PSP06. – TÉCNICAS DE PROGRAMACIÓN SEGURA

1.- PRÁCTICAS DE PROGRAMACIÓN SEGURA.

Aunque se ha avanzado mucho en los mecanismos para escribir código libre de fallos, todavía se siguen creando programas incorrectos. Se estima que por cada 1000 líneas de código que se escriben, aparecen entre 5 y 10 errores. Sólo una pequeña parte de esos errores de programación se pueden considerar vulnerabilidades de seguridad.

Pero, ¿tan difícil es escribir código seguro? A continuación se incluye una lista de buenas prácticas y otra de malas prácticas a la hora de escribir código en las aplicaciones.

1.1.- LISTA DE BUENAS PRÁCTICAS AL ESCRIBIR CÓDIGO EN LAS APLICACIONES.

Una breve lista de cuestiones a tener siempre presentes cuando se desarrolla software es la siguiente:

Informarse

- Una forma de evitar fallos es estudiar y comprender los errores que otros hayan cometido a la hora de desarrollar software
- Internet es el hogar de una gran variedad de foros públicos donde se debaten a veces problemas de vulnerabilidad de software
- Leer libros y artículos sobre prácticas de codificación segura, así como análisis de los defectos del software
- Explorar el software de código abierto, ya que hay muchos ejemplos de cómo llevar a cabo diversas acciones de programación

Precaución en el manejo de datos. La mayoría de los programas aceptan la entrada de algún tipo de dato. Los datos pueden ser adquiridos desde múltiples fuentes de entrada. El programador debería verificar cada pieza de entrada de datos al programa. Algunas de las prácticas a seguir en este apartado son las siguientes:Precaución en el manejo de datos. La mayoría de los programas aceptan la entrada de algún tipo de dato. Los datos pueden ser adquiridos desde múltiples fuentes de entrada. El programador debería verificar cada pieza de entrada de datos al programa. Algunas de las prácticas a seguir en este apartado son las siguientes:

- Limpiar los datos: es el proceso de examen de los datos de entrada. Los atacantes a menudo intentan introducir contenido que está más allá de lo que el programador prevé para la entrada de datos del programa. Por ejemplo: la alteración del juego de caracteres, uso de caracteres no permitidos y desbordamiento del búfer de datos.
- Realizar la comprobación de límites: un problema típico es el desbordamiento del búfer. Es preciso asegurarse de verificar que los datos proporcionados al programa puedan caber en el espacio que se asigna para ello. En los arrays hay que revisar los índices para garantizar que permanecen dentro de sus límites.
- Revisar los ficheros de configuración: es necesario validarlos, como si se tratase de una entrada de datos por teclado, ya que pueden ser manipulados por un atacante.
- Comprobar los parámetros de línea de comandos.
- No fiarse de las URLs web: muchos diseñadores de aplicaciones web utilizan URLs para insertar variables y sus valores. El usuario puede alterar la URL directamente dentro de su navegador por variables de ajuste y/o de sus valores a cualquier configuración que elija.
- Cuidado con los contenidos web: algunas aplicaciones web insertan variables en campos HTML ocultos. Tales campos también pueden ser modificados por el usuario en una sesión de navegación.
- Comprobar las cookies web. Los valores pueden ser modificados por el usuario final y no se debe confiar en ellas.
- Comprobar las variables de entorno: un uso común de las variables de entorno es pasar configuración de preferencias a los programas. Los atacantes pueden proporcionar variables de entorno no previstas por el programador.
- Establecer valores iniciales válidos para los datos: es un buen hábito inicializar correctamente las variables.
- Comprender las referencias de nombre de fichero (rutas de acceso de ficheros y directorios) y utilizarlas correctamente dentro de los programas.
- Proteger la confidencialidad e integridad de información considerada como confidencial: contraseñas, números de cuenta de tarjetas de crédito, etc.

Reutilizar código bueno siempre que sea posible. Se refiere a la reutilización de software que ha sido completamente revisado y probado, y ha resistido las pruebas del tiempo y de los usuarios.

Revisar los programas. Siempre es aconsejable seguir una práctica de revisión de los fallos de seguridad en el código fuente. Si un programa es confiado a varias personas, todas deben participar en su revisión. Algunas prácticas comúnmente utilizadas son:

- Realizar una revisión por pares (dos o más revisores). Para entornos de desarrollo relativamente informales, un proceso de revisión de código por partes puede ser suficiente.
- Realizar una validación y verificación independiente. Algunos proyectos de programación necesitan una revisión más formal que implica revisar el código fuente de un programa, línea por línea, para garantizar que se ajusta a su diseño, así como a otros criterios (los de seguridad, por ejemplo).
- Identificar y utilizar las herramientas de seguridad disponibles. Hay herramientas de software disponibles que ayudan en la revisión de fallos en el código fuente. Son útiles para la captura de errores comunes, aunque no son tan útiles para detectar cualquier otro error. Algunas de estas herramientas son:
- **PMD:** herramienta de auditoría y verificación de código estático que permite detectar errores potenciales en las aplicaciones, en base a un conjunto de reglas parametrizables. Tiene un plugin para el IDE Eclipse. Ver https://pmd.github.io/
- **Visual Code Grepper:** herramienta de revisión automática de la seguridad del código disponible para los lenguajes de programación C++, C#, VB, PHP, Java y PL/SQL. Pensada para acelerar el proceso de revision del código fuente. Ver https://sourceforge.net/projects/visualcodegrepp/
- **FindBugs:** detección de errores de código Java (incluyendo algunos fallos de seguridad). Tiene un plugin para el IDE Eclipse. Ver http://findbugs.sourceforge.net

Utilizar listas de control de seguridad. Estas listas pueden ser muy útiles para asegurarse de que se han cubierto todas las fases durante su ejecución. Un posible ejemplo sería el siguiente:

- La aplicación requiere una contraseña para que los usuarios puedan acceder.
- Todos los inicios de sesión de usuario son únicos.
- La aplicación utiliza el sistema de control de acceso basado en roles.
- Las contraseñas no se transmiten a través de la red en texto plano.
- El cifrado se utiliza para proteger los datos que se transfieren entre servidores y clientes.

