado, extrae su clave pública y comprueba que el número haya sido firmado correctamente.
4. El cliente genera una clave de sesión, la encripta con la clave pública del servidor y se la envía.
5. El servidor usa la clave de sesión para calcular el MAC de toda la comunicación (usando la clave de sesión) y se lo envía al cliente.
6. El cliente calcula ahora el MAC de toda la comunicación, y se lo envía al servidor.

La elección del algoritmo se realiza en los pasos 1 y 2, mientras que el key agreement se realiza en los pasos 1, 2, 3 y 4.

Los pasos 5 y 6 evitan un ataque activo en el que el atacante cambia el mensaje del paso 1 para que el cliente sólo ofrezca una lista de algoritmos criptográficos débiles.

Aquí hemos visto cómo se realiza el handshake con RSA. En caso de que el handshake se haga con DH el protocolo sería ligeramente distinto.

5.3.2 Transmisión segura de datos

La transmisión segura de datos busca principalmente dos objetivos:

- **Encriptación**, que evite desvelar el texto transmitido
- **Integridad**, para evitar que los datos transmitidos se modifiquen

SSL trasporta los datos en bloques llamados **registros SSL** los cuales se obtienen siguiendo el siguiente protocolo (véase Figura 6.18):

1. Los datos se parten en bloques llamados **fragmentos**, los cuales pueden tener longitud variable.
2. A cada fragmento se le calcula un **MAC** (usando la clave de sesión), y se le concatena al fragmento.

3. Se encripta el fragmento y el MAC usando la clave de sesión, para obtener el llamado **registro SSL**.

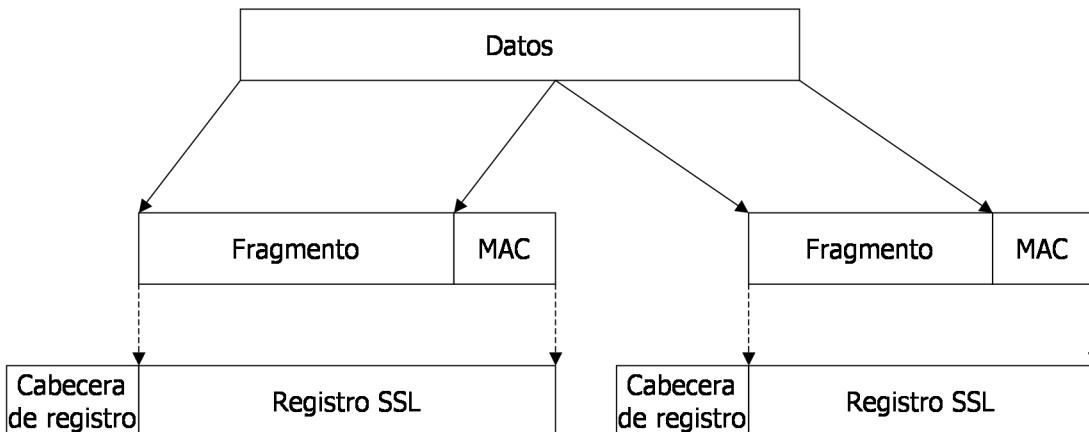


Figura 6.18 Obtención de los fragmentos SSL

Al registro SSL se le añade una **cabecera de registro** con información sobre el contenido del registro. La cabecera de registro tiene tres campos: (1) Un **número de versión**, el cual es redundante y sólo sirve para comprobar que todo va bien, (2) la **longitud** en bytes del registro SSL, (3) y el **Content Type** que indica el tipo de datos que transporta el registro SSL, y que puede ser uno de los que aparecen en la Tabla 6.4:

Content Type	Descripción
application_data	Datos normales
handshake	Se usan para renegociar el handshake durante la comunicación
change_cipher_spec	Permite cambiar el algoritmo criptográfico durante la comunicación
alert	Se usa para enviar mensajes de error, y para indicar el final de la comunicación

Tabla 6.4: Tipos de datos que puede transportar el registro SSL

5.3.3 Cierre de la conexión

Es importante que las conexiones SSL se cierren con un par de mensajes de `alert` con el subtipo `close_notify` (como muestra la Figura 6.19), porque si no se puede hacer un ataque en el que el receptor reciba sólo parte de los datos (p.e. sólo un trozo de una página web).

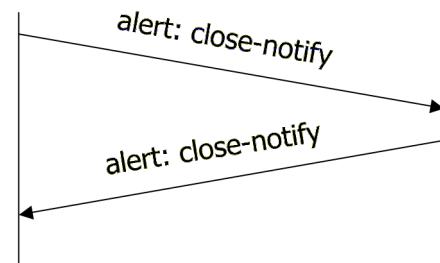


Figura 6.19 Cierre de la conexión

Estos mensajes de cierre de la conexión evitan

que el atacante activo envíe un segmento FIN de TCP que corte prematuramente la conexión.

En caso de que este cierre no se haga correctamente, los datos recibidos no se deben considerar válidos, aunque hay implementaciones de SSL que ignoran esta regla.

5.4 Sesiones

El proceso de handshake es un proceso costoso debido a que usa criptografía de clave pública. Para reducir este coste SSL permite que el handshake sólo se realice en la primera conexión, de forma que las siguientes conexiones sigan usando las mismas claves. Para conseguir esto, la primera vez que nos conectamos al servidor, éste devuelve al cliente un `session_id` que el cliente entregará en las demás conexiones.

El tiempo de vigencia de la sesión es personalizable en el servidor, pero si por alguna razón la conexión se corta indebidamente, la sesión se debe invalidar.

5.5 Identificación de clientes

SSL proporciona un mecanismo para que el cliente se pueda identificar usando un certificado de usuario. Para que el cliente se autentique debe ser el servidor quien se lo pida durante el proceso de handshake, o bien al realizar un rehandshake que explicaremos en el siguiente apartado.

5.6 Rehandshake

El rehandshake se ha introducido en SSL básicamente por dos motivos:

El primero fue para permitir que el cliente de un servidor seguro se identifique cuando entra en una zona en la que requiera autorización especial.

El otro motivo fue debido a las restricciones de exportación de los EEUU, donde se diseñó un sistema en el que primero se establecía una conexión de criptografía débil, y luego, si el usuario estaba dentro de EEUU o tenía un certificado que le permitiera usar criptografía fuerte, se renegociaba el handshake para cambiar los algoritmos criptográficos. A esta técnica Microsoft la llamó **Server Gated Cryptography**, y Netscape la llamó **Step Up**.

Estos certificados que permitían usar criptografía fuerte sólo los expedía Verisign Inc. y Thawte, aunque actualmente esta técnica ya no se usa debido a que estas restricciones de exportación se quitaron.

6 OpenSSL

OpenSSL es una implementación open source de SSL. Consta de una librería C que permite a los programadores realizar todas las operaciones criptográficas típicas (no sólo las relacionadas con SSL). Entre estas operaciones encontramos: encriptar ficheros, firmarlos, crear certificados digitales, mensajes de correo S/MIME, etc.

Además OpenSSL consta de una interfaz de comandos, que es la que nosotros vamos a estudiar en este apartado.

Suponemos que el lector conoce los conceptos criptográficos básicos de encriptación, firmas digitales y certificados digitales vistos en los temas anteriores, con lo que nos centraremos en ver cómo se implementan estos conceptos en OpenSSL.

Esta interfaz es el comando `openssl`, el cual tiene dos modos de operación: modo interactivo y modo comando. El **modo interactivo** es en el que entramos cuando escribimos el comando sin argumentos, para salir del intérprete de comandos podemos usar el comando `quit`. El **modo comando** es el que usamos cuando pasamos argumentos al programa, y es el que nosotros vamos a estudiar.

En general, `openssl` en modo comando recibe como argumento un **nombre de comando** (p.e. `enc`, `dgst`, `sign`, etc) y una serie de **opciones** las cuales se preceden por guión (p.e. `-pass`, `-file`, `-noout`, etc). Además, a las opciones las puede seguir uno o más **argumentos**. Luego el formato general del comando `openssl` es:

```
openssl command [ -command_opts ] [ command_args ]
```

6.1 Encriptación de ficheros

OpenSSL soporta una amplia gama de cifradores (Blowfish, AES, CAST5, DES, 3DES, IDEA, RC2, RC4 y RC5 entre otros). También soporta los diferentes modos de encriptación (ECB, CBC, CFB, OFB), y el modo por defecto es CBC, que es el que usa cuando no se especifica modo.

Tanto para encriptar como para desencriptar se usa el comando `enc`, sólo que en caso de querer desencriptar se añade la opción `-d`. Alternativamente podemos usar el nombre del algoritmo criptográfico que queremos usar para encriptar o desencriptar como comando (p.e. `des`, `des3`, `rc2`, `rc5`, etc).

Para indicar los nombres de los ficheros de entrada y salida se usan las opciones `-in` y `-out` seguidas del correspondiente nombre de fichero.

Por ejemplo, para encriptar el fichero `mensaje.txt` con DES haríamos:

```
$ openssl enc -des -in mensaje.txt -out mensaje.txt.des
enter des-cbc encryption password:*****
Verifying - enter des-cbc encryption password:*****
```

Alternativamente podríamos haber usado el comando `des` directamente así.

```
$ openssl des -in mensaje.txt -out mensaje.txt.des
enter des-cbc encryption password:*****
Verifying - enter des-cbc encryption password:*****
```

Si ahora queremos desencriptar el fichero usaremos la opción `-d` así:

```
$ openssl des -d -in mensaje.txt.des -out mensaje.txt
enter des-cbc decryption password:*****
```

La opción de encriptación también puede usarse para convertir un texto a base 64 usando como algoritmo criptográfico `base64`, y de hecho éste es el cifrador por defecto cuando no se especifica cifrador.

Obsérvese que, para encriptar, OpenSSL nos está pidiendo un `password` al cual le realiza un PBE. En caso de usar `password` es recomendable pedir al algoritmo que use `salt` para aumentar su seguridad, para lo cual podemos usar la opción `-salt`.

También podemos pedirle que use una clave binaria, pero en este caso debemos pasársela la clave binaria en hexadecimal (y, si procede, el vector de inicialización) con la opción `-K` y `-iv` respectivamente:

```
$ openssl bf -in mensaje.txt -out mensaje.txt.bf -K
4587445A452B423F432D45E3 -iv E423D2342
```

En este ejemplo estamos pidiendo que encripte con Blowfish el fichero `mensaje.txt` con la clave y IV pasados como argumento.

También podemos pasársela a `openssl` el fichero donde está la clave con la opción `-kfile`.

6.2 Pasar el password

El `password` también se puede pasar como una opción más, aunque la forma de pasarlo por desgracia depende del comando a ejecutar. En los comandos que sólo llevan asociado un `password` se usa la opción `-pass`, mientras que

en los que pueden tener dos passwords (un password antiguo y uno nuevo) se suelen usar las opciones `-passin` y `-passout`.

A continuación de la opción se indica de dónde obtener el password, pudiendo éste proceder de distintas fuentes, unas más seguras que otras. Estas fuentes son:

`-pass:password`

Escribimos el password directamente en la línea de comandos.

`-pass:var_entorno`

Leemos el password de la variable de entorno `var_entorno`.

`-pass:fichero`

Leemos el password del `fichero`.

