

USO DE OPENSLL PARA CIFRADO DE MENSAJES Y FICHEROS

Laura Bezanilla Matellán

23 de noviembre de 2022

Índice

PARTE 1.....	2
1. Información sobre el uso de la herramienta OpenSSL para cifrar y descifrar ficheros	2
2. Experimentación con diferentes algoritmos y modos de cifrado.....	3
3. Concepto de entropía de Shanon. Obtener la entropía de los ficheros anteriores	6
4. Cifrar una imagen en formato BMP usando AES con modos ECB y CBC	8
 PARTE 2.....	 10
1. Puertos a nivel de transporte que utiliza un servidor público para brindar un servicio web.....	10
2. Analizar el certificado digital presente en el servidor web de la UVa.....	13
a. Protocolos criptográficos y versión utilizado a nivel de transporte	13
b. Algoritmo de criptografía asimétrica utilizado. Longitud de la clave pública	14
c. Indica la clave pública presente en el certificado digital. ¿Por qué se utiliza esa?	14
d. Algoritmo de criptografía simétrica utilizado. Longitud de la clave privada	16
e. Algoritmo de firma digital utilizado en el certificado digital	16
3. Diagrama de secuencia de la criptografía de clave pública y privada en certificados	17
4. Uso de la herramienta OpenSSL para la generación de un certificado digital de 1024 bits. Analizarlo con ssllscan y ssltest.	20

PARTE 1

1. Información sobre el uso de la herramienta OpenSSL para cifrar y descifrar ficheros

OpenSSL es un conjunto de herramientas de criptografía que sirven para hacer más segura la comunicación a través de diferentes redes. Esta biblioteca se usa principalmente para crear, convertir y gestionar certificados SSL, es decir *Secure Sockets Layer*. Actualmente es uno de los programas más utilizados dentro del área de la criptografía moderna. A continuación, se comentarán sus opciones más importantes para cifrar y descifrar ficheros:

- *version* : comprobar la version de la implementación de Openssl.

```
(thegoodhacker@kali)-[~]
$ openssl version
OpenSSL 3.0.3 3 May 2022 (Library: OpenSSL 3.0.3 3 May 2022)
```

Figura 1. Versión de OpenSSL

- *ciphers -v* : mostrar la lista con los algoritmos de cifrado disponibles.

```
(thegoodhacker@kali)-[~]
$ openssl ciphers -v
TLS_AES_256_GCM_SHA384          TLSv1.3 Kx=any
TLS_CHACHA20_POLY1305_SHA256   TLSv1.3 Kx=any
TLS_AES_128_GCM_SHA256         TLSv1.3 Kx=any
ECDHE-ECDSA-AES256-GCM-SHA384   TLSv1.2 Kx=ECDH
ECDHE-RSA-AES256-GCM-SHA384     TLSv1.2 Kx=ECDH
DHE-DSS-AES256-GCM-SHA384      TLSv1.2 Kx=DH
```

Figura 2. Algoritmos de cifrado disponibles

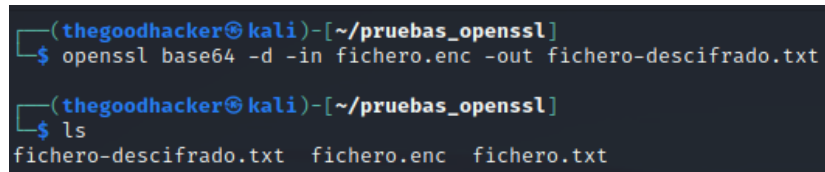
- *-in <nombre_fichero>*: nombre del archivo de entrada.
- *-out <nombre_fichero>*: nombre del archivo de salida.
- *genrsa* : generación de la clave privada RSA.
- *rsa* : gestión de las claves RSA.
- *req* : gestión de solicitudes de firma de certificados (CSR).
- *verify* : verificación de certificados X.509.
- *-e* : encriptar los datos de entrada. El comportamiento por defecto de la herramienta OpenSSL es codificar, así que este parámetro no haría falta.

```
(thegoodhacker@kali)-[~/pruebas_openssl]
$ openssl base64 -in fichero.txt -out fichero.enc

(thegoodhacker@kali)-[~/pruebas_openssl]
$ cat fichero.enc
aG9sYSBtdW5kbyBwZXF1Zc0xb3MgaGFja2VycyEK
```

