

# Curso: (62612) Diseño de aplicaciones seguras

Fernando Tricas García

Departamento de Informática e Ingeniería de Sistemas  
Universidad de Zaragoza

<http://webdiis.unizar.es/~ftricas/>

<http://moodle.unizar.es/>

[ftricas@unizar.es](mailto:ftricas@unizar.es)

# Tema VIII: Criptografía

Fernando Tricas García

Departamento de Informática e Ingeniería de Sistemas  
Universidad de Zaragoza

<http://webdiis.unizar.es/~ftricas/>

<http://moodle.unizar.es/>  
[ftricas@unizar.es](mailto:ftricas@unizar.es)



# Utilización de la criptografía

- ▶ Saber poco de criptografía es más peligroso que no saber nada
- ▶ Errores frecuentes
  - ▶ No identificar la necesidad
  - ▶ Aplicarla mal
- ▶ Vamos a discutir errores frecuentes y algunas bibliotecas disponibles



# Recomendaciones generales

- ▶ Los desarrolladores no son criptógrafos (nunca inventar sistemas)
  - ▶ Un algoritmo criptográfico es algo difícil
  - ▶ Las técnicas de criptoanálisis modernas son muy potentes
- ▶ Lo mejor es usar algoritmos publicados y ampliamente utilizados
- ▶ Tampoco diseñar protocolos criptográficos (incluso los conocidos y utilizados han sido rotos por algún problema en su realización)



# Integridad de los datos

- ▶ Uno de los fallos más frecuentes es pensar que la criptografía garantiza la integridad de los datos
    - ▶ Algo tan simple como cambiar los datos por basura podría ser desastroso
  - ▶ Colocar 'testigos' no siempre es suficiente
    - ▶ Utilizar MAC (Message Authentication Code) para dar cuenta de la integridad y gestionar adecuadamente los fallos de integridad
- Sólo esta parte ya es un buen motivo para utilizar un protocolo conocido



# Leyes de exportación

- ▶ Ya no hay tantas restricciones (especialmente las de EEUU)
- ▶ Hay empresas que, en lugar de exportar la parte criptográfica, la importan!
- ▶ Antes de ... comprobar



# Programando con criptografía

La criptografía viene de serie en Java, a partir de JDK 1.4

- ▶ JCA (Java Cryptography Architecture)
- ▶ JCE (Java Cryptography Extension)
- ▶ `java.security` clases no sujetas a controles de exportación
- ▶ `javax.crypto` las otras



# Programando con criptografía

## Autenticación y control de acceso.

- ▶ Java Security Architecture
- ▶ JAAS (Java Authentication and Authorization Service)
  - ▶ `javax.security.auth.login`

## Comunicaciones seguras

- ▶ JSSE (Java Secure Sockets Extension)
  - ▶ `javax.net.ssl.SSLSocket`
- ▶ JGSS (Java General Security Service)
  - ▶ `org.ietf.jgss`
- ▶ Java SASL API (Simple Authentication and Security Layer)
  - ▶ `javax.security.sasl.Sasl`





# Programando con criptografía

## Infraestructura de clave pública

- ▶ Certificados X.509 y listas de revocación
- ▶ CertPath API (Java Certification Path API)
- ▶ On-line Certificate Status Protocol (OCSP)
- ▶ Java PKCS
  
- ▶ `java.security.KeyStore`
- ▶ `java.secrutiy.cert.CertStore`

## Herramientas PKI

- ▶ `keytool`
- ▶ `jarsigner`



# Es Extensible

- ▶ Se pueden añadir funciones mediante proveedores (*providers*)
- ▶ En `j2.../.../lib/ext` las bibliotecas (es un directorio)
- ▶ En `j2.../.../lib/security/java.security` (es un fichero)  
`security.provider.X= ....`

Se 'recorren' en orden, pero se puede seleccionar



# Capacidades

- ▶ Resúmenes de mensajes
- ▶ Cifrado de clave privada
- ▶ Cifrado de clave pública
- ▶ Firmas digitales
- ▶ Certificados digitales
- ▶ Firma de código
- ▶ SLL/TLS