Por ejemplo para encriptar un fichero indicando el password en la línea de comando usamos:

```
$ openssl des -in mensaje.txt -out mensaje.txt.des -pass  
pass:sesamo
```

6.3 Message digest

El comando `openssl` permite calcular el hash de un fichero usando diversos algoritmos (MD2, MD4, MD5, MDC2, SHA1). Para ello usamos el comando `dgst` seguido de una opción con el algoritmo a aplicar (p.e. `-md2`, `-md4`, `-md5`, `-mdc2`, `-sha1`), o bien el nombre del algoritmo hash a aplicar directamente (p.e. `md2`, `md4`, `md5`, `mdc2`, `sha1`).

Por ejemplo para calcular el hash SHA1 del fichero `mensaje.txt` haríamos:

```
$ openssl sha1 mensaje.txt  
SHA1(mensaje.txt) = 8779409b80bb55800b7685b04003f5d1cae26ca1
```

Por defecto el hash nos lo devuelve por la salida estándar pero también lo podemos mandar a un fichero con la opción `-out`:

```
$ openssl sha1 -out mensaje.txt.sha1 mensaje.txt  
$ cat mensaje.txt.sha1  
SHA1(mensaje.txt) = 8779409b80bb55800b7685b04003f5d1cae26ca1
```

El hash sale en texto hexadecimal, pero también podemos sacar el hash en binario con la opción `-binary`.

6.4 Certificados

En este apartado vamos a aprender todo lo referente a la gestión de certificados con OpenSSL. Para ello vamos a suponer que queremos actuar como una CA, vamos a empezar configurando el entorno de OpenSSL para luego expedir un certificado raíz, y por último firmar certificados de usuarios que queremos meter en nuestra organización.

6.4.1 Crear un directorio para los certificados

El primer paso que tenemos que dar para crear una CA es crear un directorio donde almacenar la clave privada de la CA, así como los certificados y CRL que firme la CA.

Nosotros vamos a suponer que `macprogramadores.org` es una organización que quiere firmar los certificados de sus miembros, con lo que vamos a crear el directorio `$HOME/macproCA` donde meteremos los ficheros de configuración de la CA.

Dentro de este subdirectorio debemos meter dos subdirectorios¹: `certs` donde vamos a meter los certificados que creemos, y `private` donde meteremos la clave privada de la CA. La CA puede tener varios certificados raíz y varias claves privadas para firmar certificados, en cuyo caso todas ellas se meterían en sus respectivos subdirectorios. En principio la información que tengamos en `macproCA` es pública y puede ser accesible por otros usuarios, la única excepción son las claves privadas del directorio `private`, este directorio le debemos proteger restringiendo los permisos al máximo para evitar que otros usuarios accedan a él.

Después necesitamos crear una serie de ficheros estándar que debe contener el directorio `macproCA`:

El primero de estos ficheros será `macproCA/serial` y en él se meterán los números de serie de los certificados que creemos. Es importante que este número de serie sea único para cada certificado, con lo que empezaremos metiendo un 1 en el fichero. Además la herramienta OpenSSL exige que este número esté en hexadecimal y que tenga al menos dos dígitos con lo que meteremos un 0 delante.

El otro fichero que tenemos que crear es `macproCA/index.txt` que es una base de datos en la que OpenSSL almacena los certificados que ha expedido. Inicialmente este fichero debe existir aunque no hayamos creado certificados, con lo que lo vamos a crear vacío.

¹ Estos nombres de directorios están recomendados por OpenSSL

Es decir lo que haremos será:

```
$ cd $HOME
$ mkdir macproCA
$ cd macproCA
$ mkdir certs private
$ chmod g-rwx,o-rwx private
$ echo '01' > serial
$ touch index.txt
```

6.4.2 El fichero de configuración

Lo siguiente que vamos a hacer es crear un fichero de configuración para la CA. Aunque este fichero no es estrictamente necesario, ya que si no existe OpenSSL usa valores por defecto para las opciones que no le pasemos, este fichero es muy recomendable para gestionar la CA.

Normalmente durante la instalación de OpenSSL se crea un fichero de configuración global que en el caso de Mac OS X está en `/System/Library/OpenSSL/openssl.cnf`¹, en el caso de Linux este fichero se encuentra en `/etc/ssl/openssl.cnf`, y en otros sistemas operativos se podría encontrar en otro directorio parecido.

Para saber en qué directorio se encuentra este fichero de configuración podemos ejecutar el comando:

```
$ openssl ca
```

Este comando dará la ruta del fichero global de configuración del CA y una serie de errores debido a que el fichero no está bien configurado.

Nosotros no vamos a usar este fichero sino que crearemos otro fichero de configuración personalizado (aunque animamos al lector a que abra el fichero de configuración por defecto para ver un ejemplo de qué forma tienen estos ficheros).

El Listado 6.2 muestra la forma que va a tener nuestro fichero de configuración. El fichero tiene una serie de secciones, cada una de ellas corresponde a uno de los comandos que puede usar el comando `openssl`, y que en principio son actualmente sólo tres: `ca`, `req` y `x509`. Dentro de cada sección encontramos pares clave-valor asociados a esa sección. Además se pueden crear pares clave-valor globales, que estarán antes de empezar las secciones.

¹ Si tenemos instalado `openssl` de Fink la ruta del fichero de configuración también podrá ser `/sw/etc/ssl/openssl.cnf`

La primera sección que encontramos es la sección `ca`. Esta sección enumera las CA de primer nivel de que disponemos, que en nuestro caso es sólo una: `macpro_ca`. Podemos pedir que esta CA sea la que se use por defecto cuando no se especifique nombre de CA en las opciones de la línea de comandos usando la entrada `default_ca` como hace nuestro ejemplo.

La entrada anterior nos lleva a la sección `macpro_ca` en la que configuramos nuestra CA. La entrada `dir` la usamos para indicar el directorio donde están los ficheros de configuración de nuestra CA, y esta entrada se puede expandir como si fuera un macro (aunque sólo dentro de su sección) precediéndola por el símbolo `$` como muestran las siguientes entradas.

`certificate` indica dónde se encuentra el certificado de nuestra CA, `database` indica en qué fichero se guardará información sobre los certificados creados, `new_certs_dir` indica en qué directorio guardar los certificados que creamos, `private_key` indica donde está la clave privada de nuestra CA y `serial` permite a OpenSSL seguir la cuenta de los números de serie asignados.

```
[ca]
default_ca = macpro_ca

[macpro_ca]
dir = /Users/fernando/macproCA
certificate = $dir/certs/cacert.cer
database = $dir/index.txt
new_certs_dir = $dir/certs
private_key = $dir/private/cakey.private
serial = $dir/serial

default_crl_days = 7
default_days = 365
default_md = md5

policy = macpro_policy
x509_extensions = extensiones

[macpro_policy]
commonName = supplied
stateOrProvinceName = optional
countryName = optional
emailAddress = supplied
organizationName = match
organizationalUnitName = supplied

[extensiones]
basicConstraints = CA:false
```

Listado 6.2: Fichero de configuración de un CA

Las otras entradas que encontramos son: `default_crl_days` que indica cada cuantos días queremos actualizar nuestros CRL, `default_days` que indica el periodo de validez por defecto de los certificados que expidamos, `default_md` para indicar el algoritmo de message digest a usar en los certificados creados.

`policy` es una entrada que indica los valores que deberá tener el DN de los certificados expedidos a partir de los CSR (Certificate Signature Request) que recibamos de los usuarios. Si nos llegan CSR que tienen valores para el DN que no están en estos campos, estos valores serán ignorados. Cada uno de estos campos puede tomar los valores: `supplied` que indica que el CSR debe tener este campo, `optional` que permite que el CSR no tenga este campo, y en consecuencia no lo tenga el certificado que firmemos, y `match` que indica que el valor del CSR debe coincidir con el valor de nuestro certificado raíz para poder ser firmado (p.e. el nombre de la organización).

Por último la entrada `x509_extensions` nos lleva a otra sección en la que indicamos las extensiones que deben tener los certificados que expidamos. En nuestro ejemplo sólo hemos pedido la extensión `basicConstraints` poniendo el valor `CA:false` que significa que no queremos que los dueños de los certificados que expidamos puedan actuar como CAs firmando certificados.

Como no queremos que OpenSSL utilice el fichero de configuración global, sino que queremos que use el de nuestro ejemplo debemos indicárselo creando la siguiente variable de entorno:

```
$ OPENSSL_CONF=/Users/fernando/macproCA/openssl.cnf  
$ export OPENSSL_CONF
```

6.4.3 Crear el certificado raíz

Antes de empezar a expedir certificados necesitamos crear un certificado raíz self-signed para nuestra organización con el que firmar los demás certificados.

Para ello usaremos el comando `req`, que sirve tanto para crear un CSR, como para crear un certificado raíz. Pero antes de crear este certificado necesitamos añadir información adicional a nuestro fichero de configuración. En concreto necesitamos añadir una nueva sección para el comando `req` como muestra el Listado 6.3.

`defaults_bits` indica el tamaño de la clave. Como por defecto es de 512 bits (que es un poco pequeña), vamos a fijar que por defecto sea de 1024 bits.

`prompt` y `distinguished_name` indican la información necesaria para el DN que necesita OpenSSL.

Por último la sección `extensionesCA` la usaremos desde la línea de comandos cuando queremos crear un certificado con permisos para firmar certificados.

```
[req]
default_bits          = 1024
prompt                = no
distinguished_name    = macpro_dn

[macpro_dn]
commonName            = MacProgramadores CA
stateOrProvinceName   = Madrid
countryName           = ES
emailAddress          = webmaster@macprogramadores.org
organizationName      = macprogramadores.org

[extensionesCA]
basicConstraints       = CA:true
```

Listado 6.3: Secciones para la creación de certificados

Ahora podemos pedir que nos genere el certificado raíz con el comando:

```
$ openssl req -x509 -newkey rsa:2048 -keyform PEM -outform PEM
-keyout macproCA/private/cakey.private -out
macproCA/certs/cacert.cer -extensions extensionesCA
Generating a 2048 bit RSA private key
.....+
.....+
writing new private key to 'private/cakey.private'
Enter PEM pass phrase:*****
Verifying - Enter PEM pass phrase: *****
```

Cuando al comando `req` le pasamos la opción `-x509` le estamos diciendo que queremos que nos cree un certificado. Si no le pasamos la opción `-x509`, lo que estamos pidiendo es un CSR. La opción `-newkey` indica que también queremos crear una clave privada, si a esta opción no la indicamos nada más crearía una clave privada del tipo y tamaño por defecto (1024 bits que es el tamaño que damos en el fichero de configuración), pero como se trata del certificado raíz merece la pena que sea un poco más fuerte y por eso lo hemos subido a 2048 bits. `-keyform` y `-outform` indican el formado del fichero de clave privada y de certificado. El formato PEM es el formato más usado para estos documentos. `-keyout` y `-out` indican los nombres de los ficheros de clave privada y de certificado. Por último `-extensions` indica que queremos usar las extensiones de la sección `[extensionesCA]`, en la que decimos que queremos que este certificado pueda firmar otros certificados. Es extremadamente importante que guardemos bien el password que nos pide esta vez `openssl`, ya que es el que daría acceso a un atacante a

falsificar cualquier certificado de nuestra organización. Este password lo usa `openssl` para encriptar el fichero de clave privada `cakey.private` con 3DES.

6.4.4 Hacer un CSR

Ahora los demás miembros de la organización deberán enviarnos un CSR con los datos que quieren que les firmemos.

