Apuntes

Ingeniería Superior en Informática
II55 - Seguridad y Protección de la Información II

Fernando Marco Sales

6 de mayo de 2008

Índice

1. Infraestructuras de clave pública **3**
1.1. Funciones Hash. Tipos y funciones . . . 3
1.2. Repaso del modelo x509 . . . 6
1.3. Autoridades de certificación x509 . . . 7
1.3.1. Declaración de prácticas de certificación (CPS) . . . 7
1.3.2. Tipos de certificados habituales . . . 7
1.3.3. Modelos de expedición de certificados . . . 8
1.4. Dispositivos criptográficos. El clauer . . . 10

2. Librerías y arquitecturas criptográficas **11**
2.1. OpenSSL . . . 11
2.1.1. Openssl desde la línea de comandos . . . 11
2.1.2. Programación . . . 13
2.2. CrytoAPI . . . 15
2.2.1. Arquitectura . . . 15
2.2.2. CSP y Certificate Store Provider . . . 15
2.2.3. Capicom . . . 16
2.3. Pkcs11 . . . 16

3. Protocolos criptográficos **18**
3.1. Firma ciega de Chaum . . . 18
3.2. Lanzar una moneda . . . 18
3.3. Compartición de secretos . . . 20
3.4. Dinero electrónico . . . 21
3.5. Voto telemático . . . 22
3.6. Voto electrónico . . . 24
3.7. Firma digital en la práctica . . . 25

4. Anexo A: Generación de números aleatorios **27**

5. Anexo B: Generación de números primos y Aritmética modular **28**
5.1. Aritmética modular . . . 28
5.2. Generación de números primos . . . 28

1. Infraestructuras de clave pública

1.1. Funciones Hash. Tipos y funciones

Hay diferentes tipos de funciones *Hash*, entre las que tenemos:

- md5: 128 bits
- sha1: 160 bits
- sha2: 256 bits

Algunas características de éste tipo de funciones son:

- Dados 2 mensajes distintos, en la práctica nunca tendrán el mismo *Hash*
- De un *Hash*, es imposible sacar el mensaje

Este tipo de funciones son las que su utilizan a la hora de firmar los documentos y posteriormente comprobar que no estan corruptos por cualquier motivo. Esto lo podemos hacer calculando el md5 de un mensaje y cifrándolo con K_A, más tarde para comprobar que el documento no está corrupto calculamos el md5 del mensaje y comprobamos que es igual al que el documento lleva cifrado, si es así podemos tener la certeza de que el documento esta bien (ver. figura 1).

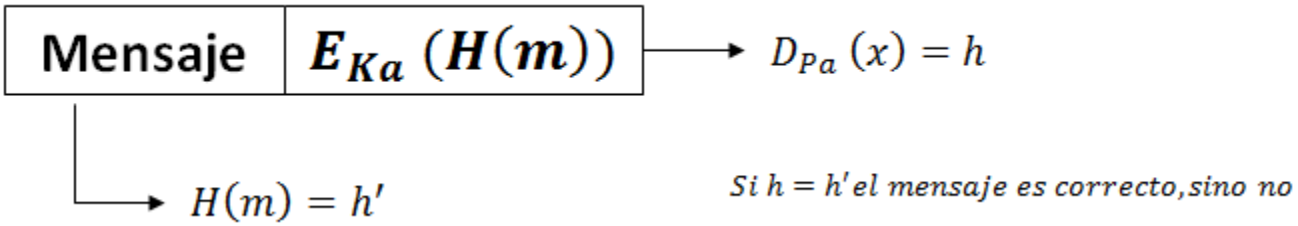

Figura 1: Firma de un documento

En cambio para cifrar se creo una llave única de sesión. Con esta llave ciframos el mensaje. Despues enviamos la llave de sesión cifrada con P_B junto con el mensaje cifrado (ver. figura 2).

Ademas de todo sabemos que si tenemos un mismo mensaje, podemos generar 2 mensajes distintos y conseguir que tengan el mismo *Hash*. Mas tarde si pegamos un mismo mensaje en ambos seguiran teniendo el mismo *Hash* (ver. figura 3) . Por lo tanto sabiendo esta vulnerabilidad del md5 y del sha1, hoy en día a la hora de firmar lo que se hace es calcular el md5, despues el sha1 y por último juntar ambos resultados.

Pero en lo referido a la firma, hay un factor que también nos afecta como es la confianza. Algunos aspectos que tendremos que tener en cuenta son los siguientes:

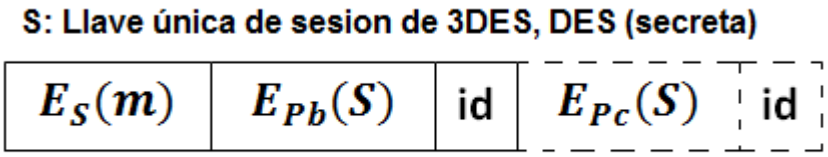

Figura 2: Cifrado de un documento

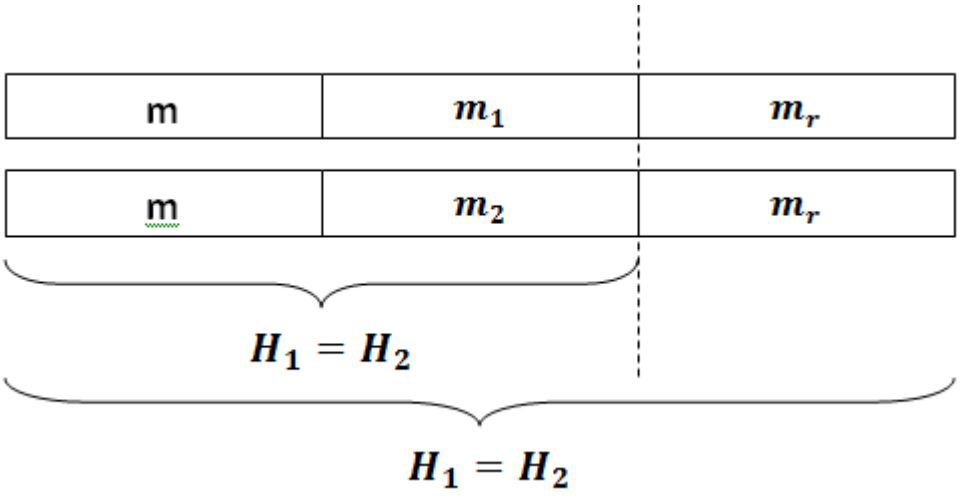

Figura 3: Propiedades del Hash

- La confianza en el Software instalado, ya que hoy en día muchos de ellos contienen Spyware
- La confianza en el equipo que utilizaremos el Software
- La confianza en la llave pública. El saber que una llave es de un persona, no nos lo dice ningún programa, sino que es todo por confianza, directa o indirecta.
  - Directa: El propietario te da su llave pública
  - Autoadquirida: Un ejemplo es con *ssh*, ya que tu aceptas la primera vez la llave y despues conforme interactuas conla máquina te cercioras de que realmente es quien tu creias que era.

