1 Cifrado asimétrico

Para realizar el cifrado asimétrico en Javascript se necesita una clave pública y otra privada. Estas claves se pueden generar desde Javascript (para un uso temporal) o desde línea de comandos (para un uso permanente).

Para cifrar un socket se necesitan los certificados públicos y privados. Para ello se puede usar el comando:

```
 $ openssl req -nodes -new -x509 -keyout server.privada -out server.publica
```

Posteriormente las debemos cargar en nuestro código:

```
const fs = require("fs");

const publicKey = fs.readFileSync('server.publica', 'utf-8');
const privateKey = fs.readFileSync('server.privada', 'utf-8');
```

Pero si el uso va a ser temporal, por ejemplo, para establecer una conexión temporal mediante un socket, se pueden generar desde Javascript, para ello se debe usar la biblioteca crypto:

```
const crypto = require("crypto");
const { publicKey, privateKey } = crypto.generateKeyPairSync("rsa",
{modulusLength: 2048});
```

Como se puede ver, se está usando el algoritmo “rsa” para generar las claves.

El cifrado rsa tiene limitaciones, **no es posible cifrar un texto que sea mayor que las claves generadas**. Por ello, cuando hay que cifrar grandes cantidades de datos, se genera un contraseña que se cifra por criptografía asimética. Después se cifra el texto que se desee con dicha contraseña. Para mandar el mensaje, se manda el mensaje cifrado de forma simétrica con la contraseña cifrada de forma asimétrica.

2 Formatos base64 y hexadecimal

Los textos cifrados suelen ser un conjunto ilegibles de bytes, por lo que es normal representarlos en otros formatos más manejables. Los habitual es usar base64 o hexadecimal.

En el **formato hexadecimal** se usan sólo 4 bits para representar la información. Lo que se hace es dividir la cadena de bytes en bloques de 4 bits. Con 4 bits sólo se pueden representar 16 símbolos que son:

```
0123456789abcdef
```

En el **formato base64** se usan sólo 6 bits para representar la información. Lo que se hace es dividir la cadena de bytes en bloques de 6 bits. Con 6 bits sólo se pueden representar 64 símbolos que son:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

El formato base64 es usado actualmente para incluir archivos adjuntos en los correos electrónicos.

Las funciones que se han usado y se van a usar en estos apuntes para el cifrado de datos, tienen métodos para indicar si la información se desea en base64 o hexadecimal. Concretamente se usa

mucho la clase Buffer. Esta clase nos permite transformar un texto a base64 o hexadecimal y viceversa:

```
// Para base64:
console.log(Buffer.from("Hello World").toString('base64'));
// SGVsbG8gV29ybGQ=
console.log(Buffer.from("SGVsbG8gV29ybGQ=", 'base64').toString('utf-8'))
// Hello World

// Para hexadecimal:
console.log(Buffer.from("Hello World").toString('hex'));
// 48656c6c6f20576f726c64
console.log(Buffer.from("48656c6c6f20576f726c64", 'hex').toString('utf-8'))
// Hello World
```

Como se puede ver en los ejemplos, con Buffer.from() se transforma un texto a Buffer. Además se le puede pasar un segundo argumento indicando en qué formato está el texto. Por ejemplo, si el texto está en formato hexadecimal:

```
const buffer = Buffer.from("48656c6c6f20576f726c64", 'hex')
```

3 Cifrado y descifrado asimétrico

Para cifrar y descifrar usando cifrado asimétrico se puede usar:

```
const crypto = require("crypto");

// Se generan las claves publica y privada:
const { publicKey, privateKey } = crypto.generateKeyPairSync("rsa",
{modulusLength: 2048});

// Texto a cifrar
const data = "Hola mundo";

// Para cifrar:
const encryptedData = crypto.publicEncrypt(
  {
    key: publicKey,
    padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
    oaepHash: "sha256",
  },
  // Se usa "Buffer.from" para transformar el texto a tipo Buffer
  Buffer.from(data)
);

// El texto cifrado está en formato binario, se puede usar "base64" o
// "hexadecimal" para tener un formato más legible:
console.log("\nTexto cifrado:\n" + encryptedData.toString("base64"));

// Para descifrar:
const decryptedData = crypto.privateDecrypt(
  {
    key: privateKey,
    padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
    oaepHash: "sha256",
  },
  encryptedData
);

// Los datos descifrados están en formato Buffer, se usa toString para
// recuperar el texto original:
```

```
console.log("\nTexto descifrado:\n" + decryptedData.toString(""));
```

Ejercicios:

1. Modifique el ejemplo anterior para que usar claves públicas y privadas generadas desde línea de comandos.
2. Haga un programa al que se le pase por línea de comandos 3 argumentos, el nombre del archivo que guarda la clave pública, el nombre de un archivo y el nombre del archivo de salida. Se debe guardar el archivo cifrado en el archivo de salida. Guarde el texto en formato base64.
3. Haga ahora un programa que descifre el archivo generado en el ejemplo anterior. Se le deben de pasar 3 argumentos, el nombre del archivo que guarda la clave privada, el nombre del archivo cifrado y el nombre del archivo en el que se guardará la salida.
4. Verifique que con los programas anteriores no se pueden cifrar archivos grandes.

4 Firma y verificación de datos

Hay situaciones en las que sólo se desea tener un método para saber si unos datos han sido modificados o alterados al viajar por la red.

Para ello se puede hacer una firma de los datos. Hay muchas formas, por ejemplo, generar el HASH de los datos y después cifrar estos datos usando la clave pública. Si se desea verificar que los datos son correctos, sólo hay que seguir el proceso inverso.