Para ello usarán el comando `req` (sin la opción `-x509`) así¹:

```
$ openssl req -newkey rsa -keyform PEM -outform PEM -keyout
jose.private -out jose.csr
Generating a 1024 bit RSA private key
.....+++++
.....+++++
writing new private key to jose.private'
Enter PEM pass phrase:*****
Verifying - Enter PEM pass phrase:*****
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Murcia
Locality Name (eg, city) []:Murcia
Organization Name (eg, company) [Internet Widgits Pty
Ltd]:macprogramadores.org
Organizational Unit Name (eg, section) []:Documentacion
Common Name (eg, YOUR name) []:Jose Abel Marcos
Email Address []:jose@macprogramadores.org

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

El resultado de este proceso es la creación de dos ficheros en el directorio actual:

- `jose.private` con la clave privada de José protegida con el password que indicó.

¹ Es importante que este comando lo ejecutemos desde otro terminal donde no esté fijada la variable de entorno `OPENSSL_CONF`, con el fin de que se use el fichero de configuración global y no el que creamos para el CA.

- `jose.csr` con el CSR que José debe enviar al CA para que lo firme.

El programa nos pide todos los datos que quiere José que tenga su certificado, y después nos acaba pidiendo un segundo password de challenge que es el que usará el CA para desencriptar el CSR. Si no nos interesa poner este password podemos dejar el campo en blanco (como hemos hecho en este caso).

6.4.5 Firmar una CSR

Ahora le toca al CA firmar el CSR que le envíe José. Antes de firmar el certificado puede ver su contenido con el comando:

```
$ openssl req -in jose.csr -text -noout
Certificate Request:
Data:
Version: 0 (0x0)
Subject: C=ES, ST=Murcia, L=Murcia, O=macprogramadores.org,
OU=Documentacion, CN=Jose Abel
Marcos/emailAddress=jose@macprogramadores.org
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
00:c1:98:40:d4:be:3f:3f:b9:82:b0:36:9b:b0:81:
c5:72:9c:63:a8:69:ac:a1:e4:80:4b:ef:91:44:25:
f9:e8:72:40:d0:a5:89:3e:c6:5a:12:49:b9:9e:06:
8f:ad:b8:6b:45:ca:cf:84:d8:6d:e7:e6:31:ac:90:
26:f2:ab:9c:3b:b1:1c:16:16:b1:64:d5:23:c9:d7:
66:63:21:49:37:2d:bf:7c:b2:68:9e:3f:14:04:87:
f8:85:b9:0c:86:d4:b3:21:50:78:a5:8f:f8:35:fe:
7e:e5:44:81:13:31:e7:1e:cd:89:ed:ce:19:e8:1d:
24:aa:1d:d5:05:d4:a4:d1:bf
Exponent: 65537 (0x10001)
Attributes: challengePassword:
Signature Algorithm: md5WithRSAEncryption
76:17:de:d0:b5:48:e6:3b:ad:ac:7f:81:17:b9:99:7f:05:be:
63:9f:46:16:8d:d2:ed:9a:4b:9c:bc:14:2e:1f:64:0c:ad:4b:
74:75:4a:df:ac:12:ed:96:cf:af:3a:93:eb:48:3e:2a:27:3f:
c4:f7:f2:6c:3a:e7:cb:d0:4e:41:72:82:dd:fa:59:cb:1c:f3:
0c:a3:64:4b:ef:46:79:4a:1b:14:bd:d3:d9:69:8e:63:26:c4:
3f:16:a2:0e:82:1f:95:f3:4b:40:51:ba:55:a8:8e:65:89:71:
03:a5:90:df:94:8c:f0:1d:6f:74:0e:a7:ca:96:4f:97:f1:26:
04:ca
```

Si la CA está de acuerdo con los datos del CSR puede firmarlo con el comando:

```
$ openssl ca -keyfile macproCA/private/cakey.private -in
jose.csr
Using configuration from /Users/fernando/macproCA/openssl.cnf
```

```
Enter pass phrase for private/cakey.private: ****
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName          :PRINTABLE:'ES'
stateOrProvinceName :PRINTABLE:'Murcia'
localityName         :PRINTABLE:'Murcia'
organizationName     :PRINTABLE:'macprogramadores.org'
organizationalUnitName:PRINTABLE:'Documentacion'
commonName           :PRINTABLE:'Jose Abel Marcos'
emailAddress         :IA5STRING:'jose@macprogramadores.org'
Certificate is to be certified until Feb 12 20:10:58 2005 GMT
(365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Para firmar certificados el comando que se usa no es `req` sino `ca`. Lo primero que hace OpenSSL es pedirnos el password para acceder al fichero de la clave privada, después nos muestra los datos a firmar, y si estamos de acuerdo los firma.

OpenSSL nos escribe el certificado expedido en el subdirectorio `certs`, y actualiza los ficheros `serial` y `index.txt` convenientemente:

```
$ cd macprogCA
$ ls certs
01.pem
cacert.cer
$ cat serial
02
$ cat index.txt
V 050212201058Z 01 unknown /CN=Jose Abel Marcos /ST=Murcia
/C=ES /emailAddress=jose@macprogramadores.org
/O=macprogramadores.org /OU=Documentacion
```

El fichero `01.pem` es el que contiene el certificado que hemos firmado. Si queremos podemos copiar este fichero con el nombre `jose.cer`, y entregárselo a José.

6.4.6 Revocar certificados

Por último vamos a revocar el certificado que hemos expedido para ver cómo se realizaría este proceso. Para ello simplemente tenemos que usar el comando `ca` con la opción `-revoke` e indicar el certificado a revocar:

```
$ openssl ca -keyfile macproCA/private/cakey.private -revoke
macproCA/certs/01.pem
Using configuration from /Users/fernando/macproCA/openssl.cnf
```

```
Enter pass phrase for private/cakey.private:*****
Revoking Certificate 01.
Data Base Updated
```

Esta operación no modifica el certificado, sino que simplemente modifica el contenido del fichero de base de datos `index.txt`:

```
$ cat index.txt
R 050212201058Z 040213204855Z 01 unknown /CN=Jose Abel
Marcos/ST=Murcia /C=ES /emailAddress=jose@macprogramadores.org
/O=macprogramadores.org /OU=Documentacion
```

Vemos que ahora el fichero está precedido por una `R` (Revoked).

Lo siguiente que tenemos que hacer es distribuir los CRL. Estos documentos se expiden periódicamente con una periodicidad que indicamos en `default_crl_days`. Además los CRL tienen un campo `NextUpdate` del CRL que indican la próxima vez que se volverán a actualizar, es decir, a los CRL se les suele dar fecha de caducidad para que los usuarios sepan cuando vamos a actualizarlos.

Cuando vayamos a expedir un CRL debemos usar el comando `ca` junto con la opción `-gencrl` así:

```
$ openssl ca -gencrl -keyfile macproCA/private/cakey.private -
out macpro.crl
Using configuration from /Users/fernando/macproCA/openssl.cnf
Enter pass phrase for private/cakey.private:*****
```

El CRL lo debemos publicar (por ejemplo en un servidor HTTP) para que nuestros usuarios lo puedan usar.

Por último comentar que los usuarios deberían comprobar que el CRL no haya sido modificado por un atacante, para lo cual pueden usar el comando:

```
$ openssl crl -in macpro.crl -CAfile jose.cer
verify OK
```

Donde `jose.cer` es una copia del fichero `macproCA/certs/01.pem`.

6.5 Firmar documentos

Las firmas digitales, al igual que las funciones hash, se calculan con el comando `dgst`, pero añadiendo la opción `-sign` o `-verify`. La opción `-sign` debe ir precedida por el fichero de la clave privada a usar para firmar, y la opción `-verify` debe ir seguida del fichero con el certificado que queremos usar para verificar la firma.

Por ejemplo, si José quiere firmar un mensaje con su certificado puede usar el comando:

```
$ openssl dgst -sha1 -sign jose.private -out mensaje.txt.sign
mensaje.txt
```

Ahora Fernando puede verificar la firma de José. Para ello primero debe extraer la clave pública del certificado de José con el comando:

```
$ openssl x509 -pubkey -in jose.cer -noout > jose.public
```

Y después puede comprobar la firma con el comando:

```
$ openssl dgst -sha1 -verify jose.public -signature mensaje.txt.sign
mensaje.txt
Verified OK
```

6.6 Documentos S/MIME

OpenSSL también nos permite encriptar, desencriptar, firmar, y verificar mensajes S/MIME.

Para firmar un mensaje usamos el comando:

```
$ openssl smime -sign -signer fernando.cer -inkey
fernando.private -in mensaje.txt -from "Fernando Lopez
Hernandez <fernando@macprogramadores.org>" -subject "Mensaje
de prueba" -to "Jose <jose@macprogramadores.org>"
Enter pass phrase for fernando.private:*****
To: Jose <jose@macprogramadores.org>
From: Fernando Lopez Hernandez <fernando@macprogramadores.org>
Subject: Mensaje de prueba
MIME-Version: 1.0
Content-Type: multipart/signed; protocol="application/x-pkcs7-
signature"; micalg=sha1; boundary="----"
66D4E2BA031B24CE266CD9384 6DD555E"
```

This is an S/MIME signed message

```
-----66D4E2BA031B24CE266CD93846DD555E
Hola este es un mensaje para hacer pruebas con S/MIME.
```

Espero que no te moleste.

Saludos.

```
-----66D4E2BA031B24CE266CD93846DD555E
Content-Type: application/x-pkcs7-signature; name="smime.p7s"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="smime.p7s"
```

MIIIFWQYJKoZIhvCNQcCoIIFSjCCBUYCAQExCzAJBgUrDgMCGgUAMAsGCSqGSIb3DQEHAaCCAzYwggMyMIICGqADAgECAgECMA0GCSqGSIB3DQEBAUAMIGKMRwwGgYDVQQDExNNYWNQcm9ncmFtYWRvcmVzIENBMQ8wDQYDVQQIEwZNYWRyaWQxCzAJBgNVBAYTAKVTMS0wKwYJKoZIhvCNQkBFh53ZWJtYXN0ZXJabWFjcHJvZ3JhbWFkb3J1cy5vcmcxtHTAbBgNVBAoTFG1hY3Byb2dyYW1hZG9yZXMu3JnMB4XDTA0MDIxNDA4Mzc0M1oXDTA1MDIxMzA4Mzc0M1owgaIxITAfBgNVBAMTGEZ1cm5hbmRvIEsvcGV6IEh1cm5hbmRlejEPMA0GA1UECBMGTWFkcm1kMQswCQYDVQQGEwJFUzEsMCoGCSqGSIB3DQEJARYdZmVybmuZG9AbWFjcHJvZ3JhbWFkb3J1cy5vcmcxtHTAbBgNVBAoTFG1hY3Byb2dyYW1hZG9yZXMu3JnMRIwEAYDVQQLEw1Gb3JtYWNpb24wgZ8wDQYJKoZIhvCNQEBBQADgY0AMIGJAoGBAKZNrucMIUezvucOW/H/UznYdhMITJUt25+Yxf165ZfQo/RK9JxPCZwtc4XFsnxKBVjOqAX4b/mmrsRDpvi/TN6B7MP1u58Q/DBHWFPbQwCZTCeV6Gnq2BpZXdarfTlxOV63U+8Ui/HWF9oXRSk0j0YrwOZxgGsoi4PrYKedgXNAgMBAAGjDTALMAkGA1UdEwQCMAAwDQYJKoZIhvCNQEEBQADggEBACZtvgoYifkTFPF21vid7zrcJP6zT8euKvbVSrlqhw5cusY4tc/hv3ueqGQWaeZ5RoCFnqvBhxUaOkDmz/61bFUr0X675x3Z9UGe2rE2gAEr7XsZw1qvfqg/pezs3fw+WaL720IiHstruX8BMMzp58fBwfPUi5WtYapUX00F4R69RrFLAcLWaWbhna44JOaR5cdJODeLD3ICeEJ9fxcltxQlsgBjfd3JC0MaIvrSACJPVsNCOhsCq0D07YdwOXUoKyNsXckPYMY2aug6jhdfuhP3xh9t4wQmbC3hj/3903MCikxxAgh3dheMYjEyN7DPAAxNIkUtMxXp5Di8GhIbW0xggHrMIB5wIBATCBkDCBiEcMB0GA1UEAxMTTWfjUHJvZ3JhbWFkb3J1cyBDQTEPMA0GA1UECBMGTWFkcm1kMQswCQYDVQQGEwJFUzEtMCsGCSqGSIB3DQEJARYed2VibWFzdGVyQG1hY3Byb2dyYW1hZG9yZXMu3JnMR0wGwYDVQQKExRtYWNwcm9ncmFtYWRvcmVzLm9yZwIBAjAJBgUrDgMCGgUAoIGxMBgGCSqGSIB3DQEJAzELBkgqhkig9w0BBwEwHAYJKoZIhvCNQkFMQ8XDTA0MDIxNDE3MTYyOVowIwYJKoZIhvCNAAQkEMRYEFFKGt+HCW/dPc5FQRicHdhdyu5NxMFIGCSqGSIB3DQEJDzFFMEMwCgYIKoZIhvCNAwcwDgYIKoZIhvCNAwICAgCAMA0GCCqGSIB3DQMCAGFAMAcGBSSOAwIHMA0GCCqGSIB3DQMCAGEoMA0GCSqGSIB3DQEBAQUABIGAf2oYu1XQhqabCpRR6mOu5xIV9rvB1BtPfG39zwpGWPPRGUMiWBHDW1V92ezmqBe/uRI9n+IhJu6Pa3X7Csf+XoNAW8R7gr2KVRQawkWgNZa0alpeQFHMYpQdYrRtUfmR0bhpalLpjEkJcVK46DKt19Zz6vt/Mu2KqFJ6kn6SgU=