# Integridad de los mensajes

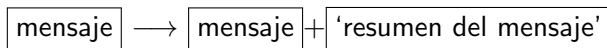
- ▶ Resúmenes:

mensaje



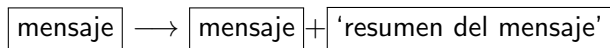
# Integridad de los mensajes

## ► Resúmenes:



# Integridad de los mensajes

## ▶ Resúmenes:

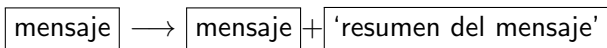


El resumen funciona en una sola dirección

- ▶ checksum
- ▶ hash

# Integridad de los mensajes

- ▶ Resúmenes:



El resumen funciona en una sola dirección

- ▶ checksum
- ▶ hash

MD2, MD5 (128 bits), SHA-1 (160), SHA-256, SHA-383,  
SHA-512

```
MessageDigest.getInstance("MD5")
```

```
MessageDigest.getInstance("MD5", "ProveedorX")
```

```
.update(textoPlano)
```

```
.digest()
```



# Ejemplo

<http://www.spiration.co.uk/post/1199/Javamd5examplewithMessageDigest>

```
include java.security.*;

... etc

sessionId="12345";

byte [] defaultBytes = sessionId.getBytes();
try{
    MessageDigest algorithm = MessageDigest.getInstance("MD5");
    algorithm.reset();
    algorithm.update(defaultBytes);
    byte messageDigest [] = algorithm.digest();

    StringBuffer hexString = new StringBuffer();
    for (int i=0;i<messageDigest.length;i++) {
        hexString.append(Integer.toHexString(0xFF & messageDigest[i]));
    }
    String foo = messageDigest.toString();
    System.out.println(" sessionId_" + sessionId + "_md5_version_" +
        hexString.toString());

    sessionId=hexString+"";
} catch (NoSuchAlgorithmException nsae){
}
}
```





## Algoritmos, clases, ...

Si se usa con clave, sirve para autenticar  
(*Message Authentication Code, MAC*)  
HMAC/SHA-1, HMAC/MD5

```
KeyGenerator.getInstance("HmacMD5")  
.generateKey("HmacMD5")  
Mac.getInstance("HmacMD5")  
.init(MD5key)  
.update(textoPlano)  
.doFinal()
```



# Otros usos de los hash criptográficos

Contención en el uso de recursos ('hash cash'). Puede haber otros esquemas

- ▶ El cliente debe producir una cadena aleatoria cuyo hash tiene ciertas características
  - ▶ como no valen todas, tarda un poco en conseguirlo, repitiendo.
- ▶ El servidor valida la cadena con una operación de hash sencilla
  - ▶ una sola vez
- ▶ Si el hash es bueno, la única posibilidad es iterar y comprobar
- ▶ Comprobación muy simple, cálculo costoso

<http://hashcash.org/>



# Criptografía simétrica

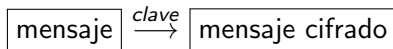
O de clave secreta (privada)

mensaje



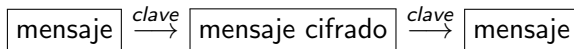
# Criptografía simétrica

O de clave secreta (privada)



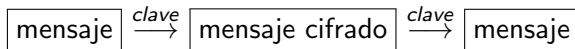
# Criptografía simétrica

O de clave secreta (privada)



# Criptografía simétrica

O de clave secreta (privada)



- ▶ En bloques (64 bits). Si no se tiene 64 bits, se completan (*padding*)
- ▶ También por bits (*stream ciphers*)

DES usa una clave de 56 bits (estándar EEUU), mejor usar 128 bits,

# Cuidado con los 'rellenos'

- ▶ Hay que usarlos cuando trabajamos por bloques y la longitud del mensaje no es múltiplo del tamaño del bloque  
Disponibles:

- ▶ Sin relleno
- ▶ PKCS5
- ▶ OAEP
- ▶ SSL3
- ▶ ...