Mantener el código en buen estado. El mantenimiento del código puede ser de vital importancia para la seguridad del software en el transcurso de su vida útil. Seguiremos las siguientes prácticas de mantenimiento del código:

- Utilizar normas. Se pueden tener normas con respecto a cosas como la documentación en línea del código fuente, la selección de los nombres de las variables, etc. Esto permite hacer más fácil la vida a los desarrolladores de software que luego mantendrán el código. El código modular, que está bien documentado y que es fácil de seguir, se mantiene mejor.
- Retirar código obsoleto.
- Analizar todos los cambios en el código.

1.2.- LISTA DE MALAS PRÁCTICAS AL ESCRIBIR CÓDIGO EN LAS APLICACIONES.

A continuación se incluye una lista no exhaustiva de malas prácticas a la hora de escribir código en las aplicaciones:

- Escribir código que utilice nombres de fichero relativos: la referencia a un nombre de fichero debe ser completa.
- Referirse dos veces en el mismo programa a un fichero por su nombre. Se recomienda abrir el fichero una vez por su nombre y utilizar un identificador a partir de entonces.
- Invocar programas no confiables dentro de los programas.
- Asumir que los usuarios no son maliciosos.
- Dar por sentado el éxito. Cada vez que se realiza una llamada al sistema, comprobar el valor de retorno por si la llamada fallase.
- Invocar dentro del programa a un shell o una línea de comandos.
- Utilizar áreas de almacenamiento con permisos de escritura. Si es absolutamente necesario, hay que suponer que la información pueda ser manipulada, alterada o destruida por cualquier persona o proceso que así lo desee.
- Guardar datos confidenciales en una base de datos sin protección de contraseña.
- Hacer eco de las contraseñas o mostrarlas en la pantalla del usuario.
- Enviar contraseñas vía e-mail.
- Distribuir mediante programación, información confidencial a través de correo electrónico.

- Guardar las contraseñas sin cifrar.
- Transmitir entre los sistemas contraseñas sin encriptar.
- Tomar decisiones de acceso basadas en variables de entorno o parámetros de línea de comandos que se pasan en tiempo de ejecución.
- Evitar en la medida de lo posible, el uso de software o los servicios de terceros para operaciones críticas.

2.- TÉCNICAS DE SEGURIDAD. VISIÓN GENERAL.

A continuación se presentan algunas de las técnicas y mecanismos más importantes para asegurar sistemas y aplicaciones: criptografía, certificados digitales y control de acceso.

2.1.- CRIPTOGRAFÍA.

El término criptografía es un derivado de la palabra griega kryptos, que significa “oculto”. El objetivo de la criptografía es ocultar el significado de un mensaje mediante el cifrado o codificación del mensaje.

El proceso general de cifrado y descifrado de mensajes se muestra en la siguiente figura:

(link: https://www.google.es)

- Si a un texto legible se le aplica un algoritmo de cifrado, que en general depende de una clave, esto arroja como resultado un texto cifrado que es el que se envía o guarda. A este proceso se le llama **cifrado o encriptación.**
- Si a ese texto cifrado se le aplica el mismo algoritmo, dependiente de la mima clave o de otra clave (esto depende del algoritmo), se obtiene el texto legible original. A este segundo proceso se le llama **descifrado o desencriptación.**

Existen tres **clases de algoritmos criptográficos:**

- **Funciones de una sola vía (o funciones hash):** convierten un mensaje de cualquier tamaño en un mensaje de longitud constante. Permiten mantener la integridad de los datos, tanto en el almacenamiento como en el tráfico de redes. También se usan como parte de los mecanismos de firma digital. Reciben su nombre debido a su naturaleza matemática. Dado un mensaje x, es muy fácil calcular el resultado de f(x). A este f(x) se le denomina el hash, resumen de x o message digest de x. La clave está en que es prácticamente imposible calcular x a partir de las f(x). Este tipo de funciones tienen un amplio abanico de usos en la seguridad informática. Casi cualquier protocolo usa este tipo de funciones así como la autenticación por firmas digitales. Los dos algoritmos de una sola vía más utilizados son el MD5 o SHA-1.
- **Algoritmos de clave secreta o de criptografía simétrica.** El emisor y el receptor comparten el conocimiento de una clave que no debe ser revelada a nadie más. La clave se usa tanto para cifrar como para descifrar el mensaje. La comunicación de las claves entre el emisor y el receptor (diciéndola en alto, mandándola por correo electrónico o postal, haciendo una llamada telefónica, ...) constituye el punto débil de este tipo de criptografía. El algoritmo de cifrado simétrico más popular es el **DES**, que utiliza claves de 56 bits y un cifrado de bloques de 64 bits. Una variante de éste es el Triple DES o 3DES, cuya clave es de 128 bits (112 de clave y 16 de paridad). Otro algoritmo es el **AES** con un tamaño de clave variable siendo el estándar el de 256 bits.
- **Algoritmos de clave pública o de criptografía asimétrica.** El emisor de un mensaje emplea una clave pública (difundida previamente por el receptor) para encriptar el mensaje. El receptor emplea la correspondiente clave privada (no debe ser revelada nunca) para desencriptar el mensaje y sólo él puede desencriptar el mensaje gracias a su clave privada. El algoritmo simétrico más popular es el **RSA**, cuyo uso es prácticamente universal como método de autenticación y firma digital. Además, es componente de protocolos y sistemas como IPSec, SSL, PGP, etc. Una **firma digital** está compuesta por una serie de datos asociados a un mensaje, estos datos permiten asegurar la identidad del firmante (emisor del mensaje) y la integridad del mensaje. El método de firma digital más extendido es el RSA. Su funcionamiento es el siguiente:

- El emisor:
  - Genera un hash del mensaje a enviar mediante una función acordada.
  - Este hash es cifrado con su clave privada. El resultado es lo que se conoce como firma digital que se envía adjunta al mensaje.
  - El emisor envía el mensaje y su firma digital, es decir, el mensaje firmado.