Figura 3. Cifrar un fichero de prueba

- **-d** : descriptar los datos del fichero de entrada.



```
(thegoodhacker@kali)-[~/pruebas_openssl]
$ openssl base64 -d -in fichero.enc -out fichero-descifrado.txt

(thegoodhacker@kali)-[~/pruebas_openssl]
$ ls
fichero-descifrado.txt  fichero.enc  fichero.txt
```

Figura 4. Descifrar fichero de prueba

2. Experimentación con diferentes algoritmos y modos de cifrado

Todos los algoritmos utilizados en esta sección son de cifrado simétrico, es decir, utilizan la misma clave tanto para el cifrado como para el descifrado de la información. A continuación, se hará una breve explicación de cada uno de ellos.

- **AES**: Se basa en el cifrado por bloques. Se manejan bloques de 128 bits con soporte para claves de diferente longitud, como pueden ser 128, 192 o 256 bits. Este algoritmo es seguro, rápido y eficiente, siendo así el algoritmo de cifrado por bloques más utilizado en la actualidad. El nivel de seguridad es tan alto que lo utiliza la Agencia de Seguridad Nacional de Estados Unidos (NSA) en sus documentos confidenciales.
- **DES**: Utiliza un tamaño de bloque de 64 bits con una clave de la misma longitud. Sin embargo, sólo 56 bits son usados por el algoritmo. Los ocho restantes se utilizarán únicamente para comprobar la paridad. Durante el proceso se realizan 16 iteraciones diferentes. Para cifrar se usa una función matemática llamada F de Feistel. Tiene la desventaja de que la clave es demasiado corta, por lo que se pueden llevar a cabo ataques por fuerza bruta.
- **CAMELLIA**: De la misma forma que el algoritmo DES, también utiliza la función de Feistel con 18 rondas (claves con una longitud de 128 bits) o 24 rondas (claves de 192 o 256 bits) dependiendo del tamaño de la clave. Cada seis rondas se aplica una capa de transformación lógica: la función FL.

Por otro lado, se usarán diferentes modos de cifrado a lo largo del apartado. Algunos de ellos son:

- **ECB**: Es el modo de cifrado más simple, en el cual el mensaje es dividido en bloques de tamaño fijo para que cada uno de ellos se cifre de manera separada. Su desventaja es que bloques idénticos del mensaje sin cifrar producirán textos cifrados idénticos. Esto hace posible los ataques basados en diccionario, por lo que no es muy recomendable su uso para protocolos criptográficos. Sin embargo, tiene la ventaja de poder cifrar los bloques en paralelo.

- **CBC:** Este modo de cifrado es una extensión de ECB que añade cierta seguridad. En este modo, antes de ser cifrado, a cada bloque se le aplica una operación XOR con el previo bloque ya cifrado. De esta forma, cada bloque cifrado depende de todos los bloques de texto claro usados hasta ese punto.
- **CFB:** Este modo es análogo al anterior, solo que en lugar de cifrar el texto plano y hacer la operación XOR con el mensaje anterior, lo que hace es cifrar el mensaje anterior y después hacer el XOR con el texto plano.

A continuación, se presentan los diferentes ficheros que se utilizarán durante los próximos ejemplos:

- **Fichero de texto aleatorio:** Generado por medio de un script en Python y la redirección por la línea de comandos de Kali Linux. Se obtendrá un fichero con números aleatorios del 1 al 100, con un incremento de 2 en 2, separados por un espacio.
- **Fichero binario:** Obtenido de la red.