Además tenemos 2 tipos de modelos de confianza, que son:

- Horizontal:
  Todos los usuarios estan en el mismo nivel. Se rige por la regla: *"Los amigos de mis amigos, son mis amigos"*. Estos sistemas requieren de usuarios expertos, para que sepan adaptar niveles de confianza.

- Vertical:
  No requieren de usuarios expertos. Estos sistemas requieren de TDC(Terceros de confianza). Este es aquel en que todos confian en una tercera persona, como por ejemplo las CA's.

Por lo tanto cuando me hago un certificado, tengo que confiar que la CA, hace su trabajo correctamente y que he obtenido su llave pública de una forma segura.

1.2. Repaso del modelo x509

Este modelo nos dice como tiene que ser el certificado y la CRL. Los campos de los que se compone son:

- Issuer y Subject = DN(Distinguished Name)
  - CN(Common Name)
  - O(Organization)
  - OU(Organizational Unit)
  - L(Location)
  - ST(State)
  - C(Country)
- Número de serie
- CA emisora
- Fecha inicio - fecha fin
- Versión
- Extensiones
- Algoritmo utilizado
- Llave pública
- Firma del certificado
- Proposito del certificado

La sintaxis del X509 se define mediante el lenguaje ASN1(para su estructura) y los formatos de codificación más comunes son el DER o el PEM. Según la notación ASN1 un certificado se puede agrupar en 3 grandes grupos como son:

1. El primer grupo contiene los datos que identifican a la persona propietaria del certificado(CN, O, OU, L, ST y C), además de el nº de serie, el tiempo de validez, la CA emisora, ...
2. Consta de dos campos para almacenar la llave pública, en el primero de ellos nos dice el algoritmo utilizado para la generación de la llave y el segundo contiene la llave pública
3. Este grupo contiene 3 atributos como son el algoritmo de firma utilizado, el hash de la firma y la propia firma digital

Una cosa que no especifican es el contenido obligatorio de *extensiones*, por lo que se utiliza para poder meter toda aquella información que no se recoge en los demás campos(por ejemplo, el DNI en españa). El problema que genera este campo *extensiones*, es el necesitar para cada tipo de certificado su SW correspondiente, ya que al poder variar este campo, no existe ningún SW que los reconozca todos. Como en todo, existe siempre la posibilidad de que perdamos el certificado, nos lo roben, etc ..., es por ello que el modelo X509, obliga a todas las CA, a que tengan una CRL (*Certificate Revocation List*). La CRL, es un documento firmado por la propia CA, en el cual aparecen todos los números de serie de los certificados que han sido rebocados, el cual podremos utilizar para comprobar la validez del certificado que esten usando. Pero este método es un poco farragoso, ya que tienes que descargarte siempre la CRL de la página de la CA, para poder realizar las comprobaciones oportunas, es por eso que se invento el OCSP (*Online Certificate Status Protocol*), que es un método para comprobar la validez de los certificados vía web.

Anteriormente hemos comentado que los certificados pueden ser rebocados por los titulares de los mismos en cualquier momento, pero no hemos comentado nada de los posibles hechos por los que se permite la rebocación del mismo. Algunos de los motivos admitidos para la rebocacion de los certificados son:

- Perdida del certificado
- Robo del certificado
- Voluntad del usuario (raro pero cierto)

1.3. Autoridades de certificación x509

1.3.1. Declaración de prácticas de certificación (CPS)

El CSP (Certificate Practice Statement), es un documento obligatorio que toda CA debe tener. En el viene recogida toda la política que sigue esta CA, los procedimientos que utiliza para otorgar los certificados, etc... Este documento viene auditado por una autoridad competente para ello, la cual nos da seguridad de que dicha CA cumple a la perfección su CSP. Hoy en día este tipo de documentos son auditados por el *WebTrust CA*.

1.3.2. Tipos de certificados habituales

En el mundo de los certificados digitales, existen mas tipos aparte del certificado personal, que vemos en clase, algunos de estos tipos son:

- Certificado de pertenencia a empresa:
  Acredita tanto la identidad personal del titular como su vinculación con la entidad para la cual trabaja

- Certificado de representante:
  Idéntico que el anterior, pero además acredita los poderes del titular dentro de la empresa
- Certificado de persona jurídica:
  Identifica a una empresa o organización, como tal a la hora de hacer trámites ante las administraciones o instituciones
- Certificado de servidor seguro:
  Se utiliza en los servidores que quieren proteger contra terceros el intercambio de informacion con los usuarios
- Certificado de firma de código:
  Para garantizar la autoría y la no modificación del código de aplicaciones informáticas

1.3.3. Modelos de expedición de certificados

Generalitat Valenciana

Tiene dos tipos, uno de ellos es el *soft*, que guarda la llave en algun disco de almacenamiento(disquet, DD, etc...). El otro tipo es el que usa la tarjeta electrónica. El procedimiento es muy similar para los dos tipos, ya que lo único que cambia es la forma de generar la llave privada, de todos modos pasaremos a explicar como funcionan:

1. Acudir en persona la punto de registro para la obtención del certificado
2. Presentar DNI/Pasaporte, para la autenticación
3. Obtención de llaves
   a) Tipo *soft*
      - floppy
        - pkcs12 firma (más el certificado de autenticación)
        - pcks12 cifrado (se quedan una copia en la GVA)
      - clauer, tanto el pcks12 de firma con el de cifrado van dentro
   b) Tipo *tarjeta*
      - llave de firma (privada): se genera dentro de la tarjeta
      - llave de cifrado (pública): se genera externamente y se mete en la tarjeta

La tarjeta te da la garantía de que no se pueden sacar las llaves de la misma, tanto como si se generan dentro de ella como si se generan externamente y se meten posteriormente. Eso si esta tarjeta puede destruirse bien rascando el chip que lleva incorporado o si intentamos hacer una radiografía de la misma.

Un riesgo que corre la GVA, es el de firma los certificados on-line, exponiendo asi la llave de firma, es por eso que la GVA no firma con su certificado raiz, sino que se vale de una CA secundaria que es la que se encarga de firma los certificados y así mantener segura la llave de firma.

Fábrica Nacional de la Moneda y Timbre

El método de la FNMT, se divide en 3 fases:

1. Generación
   El usuario genera en el ordenador de su casa, una petición de certificado, mediante un *pcks10*, poniendo el DNI en este. Mas tarde lo envia y recibe un *ID*.

2. Identificación
   El usuario acude al punto de registro con el DNI, una fotocopia del DNI y el *ID*. De esta forma se comprueba que la persona que ha pedido el certificado, es realmente quien dice ser.

3. Obtención
   Al cabo de un dia de haber acudido al punto de registro el usuario podra descargarse en su casa si certificado firmado.