-----66D4E2BA031B24CE266CD93846DD555E--

El fichero `fernando.private` contiene la clave privada de Fernando, y el fichero `fernando.cer` su certificado X.509 (y será una copia de un fichero `.pem` de la carpeta `macproCA/certs`). El programa nos ha generado un mensaje S/MIME firmado que podemos también guardar en un fichero con la opción `-out` con el fin de enviarlo con un programa de correo (como el comando `mail`). Obsérvese que las cabeceras no se firman ni se encriptan sino que aparecen antes del contenido del mensaje.

Si tenemos un mensaje S/MIME firmado podemos comprobar la firma con:

```
$ openssl smime -verify -in mensaje.smime -signer fernando.cer -CAfile cacert.cer
```

Hola este es un mensaje para hacer pruebas con S/MIME.

Espero que no te moleste.

Saludos.

Verification successful

Obsérvese que debemos pasarle tanto el certificado del firmante del mensaje `fernando.cer`, como el del CA que avala su validez `cacert.cer`

Si lo que queremos es encriptar un mensaje S/MIME, podemos usar la opción `-encrypt`. Por ejemplo, si Fernando quiere enviar un mensaje encriptado a José, debe disponer primero del certificado de José `jose.cer`, y usarlo para con la clave pública del certificado encriptar el mensaje así:

```
$ openssl smime -encrypt -in mensaje.txt -out mensaje.enc
jose.cer
```

Ahora José puede desencriptar el mensaje con su clave pública así:

```
$ openssl smime -decrypt -inkey jose.private -in mensaje.enc -
recip jose.cer
Enter pass phrase for jose.private:*****
Hola este es un mensaje para hacer pruebas con S/MIME.
```

Espero que no te moleste.

Saludos.

También podemos encriptar y firmar un mensaje. Para ello, primero firmamos el mensaje en un fichero intermedio con los comandos que vimos antes, y luego lo encriptamos usando el comando para encriptar:

```
$ openssl smime -sign -signer fernando.cer -inkey
fernando.private -in mensaje.txt -out mensaje.smime
Enter pass phrase for fernando.private:*****
```

Este comando genera el fichero `mensaje.smime` firmado que ahora vamos a encriptar:

```
$ openssl smime -encrypt -from "Fernando Lopez Hernandez
<fernando@macprogramadores.org>" -subject "Mensaje de prueba"
-to "Jose <jose@macprogramadores.org>" -in mensaje.smime -out
mensaje.smime.enc jose.cer
$ cat mensaje.smime.enc
To: Jose <jose@macprogramadores.org>
From: Fernando Lopez Hernandez <fernando@macprogramadores.org>
Subject: Mensaje de prueba
MIME-Version: 1.0
Content-Disposition: attachment; filename="smime.p7m"
Content-Type: application/x-pkcs7-mime; name="smime.p7m"
Content-Transfer-Encoding: base64
MIILFQYJKoZIhvcNAQcDoIILBjCCCwICAQAxggEsMIIBKAIBADCBkDCBijEcMB
oGA1UEAxMTTWFjUHJvZ3JhbWFkb3JlcyBDQTEPMA0GA1UECBMGTWFkcm1kMQsw
CQYDVQQGEwJFUzEtMCsGCSqGSIb3DQEJARYed2VibWFzdGVyQG1hY3Byb2dyYW
1hZG9yZXMu3JnMR0wGwYDVQQKExRtYWNwcm9ncmFt
.....
.....
```

```
YWRvcmVzLm9yZwIBATANBgkqhkiG
```

Obsérvese que las cabeceras se ponen durante la encriptación, y no durante la firma, porque las cabeceras S/MIME viajan sin encriptar:

Para desencriptar y comprobar la firma seguimos el proceso inverso:

```
$ openssl smime -decrypt -inkey jose.private -in
mensaje.smime.enc -out mensaje.smime -recip jose.cer
Enter pass phrase for jose.private:*****
$ openssl smime -verify -in mensaje.smime -signer fernando.cer
-CAfile cacert.cer
Hola este es un mensaje para hacer pruebas con S/MIME.
```

Espero que no te moleste.

Saludos.

Verification successful

Primero hemos desencriptado `mensaje.smime.enc` para obtener `mensaje.smime`, y luego hemos comprobado la firma de `mensaje.smime`

6.7 Encriptar con RSA

Aunque para encriptar y desencriptar ficheros se recomienda la criptografía simétrica, que es mucho más rápida, también podemos encriptar un fichero con criptografía asimétrica, para ello encriptaremos con la clave pública del receptor y desencriptaremos con la clave privada del receptor.

Por ejemplo, si Fernando quiere enviar un mensaje a José, lo encripta con la clave pública de José haciendo:

```
$ openssl rsautl -encrypt -pubin -inkey jose.public -in
mensaje.txt -out mensaje.txt.rsa
```

La opción `-pubin` es necesaria para indicar que en `-inkey` no va una clave privada, sino una clave pública.

Cuando José reciba el mensaje lo podrá desencriptar con su clave privada haciendo:

```
$ openssl rsautl -decrypt -inkey jose.private -in
mensaje.txt.rsa -out mensaje.txt
Enter pass phrase for jose.private:*****
```

6.8 Crear un cliente y servidor SSL

OpenSSL dispone de los comandos `s_server` y `s_client` que nos permiten emular un cliente y servidor SSL.

Si Fernando quiere ejecutar un servidor SSL con su certificado, puede ejecutar el comando:

```
$ sudo openssl s_server -accept 443 -cert fernando.cer -key  
fernando.private -CAfile macproCA/certs/cacert.cer  
Using default temp DH parameters  
Enter PEM pass phrase:*****  
ACCEPT
```

La opción `-accept` indica el puerto donde ejecutar el servidor (si es un puerto inferior a 1024, como en este ejemplo, debemos lanzar el comando con `sudo` para que tenga permisos de superusuario). Las opciones `-cert` y `-key` indican el certificado público y clave privada a usar por el servidor. Por último, la opción `-CAfile` indica donde está el certificado de la CA en la que confiamos, y que firma los certificados de nuestros clientes.

Ahora José puede lanzar un cliente que se conecte al servidor usando el comando:

```
$ openssl s_client -connect localhost:443 -CAfile  
macproCA/certs/cacert.cer  
CONNECTED(00000003)  
depth=1 /CN=MacProgramadores  
CA/ST=Madrid/C=ES/emailAddress=webmaster@macprogramadores.org/  
O=macprogramadores.org  
verify return:1  
depth=0 /CN=Fernando Lopez  
Hernandez/ST=Madrid/C=ES/emailAddress=fernando@macprogramadore  
s.org/O=macprogramadores.org/OU=Formacion  
verify return:1  
---  
Certificate chain  
0 s:/CN=Fernando Lopez  
Hernandez/ST=Madrid/C=ES/emailAddress=fernando@macprogramadore  
s.org/O=macprogramadores.org/OU=Formacion  
    i:/CN=MacProgramadores  
CA/ST=Madrid/C=ES/emailAddress=webmaster@macprogramadores.org/  
O=macprogramadores.org  
    1 s:/CN=MacProgramadores  
CA/ST=Madrid/C=ES/emailAddress=webmaster@macprogramadores.org/  
O=macprogramadores.org  
        i:/CN=MacProgramadores  
CA/ST=Madrid/C=ES/emailAddress=webmaster@macprogramadores.org/  
O=macprogramadores.org  
---  
Server certificate
```

```

-----BEGIN CERTIFICATE-----
MIIDMjCCAhqgAwIBAgIBAjANBgkqhkiG9w0BAQQFADCBijEcMB0GA1UEAxMTTW
FjUHJvZ3JhbWFkb3J1cyBDQTEPMA0GA1UECBMGTWFkcm1kMQswCQYDVQQGEwJF
UzEtMCsGCSqGSIB3DQEJARYed2VibWFzdGVyQG1hY3Byb2dyYW1hZG9yZXMuB3
JnMR0wGwYDVQQKExRtYWNwcm9ncmFtYWWRvcnVzLm9yZzAeFw0wNDAYMTQwODM3
NDNaFw0wNTAyMTMwODM3NDNaMIGiMSEwHwYDVQQDExhGZXJuYW5kbyBmB3Blei
BIZXJuYW5kZXoxDzANBAgTBk1hZHJpZDELMAkGA1UEBhMCRVMxLDAqBqkq
hkiG9w0BCQEWHWZ1cm5hbmcRvQG1hY3Byb2dyYW1hZG9yZXMuB3JnMR0wGwYDVQ
QKExRtYWNwcm9ncmFtYWWRvcnVzLm9yZzESMBAGA1UECxMJRm9ybWFjaW9uMIGf
MA0GCSqGSIB3DQEBAQUAA4GNADCBiQKBgQCmTa7nDCFhs77nD1vx/1M52HYTCL
SVLdufmMX9euWX0KP0SvScTwmcLXOFxbDcSgVYzqgF+G/5pq0kQ6b4v0zegezD
9bufEPwwR1haW0MAmUwQnlehp6tgaWV3Wq305tlet1PvFIvx1hfaF0UpNI9GK
8DmcYBrKIuD62CnnYFzQIDAQABow0wCzAJBgNVHRMEAjAAMA0GCSqGSIB3DQE
BAUAA4IBAQAmbb4NGIn5ExTxdtb4ne863CT+s0/Hrir21Uq5aoctOLrGOLQv4b
97nqhkfMnmeUaAhZ6rwYcVGjpA5s/+pwXVK9F+u+cd2fVBntqxNoABK+17GcNa
r36oP6XmbN38Plmi+9tCIh7La71/ATDGaefHwcHz1IuVrWGqVF9NBeEevUaxSw
HC1mlm4Z2uOCTmkeXHSTg3iw9yAnhCfx13JbX8UJbIAy33dyQtDGiL60gHCT1b
DQjobAqtA9O2HcDl1KCsjbF3JD2DGNmroOo4XX7oT98YfbeMEJmwt4Y/9/TtzA
opMVwiId3YXjGIxMjewzwAFzYpFLTMV6eQ4vBoSG1t
-----END CERTIFICATE-----
subject=/CN=Fernando Lopez
Hernandez/ST=Madrid/C=ES/emailAddress
=fernando@macprogramadores.org/O=macprogramadores.org/OU=Forma
cion
issuer=/CN=MacProgramadores
CA/ST=Madrid/C=ES/emailAddress=webmaster@macprogramadores.org/
O=macprogramadores.org
---
No client certificate CA names sent
---
SSL handshake has read 2194 bytes and written 276 bytes
---
New, TLSv1/SSLv3, Cipher is DHE-RSA-AES256-SHA
Server public key is 1024 bit
SSL-Session:
    Protocol : TLSv1
    Cipher   : DHE-RSA-AES256-SHA
    Session-ID:
        520A07C37A2EE6CFF99D38C462A0D748A01EF58F60497F89AC5FF617DB85B6
        4E
    Session-ID-ctx:
    Master-Key:
        5E8AE6396FDC76DA064601280C2F8AB24443B7EC268608C3536D62DA0A24F0
        4E05CAB1F2EFF09D3ED3F90FCCC1FE0608
    Key-Ag      : None
    Start Time: 1077176974
    Timeout     : 300 (sec)
    Verify return code: 0 (ok)
Hola estamos conectados
```