# Cuidado con los 'rellenos'

- ▶ Hay que usarlos cuando trabajamos por bloques y la longitud del mensaje no es múltiplo del tamaño del bloque
- Disponibles:

- ▶ Sin relleno
- ▶ PKCS5
- ▶ OAEP
- ▶ SSL3
- ▶ ...

La forma en que se hará el cifrado

- ▶ ECB (Electronic Code Book Cipher)
- ▶ CBC (Cipher Block Chaining). XOR con el bloque anterior cifrado.
- ▶ CFB (Cipher Feedback)
- ▶ OFB (Output Feedback)
- ▶ PCBC (Plain Cipher Block Chaining)





# Los algoritmos

- ▶ DES (56 bits, por bloques)
- ▶ TripleDES (112 bits)
- ▶ AES (Rinjdael bloques de 128, con 128, 192, 256 bits de clave)
- ▶ RC2, RC4, RC5 (de RSA)
- ▶ Blowfish (Bruce Schneider, longitudes de clave desde 32 a 448 bits -múltiplos de 8-) Adecuado para microprocesadores.
- ▶ PBE (Password Base Encryption)



# Los algoritmos

```
KeyGenerator.getInstance("DES")  
    .init(56)  
    .generateKey  
Cipher.getInstance("DES/ECB/PKCS5Padding")  
    .init(Cipher.ENCRYPT_MODE, key)  
    .doFinal(textoPlano)  
    .init(Cipher.DECRYPT_MODE, key)  
    .doFinal(textoCifrado)
```



# Ejemplo

<http://ccia.ei.uvigo.es/docencia/SSI/practicas/jce.html>

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

public class EjemploDES {
    /* Ejemplo de uso de funciones de resumen Hash
     * carga el fichero que recibe como parametro, lo cifra y lo descifra
     */
    public static void main(String[] args) throws Exception {
        // Comprobar argumentos
        if (args.length != 1) {
            mensajeAyuda();
            System.exit(1);
        }

        /* Cargar "provider" (solo si no se usa el que viene por defecto) */
        // Security.addProvider(new BouncyCastleProvider()); // Usa provider BC
        //

        /* PASO 1: Crear e inicializar clave */

        System.out.println("1._Generar_clave_DES");
        KeyGenerator keyGen = KeyGenerator.getInstance("DES");
        keyGen.init(56); // clave de 56 bits
        SecretKey clave = keyGen.generateKey();

        System.out.println("CLAVE:");
        mostrarBytes(clave.getEncoded());
        System.out.println();
    }
}
```



# Ejemplo

```
/* PASO 2: Crear cifrador */
Cipher cifrador = Cipher.getInstance("DES/ECB/PKCS5Padding");
// Algoritmo DES
// Modo : ECB (Electronic Code Book)
// Relleno : PKCS5Padding
//

/*****/
System.out.println("2. Cifrar con DES el fichero "+args[0]+
    ", dejar el resultado en "+args[0]+".cifrado");

/* PASO 3a: Inicializar cifrador en modo CIFRADO */
cifrador.init(Cipher.ENCRYPT_MODE, clave);

/* Leer fichero de 1k en 1k y pasar fragmentos leídos al cifrador */
byte [] buffer = new byte[1000];
byte [] bufferCifrado;

FileInputStream in = new FileInputStream(args[0]);
FileOutputStream out = new FileOutputStream(args[0]+".cifrado");

int bytesLeídos = in.read(buffer, 0, 1000);
while (bytesLeídos != -1) { // Mientras no se llegue al final del fichero
    bufferCifrado = cifrador.update(buffer, 0, bytesLeídos);
    // Pasa texto claro leído al cifrador
    out.write(bufferCifrado); // Escribir texto cifrado
    bytesLeídos = in.read(buffer, 0, 1000);
}
bufferCifrado = cifrador.doFinal(); // Completar cifrado (puede devolver texto)
out.write(bufferCifrado); // Escribir final del texto cifrado (si lo hay)

in.close();
out.close();
```



# Ejemplo

```
/******  
System.out.println(" 3. Descifrar con DES el fichero "+args[0]+" .cifrado"+  
    ",_dejar_el_resultado_en"+args[0]+" .descifrado");  
  