El paso 1 y el paso 3, es aconsejable hacerlo en el mismo ordenador, ya que sino la llave privada y el certificado no se asocian. Este método comparado con el de la GVA es bastante más seguro, ya que en el momento de la firma de certificados, no exponemos la llave de firma, al no realizarse de forma on-line, como en la GVA.

CATcert

La autoridad certificadora de cataluña, digamos que tiene 3 modelos diferentes mediante los cuales uno puede conseguir un certificado. El modelo 1 es bastante parecido al que sigue la FNMT pero con alguna ligera modificación, el modelo 2 te dan un código mediante el cual uno se descarga el certificado en casa y el modelo 3 es idéntico al de la GVA, pero exclusivamente con el soporte del clauer. Ahora pasaremos a detallar los pasos de consecución del certificado en cada uno de los modelos:

- Modelo 1:

  1. Rellenar un formulario web (insufrible) y generar la petición de certificado
  2. Acudir en persona la punto de registro para la obtención del certificado
  3. Presentar DNI/Pasaporte, para la autenticación
  4. Recoger el certificado

- Modelo 2:

1. Acudir en persona la punto de registro para la obtención del certificado
2. Presentar DNI/Pasaporte, para la autenticación y recoger el código de descarga
3. Generar la petición de certificado y posteriormente, pasado un tiempo, descargartelo mediante el código

- Modelo 3:

1. Acudir en persona la punto de registro para la obtención del certificado
2. Presentar DNI/Pasaporte, para la autenticación
3. Obtención del certificado, así como de las llaves

1.4. Dispositivos criptográficos. El clauer

El *clauer*, es un proyecto de la *UJI* para implantar el uso de la firma electrónica dentro de la comunidad universitaria. Este consiste en un *software*, el cual una vez instalado puede convertir un dispositivo de almacenamiento, en una herramienta para el tranporte, almacenamiento y uso de certificados digitales, así como las llaves de los mismos. Actualmente existen dos formatos en los que el usuario puede tener el *clauer*, estos son:

- Disco Clauer
  Se utiliza un disco USB, el cual se particiona en 2, para poder así poner el software del clauer, y evitar que el usuario pueda borrar accidentalmente. Despues el usuario mediante las herramientas apropiadas podra introducir/eliminar los certificados, así como hacer uso de el para firmar, ...

- Fichero Clauer
  Se puede utilizar un floppy, un CD o un disco USB, como el usuario prefiera. Esto consiste en generar unos archivos del tipo CRYF_000.cla, CRYF_001.cla, ... y meterlos en alguno de los dispositivos anteriores. Pero este método tiene varios inconvenientes, ya que si se guarda en un CD, aunque resulte bastante barato, si queremos cambiar la contraseña o modificarlo, tenemos que volver a realizar otra copia con las modificacione y volverlas a grabar, destruyendo las anteriores. En cambio si utilizamos un USB, tambíen existe la posibilidad de en un descuido poder borrar todo.

Pero el uso del *clauer* tiene que ser cuidadoso, ya que el dispositivo no tiene autonomía criptográfica, debido a que simplemente es un dispotivo de almacenamiento, en el cual se han puesto los certificados y la llaves de los mismos generados desde el exterior, y en el cual hay que extraer estos certificados para poder usarlos. Con todo esto habrá que tener el suficiente miramiento de utilizarlo en equipos que tengamos plena confianza en que estan administrados correctamente. Algunas de la ventajas e inconvenientes que todo esto provoca son:

- Ventajas:

  - Se pueden almacenar cientos de certificados, con sus contraseñas en el dispositivo
  - El precio es bastante económico, ya que si optamos por el CD, estabamos ablando de centimos, y si nos da por utilizar un disco USB, al poder usarse tambíen como dispositivo de almacenamiento masivo no es caro
  - Hoy en día casi todos los ordenadores disponen de lectores de CD-ROM o de puertos USB, en cambio pocos tienen lectores de tarjetas criptográficas
- Inconvenientes
  - En los equipos inseguros las llaves y los certificados pueden ser copiados, en cambio si utilizaramos un tarjeta criptográfica este no ocurriría
  - Al poder duplicar los certificados, perdemos unicidad y seguridad sobre ellos

Para mas información mirar en `http://clauer.nisu.org`

2. Librerías y arquitecturas criptográficas

2.1. OpenSSL

Es una aplicación y una librería al mismo tiempo. Ya que desde su linea de comandos puedes hacer varias llamadas a sus funciones de librería. Pero también se utiliza para programación crytográfica como librería. El openssl es una librería crytográfica básica, ya que tiene funciones de cálculo de *Hash*, generación de llave simétrica y asimétrica, librería de números grandes, etc... La otra parte del openssl, es la que tiene una librería *ssl (Security Sockects Layer)*, a la cual da soporte, y es con la que se pueden montar clientes ssl o servidores ssl.

El openssl se utiliza mucho sin que el cliente lo sepa, ya que por ejemplo los servidores Apache utilizan este librería para implementar su parte de ssl. La librería EVP, esta pensada como una máquina de cifrado, a la cual se le meten los datos y te los devuelve cifrados. Esta máquina tiene el mismo funcionamiento, lo único que cambia es el proceso, dependiendo del método de cifrado. El *Hash* tambíen se calcula con la librería EVP, ya que tiene funcionamiento de cifrado.

2.1.1. Openssl desde la línea de comandos

El openssl tiene una línea de comandos muy potente desde la que podemos hacer infinidad de cosas, por lo que ahora pasaremos a mostrar algunas ordenes básicas para el manejo de certificados digitales, para firmar, para cifrar o para descifrar. Las ordenes son las siguientes:

1. **Funciones *digest* o de resumen**
   - openssl dgst -md5 archivo
   - openssl md5 archivo

Ademas del MD5, el openssl soporte tambíen los tipos MD2, MD4, SHA, SHA1, ...

2. **Funciones de cifrado**
   El openssl soporta varios algoritmos de clave simétrica para cifrar, entre los que se encuentran DES, 3DES, IDEA, Blowfish y AES. Ademas tambíen soporta el estandar base64, que nos resulta tan útil para el envio de correos. Algunos ejemplos de su uso son:

   - openssl bf -e -in archivo_entrada -out archivo_salida
   - openssl enc -e -3des -in archivo_entrada -out archivo_salida
   - openssl enc -base64 -e -aes128 -in archivo_entrada -out archivo_salida

3. **Funciones de descifrado**
   Al igual que permite el cifrado de datos, tambíen permite su contrario que es el descifrado, con el único cambio en la línea de comando del -e por un -d. Por ejemplo:

   - openssl enc -3des -d -in archivo_entrada -out archivo_salida

4. **Funciones para certificados**
   Las siguientes lineas, nos muestran como hacer un peticion de certificado y posteriormente ver su contenido para poder verificar que lo hemos hecho correctamente.