- El receptor:
  - Separa el mensaje de la firma digital.
  - Genera el resumen del mensaje recibido usando la misma función que el emisor.
  - Descifra la firma digital con la clave pública del emisor, obteniendo el hash original.
  - Compara los dos hash y, si son iguales, el mensaje recibido es correcto y con garantías de que nos lo ha enviado el emisor.

En resumen, la criptografía juega 3 papeles principales en la implementación de sistemas seguros:

- Se usa para mantener el secreto y la integridad de la información, donde quiera que pueda estar expuesta a ataques.
- Se utiliza como base para los mecanismos de autentación de la comunicación entre pares de principales (un principal puede ser un usuario o un proceso).
- Se usa para implementar el mecanismo de la firma digital.

2.2.- CERTIFICADOS DIGITALES.

Un **certificado digital** o certificado electrónico es un fichero informático generado por una entidad de servicios de certificación que asocia unos datos de identidad a una persona física, organismo o empresa confirmando de esta manera su identidad digital en Internet. El certificado digital es válido principalmente para autenticar a un usuario o sitio web en internet, por lo que es necesaria la colaboración de un tercero que sea de confianza para cualquiera de las partes que participe en la comunicación.

Estos terceros que son de confianza se llaman **Autoridades de Certificación (AC)**. Son entidades que se encargan de emitir y gestionar tales certificados y que tienen una propiedad muy importante: que se puede confiar en ellas. La forma en la que la AC hace válido el certificado es firmándolo digitalmente. Al aplicar el algoritmo de firma digital al documento se obtiene un texto, una secuencia de datos que permiten asegurar que el titular de ese certificado ha firmado electrónicamente el texto y que éste no ha sido modificado.

Un certificado digital tiene un formato estándar universalmente aceptado que se llama **X.509**, cuya estructura principal está formada por los siguientes campos:

- Versión: indica la versión del formato del certificado, normalmente X.509v3.
- Número de serie: identificador numérico único dentro del dominio de la AC.
- Algoritmo de firma y parámetros, que identifican el algoritmo asimétrico y la función de una sola vía que se usa para firmar el certificado.
- Emisor del certificado: el nombre X.500 de la AC.
- Fechas de inicio y final de validez, que determinan el periodo de validez del certificado.
- Nombre del propietario de la clave pública que se está firmando.
- Identificador del algoritmo que se está utilizando, la clave pública del usuario y otros parámetros si son necesarios.
- La firma digital de la AC, es decir, el resultado de cifrar mediante el algoritmo asimétrico y la clave privada de la AC, el hash obtenido del documento X.509.

Para ver el almacén de certificados del navegador **Mozilla Firefox** pulsamos en la opción de menú Herramientas > Opciones > Avanzado, pestaña de cifrado y pulsamos en el botón Ver Certificados. Desde el navegador Internet Explorer pulsamos en la opción de menú Herramientas > Opciones de internet > pestaña Contenido y pulsamos en el botón Certificados.

Podemos ver el certificado de la página que estamos visitando pinchando sobre el candado.

Las principales **aplicaciones de los certificados digitales** son:

- Autenticar la identidad del usuario de forma electrónica ante terceros.
- Trámites electrónicos ante organismos públicos: la Agencia Tributaria, la Seguridad Social, las Cámaras, ...

- Trabajar con facturas electrónicas.
- Firmar digitalmente correos electrónicos y todo tipo de documentos.
- Cifrar datos para que solo el destinatario del documento pueda acceder a su contenido.

Algunas AC españolas son: FNMT (Fábrica Nacional de Moneda y Timbre), Izenpe y la Agencia Catalana de Certificació (CATCert).

2.3.- CONTROL DE ACCESO.

En general, cualquier empresa u organización tiene diversos tipos de recursos de carácter privado y secreto que necesita asegurar, de manera que solo las personas indicadas puedan acceder a ellos para realizar las tareas requeridas. Esos recursos pueden ser físicos (por ejemplo un equipo informático muy caro), informativos (datos confidenciales) o de personal (los empleados).

Se hace necesario realizar un control de acceso a los recursos, pero este control de acceso es algo más que simplemente requerir nombres de usuario y contraseñas. Hay tres componentes importantes de control de acceso:

- **Identificación:** proceso mediante el cual el sujeto suministra información diciendo quién es.
- **Autenticación:** es cualquier proceso por el cual se verifica que alguien es quien dice ser. Esto implica generalmente un nombre de usuario y una contraseña, pero puede incluir cualquier otro método para demostrar la identidad, como una tarjeta inteligente, exploración de la retina, reconocimiento de voz o huellas dactilares.
- **Autorización:** es el proceso de determinar si el sujeto, una vez autenticado, tiene acceso al recurso. La autorización es equivalente a la comprobación de la lista de invitados de una fiesta.

El sistema de control de acceso debe permitir el acceso al usuario correctamente autenticado y debe impedir el acceso a los demás. Debería también guardar un buen registro de auditoría de todas las entradas e intentos fallidos.

Las medidas de identificación y autenticación se centran en una de estas tres formas:

- Algo que se sabe, algo que se conoce, típicamente las **contraseñas**, son la más extendida.
- Algo que se es, medidas que utilizan la biometría (identificación por medio de la voz, la retina, la huella dactilar, **geometría** de mano, etc.)
- Algo que se tiene, las **tarjetas**.

3.- CRIPTOGRAFÍA CON C#.

3.1.- RESÚMENES DE MENSAJES.