/* PASO 3b: Poner cifrador en modo DESCIFRADO */  
cifrador.init(Cipher.DECRYPT_MODE, clave);  
  
in = new FileInputStream(args[0]+" .cifrado");  
out = new FileOutputStream(args[0]+" .descifrado");  
byte [] bufferPlano;  
  
bytesLeidos = in.read(buffer, 0, 1000);  
while (bytesLeidos != -1) { // Mientras no se llegue al final del fichero  
    bufferPlano = cifrador.update(buffer, 0, bytesLeidos); // Pasa texto claro  
    out.write(bufferPlano); // Escribir texto descifrado  
    bytesLeidos = in.read(buffer, 0, 1000);  
}  
bufferPlano = cifrador.doFinal(); // Completar descifrado (puede devolver texto  
out.write(bufferPlano); // Escribir final del texto descifrado (si lo hay)  
  
in.close();  
out.close();  
} // Fin main()  
  
public static void mostrarBytes(byte [] buffer) {  
    System.out.write(buffer, 0, buffer.length);  
}  
  
public static void mensajeAyuda() {  
    System.out.println(" Ejemplo_cifrado_DES" );  
    System.out.println("\t Sintaxis: java EjemploDES_fichero" );  
    System.out.println();  
}  
}
```

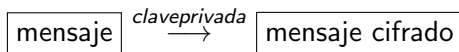


# Criptografía de clave pública

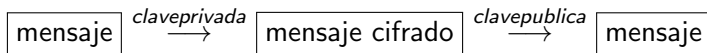
mensaje



# Criptografía de clave pública

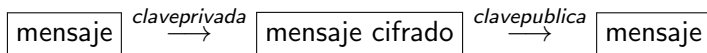


# Criptografía de clave pública

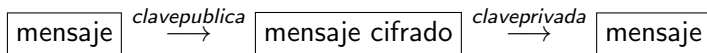




# Criptografía de clave pública



O también:



# Firma digital

mensaje

No hay cifrado. Sólo garantía del origen.



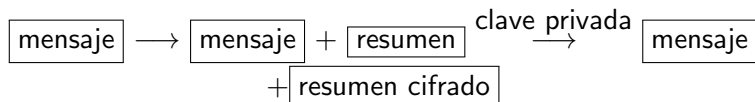
# Firma digital



No hay cifrado. Sólo garantía del origen.

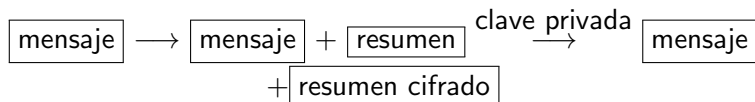


# Firma digital



No hay cifrado. Sólo garantía del origen.

# Firma digital



No hay cifrado. Sólo garantía del origen.

# Algoritmos

- ▶ MD2/RSA
- ▶ MD5/RSA
- ▶ SHA1/DSA
- ▶ SHA1/RSA

Se puede hacer 'a lo bruto' o ...



# Algoritmos

- ▶ MD2/RSA
- ▶ MD5/RSA
- ▶ SHA1/DSA
- ▶ SHA1/RSA

Se puede hacer 'a lo bruto' o ...  
Utilizar la clase signature