El script para generar dicho fichero de números aleatorios se muestra en la siguiente imagen:

```
import random

def generaFichero(tamaño):
    for i in range(100):
        print(random.randrange(1, 100, 2), end=" ")

    return None

generaFichero(100)
```

Figura 5. Script en Python para generar un fichero de texto aleatorio

Para obtener ese fichero se utiliza la redirección de la salida de los comandos que nos ofrece el sistema operativo Kali Linux de la siguiente forma.

```
(thegoodhacker@kali)-[~/ejer2]
$ python random-text.py > fichero-aleatorio
```

Figura 6. Obtener fichero de texto aleatorio

Debido a que son muchas combinaciones de algoritmos con sus respectivos modos de cifrado, se ha creado un Shell script que escribe en un fichero los resultados de tiempo en ejecución de cada algoritmo. Esto tiene como objetivo automatizar el proceso de cifrado y descifrado de los distintos ficheros. A continuación, se muestra su código fuente.

```
#!/bin/bash

#
# Descripcion: Obtener resultados del tiempo de ejecucion de los distintos algoritmos
#

Passwords="P4ssWo-rd0"

AlgCifrado=( "aes-256" "des" "camellia-256")
ModoCifrado=( "ecb" "cbc" "cfb")

mkdir -p encriptacion/$1

exec 3>&1 4>&2 >encriptacion/$1/resultados_tiempo 2>&1

for i in "${AlgCifrado[@]}"
do
    for j in "${ModoCifrado[@]}"
    do
        echo
        echo -e $1 $i-$j "\n"
        echo "Cifrado:"
        time openssl $i-$j -pbkdf2 -in $1 -out encriptacion/$1/$i-$j -k ${Passwords}
        echo -e "\n"
        echo "Descifrado:"
        time openssl $i-$j -pbkdf2 -d -in encriptacion/$1/$i-$j -out $1 -k ${Passwords}
        echo -e "\n"
    done
done

# Reestablecer valores de stdout and stderr
exec 1>&3 2>&4
```

Figura 7. Shell script utilizado para cifrar y descifrar los diferentes ficheros de entrada

Este script creará un nuevo directorio donde almacenará los ficheros de salida encriptados con los distintos algoritmos y los tres modos de cifrado. Además, se creará un fichero con los tiempos de ejecución de cada algoritmo gracias al comando *time*. A continuación, se muestra un ejemplo con el fichero aleatorio creado anteriormente.

```
(thegoodhacker@kali)~[/ej2]
$ ./cifrar.sh fichero-aleatorio

(thegoodhacker@kali)~[/ej2]
$ ls
binario  cifrar.sh  encriptacion  fichero-aleatorio  random-text.py

(thegoodhacker@kali)~[/ej2]
$ cd encriptacion/fichero-aleatorio/

(thegoodhacker@kali)~[/ej2/encriptacion/fichero-aleatorio]
$ ls
fichero-aleatorio.aes-256-cbc  fichero-aleatorio.camellia-256-cfb  fichero-aleatorio.des-ecb
fichero-aleatorio.aes-256-cfb  fichero-aleatorio.camellia-256-ecb  resultados_tiempo
fichero-aleatorio.aes-256-ecb  fichero-aleatorio.des-cbc
fichero-aleatorio.camellia-256-cbc  fichero-aleatorio.des-cfb
```

Figura 8. Ejemplo de ejecución del script

Una vez ejecutado este script se puede comprobar el contenido del fichero que contiene los distintos tiempos de ejecución para cada algoritmo.

```
(thegoodhacker@kali)-[~/ejer2/encriptacion/fichero-aleatorio]
$ cat resultados_tiempo

fichero-aleatorio aes-256-ecb

Cifrado:

real    0m0.033s
user    0m0.032s
sys     0m0.000s

Descifrado:

real    0m0.021s
user    0m0.017s
sys     0m0.004s
```

Figura 9. Ejemplo de tiempo de ejecución para el algoritmo AES

```
fichero-aleatorio des-cbc

Cifrado:

real    0m0.020s
user    0m0.016s
sys     0m0.004s

Descifrado:

real    0m0.020s
user    0m0.016s
sys     0m0.004s
```