José sólo necesita especificar la dirección a la que conectarse con `-connect` y el certificado raíz en el que confía con `-CAfile`, y que avalará la identidad del servidor. A partir de este momento, en la consola del cliente podemos escribir texto, y este texto se enviará al servidor de SSL.

Observe que en pantalla se muestra cuál es la cadena de certificación (confió en Fernando porque confío en el CA de macpro).

6.9 La herramienta ssldump

El comando `ssldump`¹ nos permite analizar el tráfico a través de una conexión SSL. El programa es capaz de descodificar gran parte del handshake, así como de los registros SSL que se transmiten. Lógicamente, para que pueda desencriptar el tráfico encriptado que se está transmitiendo necesita que le pasemos el fichero de clave privada del servidor.

Por ejemplo, si queremos analizar el tráfico entre el cliente y el servidor del apartado anterior podemos hacer:

```
$ ssldump -ilo port 443 and host localhost -d -k
fernando.private
New TCP connection #1: localhost(44763) <-> localhost(443)
1 1 0.0031 (0.0031)  C>S SSLv2 compatible client hello
    Version 3.1
    cipher suites
        TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
        TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
        TLS_RSA_WITH_3DES_EDE_CBC_SHA
        SSL2_CK_3DES
        TLS_DHE_DSS_WITH_RC4_128_SHA
        TLS_RSA_WITH_IDEA_CBC_SHA
        TLS_RSA_WITH_RC4_128_SHA
        TLS_RSA_WITH_RC4_128_MD5
        SSL2_CK_IDEA
        SSL2_CK_RC2
        SSL2_CK_RC4
        SSL2_CK_RC464
        TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
        TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
        TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5
        TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
        TLS_DHE_RSA_WITH_DES_CBC_SHA
        TLS_DHE_DSS_WITH_DES_CBC_SHA
        TLS_RSA_WITH_DES_CBC_SHA
        SSL2_CK_DES
        TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
        TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
        TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
        TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
        TLS_RSA_EXPORT_WITH_RC4_40_MD5
```

¹ Este comando no viene preinstalado en Mac OS X, pero podemos descargarlo del proyecto Fink.

```

SSL2_CK_RC2_EXPORT40
SSL2_CK_RC4_EXPORT40
1 2 0.0174 (0.0143) S>C Handshake
    ServerHello
        Version 3.1
        session_id[32]=
            34 5c 7c 71 92 70 84 54 46 65 6a 7e 06 29 c9 0a
            7c af 56 a0 98 2e c0 21 d4 90 bf 52 cc 14 4a d7
        cipherSuite          TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
        compressionMethod    NULL
1 3 0.0174 (0.0000) S>C Handshake
    Certificate
1 4 0.0174 (0.0000) S>C Handshake
    ServerKeyExchange
1 5 0.0174 (0.0000) S>C Handshake
    ServerHelloDone
1 6 0.0336 (0.0161) C>S Handshake
    ClientKeyExchange
        DiffieHellmanClientPublicKey[64]=
            48 4f 38 b7 0c 50 1c 90 91 f6 18 f7 8e 40 5d a6
            b8 48 14 b9 04 26 b8 5c 3f f3 91 17 d9 db fd 6b
            e5 ed 2d 35 06 4e 48 9b 9e 78 0d 20 44 c5 46 61
            46 2c 80 b8 7f 3f 88 d8 0e 6f 92 f2 c9 01 14 1c
1 7 0.0336 (0.0000) C>S ChangeCipherSpec
1 8 0.0336 (0.0000) C>S Handshake
1 9 0.0391 (0.0054) S>C ChangeCipherSpec
1 10 0.0391 (0.0000) S>C Handshake
1 11 1.2460 (1.2069) C>S application_data
1 12 1.2460 (0.0000) C>S ChangeCipherSpec
1 13 1.9037 (0.6577) C>S Alert
1 1.9038 (0.0000) C>S TCP FIN
1 1.9066 (0.0027) S>C TCP FIN

```

En la salida se ve el tráfico de registros SSL entre en cliente y el servidor. Cada registro SSL aparece numerado con dos números que indican el número de conexión y el número de registro. Por ejemplo 1 7 indica que se trata del registro SSL séptimo de la primera conexión. A los números de registro les siguen dos timestamp, el primero indica el tiempo transcurrido desde el inicio de la conexión, y el segundo el tiempo transcurrido desde el anterior registro SSL. Además se indica si el tráfico es del cliente al servidor c > s, o del servidor al cliente s > c. Por último se indica el tipo de registro (Handshake, ChangeCipherSpec, ChangeCipherSpec, Alert).

Vemos que, en el primer registro el cliente presenta los cifradores disponibles, y en los registros 2 al 10 se realiza el handshake. En el registro 11 el cliente envía datos al servidor, en el registro 13 el cliente manda la notificación de cierre, y finalmente se cierra el socket.

7 SSL en Java

7.1 JSSE

JSSE (Java Secure Socket Extensions) proporciona un API estándar para acceder a conexiones SSL y TLS.

Hasta Java 1.3 JSSE se distribuía como una extensión estándar pero, a partir de Java 1.4, con la eliminación de las restricciones de exportación de los EEUU, JSSE se empezó a distribuir junto a la máquina virtual.

Las clases de JSSE se encuentran en el paquete `javax.net.ssl.*`, y la Figura 6.20 muestra la relación de estas clases tanto entre ellas como con otros paquetes.

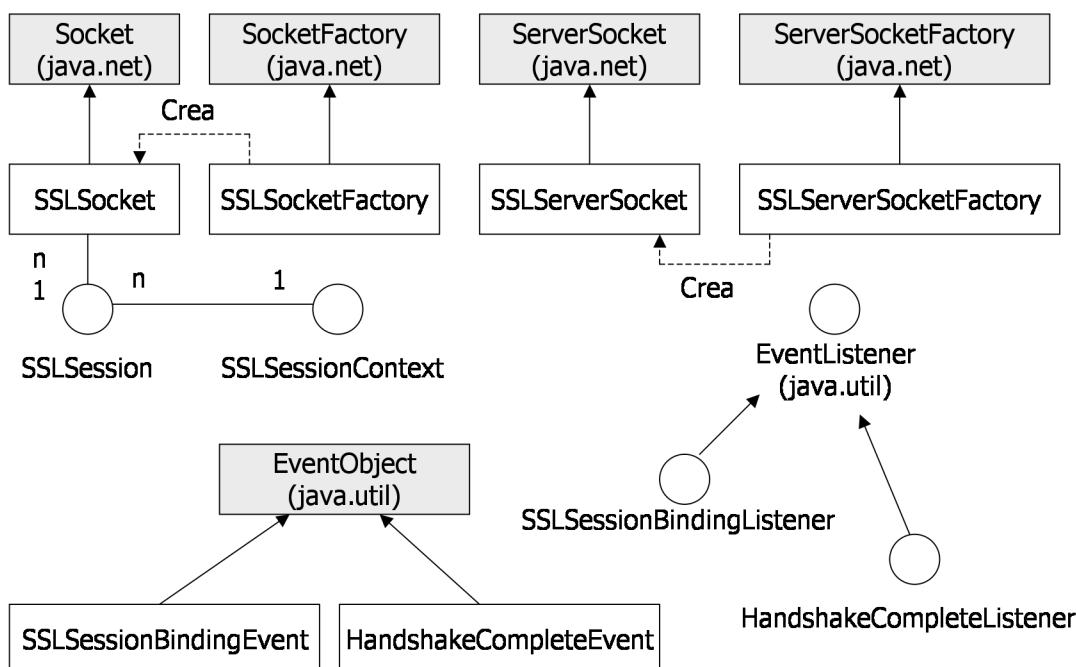


Figura 6.20 Principales clases del JSSE

`SSLSocket` y `SSLServerSocket` son derivadas de `Socket` y `ServerSocket` respectivamente que representan sockets seguros.

`SSLSession` y `SSLSessionContext` se usan para la gestión de sesiones SSL.

Además se incorporan dos nuevos eventos:

- `HandshakeCompleteEvent` que se produce cuando se acaba la fase de handshake y puede empezar la comunicación.

- `SSLSessionBindingEvent` que se produce cuando un socket seguro se une o abandona una sesión.

7.2 Keystores y truststores

JSSE introduce un nuevo tipo de keystore llamado **truststore**, que es una base de datos con los certificados de las principales CA.

Este fichero se encuentra en `$JRE_HOME/lib/security/cacerts`, y como vimos en el apartado 2.2 se puede consultar con el comando:

```
$ keytool -keystore $JRE_HOME/lib/security/cacerts -list
```

Usando el password "changeit".

Es importante tener clara la diferencia entre lo que es un keystore (que vimos en el apartado 2.2) y un truststore:

- El truststore se usa para comprobar la validez de los certificados que recibimos, y normalmente sólo lo usa el cliente, aunque lo puede usar el servidor si el cliente se está identificando con un certificado para ver si este certificado es válido.
- El keystore es donde un programa almacena sus certificados que usa para identificarse en el establecimiento de una conexión SSL. Normalmente sólo lo usa el servidor, aunque puede usarlo el cliente si quiere identificarse usando un certificado.

Por defecto JSSE usa el truststore `$JRE_HOME/lib/security/cacerts` aunque podemos cambiar el truststore por defecto modificando las propiedades del sistema `javax.net.ssl.trustStore` y `javax.net.ssl.trustStorePassword` para indicar qué fichero usar como truststore (p.e. al valor `$HOME/.keystore`).