Un message digest o resumen de mensajes (también conocido como función hash) es una marca digital de un bloque de datos. Existe un gran número de algoritmos diseñados para procesar estos resúmenes, siendo dos de los más conocidos SHA y MD5. Los resúmenes de mensaje son funciones hash seguras de una sola vía (obtienen una salida segura de longitud fija, partiendo de una entrada de longitud variable).

Un ejemplo típico del uso de resúmenes de mensajes se puede observar en las tablas con la información de login de los usuarios. En la tabla en vez de almacenar la clave de acceso en texto plano, se almacena el resumen de la clave. Además, suele ser común añadir relleno al texto para dificultar el acceso a la base de datos desde equipos que no conocen el relleno. El proceso de validación de los datos de login es el siguientes:

1. Calcular el resumen de la clave proporcionada
2. Comprobar si existe un usuario con el resumen calculado y el nombre de usuario proporcionado.

Ejemplo

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Resumen
{
```

```
    public class Program
    {
        public static void Main()
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                string hash = GetHash(sha256Hash, source);

                Console.WriteLine($"The SHA256 hash of {source} is: {hash}.");

                Console.WriteLine("Verifying the hash...");

                if (VerifyHash(sha256Hash, source+" ", hash))
                {
                    Console.WriteLine("The hashes are the same.");
                }
                else
                {
                    Console.WriteLine("The hashes are not same.");
                }
            }
        }

        private static string GetHash(HashAlgorithm hashAlgorithm, Object input)
        {

            // Convert the input string to a byte array and compute the hash.
            // Se puede convertir un objeto que haya sido serializado de la misma forma a partir
            // de los bytes que se van a enviar por el socket
            byte[] data = hashAlgorithm.ComputeHash(Encoding.UTF8.GetBytes((String)input));

            // Create a new Stringbuilder to collect the bytes
            // and create a string.
            var sBuilder = new StringBuilder();

            // Loop through each byte of the hashed data
            // and format each one as a hexadecimal string.
            for (int i = 0; i < data.Length; i++)
            {
                sBuilder.Append(data[i].ToString("x2"));
            }

            // Return the hexadecimal string.
            return sBuilder.ToString();
        }

        // Verify a hash against a string.
        private static bool VerifyHash(HashAlgorithm hashAlgorithm, string input, string hash)
        {
            // Hash the input.
            var hashOfInput = GetHash(hashAlgorithm, input);

            // Create a StringComparer an compare the hashes.
            StringComparer comparer = StringComparer.OrdinalIgnoreCase;