Figura 10. Ejemplo de tiempo de ejecución para el algoritmo DES

```
fichero-aleatorio camellia-256-cfb

Cifrado:

real    0m0.032s
user    0m0.028s
sys     0m0.004s

Descifrado:

real    0m0.033s
user    0m0.033s
sys     0m0.000s
```

Figura 11. Ejemplo de tiempo de ejecución para el algoritmo CAMELLIA

Los resultados obtenidos del tiempo de ejecución de cada algoritmo se dividen en tres tiempos diferentes:

- **Real:** Tiempo transcurrido entre la ejecución y la finalización del proceso.
- **User:** Tiempo de CPU gastado en código de modo usuario.
- **Sys:** Tiempo de CPU que transcurre en el núcleo.

3. Concepto de entropía de Shanon. Obtener la entropía de los ficheros anteriores

En el ámbito de la Teoría de la Información la entropía, también llamada la entropía de Shannon, es una medida de incertidumbre asociada con variables aleatorias. Dicha entropía se usa para poder reducir o eliminar la incertidumbre.

La entropía, definida por Shannon, hace referencia a la cantidad media de información que contiene una variable aleatoria.

Después de investigar su funcionamiento, se ha creado un script en Python para poder hacer el cálculo de la entropía de los bytes de los ficheros del apartado anterior, el cual recibe el nombre del fichero como argumento e imprime por la salida estándar el resultado de la entropía.

```
#
# Descripcion: Calcula la entropia de Shanon del fichero pasado como argumento argv[1]
#
import math

nombreFichero = "fichero-aleatorio"

with open(nombreFichero, "rb") as fichero:
    contadores = {byte: 0 for byte in range(2 ** 8)} # Inicializamos los contadores con ceros

    for byte in fichero.read(): # Leemos por trozos un archivo grande
        contadores[byte] += 1 # Aunmetamos el contador para el byte especificado

    ficheroSize = fichero.tell() # Obtenemos el tamaño del archivo leyendo la posicion actual

    probabilidades = [contador / ficheroSize for contador in contadores.values()] # Calcular las probabilidades de cada byte

    entropia = -sum(probabilidad * math.log2(probabilidad) for probabilidad in probabilidades if probabilidad > 0) # Suma final

    print("\nLa entropia del fichero es: " + str(entropia) + " por byte.")
```

Figura 12. Script en Python utilizado para el cálculo de la entropía

Para comprobar que este script está realizando correctamente los cálculos, se va a utilizar la herramienta *ent* que proporciona Linux. Su funcionalidad es calcular la entropía de un fichero, es decir la densidad de información del contenido del archivo, expresada en número de bits.

```
(thegoodhacker@kali)-[~/ejer3]
$ ent fichero-aleatorio
Entropy = 2.850395 bits per byte.

Optimum compression would reduce the size
of this 287 byte file by 64 percent.

Chi square distribution for 287 samples is 13129.36, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 45.5296 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is -0.519116 (totally uncorrelated = 0.0).
```

Figura 13. Cálculo de la entropía del fichero aleatorio descifrado con la herramienta *ent*

A continuación, se muestra como el script que se ha creado anteriormente realiza el cálculo obteniendo el mismo resultado.

```
(thegoodhacker@kali)-[~/ejer3]
$ python entropia.py

La entropía del fichero es: 2.8503952230654184 por byte.
```

Figura 14. Cálculo de la entropía con el Shell script

Para no alargar mucho el informe, se va a hacer como un último ejemplo el cálculo de la entropía de una imagen PNG.

```
(thegoodhacker@kali)-[~/ejer2]
$ python entropia.py

La entropía del fichero es: 7.956165781549262 por byte.

(thegoodhacker@kali)-[~/ejer2]
$ ent binario
Entropy = 7.956166 bits per byte.

Optimum compression would reduce the size
of this 57058 byte file by 0 percent.

Chi square distribution for 57058 samples is 5782.08, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 125.9841 (127.5 = random).
Monte Carlo value for Pi is 3.165422232 (error 0.76 percent).
Serial correlation coefficient is 0.049948 (totally uncorrelated = 0.0).
```

Figura 15. Cálculo de la entropía de una imagen

A diferencia del resultado obtenido en la *Figura 13*, un valor pequeño, en este caso no sucede. Este resultado indica que la imagen es extremadamente densa en información, esencialmente aleatoria. Por lo tanto, es poco probable que la compresión del archivo reduzca su tamaño.