   - openssl req -new -outform PEM -keyform PEM - keyout llave_salida -out archivo_salida rsa:1024
   - openssl -req -in peticion_cert -text

   Ademas tambíen existen funciones para poder ver los certificados una vez creados, dejandonos el openssl verlos en dos formatos distintos como son el PEM y el DER. Ademas de ello nos permite pasar el certificado de un formata al otro. Un ejemplo de ello es:

   - openssl x509 -in certificado -inform PEM -text -noout
   - openssl x509 -in certificado -inform DER -text -noout
   - openssl x509 -in certificado.pem -inform PEM -out certificado.der -outform DER
   - openssl x509 -in certificado.der -inform DER -out certificado.pem -outform PEM

   Por último saber que el openssl nos permite tambíen importar y exportar certificados de una base de datos *pcks12* y transformarlos a formato PEM/DER, de la siguiente forma:

   - openssl pkcs12 -export BBDD -in certificado.pkcs12 -out certificado.p12
   - openssl pkcs12 -in certificado.p12 -out certificado.pem

2.1.2. Programación

RSA

El algoritmo RSA, es asimétrico (llave privada y llave pública), pero se comporta de forma simétrica, ya que puedes cifrar y descifrar tanto con la llave pública como con la llave privada. Además está totalmente integrado en todas las aplicaciones. El cifrado puede ser tanto con llave pública como con la privada, lo mismo que para descifrar, por lo tanto hay 4 funciones para hacer cada una de las opciones:

- RSA_public_encrypt (fichero, llave_RSA, tipo_de_relleno)
  Ya que cuando se cifra, el RSA lo hace en bloques de 256, puede darse el caso de que se queden 100 bytes para formar un bloque, y al tener que ser estos de 256 bytes, los 1566 bytes son de relleno. El mismo RSA sabe diferenciar lo que es relleno de lo que son datos. Esta función solo puede cifrar 1 bloque, con lo que no puede ser mas grande de n
- RSA_private_decrypt(fichero, llave_RSA, tipo_de_relleno)
  Al igual que en el anterior el tipo de relleno recomendado es RSA_PKCS1_PADDING
- RSA_sign(tipo_Hash, fichero, tam_fichero, buffer_salida, tam_buffer_salida, llave_RSA)
  Hace un cifrado(firma) de *Hash* del fichero que se le pasa con la *llave_RSA* proporcionado, almacenandolo en *buffer_salida*.
- RSA_verify(tipo_Hash, fichero, tam_fichero, buffer_salida, tam_buffer_salida, llave_RSA)
  Comprueba que la firma es válida.

EVP

Cuando se cifra con EVP, se obtiene un buffer con los datos cifrados y un array con las llaves de sesion cifradas para cada destinatorio, además del tamaño del buffer y del array. En la práctica posteriormente se crea un *pkcs7* con el texto y todos los *recipe* y se envia a cada destinatario. Algunas funciones para el manejo del EVP son:

- EVP_signinit(contexto_para_hash, tipo_Hash)
  Inicializa la firma. En el *tipo_de_hash* le podemos pasar: EVP_md4, EVP_md5 o EVP_sha1, si estamos trabajando con una llave RSA para firmar.
- EVP_verifyinit(contexto_para_hash, tipo_Hash)
  Se le pasan los mismos datos que a la función anterior y nos dice si esta es válida o no.
- EVP_signupdate(contexto_para_hash, buffer, tam_buffer)
  Va actualizando el *Hash* con los datos que le pasamos en el *buffer*(*longitud_buffer* marca el tamaño del mismo). El *contexto_para_hash*, debe haber sido inicializado con *EVP_signinit()*, para que la función sea válida.

- EVP_verifyupdate(contexto_para_hash, buffer, tam_buffer)
  Se le pasan los mismos datos que a la función anterior y nos dice si esta es válida o no.

- EVP_signFinal(contexto_para_hash, buffer_devolución, tam_buffer, llave_privada)
  Para acabar la firma, le tenemos que pasar un puntero al buffer donde queremos tener la firma de los datos pasados a la función que será *buffer_devolución*, así como la llave con la que queremos firmar(RSA o DSA). La función nos devolverá en *tamaño_buffer*, el tamaño que tiene el mismo, esto tambíen lo podemos conseguir con la función *EVP_PKEY_size()*. Como en el apartado anterior *contexto_para_hash*, debe haber sido inicializado con *EVP_signinit()*, para que la función sea válida.

- EVP_verifyFinal(contexto_para_hash, buffer_devolución, tam_buffer, llave_privada)
  Se le pasan los mismos datos que a la función anterior y nos dice si esta es válida o no.

- EVP_sealinit(contexto_para_hash, tipo_cifrado, buffer_ini,buffer_llaves,n_llaves)
  Inicializa los diferentes *recipes*, donde albergar cifrada la $llave\_de\_sesion_i$ con la $llave\_pública_i$ del $usuario_i$ requerido.

- EVP_sealupdate(contexto_para_Hash, buffer_salida, buffer_tam_salida, buffer_entrada, buffer_tam_entrada)
  Va actualizando los *buffer_salida* con los datos que le pasamos en el *buffer*(*longitud_buffer* marca el tamaño del mismo). El *contexto_para_hash*, debe haber sido inicializado con *EVP_sealinit()*, para que la función sea válida.

- EVP_sealfinal(contexto_para_Hash, buffer_salida, buffer_tam_salida)
  Finaliza el cifrado de las llaves de sesion y las devuelve junto con su tamaño en *buffer_salida* y *buffer_tam_salida* respectivamente.

- EVP_openinit(contexto_para_Hash, tipo_Hash, llave_sesion_cifrada, tam_llave_cifrada, buffer_ini, llave_privada_RSA)
  Inicializa el descifrado de una llave de sesion pasada en *llave_sesion_cifrada*, con el algoritmo *tipo_Hash* y con la llave privada *llave_privada_RSA*.

- EVP_openupdate(contexto_para_Hash, buffer_salida, buffer_tam_salida, buffer_entrada, buffer_tam_entrada)
  Va actualizandolosl *buffer_salida* con los datos que le pasamos en el *buffer*(*longitud_buffer* marca el tamaño del mismo). El *contexto_para_hash*, debe haber sido inicializado con *EVP_openinit()*, para que la función sea válida.

- EVP_openfinal(contexto_para_Hash, buffer_salida, buffer_tam_salida)
  Termina el descifrado de la llave de sesion, devolviendola en *buffer_salida*, así como su tamaño en *buffer_tam_salida*.

2.2. CrytoAPI

Método utilizado por Microsoft, el cual consiste en que este instala todos los CSP(Cryptographic Service Provider) en el SO y cada aplicación coge los que le hacen falta. Los de Microsoft, han sacado el CryptoAPI, ya que dicen que el *pkcs12*, no tiene ninguna validez para ellos. Los objetos *pkcs12* no se pueden sacar en formato PEM, ya que son binarios y van cifrados.