            return comparer.Compare(hashOfInput, hash) == 0;
        }

    }
}
// The example displays the following output:
//    The SHA256 hash of Hello World! is: 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069.
//    Verifying the hash...
//    The hashes are the same.
```

3.2.- FIRMAS DIGITALES.

El resumen de un mensaje no proporciona un alto nivel de seguridad. En el ejemplo anterior se puede afirmar que el fichero no es correcto si el texto que se lee no produce la misma salida que el resumen guardado. Pero alguien puede cambiar el texto y el resumen, y no se puede estar seguro de que el texto sea el que debería ser.

En este apartado se verá cómo las firmas digitales pueden autenticar un mensaje y asegurar que el mensaje no ha sido alterado y procede del emisor correcto. De esta forma el emisor no podrá negar haber enviado el mensaje y el receptor estará seguro de que el mensaje no ha sido modificado. Para crear una firma digital se necesita una clave privada y la correspondiente clave pública, con el fin de verificar la autenticidad de la firma. Se debe tener en cuenta que con la firma digital no se está encriptando la información. Toda la comunicación se realizará en texto plano.

Si el emisor A quiere enviar un mensaje a B se sigue el siguiente proceso:

El emisor:

1. A escribe el mensaje y calcula el resumen del mensaje.
2. Cifra el resumen con su clave privada
3. Envia el mensaje junto con el resumen firmado

El receptor:

1. Descifra el resumen firmado que ha recibido con la clave pública de A
2. Calcula el resumen del mensaje
3. Comprueba si el resumen calculado y el descifrado son iguales.

3.3.- CIFRADO CON C#.

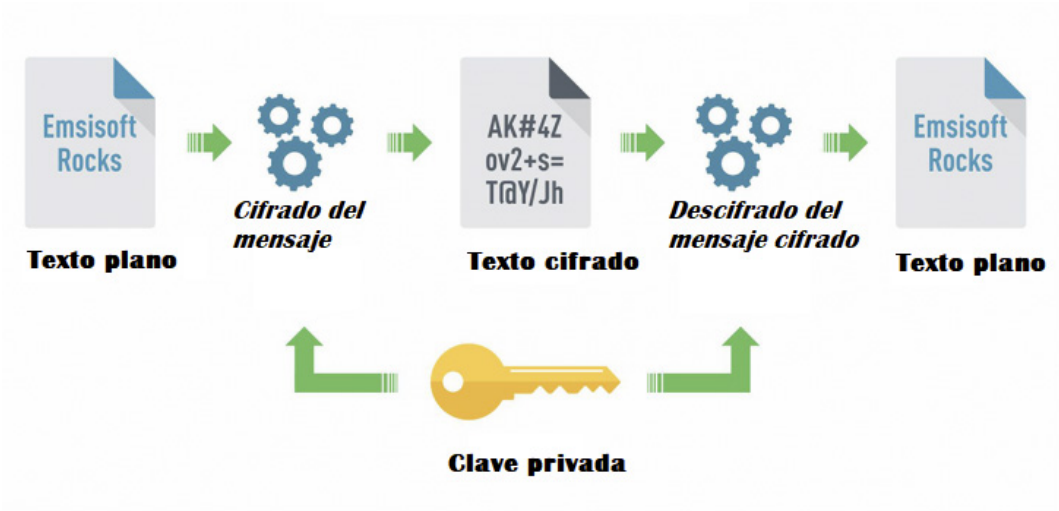
Hasta ahora se ha planteado como se puede asegurar la integridad y la autenticidad del emisor. La firma digital verifica que la información no ha cambiado. Cuando la información es autenticada su contenido queda visible y puede ser manipulado por algún intruso. Sin embargo, cuando la información está encriptada su contenido no es visible. Sólo puede desencriptarse con una clave coincidente.

La autenticación es suficiente para firmar la información, no hay necesidad de ocultarla. **La encriptación es necesaria** cuando las aplicaciones transfieren información confidencial como números de tarjetas de crédito, contraseñas u otros datos personales.

C# proporciona diferentes clases para encriptar (cifrar) o desencriptar (o descifrar) información. El cifrado de un texto legible consite en transformarlo con ayuda de una clave en un texto ilegible; el descifrado es el proceso inverso, se toman los datos ilegibles y la clave y se produce texto legible.

3.3.1.- ENCRIPTAR Y DESENCRIPTAR: CIFRADO SIMETRICO.

La siguiente figura muestra el esquema básico para encriptar y desencriptar con clave privada (o secreta). La misma clave se usará para encriptar y desencriptar la información.



(link: https://www.google.es)

Ejemplo

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Simetrico
{
    class AesExample
    {
        public static void Main()
        {
            string original = "Here is some data to encrypt!";

            // Create a new instance of the Aes
            // class.  This generates a new key and initialization
            // vector (IV).
            using (Aes myAes = Aes.Create())
            {

                // Encrypt the string to an array of bytes.
                byte[] encrypted = EncryptStringToBytes_Aes(original, myAes.Key, myAes.IV);

                // Decrypt the bytes to a string.
                string roundtrip = DecryptStringFromBytes_Aes(encrypted, myAes.Key, myAes.IV);

                //Display the original data and the decrypted data.
                Console.WriteLine("Original:   {0}", original);
                Console.WriteLine("Round Trip: {0}", roundtrip);
            }
        }
        static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (plainText == null || plainText.Length <= 0)
                throw new ArgumentNullException("plainText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");
            byte[] encrypted;

            // Create an Aes object
            // with the specified key and IV.
            using (Aes aesAlg = Aes.Create())
            {
                aesAlg.Key = Key;
                aesAlg.IV = IV;

                // Create an encryptor to perform the stream transform.
                ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

                // Create the streams used for encryption.
                using (MemoryStream msEncrypt = new MemoryStream())
                {
                    using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write
                    {
                        using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                        {
                            //Write all data to the stream.
                            swEncrypt.Write(plainText);
                        }
                        encrypted = msEncrypt.ToArray();
                    }
                }
            }


            // Return the encrypted bytes from the memory stream.
            return encrypted;

        }

        static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (cipherText == null || cipherText.Length <= 0)
```

```
            throw new ArgumentNullException("cipherText");
        if (Key == null || Key.Length <= 0)
            throw new ArgumentNullException("Key");
        if (IV == null || IV.Length <= 0)
            throw new ArgumentNullException("IV");

        // Declare the string used to hold
        // the decrypted text.
        string plaintext = null;

        // Create an Aes object
        // with the specified key and IV.
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = Key;
            aesAlg.IV = IV;

            // Create a decryptor to perform the stream transform.
            ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

            // Create the streams used for decryption.
            using (MemoryStream msDecrypt = new MemoryStream(cipherText))
            {
                using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read)
                {
                    using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                    {

                        // Read the decrypted bytes from the decrypting stream
                        // and place them in a string.
                        plaintext = srDecrypt.ReadToEnd();
                    }
                }
            }

        }

        return plaintext;

    }
  }
}
```

3.3.2.- ENCRIPTAR Y DESENCRIPTAR: CIFRADO ASIMÉTRICO.

En el **cifrado simétrico** o de clave privada se encripta y desencripta la información con la misma clave. El problema está en la distribución de la clave. Si un emisor encripta la información con una clave, el receptor de la información debe desencriptarla con la misma clave. Si el emisor cambia la clave, debe proporcionársela de nuevo al receptor, con lo que pone en peligro el cifrado, ya que alguien puede interceptar la clave.

En el **cifrado asimétrico** o de clave pública se resuelve el problema, ya que la clave para encriptar se puede compartir sin problemas entre el emisor y el receptor, y la clave para desencriptar sólo la tiene el receptor del mensaje. Supongamos que dos participantes A y B deben mantener una conversación secreta. Harían lo siguiente:

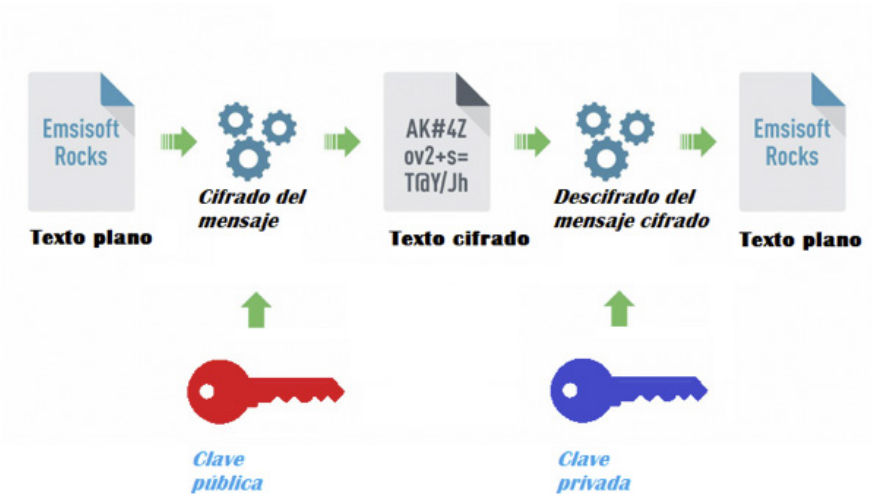
- El participante A crea un par de claves, una privada y una pública de cifrado, y manda la clave pública al participante B sin tener que tomar precauciones.
- El participante B crea su par de claves privada y pública, y manda al participante A la clave pública.
- El participante A crea un mensaje, lo cifra con la clave pública de B y le manda el mensaje. El participante B recibe el mensaje y lo descifra con su clave privada.
- El participante B crea un mensaje, lo cifra con la clave pública de A y le manda el mensaje. El participante A recibe el mensaje y lo descifra con su clave privada. Y así sucesivamente.