Sin embargo, en el ejemplo anterior el valor de la entropía es bajo. Esto significa que una compresión optima reduciría el tamaño de ese archivo de 287 bytes en un 64%.

4. Cifrar una imagen en formato BMP usando AES con modos ECB y CBC

La imagen en formato BMP que se va a utilizar en este ejercicio se adjunta a continuación.

ENCRYPTED?

Figura 16. Imagen en formato BMP

Para realizar todo el procedimiento descrito en el enunciado de esta sección, se ha creado un Shell script para poder simplificar el proceso. El código fuente se muestra en la siguiente imagen.


```
#!/bin/bash

#
# Descripción: Cifrar una imagen en formato BMP usando AES con modos ECB y CBC
#              Sustituir las cabeceras del fichero cifrado por la cabecera
#              del fichero original.
#

# Obtener la cabecera (54 bytes) del fichero original
head -c 54 image.bmp > header.txt

# Obtener el cuerpo del fichero original
tail -c +55 image.bmp > body.bin

# Cifrar la imagen usando AES con los modos ECB y CBC
openssl enc -aes-128-ecb -pbkdf2 -nosalt -k password -in body.bin -out body.ecb.bin
openssl enc -aes-128-cbc -pbkdf2 -nosalt -k password -in body.bin -out body.cbc.bin

# Trasladar las cabeceras del fichero cifrado por la cabecera del original
cat header.txt body.ecb.bin > image.ecb.bmp
cat header.txt body.cbc.bin > image.cbc.bmp

# Eliminar todos los ficheros que no sean necesarios
rm body.bin body.ecb.bin body.cbc.bin header.txt
```

Figura 17. Código fuente del Shell script

Una vez se tiene el código listo, hay que otorgarle permisos de ejecución al usuario para que pueda ejecutar dicho fichero. Esto se puede hacer con el comando *chmod* como se muestra a continuación.

```
(thegoodhacker@kali)-[~/ejer4]
$ ls
cifrado-imagen.sh  image.bmp

(thegoodhacker@kali)-[~/ejer4]
$ chmod 700 cifrado-imagen.sh
```

Figura 18. Otorgar permisos de ejecución

En este punto ya estaría todo preparado para poder ejecutar el script con el objetivo de obtener dos imágenes totalmente diferentes.

```
(thegoodhacker@kali)-[~/ejer4]
$ ./cifrado-imagen.sh

(thegoodhacker@kali)-[~/ejer4]
$ ls
cifrado-imagen.sh  image.bmp  image.cbc.bmp  image.ecb.bmp
```

Figura 19. Ejecución del script

Una vez ejecutado el script, tras examinar los resultados obtenidos se puede apreciar como en el caso del modo de cifrado ECB, el cual no tiene en cuenta el conjunto de bloques, sino que cifra cada uno independientemente se puede apreciar lo siguiente. El mensaje de la

imagen se puede visualizar a pesar de que los colores hayan cambiado. Esto se debe a que los bloques no han sido cifrados encadenadamente.

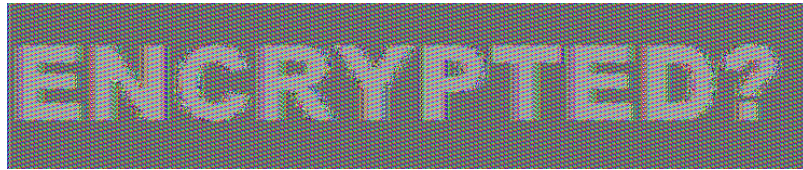


Figura 20. Imagen cifrada con el modo ECB

Por otro lado, en el caso del cifrado con el modo CBC, el cual encadena los bloques, esto no sucede. En otras palabras, el mensaje queda completamente ilegible como se muestra en la siguiente imagen.