7.3 Las socket factories

JSSE introdujo el paquete `javax.net.*` en el cual tiene dos clases:

```
javax.net.ServerSocketFactory  
javax.net.SocketFactory
```

Estas clases no tienen constructor, sino que tienen los métodos estáticos:

```
static  
ServerSocketFactory <ServerSocketFactory> getDefault()
```

```
static SocketFactory <SocketFactory> getDefault()
```

Y una vez que tenemos creados estos objetos podemos crear un `ServerSocket` o un `Socket` usando:

```
ServerSocket  
<ServerSocketFactory> createServerSocket(int port)  
Socket  
<SocketFactory> createSocket(InetAddress host, int port)
```

En Java 1.4 se introdujo este paquete como parte de la máquina virtual, y se cambiaron los constructores de `ServerSocket` y `Socket` para que llamaran a estos métodos.

La razón de este cambio fué la homogeneización de las API de programación de sockets. Lo que se pretendía hacer es que para crear tanto socket normales como sockets seguros se usaran métodos factory. La Figura 6.21 muestra la relación jerárquica entre las clases que representan socket factories normales y socket factories seguros.

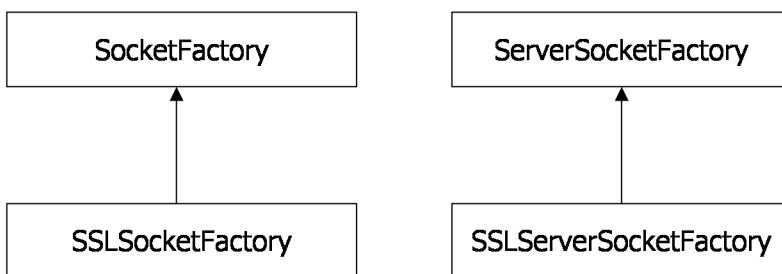


Figura 6.21 Representación de los socket factories en Java

De forma que estos objetos se crean también con:

```
static  
ServerSocketFactory <SSLServerSocketFactory> getDefault()  
static SocketFactory <SSLSocketFactory> getDefault()
```

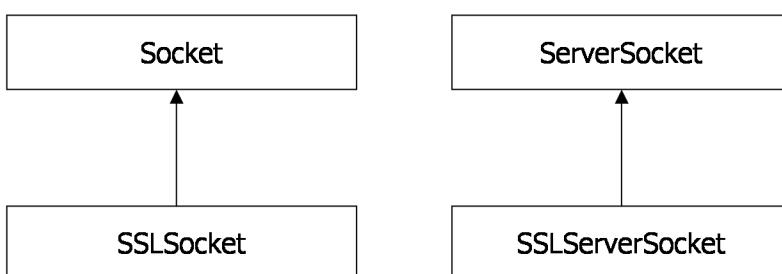


Figura 6.22 Representación de los socket seguros en Java

Y después igual que antes podemos usar:

```
ServerSocket <ServerSocketFactory> createServerSocket(
    int port)
Socket <SocketFactory> createSocket(InetAddress host
    , int port)
```

Para obtener referencias a objetos de clases derivadas del tipo `SSLServerSocket` y `SSLSocket` (véase Figura 6.22).

Luego ahora la forma de obtener un `SSLServerSocket` es:

```
ServerSocketFactory ssf =
    SSLSocketFactory.getDefault();
ServerSocket ss = ssf.createServerSocket(9096);
```

Y la de crear un `SSLSocket` es:

```
SocketFactory sf = SSLSocketFactory.getDefault();
Socket s = sf.createSocket(servidor, 9096);
```

7.4 Cliente y servidor SSL

En este apartado vamos a implementar un ejemplo de programa cliente y servidor Java que se comunican a través de un socket seguro.

Al servidor lo llamaremos `ServidorSSL`, y lo único que hace es recibir clientes y devolverles un mensaje de saludo por el socket seguro. Por su parte el cliente, al que llamaremos `ClienteSSL`, lo único que hará será conectarse al servidor, recibir el saludo y mostrárselo al usuario.

7.4.1 La clase `SSLServerSocket`

Esta clase, además de los métodos heredados de `ServerSocket` tiene métodos para consultar y modificar los cifradores y protocolos disponibles para la conexión:

```
String[] <SSLServerSocket> getEnabledCipherSuites()
String[] <SSLServerSocket> getEnabledProtocols()
void <SSLServerSocket> setEnabledCipherSuites(
    String[] suites)
void <SSLServerSocket> setEnabledProtocols(
    String[] suites)
```

También tiene métodos para pedir que el cliente se autentifique. En concreto, en el apartado 7.6 veremos cómo hacer que se autentifique el cliente con este método.

El Listado 6.4 muestra un pequeño servidor que acepta clientes SSL y les devuelve el mensaje: "Hola, soy el servidor".

```
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;

public class ServidorSSL extends Thread
{
    // La instancia de esta clase tiene un socket asociado
    Socket s;
    // Se incrementa por cada cliente que se conecta
    static int nCliente = 1;

    public static void main(String[] args ) throws Exception
    {
        // Instala un server socket en el puerto 9096
        ServerSocketFactory ssf =
            SSLSocketFactory.getDefault();
        ServerSocket ss = ssf.createServerSocket(9096);
        System.out.println("Servidor lanzado");
        // Se repite cada vez que se conecta un cliente
        while (true)
        {
            new ServidorSSL(ss.accept()).start();
        }
    }

    public ServidorSSL(Socket s)
    {
        this.s = s;
    }

    public void run()
    {
        try{
            System.out.println("Cliente " + nCliente
                + " conectado");
            PrintWriter pw =
                new PrintWriter(s.getOutputStream());
            pw.println("Hola cliente numero "+ nCliente
                + " soy el servidor");
            pw.close();
            s.close();
            System.out.println("Cliente " + nCliente++
                + " desconectado");
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
    }
}
```

Listado 6.4: Servidor SSL

7.4.2 La clase SSLSocket

Esta clase, además de métodos heredados de `Socket`, también tiene métodos para leer y modificar las suites y protocolos disponibles, así como para que el cliente se autentifique, como veremos en el apartado 7.6.

Conviene destacar que el método:

```
void <SSLSocket> startHandshake()
```

sólo se usa cuando hemos cambiado los protocolos disponibles para hacer un rehandshake, no para empezar el handshake inicial.

El Listado 6.5 muestra un cliente que conecta al servidor y lee el mensaje que le llega.

```
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;

public class ClienteSSL
{
    public static void main(String[] args ) throws Exception
    {
        if (args.length!=2)
        {
            System.out.println("Indique <host> <puerto>");
            return;
        }
        SocketFactory sf = SSLSocketFactory.getDefault();
        Socket s = sf.createSocket(
            args[0],Integer.parseInt(args[1]));
        BufferedReader br = new BufferedReader(
            new InputStreamReader(s.getInputStream()));
        System.out.println(br.readLine());
        br.close();
        s.close();
    }
}
```

Listado 6.5: Cliente SSL

Antes de poder ejecutar el servidor SSL, éste necesita disponer de un certificado self-signed que mostrar a los clientes que se conecten a él. La forma más sencilla de proporcionárselo es crear un keystore con una sola entrada en la que se almacena el certificado del servidor.

Para ello podemos usar:

```
$ keytool -genkey -keystore macprokeystore
```

Y ahora podemos indicar el keystore y password a usar por nuestro programa Java en las propiedades del sistema:

```
$ java -Djavax.net.ssl.keystore=macprokeystore  
-Djavax.net.ssl.keyStorePassword=sesamo ServidorSSL
```

De esta forma `SSLSocketFactory` coge el primer alias del keystore (en orden alfabético) y lo usa como certificado.

Para ejecutar el cliente primero necesitamos colocar el certificado en un keystore del usuario, para lograrlo tenemos dos opciones: O bien ponerlo en el fichero `cacerts` del usuario, o bien ponerlo en un keystore aparte que indicamos con las propiedades del sistema `javax.net.ssl.trustStore` y `javax.net.ssl.trustStorePassword`.

Vamos a optar por la segunda opción, para lo cual primero exportaremos el certificado de `macprokeystore` a un fichero `macpro.cer`:

```
$ keytool -export -alias mykey -file macpro.cer  
-keystore macprokeystore
```

El alias `mykey` es el alias que crea por defecto `keytool` cuando no indicamos alias (como valor al crear el certificado self-signed del servidor más arriba).

Ahora importamos el certificado a un fichero `usuariotruststore` con:

```
$ keytool -import -file macpro.cer -keystore usuariotruststore
```

Y ya podemos ejecutar el cliente con:

```
$ java -Djavax.net.ssl.keystore=usuariotruststore  
-Djavax.net.ssl.keyStorePassword=sesamo  
ClienteSSL localhost 9096
```

7.5 Sesiones

Como vimos en el apartado 5.4, el handshake de una conexión SSL se realiza sólo una vez al principio, y luego se crea una sesión que intenta reutilizar las conexiones.

El habilitar o deshabilitar el uso de sesiones en Java se gestiona con los métodos del servidor:

```
boolean <SSLServerSocket> getEnabledSessionCreation()
```

```
void <SSLServerSocket> setEnabledSessionCreation(boolean b)
```

Después, tanto el cliente como el servidor pueden acceder a la información de sesión usando el método:

```
SSLSession <SSLSocket> getSession()
```

En la sesión se pueden poner atributos que identifiquen al cliente en futuras conexiones (al estilo de las cookies) usando:

```
void <SSLSession> putValue(String name, Object value)
Object <SSLSession> getValue(String name)
```

Para invalidar una sesión podemos usar el método:

```
void <SSLSession> invalidate()
```

7.6 Identificación del cliente

Para que el servidor indique que quiere identificación por parte del usuario, tiene los métodos:

```
void <SSLSocketServer> setNeedClientAuth(boolean b)
boolean <SSLSocketServer> getNeedClientAuth()
```

También existen otros métodos que sirven para pedir al cliente identificarse, pero no le obligan (no falla la conexión si no lo hace):

```
void <SSLSocketServer> setWantClientAuth(boolean b)
boolean <SSLSocketServer> getWantClientAuth()
```

También la petición de identificación se puede llevar adelante más tarde (cuando el servidor tiene el objeto `SSLSocket`) usando el método:

```
void <SSLSocket> setNeedClientAuth(boolean b)
```

y llamando luego a:

```
void <SSLSocket> startHandshake()
```

En el Listado 6.4 y Listado 6.5 se muestra un ejemplo de cliente y servidor con identificación mutua, para ello tanto cliente como servidor necesitan tener un certificado en su keystore que presentan al otro, y a la vez necesitan el certificado del otro en su truststore.