```
KeyPairGenerator.getInstance("RSA"),  
    .initialize(1024), .generateKeyPair()  
Cipher.getInstance("MD5WithRSA")  
    .initSign(key.getPrivate())  
    .update(textoPlano) .sign()  
    .initVerify(key.getPublic()) .verify(signature)
```



# Ejemplo

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

import javax.crypto.*;
import javax.crypto.interfaces.*;
import javax.crypto.spec.*;

import java.io.*;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
// Necesario para usar el provider Bouncy Castle (BC)
// Para compilar incluir el fichero JAR en el classpath
//
public class EjemploRSA {
    public static void main (String [] args) throws Exception {

        // Anadir provider JCE (provider por defecto no soporta RSA)
        Security.addProvider(new BouncyCastleProvider()); // Cargar el provider BC

        System.out.println("1.-Creando_claves_publica_y_privada");

        // PASO 1: Crear e inicializar el par de claves RSA DE 512 bits
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA", "BC");
        // Hace uso del provider BC
        keyGen.initialize(512); // tamaño clave 512 bits
        KeyPair clavesRSA = keyGen.generateKeyPair();
        PrivateKey clavePrivada = clavesRSA.getPrivate();
        PublicKey clavePublica = clavesRSA.getPublic();

        System.out.print("2.-Introducir_Texto_Plano_(máx._64_caracteres): ");
        byte [] bufferPlano = leerLinea(System.in);
```





# Ejemplo

```
// PASO 2: Crear cifrador RSA
Cipher cifrador = Cipher.getInstance("RSA", "BC");
// Hace uso del provider BC
/*****
 * IMPORTANTE: En BouncyCastle el algoritmo RSA no funciona realmente en modo ECB
 *
 * No divide el mensaje de entrada en bloques
 *
 * Solo cifra los primeros 512 bits (tam. clave)
 *
 * Habria que hacer la division en bloques "a mano"
 *****/

// PASO 3a: Poner cifrador en modo CIFRADO
cifrador.init(Cipher.ENCRYPT_MODE, clavePublica); // Cifra con la clave publica

System.out.println("3a._Cifrar_con_clave_publica");
byte[] bufferCifrado = cifrador.doFinal(bufferPlano);
System.out.println("TEXTO_CIFRADO");
mostrarBytes(bufferCifrado);
System.out.println("\n_____");

// PASO 3b: Poner cifrador en modo DESCIFRADO
cifrador.init(Cipher.DECRYPT_MODE, clavePrivada); // Descifra con la clave privada
System.out.println("3b._Descifrar_con_clave_privada");
byte[] bufferPlano2 = cifrador.doFinal(bufferCifrado);
[...]
```

*// PASO 3a: Poner cifrador en modo CIFRADO*  
cifrador.init(Cipher.ENCRYPT\_MODE, clavePrivada); // Cifra con la clave publica  
[...]

*// PASO 3b: Poner cifrador en modo DESCIFRADO*  
cifrador.init(Cipher.DECRYPT\_MODE, clavePublica); // Descifra con la clave publica  
[...]



# ¿Quién es usted?

- ▶ La firma del mensaje garantiza que lo envió quien dice haberlo enviado (el dueño de la clave) pero...
- ▶ eso no garantiza que el que lo envió sea quien realmente dice ser
- ▶ Para eso existen las certificadoras



# Manejo de certificados

- ▶ Un fichero como repositorio de claves y certificados *keystore*
- ▶ Pueden tener nombres (*aliases*)
- ▶ Está protegido también con una clave
- ▶ *keytool*: herramienta para manipularlo
- ▶ Se puede usar para exportar una clave a un fichero, que pueda ser firmada por la autoridad certificadora
- ▶ También hay un *keystore* para almacenar los certificados (*truststore*)



## ¿Como se manejan?

- ▶ Implícitamente, cuando se usa SSL/TLS y la firma de ficheros JAR
- ▶ También puede hacerse explícitamente (`CertPath` API)
- ▶ Los certificados tienen fecha de expiración, y también listas de revocación (*Certificate Revocation Lists*)



# ¿Y el código?

- ▶ Java proporciona herramientas para firmar el código también
- ▶ Se hace con `jarsigner`
- ▶ El que lo reciba, decidirá usarlo dependiendo de quien lo firme
- ▶ También se puede implantar un sistema de control de acceso basado en la firma
- ▶ Cuando se usa un 'applet' la referencia es a un fichero de clase contenido en un JAR firmado