Figura 21. Imagen cifrada con el modo CBC

En conclusión, después de realizar todo el proceso y estudiar los resultados obtenidos, se puede decir que el modo de cifrado ECB es vulnerable cuando los bloques se repiten un gran número de veces, como sucede en el caso de las imágenes.

PARTE 2

1. Puertos a nivel de transporte que utiliza un servidor público para brindar un servicio web

Si se trata de un servidor web, eso significa que al menos presta el servicio web. Para que un servidor sea capaz de prestar un servicio es necesario que tenga algún puerto a la escucha. En este caso, mínimo va a tener el puerto 80/TCP que es el que está relacionado con el protocolo HTTP. Eso quiere decir que si se utiliza una herramienta como *nmap*, en un servidor público, para escanear el puerto 80 debería afirmar que dicho puerto está abierto.

Para demostrar que esa hipótesis es cierta, en una máquina con Kali Linux, se utiliza la aplicación *nmap* para escanear el puerto 80/TCP del servidor web público de la UVa. Para saber únicamente el estado de dicho puerto se especifica con el parámetro *-p*.

```
(thegoodhacker@kali)-[~]
$ nmap www.uva.es -p 80
Starting Nmap 7.92 ( https://nmap.org ) at 2022-11-11 13:13 EST
Nmap scan report for www.uva.es (157.88.25.8)
Host is up (0.014s latency).
rDNS record for 157.88.25.8: sostenibilidad.uva.es

PORT      STATE SERVICE
80/tcp    open  http

Nmap done: 1 IP address (1 host up) scanned in 0.21 seconds
```

Figura 22. Escaneo puerto 80/TCP del servidor web de la UVa

Uno de los problemas de HTTP es que es un protocolo sin estado, por eso es necesario la existencia de las *cookies*. Por otro lado, otro inconveniente es que la comunicación no va cifrada.

Entonces, ¿cómo se hace para que la comunicación entre el cliente y el servidor web esté cifrada? En Internet para proteger la información se utiliza el protocolo HTTPS, cuyo puerto por defecto es el 443/TCP. Sin embargo, esto tiene como objetivo que una persona que no tenga un perfil tecnológico no tenga por qué saber la existencia de este protocolo para poder navegar de forma segura.

De la misma forma que se hizo anteriormente, se va a verificar si el STIC de la UVa ha hecho bien su trabajo comprobando el estado del puerto 443.

```
(thegoodhacker@kali)-[~]
$ nmap www.uva.es -p 80,443
Starting Nmap 7.92 ( https://nmap.org ) at 2022-11-11 13:17 EST
Nmap scan report for www.uva.es (157.88.25.8)
Host is up (0.014s latency).
rDNS record for 157.88.25.8: sostenibilidad.uva.es

PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https

Nmap done: 1 IP address (1 host up) scanned in 0.52 seconds
```

Figura 23. Escaneo puerto 443/TCP del servidor web de la UVa

Como se puede ver en la *Figura 23* ambos puertos están abiertos. Esto es debido a que esa dirección pertenece a un servidor web. Para aplicar esta capa de seguridad, el servidor cuando detecta una petición insegura la va a redireccionar de HTTP en el puerto 80/TCP a HTTPS en el puerto 443/TCP. El código HTTP que provoca esta redirección es el 302.

En este punto la gente se puede preguntar, ¿qué tiene el servidor para poder proporcionar dicha capa de seguridad? Para poder saberlo se va a examinar en el servidor web de la UVa.

A lo largo de esta comprobación se va a utilizar el navegador web Brave. Se pulsará F12 para acceder a las opciones de desarrollador, para poder seleccionar el apartado de red. El siguiente paso sería buscar la dirección www.uva.es. Como se puede observar el servidor automáticamente ha puesto un candado al lado de la dirección.



Figura 24. Cambio de HTTP a HTTPS

A continuación, se comprueban las peticiones con sus respectivos estados. Se selecciona la petición cuyo estado es el 302, que es el código de respuesta HTTP cada vez que redirecciona una petición. Como se puede observar, en primer lugar se ha hecho una petición HTTP al servidor web de la UVa.