Node JS provee funciones que nos permiten firmar y verificar un mensaje:

```
const crypto = require("crypto");

// Se generan las claves publica y privada:
const { publicKey, privateKey } = crypto.generateKeyPairSync("rsa",
{modulusLength: 2048});

// Firma y verificación

// Datos a firmar
const verifiableData = "Hola mundo";

// Se genera la firma de los datos
const signature = crypto.sign("sha256", Buffer.from(verifiableData), {
  key: privateKey,
  padding: crypto.constants.RSA_PKCS1_PSS_PADDING,
});

console.log('La firma de:\n' + verifiableData + "\nes:\n" +
signature.toString("base64"));

// Se verifica la firma de los datos:
const isVerified = crypto.verify(
  "sha256",
  Buffer.from(verifiableData),
  {
    key: publicKey,
    padding: crypto.constants.RSA_PKCS1_PSS_PADDING,
  },
```

```
  signature
);

// Si isVerified es 'true', los datos no han sido modificados
console.log("\nFirma válida: " + isVerified);
```

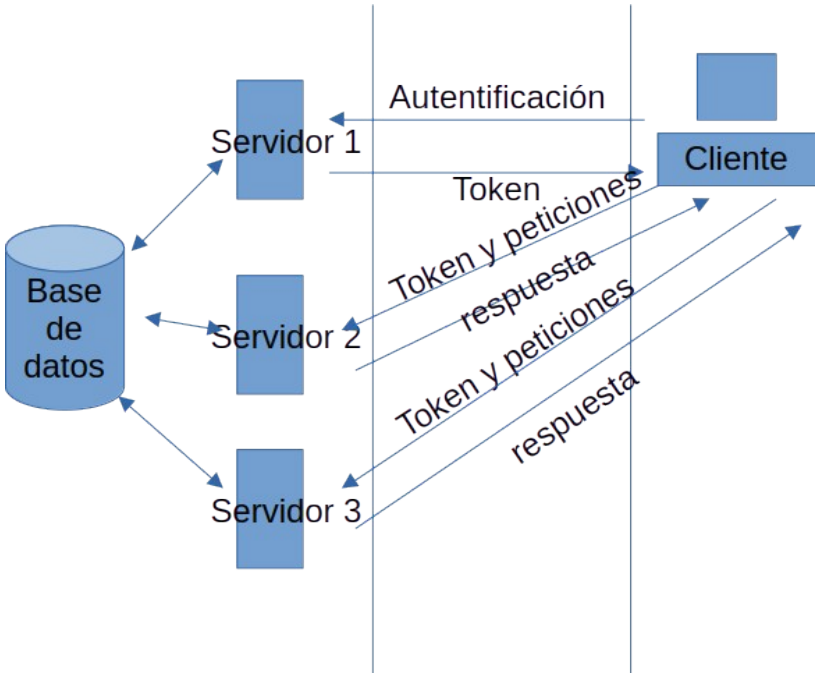
Importante: La firma se genera a parte de los datos, por lo que si se desea mandar la información y verificarla, hay que enviar los datos y la firma para que se pueda hacer la verificación.

5 Tokens

Actualmente una forma de quitar trabajo a las bases de datos es usar tokens. Al hacer un proceso de autentificación, hay que hacer consultas a la base de datos para obtener contraseñas, credenciales,… Una forma de quitar parte de este proceso es que una vez que el usuario se ha autentificado, se guarda la información que necesita la aplicación para funcionar en un JSON, por ejemplo, y después se cifra dicha información usando algún mecanismo simétrico o asimétrico. Al estar cifrada la información, el cliente recibe un paquete de datos, que no puede descifrar al no tener las claves.

El cliente en las siguientes interacciones con el servidor envía este paquete de datos al servidor. El servidor lo descifra y obtiene el JSON con los datos que le interesan. A partir de ese momento no es necesario que el servidor interactúe con la base de datos para obtener esa información.

¿Qué información se almacena en el token? Se puede almacenar información del tipo nombre de usuario, privilegios dentro de la aplicación, perfil del usuario,…



En el ejemplo de la figura se puede ver un cliente que se conecta al “Servidor 1” y realiza la autentificación. El “Servidor 1” le responde con un token. Posteriormente el cliente hace una segunda petición y envía el token. El “Servidor 2” recibe la petición, procesa el token y envía la respuesta. Normalmente no se tiene un servidor, si no una red de servidores que van procesando las peticiones según un balanceador de carga que reparte las peticiones entre ellos.

El proceso puede seguir con otras peticiones como por ejemplo el “Servidor 3”. Durante este proceso todas las peticiones que se tendrían que haber realizado a la base de datos solicitando

información sobre si el usuario está autentificado, no son necesarias pues la información va en el token.

Normalmente se suele poner una **fecha de caducidad al token**. Dentro de JSON que se cifra, se puede añadir un campo de fecha de caducidad y los servidores deben comprobar si esa fecha de caducidad ha expirado antes de proceder.

Si el usuario intenta modificar el token, el servidor no podrá descifrarlo correctamente, por lo que se le denegará el acceso. También se suele añadir una firma al token para verificar que la información no ha sido modificada.

Importante: Existen bibliotecas que sirven para generar tokens, pero no cifran la información, sólo añaden una firma a la información, por lo que la información puede ser consultada por el cliente. Habrá que verificar el funcionamiento de estas bibliotecas antes de introducir información sensible en el token.