El esquema básico de **cifrado y descifrado con clave pública** es el siguiente:

- En primer lugar, se crea el par de claves pública y privada (se puede almacenar en ficheros).
- Luego se realiza el cifrado de la información haciendo uso de la clave pública

- Más adelante en el receptor haciendo uso de la clave privada desencripta el texto cifrado.

La siguiente figura muestra el esquema básico para encriptar y desencriptar haciendo uso de un cifrado asimétrico. Se usará una clave para encriptar y otra diferente para desencriptar la información.



El cifrado mediante clave pública (asimétrico) es más lento que el cifrado mediante clave privada (simétrico). No es práctico utilizar este cifrado para encriptar grandes candidades de información. Este problema se puede solucionar combinando cifrado asimétrico con cifrado simétrico. Este esquema combinado (**cifrado híbrido**) funciona de la siguiente forma:

- El participante A crea una clave de encriptación simétrica. La utiliza para encriptar sus mensajes de texto.
- El participante A encripta la clave simétrica con la clave pública del participante B.
- El participante A envía al participante B, tanto la clave simétrica encriptada como el mensaje de texto encriptado.
- El participante B utiliza su clave privada para desencriptar la clave simétrica de A.
- El participante B utiliza la clave simétrica desencriptada para desencriptar el mensaje.

Nadie excepto el participante B puede desencriptar la clave simétrica, ya que solo él tiene la clave privada para desencriptarla. La encriptación de clave pública sólo se aplica a una pequeña porción de datos.

Ejemplo

Para el ejemplo se ha utilizado el algoritmo RSA. Se debe tener en cuenta que RSA utiliza una clave públic y una clave privada. La clave privada está compuesta por el módulo (Modulus) y el exponente de descifrado (D) y la clave pública por el módulo (Modulus) y el exponente (Exponent). No se va a profundizar en el funcionamiento del algoritmo, pero se debe comprender que las claves son diferentes. Además, si se inspecciona el valor de la clave privada se podrán ver otros atributos que se utilizan para la generación de las claves como los números primos.

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Asimetrico
{
    class RSACSPSample
    {

        static void Main()
        {
            try
            {
                //Create a UnicodeEncoder to convert between byte array and string.
                UnicodeEncoding ByteConverter = new UnicodeEncoding();
```

```
            //Create byte arrays to hold original, encrypted, and decrypted data.
            byte[] dataToEncrypt = ByteConverter.GetBytes("Data to Encrypt");
            byte[] encryptedData;
            byte[] decryptedData;

            //Create a new instance of RSACryptoServiceProvider to generate
            //public and private key data.
            using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
            {

                //Pass the data to ENCRYPT, the public key information
                //(using RSACryptoServiceProvider.ExportParameters(false),
                //and a boolean flag specifying no OAEP padding.
                encryptedData = RSAEncrypt(dataToEncrypt, RSA.ExportParameters(false), false);


                //Pass the data to DECRYPT, the private key information
                //(using RSACryptoServiceProvider.ExportParameters(true),
                //and a boolean flag specifying no OAEP padding.
                decryptedData = RSADecrypt(encryptedData, RSA.ExportParameters(true), false);

                //Display the decrypted plaintext to the console.
                Console.WriteLine("Decrypted plaintext: {0}", ByteConverter.GetString(decryptedData));

                // parametro RSA con clave privada
                RSAParameters RSAKeyInfo = RSA.ExportParameters(true);
                // parametro RSA sin clave privada
                RSAParameters RSAKeyInfo2 = RSA.ExportParameters(false);

            }
        }
        catch (ArgumentNullException)
        {
            //Catch this exception in case the encryption did
            //not succeed.
            Console.WriteLine("Encryption failed.");

        }
    }

    public static byte[] RSAEncrypt(byte[] DataToEncrypt, RSAParameters RSAKeyInfo, bool DoOAEPPadding)
    {
        try
        {
            byte[] encryptedData;
            //Create a new instance of RSACryptoServiceProvider.
            using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
            {

                //Import the RSA Key information. This only needs
                //toinclude the public key information.
                RSA.ImportParameters(RSAKeyInfo);

                //Encrypt the passed byte array and specify OAEP padding.
                //OAEP padding is only available on Microsoft Windows XP or
                //later.
                encryptedData = RSA.Encrypt(DataToEncrypt, DoOAEPPadding);
            }
            return encryptedData;
        }
        //Catch and display a CryptographicException
        //to the console.
        catch (CryptographicException e)
        {
            Console.WriteLine(e.Message);

            return null;
        }

    }

    public static byte[] RSADecrypt(byte[] DataToDecrypt, RSAParameters RSAKeyInfo, bool DoOAEPPadding)
    {
        try
        {
```

```
                byte[] decryptedData;
                //Create a new instance of RSACryptoServiceProvider.
                using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
                {
                    //Import the RSA Key information. This needs
                    //to include the private key information.
                    RSA.ImportParameters(RSAKeyInfo);