2.2.1. Arquitectura

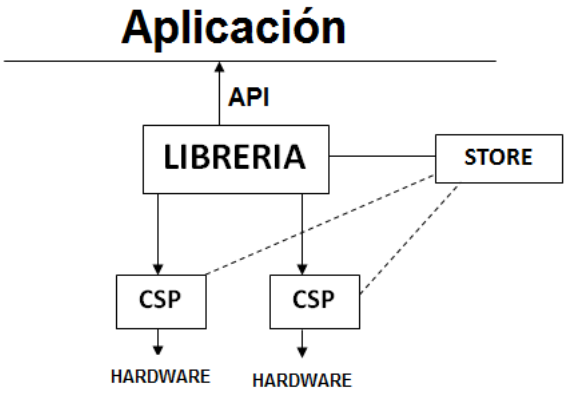

Figura 4: Arquitectura del CryptoAPI

2.2.2. CSP y Certificate Store Provider

Los CSP, no manejan los certificados, sino que estan almacenados en el *Store* (Certifycate Store Provider). Al haber funciones que pueden crear y destruir *Stores*, Microsoft crea los conceptos de *Store Lógico* y *Store Físico*, para saber donde busccar cada aplicación su *Store*.

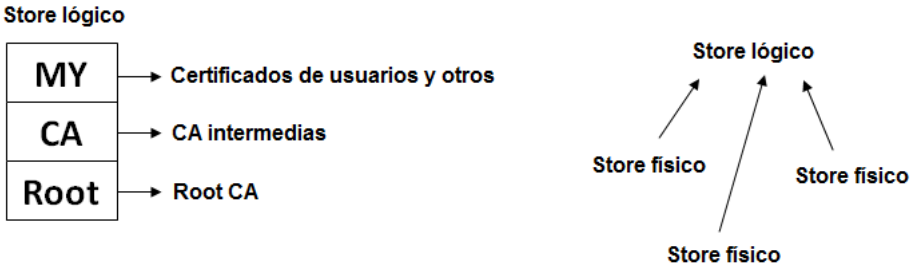

Figura 5: Store Lógico y Store Físico

Cuando se quiere construir un $CSP + Store$ para una aplicación, hay que añadir el Store al *MY* (ver. figura 5). Ademas cada *Store físico* es el encargado de decir que *Store* utiliza cada CSP.

2.2.3. Capicom

Es el módulo criptográfico creado por Microsoft para poder dar servicios a sus aplicaciones en el tema de la seguridad informática, ya que entre sus funciones estan:

- Cifrar datos
- Descifrar Datos
- Firma digital de datos
- Verificación y muestra de firmas digitales o certificados
- Añadir o eliminar certificados de los *stores*

2.3. Pkcs11

La función principal del *pkcs11*, será hablar con los dispositivos. Al final este será quien acabe firmando, codificando, etc...

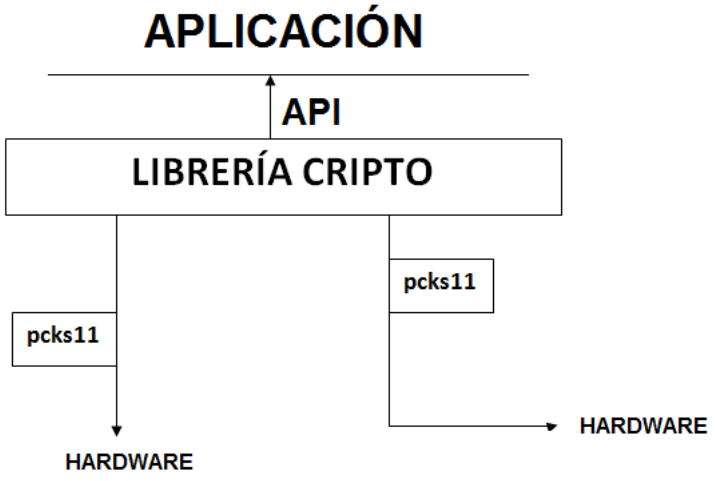

Figura 6: Arquitectura del pcks11

En el firefox, cuando ponemos el comando $< keygen >$ en un código web, si hay más de un dispositivo *pkcs11* (clauer, DNI electrónico, firefox, ...), te saca un listado con los *pkcs11* que hay para elegir donde poner las llaves. Este fue propuesto por los laboratorios RSA. Pero este

modelo tiene un inconveniente, que nos surge cuando una aplicación necesita otro *pkcs11* del que actualmente no dispone. Por lo tanto al no ser el *pkcs11* a nivel de SO, cada vez que queremos añadir un *pkcs11*, hay que añadirselo a todas las aplicaciones. Aparte de este, existen otro *pkcs*, que tambíen nos interesan por diferentes motivos, como son:

- pkcs7: SMIME, para transporte de correo
- pkcs10: Certificate Request
- pkcs12: Transporte privado de objetos (llaves, certificados, CA, CRL, ...)

Recordar que el *pkcs10* se utiliza para pedir certificados. Tiene una estructura similar a la de un certificado, pero esta autofirmado con la misma llave privada del certificado. Este contiene los datos personales y la llave pública (generada por nosotros mismos).

3. Protocolos criptográficos

3.1. Firma ciega de Chaum

La firma ciega (ver. figura 7), consiste en firmar sin saber lo que se firma ó conseguir que te firmen algo, sin saber lo que te estan firmando, todo depende del punto de vista desde el que se mire. El uso más común de éste algoritmo es la firma ciega controlada, que consiste en realizar una firma ciega total pero asegurandonos del contenido. Esto tampoco es del todo cierto, ya que nosotros nunca veremos la información del documento, que estamos firmando, al 100Una forma de poder saber el contenido es de manera estadística, es decir, nosotros le enviamos 10 documentos, la persona en cuestion abrirá 9 de ellos y así podrá comprobar que en todos pone lo mismo, con lo que te firmará el documento restante. En estos casos, el número de envios es proporcional a la importancia del documento a firmar.

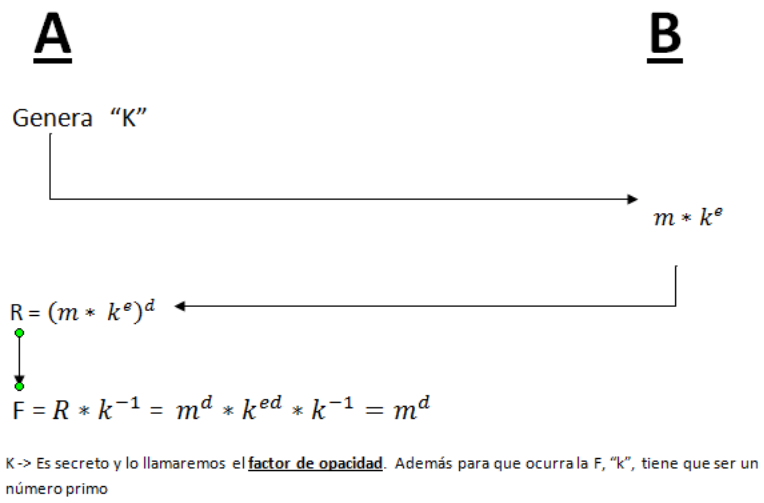

Figura 7: Posible solución para la firma ciega total

Para implementar la firma ciega, habría que hacer documentos distintos, pero todos con la misma forma, al mismo tiempo tendríamos que generar los *k's* distintos, con lo que su aplicación sería la siguiente (ver. figura 8)

Las aplicaciones para este algoritmo suelen ser el voto anónimo y el dinero electrónico.