# Algunas bibliotecas

## ▶ Cryptlib

- ▶ En C, multiplataforma. En Windows otros (ActiveX)
- ▶ Gratis sólo para usos no comerciales
- ▶ Simétricos: Blowfish, Triple DES, IDEA, RC4, RC5
- ▶ Hash: SHA-1, RIPEMD-160, MD5
- ▶ MACs: HMAC para SHA-1, RIPEMD-160, MD5
- ▶ Clave pública: RSA, El Gamal, Diffie-Hellman, DSA
- ▶ Robusta, bien escrita y eficiente
- ▶ Buena documentación, Fácil de usar

<http://www.cs.auckland.ac.nz/~pgut001/cryptlib/>



# Ejemplos

```
/* Create an envelope for the message */
cryptCreateEnvelope(&cryptEnvelope , CRYPT_UNUSED, CRYPT_FORMAT_SMIME );

/* Push in the message recipient's name */
cryptSetAttributeString(cryptEnvelope , CRYPT_ENVINFO_RECIPIENT ,
                        recipientName , recipientNameLength );

/* Push in the message data and pop out the signed and encrypted result
*/
cryptPushData(cryptEnvelope , message , messageSize , &bytesIn );
cryptPushData(cryptEnvelope , NULL , 0 , NULL );
cryptPopData(cryptEnvelope , encryptedMessage , encryptedSize , &bytesOut
);

/* Clean up */
cryptDestroyEnvelope( cryptEnvelope );
```



# Algunas bibliotecas

- ▶ OpenSSL
  - ▶ En C, multiplataforma, también otros lenguajes
  - ▶ Software libre
  - ▶ Simétricos: Blowfish, Triple DES, IDEA, RC4, RC5
  - ▶ Hash: SHA-1, RIPEMD-160, MD5
  - ▶ MACs: HMAC para SHA-1, RIPEMD-160, MD5
  - ▶ Clave pública: RSA, Diffie-Hellman, DSA
  - ▶ Buena, no tan eficiente, código (algo) confuso
  - ▶ Poca documentación, no tan fácil

<http://www.openssl.org/>





# Algunas bibliotecas

- ▶ Crypto++

- ▶ En C++, muy completa
- ▶ Simétricos: los anteriores y AES
- ▶ Hash: los anteriores y SHA-256, SHA-384, SHA-512
- ▶ MACs: HMAC, DMAC, XOR-MAC
- ▶ Clave pública: RSA, Diffie-Hellman, DSA, ElGamal
- ▶ Buena, no muy eficiente (más que openssl, más en Windows), puede hacer medidas
- ▶ Nada de documentación, fácil de usar
- ▶ Gratuita

<http://www.cryptopp.com/>



# Algunas bibliotecas

## ▶ BSAFE

- ▶ C y Java
- ▶ Simétricos: DES, RC2, RC4, RC5, RC6, AES
- ▶ Hash: MD2, MD5, SHA-1
- ▶ MACs: HMAC
- ▶ Clave pública: RSA, Diffie-Hellman, DSA y extensiones de ECC
- ▶ Muy buena y eficiente
- ▶ Buena documentación, facilidad de uso media
- ▶ Cobran un porcentaje de los beneficios obtenidos

<http://www.emc.com/security/rsa-bsafe.htm>



# Algunas bibliotecas

- ▶ Cryptix
  - ▶ Java
  - ▶ Simétricos: Varios, incluyendo AES
  - ▶ Hash: MD5, SHA-1, RIPEMD-160
  - ▶ MACs: HMAC
  - ▶ Clave pública: RSA, Diffie-Hellman, DSA, ElGamal parcialmente
  - ▶ Buena, poco eficiente
  - ▶ Documentación (JavaDOC) no muy fácil de usar
  - ▶ Gratuita

<http://www.cryptix.org/>  
¡Abandonado!