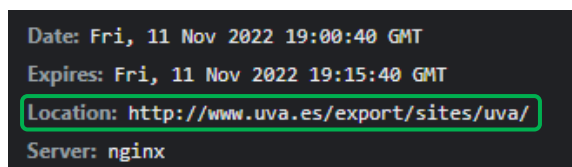


Figura 25. Petición HTTP

El servidor se da cuenta que es una conexión insegura y hace una redirección 302. Esto significa que el servidor ha tenido que aplicar una redirección para esa petición que se ha hecho por defecto al puerto 80, la cual no es segura.

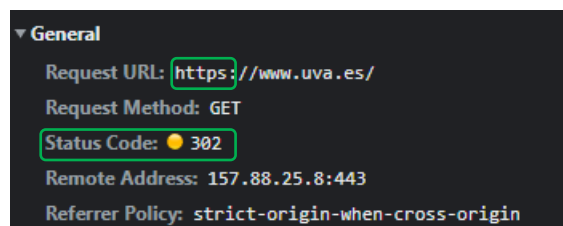


Figura 26. Verificación de la redirección

2. Analizar el certificado digital presente en el servidor web de la UVa

a. Protocolos criptográficos y versión utilizado a nivel de transporte

Para ver que protocolos criptográficos utiliza a nivel de transporte, una vez dentro de la página web de la Uva, se presiona las teclas FN + F12 para poder entrar en el menú de desarrollador del navegador Brave. En dicho menú hay que ir a la pestaña *Security* donde se podrán ver todos los datos de la conexión HTTPS que se ha establecido.

Aquí se puede ver como la conexión es segura, ya que utiliza el protocolo criptográfico TLS, *Transport Layer Security*, con la versión 1.2. Esto demuestra que esta página no tiene la mejor seguridad posible, ya que desde el año 2018 existe la versión 1.3.

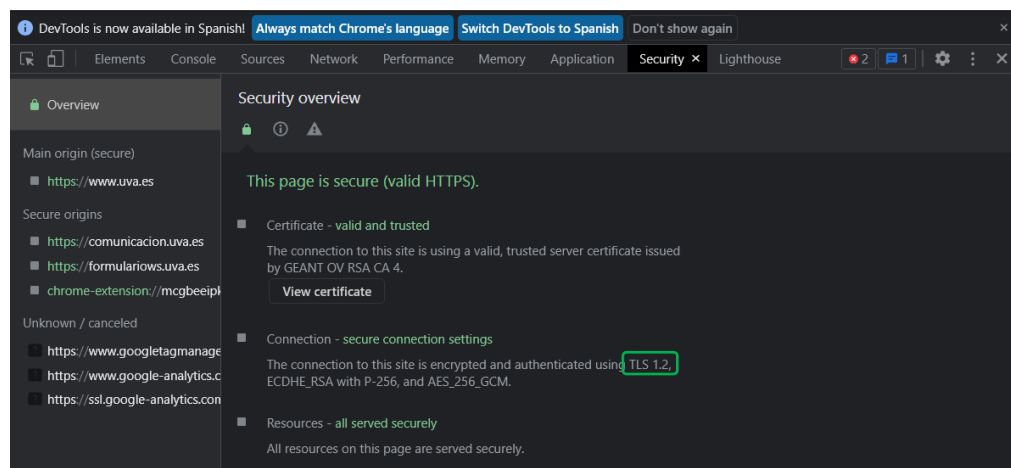


Figura 27. Comprobación del protocolo criptográfico a nivel de transporte

Otra forma de comprobarlo es utilizando una herramienta online gratuita para poder realizar un análisis profundo de la configuración de cualquier servidor web SSL en Internet.