3.2. Lanzar una moneda

El problema que se nos plantea a continuación es como lanzar una moneda en remoto, y poder estar tranquilos a la hora de realizar una apuesta de lo que saldra, sabiendo que el otro no puede modificar el resultado (o por lo menos es muy dificil que lo modifique). Una posible solucion (ver. figura 9) consiste en que A, genere un número aleatorio grande que llamaremos

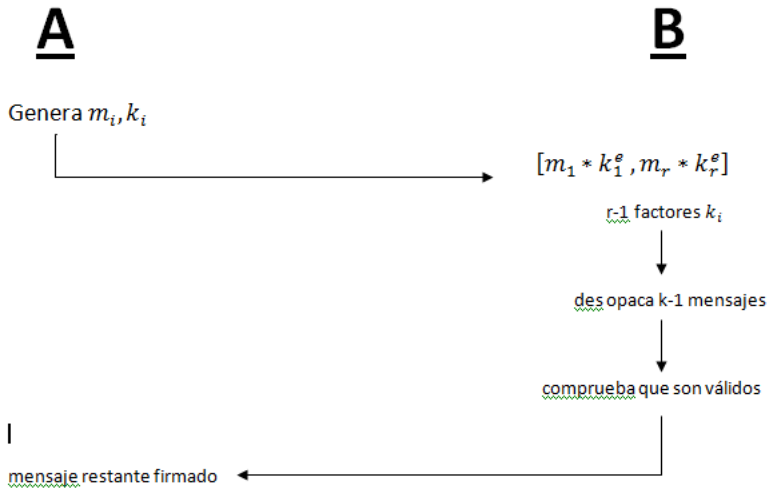

Figura 8: Posible solución para la firma ciega controlada

M y envie a B, el *md5(M)*. Una vez le llega a B, este le envia su apuesta a A, diciendole si cree que *M* es par o impar. Una vez le llegue la apuesta a A, éste le enviará a B, el número *M*, entonces B calculará el *md5(M)*, comprobará si es igual al que A le envio en primer lugar, asegurandose de esta forma de que A no ha hecho trampas, y por último vera si su apuesta era la correcta o no.

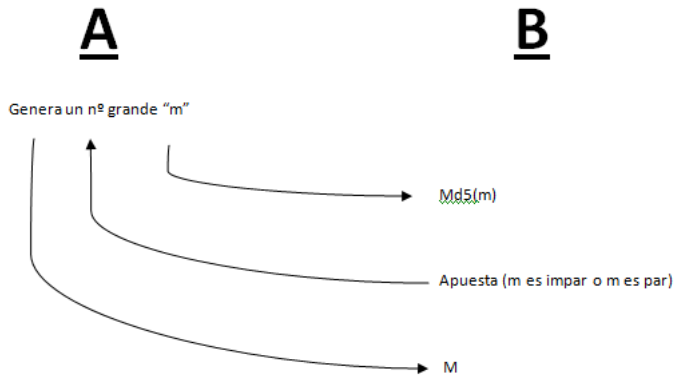

Figura 9: Posible solución para el problema de lanzar una moneda

Esta método es bastante seguro hoy en día, gracias a la dificultad de poder encontrar 2 ficheros con *md5* iguales. Pero en el momento de que alguien encontrara un número par y otro impar con el mismo *md5* la seguridad que nos aporta éste método seria nula.

3.3. Compartición de secretos

La cosa consiste en partir un documento y darle una parte a cada persona, pero que si no se juntan no lo pueden saber. Con lo que cada parte del secreto debe tener información 0. El algoritmo que utilizamos para partir un secreto se basa en la teoria de cifrado perfecto de Shannon (ver. figura 10), que consiste en hacer un *Xor* al mensaje, con una llave aleatoria del mismo tamaño que el mensaje.

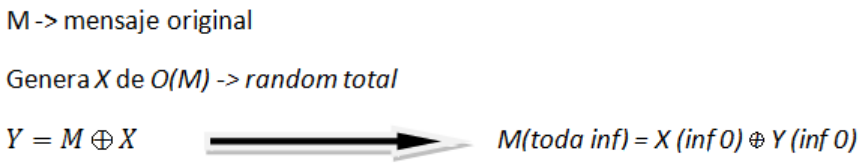

Figura 10: Partición del secreto en 2 partes

Si queremos partir en secreto en más trozos lo único que tendremos que hacer es calcular mas números primos a partir de M (ver. figura 11).

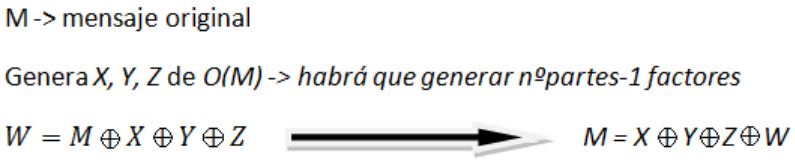

Figura 11: Partición del secreto en M partes

Este método tiene un problema, que se necesitan todas las partes para poder reconstruir el secreto. Para evitar esto, existen diferentes métodos en los que podremos partir el secreto en *m* trozos y necesitar *n* de ellos para reconstruirlo (siendo $n < m$). Nosotros estudiaremos el método de *Shammir* (se puede encontrar la explicación detallada en el Capítulo 23 del libro de Bruce Schneier `http://spi2.nisu.org/recop`), que consiste en la utilización de polinomios, los pasos a seguir son:

- Paso 1:
  Generar un número primo del tamaño de *M(mensaje)*, si *M* es muy grande se parte en trozos más pequeños. El número primo generado es público.
- Paso 2:
  Se genera un polinomio de grado *n-1*(siendo *n* el número de partes que deseamos generar), donde los coeficientes son secretos y destruidos al finalizar. De todas formas

convendría guardarse los coeficientes para poder generar más tarde nuevas partes del secreto, por lo que pueda ocurrir con las partes entregadas.

Un ejemplo del funcionamiento para este algoritmo es el siguiente:
$F(x) = (ax^2 + bx + M)mod(p)$
Daremos tantos valores a *x*, como partes distintas queramos tener.
$K_i = F(x_i)$
$F(x) = (7x^2 + 8x + 11)mod(13)$
$k_0 = (7 * 0^2 + 8 * 0 + 11)mod(13) = 0$
$k_1 = (7 * 1^2 + 8 * 1 + 11)mod(13) = 3$
$k_2 = (7 * 2^2 + 8 * 2 + 11)mod(13) = 7$
...
...
...