Para crear los certificados de cliente y servidor podemos usar:

```
$ keytool -genkey -alias servidor -keystore servidorkeystore
```

```
$ keystore -genkey -alias cliente -keystore clientekeystore
```

Ahora podemos usar el keystore del uno como truststore del otro, y lanzarlos así:

```
$ java -Djavax.net.ssl.keyStore=servidorkeystore
       -Djavax.net.ssl.keyStorePassword=sesamo
       -Djavax.net.ssl.trustStore=clientekeystore
       -Djavax.net.ssl.trustStorePassword=sesamo
ServidorAutentificadorSSL

$ java -Djavax.net.ssl.keyStore=clientekeystore
       -Djavax.net.ssl.keyStorePassword=sesamo
       -Djavax.net.ssl.trustStore=servidorkeystore
       -Djavax.net.ssl.trustStorePassword=sesamo
ClienteSSL localhost 9096
```

```
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;

public class ServidorAutentificadorSSL extends Thread
{
    // La instancia de esta clase tiene un socket asociado
    Socket s;
    // Se incrementa por cada cliente que se conecta
    static int nCliente = 1;

    public static void main(String[] args ) throws Exception
    {
        ServerSocketFactory ssf =
            SSLServerSocketFactory.getDefault();
        SSLServerSocket ss = (SSLServerSocket)
            ssf.createServerSocket(9096);
        ss.setNeedClientAuth(true);
        System.out.println("Servidor lanzado");
        // Se repite cada vez que se conecta un cliente
        while (true)
        {
            new ServidorAutentificadorSSL(
                ss.accept()).start();
        }
    }

    public ServidorAutentificadorSSL(Socket s)
    {
        this.s = s;
    }

    public void run()
    {
        try{
            System.out.println("Cliente " + nCliente
```

```
        + "conectado");
PrintWriter pw =
        new PrintWriter(s.getOutputStream());
pw.println("Hola cliente numero "+ nCliente
        + " soy el servidor");
pw.close();
s.close();
System.out.println("Ciente " + nCliente++
        + " desconectado");
}
catch(Exception ex) {
    ex.printStackTrace();
}
}
```

Listado 6.6: Servidor SSL que requiere identificación de cliente

7.7 Depurar el handshake

En caso de que estemos teniendo problemas con el handshake de SSL es muy recomendable activar la propiedad del sistema:

```
javax.net.debug="ssl:handshake"
```

La cual se puede poner por ejemplo al ejecutar el cliente o el servidor:

```
$ java -Djavax.net.ssl.keyStore=servidorkeystore
      -Djavax.net.ssl.keyStorePassword=sesamo
      -Djavax.net.ssl.trustStore=clientekeystore
      -Djavax.net.ssl.trustStorePassword=sesamo
      -Djavax.net.debug="ssl:handshake" ServidorSSL

keyStore is : servidorkeystore
keyStore type is : jks
init keystore
init keymanager of type SunX509
***  

found key for : servidor
chain [0] = [
[  

  Version: V1
  Subject: CN=Servidor, OU=dfs, O=sd, L=sdf, C=Unknown
  Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

  Key: Sun DSA Public Key
  Parameters:DSA
  p: fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400
c31e3f80 b6512669 455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55
6cd7813b 801d346f f26660b7 6b9950a5 a49f9fe8 047b1022 c24fbba9
d7feb7c6 1bf83b57 e7c6a8a6 150f04fb 83f6d3c5 1ec30235 54135a16
9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
  q: 9760508f 15230bcc b292b982 a2eb840b f0581cf5
```

```

g:      f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82
f9574c0b 3d078267 5159578e bad4594f e6710710 8180b449 167123e8
4c281613 b7cf0932 8cc8a6e1 3c167a8b 547c8d28 e0a3ae1e 2bb3a675
916ea37f 0bfa2135 62f1fb62 7a01243b cca4f1be a8519089 a883dfel
5ae59f06 928b665e 807b5525 64014c3b fecf492a
y: 395f02ba 2c32e41a d0a6bc6f a140ba80 74d4c99f 24bede1a
b766830a 4b96ed9f 01389097 9e63030c 33e1792b 3a62252b a61d55e4
1453e1c9 826f801b c490291b b8681b86 619c8da7 4c854b46 94025279
8d682887 bb3da943 acf69f35 dd30cb0b 05597095 24f59902
e52b1239 7b4b508b b602d370 79a8850f 6095f5fd 7ce8f100

Validity: [From: Wed Feb 18 08:43:52 CET 2004,
           To: Tue May 18 09:43:52 CEST 2004]
Issuer: CN=Servidor, OU=dfs, O=sd, L=sdf, C=Unknown
SerialNumber: [        403317b8]
]
Algorithm: [SHA1withDSA]
Signature:
0000: 30 2C 02 14 10 C1 A2 9E     AB F9 E4 E6 83 56 7B 18
0,.....V..
0010: 8A EE A5 21 95 BF E1 FE     02 14 3B C8 9D A7 C4 C4
...!.....;.....
0020: 01 CE 4A 8B 07 FC B5 75     61 20 DC 4A 45 EA
..J....ua .JE.

]
***  

trustStore is: clientekeystore  

trustStore type is : jks  

init truststore  

adding as trusted cert:  

Subject: CN=Cliente, OU=fds, O=ds, L=sd, ST=sd, C=es  

Issuer: CN=Cliente, OU=fds, O=ds, L=sd, ST=sd, C=es  

Algorithm: DSA; Serial number: 0x403317ca  

Valid from Wed Feb 18 08:44:10 CET 2004 until Tue May 18  

09:44:10 CEST 2004

init context  

trigger seeding of SecureRandom  

done seeding SecureRandom  

Servidor lanzado  

matching alias: servidor  

.....  

.....
```

Como vemos esta opción hace que se imprima una descripción detallada de todo el handshake que se ejecuta.

8 HTTPS

8.1 Introducción

HTTPS fue el primer protocolo de aplicación que circuló sobre una conexión SSL.

El protocolo está documentado en la RFC 2818, y realmente no es un nuevo protocolo, sino el protocolo HTTP circulando encima de una conexión SSL.

En la RFC 2818 se comentan algunas peculiaridades y problemas de seguridad que vamos a analizar en este apartado.

IANA ha asignado el puerto 443 al protocolo HTTPS, que es el que se suele usar.

Básicamente la petición de una página web segura al servidor tiene tres partes:

1. Handshaking de la conexión SSL.
2. Se envía una petición al servidor, y éste devuelve la página de respuesta.
3. Se cierra la conexión con `close_notify`, y luego se cierra el socket TCP.

Es importante que el cierre de la conexión lo realice SSL y no TCP, ya que si no la página podría llegar incompleta, y se debe considerar como inválida.

8.2 Proxies

Los proxies de HTTP son servidores intermedios que reciben una petición de un browser para que les obtenga una página. La única diferencia que hay cuando un browser accede a un proxy, en vez de a un servidor web directamente, es que en lugar de que el browser les envíe una petición de la forma:

```
GET /index.html HTTP/1.1
```

Reciben una petición de la forma:

```
GET http://www.macprogramadores.org/index.html HTTP/1.1
```

Esto se hace para que el proxy pueda saber a qué servidor conectarse.

Un problema que tiene HTTPS es que no permite la existencia de proxies, ya que la comunicación entre el cliente y el servidor web está encriptada y protegida frente a cualquier man in the middle attack. Para solucionar este problema, la RFC 2817 propone un nuevo comando de HTTP:

```
CONNECT www.macprogramadores.org:443 HTTP/1.1
```

Que lo que hace es pedir al proxy que habrá un socket con ese servidor y puerto, y que lo que nosotros le mandemos por el socket, él lo envíe por otro socket al servidor web sin alterar el tráfico, La Figura 6.23 muestra gráficamente las dos conexiones implicadas en este proceso.

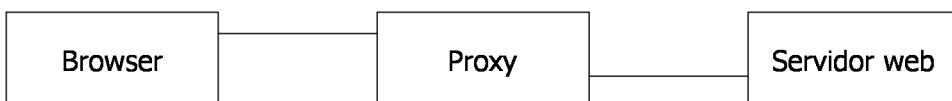


Figura 6.23 Conexión de un browser con un servidor web a través de un proxy

Este mecanismo, en principio soluciona el problema del acceso a servidores web seguros, pero abre un agujero por el que la comunicación que tenga el cliente con el servidor web se escapa del control del proxy, con lo que es decisión del administrador del proxy el habilitar esta opción.

8.3 Negociación de las conexiones HTTPS

En el apartado 5.1 vimos que había dos formas de que un protocolo de aplicación circule sobre SSL: Puertos separados y negociación de la conexión. Aunque normalmente en HTTPS se usan puertos separados, también se puede negociar la conexión.

Si un browser quiere negociar la conexión puede mandar la petición:

```
GET / HTTP/1.1
Host:www.macprogramadores.org
Upgrade: TLS/1.0
Connection: Upgrade
```

Si el servidor puede abrir una conexión segura con el cliente, le devuelve:

```
HTTP/1.1 101 Switching protocols
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
```

Y a partir de ahí inician una comunicación SSL por el puerto 80.

También es posible que el servidor sea el que pida al cliente que la conexión sea SSL para lo cual respondería:

```
HTTP/1.1 426 Upgrade required
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
```

Y a partir de ese punto (si el cliente lo soporta) se inicia un establecimiento de conexión SSL.

Esta técnica actualmente no se está usando mucho porque presenta problemas cuando el cliente está accediendo a través de un proxy que no se han solucionado.

8.4 Problemas de seguridad

En los siguientes apartados vamos a comentar algunos problemas de seguridad que se pueden presentar en HTTPS, con lo que es muy conveniente tenerlos en cuenta.

8.4.1 Comprobación de los certificados

Cuando un browser recibe un certificado, éste debe mirar el campo CommonName (CN) del DN para ver cuál es el nombre de dominio del servidor web.

Después debe realizar una búsqueda inversa en el DNS para asegurarse de que la IP a la que está conectado tiene ese nombre. Algunas veces el campo CN del DN tiene la forma: *.macprogramadores.org, lo cual indica que el certificado es válido para cualquier servidor que esté bajo ese dominio.

8.4.2 Comportamiento ante fallos

Cuando un browser detecta que el certificado del servidor presenta algún problema, éste no cierra la conexión SSL, sino que muestra el problema al usuario y le pregunta si desea seguir adelante.

8.4.3 Ataque man in the middle con un proxy

En caso de que el proxy permita acceder a conexiones seguras a través de HTTPS, los administradores de la red también pueden montar un ataque contra los usuarios de su red.

El protocolo sería el siguiente:

1. Los administradores de la red (p.e. cybercafé) instalan en los browsers un certificado que los identifica a ellos como una CA legítima.
2. Cuando un browser manda una petición `CONNECT` al proxy, éste no le abre un socket para que hable con el servidor web, sino que es el proxy quien establece una conexión segura con el browser.
3. El proxy devuelve al browser la respuesta que le da el servidor web seguro de forma que el usuario ve su página como normalmente, sólo que ahora el proxy está viendo el texto plano de toda la comunicación.

¿Cómo hace el proxy para que el browser considere como válido el certificado que le envía el proxy sea cual sea el servidor web al que se conecte el usuario?

Recuérdese que el browser comprobaba el campo `CN` del `DN`, pero el proxy puede devolver un certificado con este campo valiendo "*", con lo que el browser lo considera válido para cualquier nombre.

Para evitar este ataque los browser han empezado a rechazar los nombres de la forma "*", pero todavía se puede usar otro `CN` parecido: "*.*" o "*.*.com".

8.4.4 Enlaces inseguros

Otro posible ataque activo consiste en modificar la URL a la que accede un enlace de una página http insegura para que apunte a otro sitio donde está montado un caballo de troya.

Este ataque sólo puede ser detectado por el usuario si comprueba la URL a la que se está conectando, pero este nombre es fácil de camuflar, por ejemplo registrando el nombre `www.uno-e-secure.com`, u otro parecido.

8.4.5 Cabecera Referrer

En 1997 Daniel Klein descubrió por casualidad que la cabecera `Referrer` se enviaba de un servidor seguro a uno inseguro cuando el usuario hacía este cambio (p.e. pinchar un banner de un servidor seguro).

En principio esto no debería de importar siempre que la última petición no sea un GET parametrizado, donde podría encontrarse información confidencial. Para evitar este problema se recomienda que todos los parámetros se envíen a un servidor seguro usando POST.

8.4.6 Virtual host

Es muy típico que un servidor web aloje varios nombres de dominio a lo que se suele llamar **virtual host**. En este caso, cuando el servidor recibe una petición, usa la cabecera `Host:` para indicar a qué sitio web se está queriendo conectar el browser:

```
GET / HTTP/1.1  
Host:www.macprogramadores.org
```

Esto permite compartir una IP entre muchos dominios.