# Protocolos: SSL y TLS

- ▶ Repetimos: no inventar protocolos
- ▶ El protocolo más utilizado es el SSL, y su sucesor el TLS (Transport Layer Security)
- ▶ Proporcionan cifrado transparente a través de la red
- ▶ La autenticación se basa en la máquina (se autentica un servidor, no una aplicación, ni un usuario)



# Protocolos: SSL y TLS

- ▶ Se usa tecnología de clave pública, utilizando las *autoridades de certificación (CA)*.
- ▶ Se usan como sockets normales, en la máquina local, la infraestructura se ocupa del resto
- ▶ Los usuarios locales tienen acceso al puerto sin cifrar
- ▶ Hay que tener en cuenta algunas cosas...



# Protocolos: SSL y TLS

- ▶ No usar SSL versión 2
- ▶ Utilizar una lista de CAs
- ▶ Hay que comprobar los certificados!!!
- ▶ Hace falta números aleatorios de buena calidad (¡Recordar el caso de Debian!)
- ▶ Si la clave del servidor se ve comprometida, el daño es irreversible

En TLS hay revocación. Certificado nuevo, pero el que tenga el viejo sigue pudiendo identificarse como nosotros.

Expiración.



# Cómo funciona

(Muy esquemático)

- ▶ El cliente empieza la comunicación con su nombre y algunos datos más
- ▶ El servidor responde enviando información, en particular su clave pública
- ▶ El cliente verifica la identidad del servidor y que el nombre coincida.
- ▶ El cliente genera información aleatoria, que se usa como clave de sesión, la codifica con la clave pública del servidor y la envía al servidor. El servidor la descifrará y podrá utilizarla.
  - ▶ El cliente y el servidor utilizan esta clave para comunicarse, y además MAC.



# Protocolos: Stunnel

- ▶ Paquete para hacer túneles SSL, basado en openssl
- ▶ Es fácil proteger un servidor
- ▶ Se ejecuta en modo local





# Protocolos: Stunnel

¿Cómo?

- ▶ Se ejecuta el Stunnel en la IP externa  
`stunnel -d 192.168.100.1:imap2 -r 127.0.0.1:imap2`
- ▶ También se puede hacer con clientes, pero es más complicado
- ▶ Problemas: los del SSL. Además: el programa no puede acceder a la información del certificado



# Libreta de un solo uso

## *One time pad*

- ▶ Los algoritmos criptográficos se consideran seguros porque nadie los ha conseguido romper, no porque realmente lo sean
- ▶ Siempre usar los más usados y 'veteranos'.
- ▶ Los algoritmos simétricos relacionan su seguridad con el tamaño de la clave
- ▶ Los algoritmos basados en clave pública son matemáticamente mas robustos



# Libreta de un solo uso

- ▶ Para cada mensaje, se genera una clave aleatoria
- ▶ Se cifra mediante un XOR
- ▶ Los problemas:
  - ▶ Distribución de las claves
  - ▶ Hacen falta muchos números aleatorios (dependiendo del tamaño del mensaje ..)
  - ▶ Hay más ...



# Libreta de un solo uso

- ▶ Sincronización
  - ▶ Condiciones de carrera: (suponer que se se mandan dos mensajes cruzados, más o menos a la vez)
  - ▶ Dos mensajes codificados con el mismo pad hacen vulnerable el método
- ▶ Integridad de los datos (¿y si se pierde un mensaje o parte?)
- ▶ Muchas complicaciones!



# Algunas ideas que hay que tener en cuenta

- ▶ No almacenar datos innecesarios
- ▶ Cifrar siempre que sea necesario
  - ▶ Cualquier comunicación que potencialmente contenga información confidencial.
  - ▶ Cualquier información potencialmente confidencial almacenada en disco
    - ▶ ¡Claves!
- ▶ Cifrado insuficiente u obsoleto
- ▶ Ofuscar no es cifrar