A cada persona le entregaremos la *x* y su $F(x)$ correspondiente. Para reconstruir el secreto, cogeremos 3 de las ecuaciones generadas, montaremos el sistema de ecuaciones y despejando, podremos sacar el secreto.
$a^2 * O^2 + b * 0 + M = 0mod(p)$
$a^2 * 1^2 + b * 1 + M = 3mod(p)$
$a^2 * 2^2 + b * 2 + M = 7mod(p)$

Utilizando este método debemos de tener cuidado, ya que el sistema de ecuaciones es modular y entero. Por lo tanto la mejor forma de resolverlo es utilizar el método de los determinantes.

3.4. Dinero electrónico

Una aplicación más, que puede ser realidad gracias a la criptografía de clave pública es conocida como dinero electrónico. El dinero electrónico es físicamente un número que se genera aleatoriamente, se le asigna un valor, se cifra y firma y se envía al banco, ahí el banco valida el número y certifica el valor, y lo regresa al usuario firmado por el banco, entonces el usuario puede efectuar alguna transacción con ese billete electrónico. Las principales propiedades del dinero electrónico son las siguientes:

1. **Independencia:**
   La seguridad del dinero digital no debe depender de la el lugar físico donde se encuentre, por ejemplo en el disco duro de una PC

2. **Seguridad:**
   El dinero digital (el número) no debe de ser usado en dos diferentes transacciones

3. **Privacidad:**
   El dinero electrónico debe de proteger la privacidad de su usuario, de esta forma cuando

se haga una transacción debe de poder cambiarse el número a otro usuario sin que el banco sepa que dueños tuvo antes.

4. **Pagos fuera de línea:**
   El dinero electrónico no debe de depender de la conexión de la red, así un usuario puede transferir dinero electrónico que tenga en una ”smart card.ª una computadora, el dinero digital debe ser independiente al medio de transporte que use.

5. **Transferibilidad:**
   El dinero electrónico debe de ser transferible, cuando un usuario transfiere dinero electrónico a otro usuario debe de borrarse la identidad del primero.

6. **Divisibilidad:**
   El dinero electrónico debe de poder dividirse en valores fraccionarios según sea el uso que se da, por ejemplo en valor de 100, 50 y 25

3.5. Voto telemático

Ahora pasaremos a proponer un posible sistema de votación electrónica, del cual iremos sacando fallos y proponiendo mejoras hasta llegar a un sistema que será mas o menos fiable de utilizar. El sistema más sencillo es que un votante se autentique y tras esto deposite el voto en la urna (ver. figura 12), pero este sistema tiene un fallo muy grande, debido a que el administrador del sistema puede saber quien vota y a quien vota, es por ello que se propone una mejora con el ”sistema 2”(ver. figura 13).

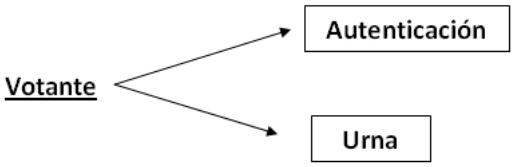

Figura 12: Sistema de votación 1

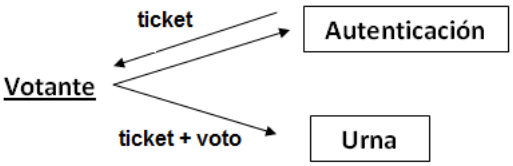

Figura 13: Sistema de votación 2

Este consiste en que el votante se autentica y tras esto recibe un ticket, despues de cerrar la sesion autentica, este envia a la urna el $ticket + voto$, el problema de este sistema es que si el administrador de las autenticaciones y el de la urna se quedan los tickets y los $tickets + voto$ respectivamente, pueden obtener mediante un *join* los resultados de la votación, así como la relación de los votos. Por lo tanto pasamos a la mejorar el "sistema 2", obteniendo con ello el "sistema 3"(ver. figura 14).

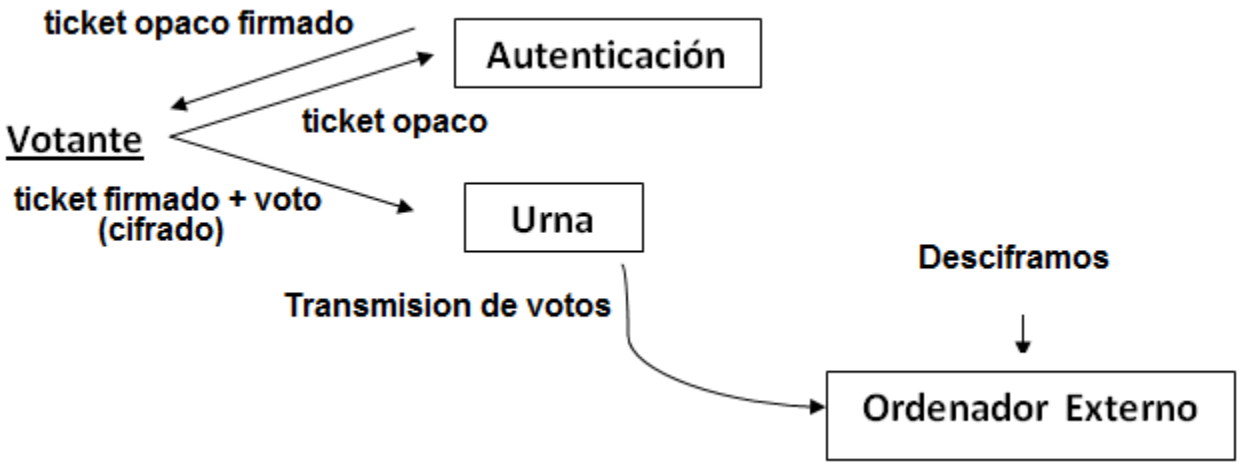

Figura 14: Sistema de votación 3

Este sistema consiste en que el votante envia un ticket opaco al sistema de autenticación, y este mediante la técnica de la firma ciega, se lo devuelve firmado, con esto evitamos que, a diferencia de los sistemas anteriores, puedan obtener una lista que relacione ticket y persona. Mas tarde el votante enviará a la urna el $ticket firmado + voto$ pero este ire cifrado, con esto podemos conseguir si el votante se guarda su voto, que mediante una lista en la que aparezcan todos los votos de la urna, este pueda comprobar que el suyo esta dentro. Ya por último, cuando finalize la votación se enviaran los votos a un ordenador externo donde se pasaran a descifrar. Ademas para asegurarnos de que no se conocen votos durante la votación debido a que estos van cifrados y mediante la llave privada se podrian descifrar, lo que se hace en estos casos es:

1. Generar el par de llaves de la urna antes de la votación
2. Partir la llave privada mediante la técnica *secret sharing*
3. Recomponer la llave privada al final de la votación

Pero este último, aunque bastante seguro no se ha podido utilizar apenas, debido a que la firma ciega estaba patentada, y por lo tanto su utilización sale bastante cara.