                    //Decrypt the passed byte array and specify OAEP padding.
                    //OAEP padding is only available on Microsoft Windows XP or
                    //later.
                    decryptedData = RSA.Decrypt(DataToDecrypt, DoOAEPPadding);
                }
                return decryptedData;
            }
            //Catch and display a CryptographicException
            //to the console.
            catch (CryptographicException e)
            {
                Console.WriteLine(e.ToString());

                return null;
            }

        }
    }
}
```

4.- COMUNICACIONES SEGURAS CON C#.

Los datos que viajan a través de la red pueden ser accedidos por personas que no son destinatarias de los mismos. Cuando incluyen información privada, como contraseñas o números de tarjetas de crédito, se deben tomar medidas para que sean incomprensibles para las partes no autorizadas. También es importante asegurarse de que los datos no se modifiquen durante el transporte, ya sea intencionadamente o no. Los protocolos SSL (Secure Sockets Layer) y TLS (Transport Layer Security) se han diseñado para ayudar a proteger la privacidad y la integridad de los datos mientras se transportan a través de una red.

Para poder utilizar streams o flujos SSL se debe disponer de certificados, como mínimo en el servidor. Gracias a los certificados y los flujos seguros, no es necesario cifrar la información en nuestras aplicaciones. Se debe tener en cuenta que el certificado del servidor debe tener la clave privada para poder descifrar la información que reciba. C# incluye la funcionalidad de encriptación de los datos, autenticación de servidores, integridad de mensajes y autenticación de clientes. De esta forma, los desarrolladores pueden ofrecer intercambio seguro de datos entre un cliente y un servidor que ejecuta un protocolo de aplicación (HTTP, Telnet, FTP, ...) a través de TCP/IP.

Ejemplo

En el siguiente ejemplo se muestra un cliente y servidor que hacen uso de SocketsSSL. En caso de no disponer de un certificado se puede modificar la función de callback ValidateServerCertificate para que acepte certificados no válidos. Tal y como se puede observar, el flujo de datos se obtiene de los sockets, por lo tanto, puede servir de ejemplo para modificar una aplicación que se comunica haciendo uso de sockets.

En el servidor, además de las tareas relacionadas con la comunicación se muestran las características del certificado.

Cliente SSL

```
using System;
using System.Collections;
using System.Net;
using System.Net.Security;
using System.Net.Sockets;
using System.Security.Authentication;
using System.Text;
using System.Security.Cryptography.X509Certificates;
using System.IO;
using System.Threading;

namespace ClienteSSL
```

```
{
    public class SslTcpClient
    {
        private static Hashtable certificateErrors = new Hashtable();

        // The following method is invoked by the RemoteCertificateValidationDelegate.
        public static bool ValidateServerCertificate(
              object sender,
              X509Certificate certificate,
              X509Chain chain,
              SslPolicyErrors sslPolicyErrors)
        {
            if (sslPolicyErrors == SslPolicyErrors.None)
                return true;

            Console.WriteLine("Certificate error: {0}", sslPolicyErrors);

            // Do not allow this client to communicate with unauthenticated servers.
         // Si no se dispone de un certificado valido para las pruebas, cambiar por return
         // true. De esta forma no se tiene en cuenta si hay algún problema
            return false;
        }
        public static void RunClient(string machineName, string serverName)
        {
            // Create a TCP/IP client socket.
            // machineName is the host running the server application.
            TcpClient client = new TcpClient("192.168.33.11", 443);
            Console.WriteLine("Client connected.");
            // Create an SSL stream that will close the client's stream.
            SslStream sslStream = new SslStream(
                client.GetStream(),
                false,
                new RemoteCertificateValidationCallback(ValidateServerCertificate),
                null
                );
            // The server name must match the name on the server certificate.
            try
            {
                sslStream.AuthenticateAsClient(serverName, null, SslProtocols.None, false);
                //sslStream.AuthenticateAsClient("ServerSSL");

            }
            catch (AuthenticationException e)
            {
                Console.WriteLine("Exception: {0}", e.Message);
                if (e.InnerException != null)
                {
                    Console.WriteLine("Inner exception: {0}", e.InnerException.Message);
                }
                Console.WriteLine("Authentication failed - closing the connection.");
                client.Close();
                return;
            }
            // Encode a test message into a byte array.
            // Signal the end of the message using the "".
            byte[] messsage = Encoding.UTF8.GetBytes("Hello from the client.");
            // Send hello message to the server.
            sslStream.Write(messsage);
            sslStream.Flush();
            // Read message from the server.
            string serverMessage = ReadMessage(sslStream);
            Console.WriteLine("Server says: {0}", serverMessage);
            // Close the client connection.
            client.Close();
            Console.WriteLine("Client closed.");
        }
        static string ReadMessage(SslStream sslStream)
        {
            // Read the  message sent by the server.
            // The end of the message is signaled using the
            // "" marker.
            byte[] buffer = new byte[2048];
            StringBuilder messageData = new StringBuilder();
            int bytes = -1;
            do
            {
```

```
                bytes = sslStream.Read(buffer, 0, buffer.Length);

                // Use Decoder class to convert from bytes to UTF8
                // in case a character spans two buffers.
                Decoder decoder = Encoding.UTF8.GetDecoder();
                char[] chars = new char[decoder.GetCharCount(buffer, 0, bytes)];
                decoder.GetChars(buffer, 0, bytes, chars, 0);
                messageData.Append(chars);
                // Check for EOF.
                if (messageData.ToString().IndexOf("") != -1)
                {
                    break;
                }
            } while (bytes != 0);

            return messageData.ToString();
        }
        private static void DisplayUsage()
        {
            Console.WriteLine("To start the client specify:");
            Console.WriteLine("clientSync machineName [serverName]");
            Environment.Exit(1);
        }
        public static int Main(string[] args)
        {
            string serverCertificateName = null;
            string machineName = null;
            if (args == null || args.Length < 1)
            {
                DisplayUsage();
            }
            // User can specify the machine name and server name.
            // Server name must match the name on the server's certificate.
            machineName = args[0];
            if (args.Length < 2)
            {
                serverCertificateName = machineName;
            }
            else
            {
                serverCertificateName = args[1];
            }
            SslTcpClient.RunClient(machineName, serverCertificateName);
            return 0;
        }
    }
}
```

Servidor SSL

```
using System;
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Net.Security;
using System.Security.Authentication;
using System.Text;
using System.Security.Cryptography.X509Certificates;
using System.IO;
using System.Threading;

namespace ServidorSSL
{
    public sealed class SslTcpServer
    {
        static X509Certificate serverCertificate = null;
        // The certificate parameter specifies the name of the file
        // containing the machine certificate.
        public static void RunServer(string certificate)
        {
            serverCertificate = new X509Certificate(certificate, "password");
            Int32 port = 443;
            TcpListener listener = new TcpListener(IPAddress.Any, port);
            listener.Start();
            while (true)
```

```
        {
            Console.WriteLine("Waiting for a client to connect...");
            // Application blocks while waiting for an incoming connection.
            // Type CNTL-C to terminate the server.