Por desgracia los servidores web seguros no funcionan bien con virtual hosting ya que cuando un browser se conecta no sabe qué certificado es el que le tiene que entregar. A este problema se le suelen aplicar dos soluciones distintas:

La primera es asignar varias direcciones IP a una sola interfaz de red del servidor web, tal como muestra la Figura 6.24, lo cual permite reutilizar el servidor web, aunque no las direcciones IP.



Figura 6.24 Servidor web con varias direcciones IP

La segunda es emitir certificados que usen comodines "*", o que tengan varios nombres de dominios separados por coma (`uno.com, dos.com, tres.com`). Esta solución permite reutilizar una sola IP, pero tiene el problema de que si queremos añadir un solo sitio web al servidor tenemos que solicitar otro certificado.

9 HTTPS en Java

9.1 Cliente y servidor HTTPS

Vamos a escribir un cliente y un servidor HTTPS.

Para hacer el cliente:

1. Creamos un objeto `URL` que represente la URL a la que queremos acceder.
1. Obtenemos una `URLConnection` a esa URL usando:

```
URLConnection <URL> openConnection()
```

2. Obtenemos un `InputStream` que nos permite leer de esa conexión con:

```
InputStream <URLConnection> getInputStream()
```

Análogamente podríamos haber escrito en esa conexión (p.e. para hacer un POST) usando:

```
OutputStream <URLConnection> getOutputStream()
```

El Listado 6.7 muestra el código fuente del cliente. Conviene aclarar que hasta Java 1.3 había que registrar un protocol handler para HTTPS fijando la propiedad del sistema:

```
java.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
```

A partir de Java 1.4 este handler ya está registrado.

```
import java.io.*;  
import java.net.*;  
  
public class ClienteHTTPS  
{  
    public static void main(String[] args) throws Exception  
    {  
        if (args.length!=1)  
        {  
            System.out.println(  
                "Use javaClienteHTTPS <URL>");  
            return;  
        }  
    }  
}
```

```

URL url = new URL(args[0]);
URLConnection uc = url.openConnection();
BufferedReader br = new BufferedReader(
    new InputStreamReader(uc.getInputStream()));
String linea = br.readLine();
while (linea!=null)
{
    System.out.println(linea);
    linea = br.readLine();
}
}
}

```

Listado 6.7: Cliente HTTPS

Para hacer el servidor, como muestra el Listado 6.8, no usamos ninguna clase nueva sino que nos limitamos a obtener un `SSLSocketFactory` que atiende a las peticiones.

```

import java.io.*;
import java.net.*;
import javax.net.ssl.*;

public class ServidorHTTPS extends Thread
{
    private Socket s;

    public static void main(String[] args) throws Exception
    {
        SSLSocketFactory ssf =
            (SSLSocketFactory)
            SSLSocketFactory.getDefault();
        ServerSocket ss = ssf.createServerSocket(8443
            ,50, InetAddress.getByName("localhost"));
        System.out.println("Servidor lanzado");
        while (true)
            new ServidorHTTPS(ss.accept()).start();
    }

    public ServidorHTTPS(Socket s)
    {
        this.s = s;
    }

    public void run()
    {
        try{
            // Obtiene streams de entrada y salida
            PrintWriter pw =
                new PrintWriter(s.getOutputStream());
            BufferedReader br = new BufferedReader(
                new InputStreamReader(s.getInputStream()));

            // Imprime la petición que llega

```

```

        String linea = br.readLine();
        while (linea!=null && !linea.equals(""))
        {
            System.out.println(linea);
            linea = br.readLine();
        }

        // Escribe la petición de respuesta
        // (HTTPS requiere un Content-Length:)
        StringBuffer buffer = new StringBuffer();
        buffer.append("<HTML>\n");
        buffer.append("<HEAD><TITLE>Mi servidor " +
                    "HTTPS</TITLE></HEAD>\n");
        buffer.append("<BODY>\n");
        buffer.append("<H1>Este es es un servidor " +
                    "HTTPS</H1>\n");
        buffer.append("</BODY>\n");
        buffer.append("<HTML>\n");
        String pagina = buffer.toString();
        pw.println("HTTP/1.0 200 OK");
        pw.println("Content-Type: text/html");
        pw.println("Content-Length:"+pagina.length());
        pw.println();
        pw.println(pagina);
        pw.flush();

        // cerramos los streams
        br.close();
        pw.close();
        s.close();
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}
}

```

Listado 6.8: Servidor HTTPS

Para ejecutar el servidor haríamos:

```
$ java -Djavax.net.ssl.keyStore=macprokeystore
      -Djavax.net.ssl.keyStorePassword=sesamo
      ServidorHTTPS
```

Y para lanzar el cliente:

```
$ java -Djavax.net.ssl.trustStore=macprokeystore
      -Djavax.net.ssl.trustStorePassword=sesamo
      ClienteHTTPS https://localhost:8443
```

Cuando ejecutemos el cliente posiblemente nos de un error diciendo que el CN del DN del certificado debería de ser localhost. Para solucionarlo

debemos volver a crearnos el certificado de `macprokeystore` poniendo este CN.

Si probamos a conectarnos al servidor desde un browser, éste nos dirá que no reconoce el certificado, y nos dará tres opciones:

- Rechazar la conexión
- Aceptar el certificado por una sesión
- Aceptar el certificado permanentemente

9.2 No comprobar el CN

Realmente el objeto que nos devuelve:

```
URLConnection <URL>
openConnection()
```

Es un objeto de tipo `HttpsURLConnection` (véase Figura 6.25), y podemos hacer un casting a este tipo.

Esta clase tiene los métodos:

```
void <HttpsURLConnection>
    setHostnameVerifier(HostnameVerifier v)

static void <HttpsURLConnection>
   setDefaultHostnameVerifier(HostnameVerifier v)
```

Que nos permiten fijar un objeto que implemente la interfaz `HostnameVerifier`, la cual tiene un solo método:

```
boolean <HostnameVerifier> verify(String hostname
    , SSLSession session)
```

El cual sirve como método de callback para preguntar si aceptamos un CN (Common Name). De esta forma podemos evitar el problema de que no coincida el CN con el nombre DNS del servidor HTTPS (p.e. porque estamos accediendo a través de su dirección IP).

El método estático nos permite fijar el `HostnameVerifier` para todo el sistema, mientras que el método de instancia nos permite fijar para una determinada conexión `HttpsURLConnection`.

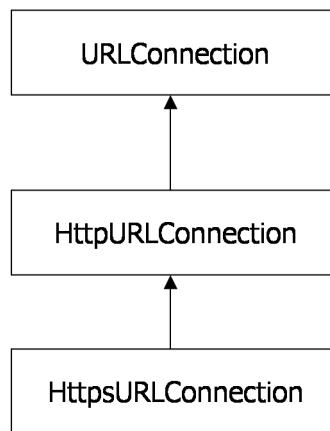


Figura 6.25 Clases base de `HttpsURLConnection`

Por ejemplo, podemos implementar la interfaz `HttpsHostnameVerifier` en la clase `ClienteHTTPS` y modificar la conexión así:

```
URLConnection uc = url.openConnection();
((HttpsURLConnection) uc).setHostnameVerifier(this);
```

Además tenemos el método:

```
static void <HttpsURLConnection>
getDefaultHostnameVerifier()
```

Que nos permite obtener el `HostnameVerifier` por defecto.

9.3 Propiedades de HTTPS

Además de la propiedad del sistema `java.protocol.handler.pkgs` que comentamos en el apartado 9.1, existen otras propiedades que se resumen en la Tabla 6.5.

Propiedad	Descripción
<code>https.proxyHost</code>	Indican donde hay un proxy HTTPS
<code>https.proxyPort</code>	
<code>http.nonProxyHost</code>	Nos permiten indicar una lista de direcciones de servidores web de la LAN para los que no queremos usar proxy. La lista se da separada por . Por ejemplo: <code>mail.comercio.org www.comercio.org</code> Esta propiedad vale tanto para HTTP como para HTTPS

Tabla 6.5: Propiedades de HTTPS

10 SMTP sobre TLS

10.1 Introducción

El protocolo original de SMTP está descrito en la RFC 821, y supondremos que el lector lo conoce. En la RFC 2487 se documenta una extensión para poder transmitir los mensajes de forma segura usando TLS.

SMTP sobre TLS es un ejemplo de un establecimiento de la conexión por negociación de la conexión (véase el apartado 5.1).

10.2 Protocolo

Para saber si el servidor al que nos vamos a conectar soporta SMTP sobre TLS debemos empezar la comunicación con el `HELO` extendido que es `EHLO`:

```
EHLO macbox.macprogramadores.org
250-Hello macbox.macprogramadores.org [213.96.114.181]
250-SIZE 52428800
250-PIPELINING
250-AUTH PLAIN LOGIN
250-STARTTLS
250 HELP
```

Si lo soporta (nos devuelve la opción `STARTTLS`) mandamos el comando:

```
STARTTLS
220 TLS go ahead
```

Y a partir de ahí comienza el handshaking de TLS. Una vez realizado el handshaking, el protocolo continua como normalmente, pero sobre una comunicación segura.

En este protocolo no es necesario que la conexión se cierre con un `close-notify` ya que el comando `QUIT` es el que realmente cierra la transacción. Hasta que no se devuelve un `250` al comando `QUIT` no se debe suponer que el mensaje ha sido aceptado por el servidor.

Un servidor de correo puede obligar a que nos conectemos a él usando SMTP sobre TLS. Para ello, al conectarnos nos devuelve el código de error:

```
530 Must issue a STARTTLS command first
```

Los mensajes que han sido enviados con este protocolo suelen indicarlo en la cabecera `Received:` así:

Received: from macbox.macprogramadores.org by
mail.macprogramadores.org with DES-CBC3-SHA encrypted SMTP

10.3 Tipos de identificación

Desgraciadamente SMTP no se tiene por qué usar en todos los nodos por los que pasa un mensaje de correo, con lo que su utilidad para la confidencialidad se ve reducida.

SMTP se puede usar con dos fines:

- **Confidencialidad de usuario.** Lo usa un servidor SMTP para dar seguridad a los mensajes de sus usuarios. Obsérvese que SMTP sobre TLS no autentifica a sus usuarios, sólo les da un canal seguro, de hecho la RFC 2487 prohíbe usar SMTP sobre TLS para autenticar a los usuarios (mediante un certificado de usuario), sino que para este fin el servidor SMTP debe usar otras técnicas como mirar la dirección IP de la que procede la conexión de usuario.
- **Identificación de replays.** En este caso SMTP puede usar SMTP sobre TLS para solicitar certificados de usuario a otros servidores que le estén usando como punto de replay. Esto permite que un servidor de SMTP sólo dé servicio a los servidores SMTP con los que está legítimamente conectado, y por ejemplo evitar así el spamming.

Referencias

- [CRLVERISIGN] CRL de Verisign Inc.
<http://crl.verisign.com>
- [DES] DES algorithm
<http://www.itl.nist.gov/fipspubs/fip46-2.htm>
- [KERBEROS] Kerberos: The Network Authentication Protocol
<http://web.mit.edu/kerberos/www/>
- [OPENSSH] OpenSSH
<http://www.openssh.org>
- [OPIE] One time Password in Everything
<http://inner.net/opie>
- [SKEY] S/Key
<http://www.sparc.spb.su/solaris/skey/>
- [SRP] Secure Remote Password
<http://srp.stanford.edu/>
- [SSH] SSH Inc.
<http://www.ssh.com>