3.6. Voto electrónico

Los requisitos para el voto por correo son:

1. Autenticidad
   Solo pueden votar las personas autorizadas
2. Acotabilidad
   Cada votante solo puede votar una vez. En votaciones anónimas se consigue mediante comprobación de IP's, en votaciones privadas, mediante autenticación
3. Anonimato
   Este reñido con el primer requisito. Imposibilidad de asociar voto y votante. Es muy dificil de conseguir. Otro factor es el evitar el rastreo del voto
4. Imposibilidad de coacción
   Que el votante no pueda demostrar lo que ha votado
5. Verificabilidad individual
   Que cada votante pueda comprobar que su voto se haya tenido en cuenta. Esta reñido con el requisito 4
6. Verificación global
   Que la mesa pueda comprobar que ...
7. Fiabilidad
8. Que sea auditable
9. Neutralidad
   Que no se conozca ningún resultado antes de terminar la votación
10. Movilidad de los votantes
11. Facilidad de uso
    Que una persona no informática pueda votar con facilidad
12. Voto rápido
13. Posibilidad de hacer voto nulo
14. Código abierto
    Esto no lo cumple nadie
15. Utilización en una red cerrada
16. Compatibilidad con mecanismos tradicionales
    Es prácticamente imposible

17. Igualdad de oportunidades

Las objecciones que Rebeca Mercury tuvo sobre este sistema son:

- Es imposible superar la venta de votos y coacción
- Forma sencilla de darle seguridad de que el voto ha sido contabilizado y el recuento sea correcto
- No da ningún control a los partidos políticos
- Los sistemas telemáticos abiertos pueden ser atacados de cualquier parte del mundo
- El sistema puede tener fallos y no darse cuenta hasta años despues
- Los mecanismos criptográficos se podrán romper y no se puede depender de ellos

3.7. Firma digital en la práctica

1. pkcs7
   Es un formato para guardar documentos firmados, en su clase *SignedData*. Esta clase tiene dos formatos:
   - No detached: Contiene los datos más la firma. Lo malo es que si no tienen la herramienta adecuada de lectura del pkcs7, no se puede leer.
   - Detached: Genera un pkcs7 únicamente con la firma y aparte tenemos el documento. El único incoveniente, es tener que ir arrastrando dos objetos en lugar de uno. Una forma de solucionar esto es crear un correo electrónico que contenga los dos objetos e ir moviendonos con este.
2. PDF
   El pdf se puede firmar, lo único que necesitaremos un lector que lo reconozca. Para esto tenemos el *Acrobat Reader*.
3. Correo Electrónico
   Se puede firmar mediante SMIME o PGP, ambos son de confianza horizontal, ya que han sido utilizados por la comunidad informática. Además de estos existe un tercer método ya obsoleto como es el PEM, del que hoy en día solo nos queda de este método el *formato PEM*
4. Documentos
   El *Openoffice* utiliza el *xml signature*, que consiste en calcular el Hash del documento y se añade al final. Hoy en día solo es válido entre *openoffice*. El *Microsoft Word* no acepta la firma.

5. Firma Xades
   Es el formato mas correcto para firma de formato *xml*. Este exige que la firma lleve una marca de tiempo, conseguida de una CA de tiempo, para saber la fecha real en que fue firmado el documento. El problema de este método es su dificil implementación

Algunas de las herramientas que se utilizan para firma son las siguientes:

- SignaCat: pcks11(*.p7* (detached) y pdf)
- Acrobat Writter: pdf
- eAuthoring: Para todo documento genera un pdf
- openoffice: odt, firmado con *xml signature*
- appelet: Son aplicaciones que cada uno se monta para firmar lo que el programador de éste quiera
- Librería itext: Tanto Java como $C\sharp$, es la librería más potente que existe

4. Anexo A: Generación de números aleatorios

La generación de números aleatorios, es imposible dentro de la informática. Por lo tanto lo que se utiliza son nº pseudoaleatorios, que vienen dados por una fórmula, que siempre genera la misma secuencia para una semilla pasada. Por ejemplo en C hay dos fórmulas, una de ellas es *rand()*, que siempre nos da la misma secuencia (la cual cosa no es del todo malo, sobretodo para hacer pruebas), la otra es *randomize(semilla)*, que genera los datos aleatorios a partir de la semilla pasada (una semilla típica suele ser la hora del reloj).

En temas de seguridad la hora del reloj no suele ser recomendable, ya que cualquiera puede acotar la hora de tu ordenador y generar él mismo la secuencia. Esto en generación de llaves de sesion puede causar muchos daños. Para alimentar bien la función, hay que utilizar una fuente de eutropía, que nos asegura un mínimo de aletoriedad. En máquinas Unix esto se consigue con */dev/random*, que es como una especie de paquete que alamacena los números aleatorios y los suelta cuando se le llama. Algunas fuentes de eutropía pueden ser: red, ratón, teclado, ... Otra función en Unix es */dev/urandom*, que se basta del random para generar los números.

El openssl tambíen tiene una función de librería para generar nº aleatorios. El fallo es que hay que darle de comer. Para darle de comer se utiliza:

RAND_load_file(/dev/random", 1024);
RAND_write_file("prgnseed.dat");

el fichero de eutropía, se guarda para asegurarse que la próxima vez tenga una fuente de eutropía, en el caso de que fallen todas las demás. Este fichero hay que guardarlo de forma segura, para evitar que nos copjan los números.

5. Anexo B: Generación de números primos y Aritmética modular

5.1. Aritmética modular

Dentro de la protección, la operativa y trabajo con números grandes, tiene que ser en forma modular. Ya que así trabajamos con números más pequeños y evitamos posibles desbordamientos, si trabajamos con nº grandes y calculáramos el módulo al final. En aritmética modular tenemos que:

$$\text{Si p primo:}$$
$$\forall x \epsilon (1...p) \exists x^{-1} / x * x^{-1} = 1$$

El cálculo de la inversa en aritmética modular, se puede realizar con *El algoritmo acumulativo de Euclide* (para ver el algoritmo, consultar la página 58 del libro `http://spi1.nisu.org/recop/Criptografia.pdf`).

5.2. Generación de números primos

Hoy en día se parte del teorema que nos dice, que no hay ninguna fórmula para generar nº primos, así que nos valemos de la fuerza bruta para ello. Del mismo modo para factorizar un nº primo tampoco hay ninguna fórmula, así que en esto se basa la seguridad RSA de generación de llaves, ya que si se pudiera factorizar se destruiría toda la seguridad de hoy en día de internet. Para saber si un nº es primo, se aplican diferentes test al número, cada vez que pasa la prueba, tenemos $1/25$ posibilidades de que no sea primo. Este test se repite como mínimo 10 veces, con lo que conseguimos $1/1000000$ posibilidades de que no sea primo. El openssl cuando genera llaves RSA, tienes dos modos de generación de llaves, que son:

- Modo normal(por defecto):
  Aplica los test anteriormente nombrados pero con bastantes menos restricciones, para evitar el masivo consumo de tiempo para la obtención de los nº primos.

- Modo seguro:
  Aplica los test anteriormente nombrados a la perfección. Para llaves de gran tamaño tarda una eternidad.