            TcpClient client = listener.AcceptTcpClient();
            ProcessClient(client);
        }
    }
    static void ProcessClient(TcpClient client)
    {
        // A client has connected. Create the
        // SslStream using the client's network stream.
        SslStream sslStream = new SslStream(
            client.GetStream(), false);
        // Authenticate the server but don't require the client to authenticate.
        try
        {
            sslStream.AuthenticateAsServer(serverCertificate, clientCertificateRequired: false, checkCertific

            // Display the properties and settings for the authenticated stream.
            DisplaySecurityLevel(sslStream);
            DisplaySecurityServices(sslStream);
            DisplayCertificateInformation(sslStream);
            DisplayStreamProperties(sslStream);

            // Set timeouts for the read and write to 5 seconds.
            sslStream.ReadTimeout = 5000;
            sslStream.WriteTimeout = 5000;
            // Read a message from the client.
            Console.WriteLine("Waiting for client message...");
            string messageData = ReadMessage(sslStream);
            Console.WriteLine("Received: {0}", messageData);

            // Write a message to the client.
            byte[] message = Encoding.UTF8.GetBytes("Hello from the server.");
            Console.WriteLine("Sending hello message.");
            sslStream.Write(message);
        }
        catch (AuthenticationException e)
        {
            Console.WriteLine("Exception: {0}", e.Message);
            if (e.InnerException != null)
            {
                Console.WriteLine("Inner exception: {0}", e.InnerException.Message);
            }
            Console.WriteLine("Authentication failed - closing the connection.");
            sslStream.Close();
            client.Close();
            return;
        }
        finally
        {
            // The client stream will be closed with the sslStream
            // because we specified this behavior when creating
            // the sslStream.
            sslStream.Close();
            client.Close();
        }
    }
    static string ReadMessage(SslStream sslStream)
    {
        // Read the  message sent by the client.
        // The client signals the end of the message using the
        // "" marker.
        byte[] buffer = new byte[2048];
        StringBuilder messageData = new StringBuilder();
        int bytes = -1;
        do
        {
            // Read the client's test message.
            Thread.Sleep(3000);
            bytes = sslStream.Read(buffer, 0, buffer.Length);

            // Use Decoder class to convert from bytes to UTF8
            // in case a character spans two buffers.
```

```
            Decoder decoder = Encoding.UTF8.GetDecoder();
            char[] chars = new char[decoder.GetCharCount(buffer, 0, bytes)];
            decoder.GetChars(buffer, 0, bytes, chars, 0);
            messageData.Append(chars);
            // Check for EOF or an empty message.
            if (messageData.ToString().IndexOf("") != -1)
            {
                break;
            }
        } while (bytes != 0);

        return messageData.ToString();
    }
    static void DisplaySecurityLevel(SslStream stream)
    {
        Console.WriteLine("Cipher: {0} strength {1}", stream.CipherAlgorithm, stream.CipherStrength);
        Console.WriteLine("Hash: {0} strength {1}", stream.HashAlgorithm, stream.HashStrength);
        Console.WriteLine("Key exchange: {0} strength {1}", stream.KeyExchangeAlgorithm, stream.KeyExchangeSt
        Console.WriteLine("Protocol: {0}", stream.SslProtocol);
    }
    static void DisplaySecurityServices(SslStream stream)
    {
        Console.WriteLine("Is authenticated: {0} as server? {1}", stream.IsAuthenticated, stream.IsServer);
        Console.WriteLine("IsSigned: {0}", stream.IsSigned);
        Console.WriteLine("Is Encrypted: {0}", stream.IsEncrypted);
    }
    static void DisplayStreamProperties(SslStream stream)
    {
        Console.WriteLine("Can read: {0}, write {1}", stream.CanRead, stream.CanWrite);
        Console.WriteLine("Can timeout: {0}", stream.CanTimeout);
    }
    static void DisplayCertificateInformation(SslStream stream)
    {
        Console.WriteLine("Certificate revocation list checked: {0}", stream.CheckCertRevocationStatus);

        X509Certificate localCertificate = stream.LocalCertificate;
        if (stream.LocalCertificate != null)
        {
            Console.WriteLine("Local cert was issued to {0} and is valid from {1} until {2}.",
                localCertificate.Subject,
                localCertificate.GetEffectiveDateString(),
                localCertificate.GetExpirationDateString());
        }
        else
        {
            Console.WriteLine("Local certificate is null.");
        }
        // Display the properties of the client's certificate.
        X509Certificate remoteCertificate = stream.RemoteCertificate;
        if (stream.RemoteCertificate != null)
        {
            Console.WriteLine("Remote cert was issued to {0} and is valid from {1} until {2}.",
                remoteCertificate.Subject,
                remoteCertificate.GetEffectiveDateString(),
                remoteCertificate.GetExpirationDateString());
        }
        else
        {
            Console.WriteLine("Remote certificate is null.");
        }
    }
    private static void DisplayUsage()
    {
        Console.WriteLine("To start the server specify:");
        Console.WriteLine("serverSync certificateFile.cer");
        Environment.Exit(1);
    }
    public static int Main(string[] args)
    {
        string certificate = null;
        if (args == null || args.Length < 1)
        {
            DisplayUsage();
        }
        certificate = args[0];
        SslTcpServer.RunServer(certificate);
```

```
            return 0;
        }
    }
}
```