Protocols		
TLS 1.3		No
TLS 1.2		Yes
TLS 1.1		Yes
TLS 1.0		Yes
SSL 3		No
SSL 2		No

Figura 28. Comprobación de los protocolos a nivel de transporte con la herramienta SSLTest

Por otro lado, la herramienta *ssls* que está disponible en Kali Linux, ofrece una información muy parecida a la anterior. En este caso se especifica que versiones de los protocolos TLS y SSL, cuyo significado es *Secure Sockets Layer*, están disponibles en esta página web.

```
Testing SSL server www.uva.es on port 443 using SNI name www.uva.es

SSL/TLS Protocols:
SSLv2      disabled
SSLv3      disabled
TLSv1.0    enabled
TLSv1.1    enabled
TLSv1.2    enabled
TLSv1.3    disabled
```

Figura 29. Comprobación de los protocolos a nivel de transporte con la herramienta *SSLScan*

b. Algoritmo de criptografía asimétrica utilizado. Longitud de la clave pública

Como se puede ver en las siguientes imágenes el algoritmo de criptografía asimétrica de clave pública es RSA, con una longitud de la clave pública de 2048 bits.

Algoritmo de clave pública de la entidad receptora

Clave pública de la entidad receptora

Extensiones

ID de clave de la entidad emisora de certificados

ID de clave de la entidad receptora del certificado

Valor de campo

PKCS #1 con cifrado RSA

Figura 30. Algoritmo de criptografía asimétrica

RSA Key Strength: 2048

Figura 31. Longitud de la clave pública

c. Indica la clave pública presente en el certificado digital. ¿Por qué se utiliza esa?

La clave pública está presente, como se puede ver en la *Figura 32*, dentro del certificado en el apartado Detalles en el navegador Brave.

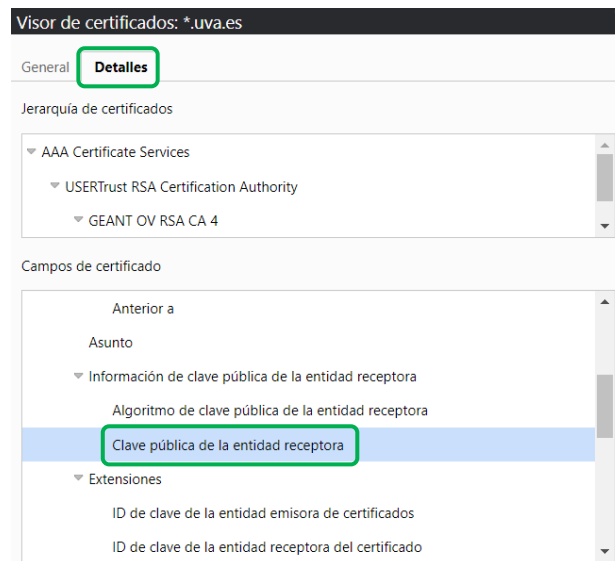


Figura 32. Localización de la clave pública en el certificado digital

Otra forma de verlo es utilizando la aplicación *sslsan* con la opción *–show-certificate* para obtener más información del certificado. En la siguiente imagen se puede observar tanto el módulo como el exponente de la clave pública.

```
RSA Public-Key: (2048 bit)
Modulus:
 00:e4:dc:6a:0e:f7:ce:8d:30:55:b0:70:77:79:92:
 0e:95:47:b8:cc:27:f0:08:e0:bf:47:a3:48:36:16:
 d1:82:17:bc:0c:c8:2b:c2:bb:0b:45:54:61:23:fe:
 0a:f3:b9:9f:d7:b2:9b:d0:d5:ed:9a:05:2c:b4:fb:
 14:b7:47:ab:b5:99:04:a4:5a:17:ab:65:4d:6d:88:
 67:b3:71:33:4a:c0:a5:cf:c3:03:53:7b:2a:d5:49:
 44:ca:f2:d0:cc:62:75:6e:0a:09:a6:35:e2:22:c8:
 2c:2f:2e:c7:7e:37:7f:3c:76:a1:37:44:c8:1c:66:
 ee:cd:b9:88:94:82:a5:5a:07:8e:c8:2e:a5:fe:b5:
 6d:94:22:99:a4:7b:b8:73:9c:19:ce:c2:d5:98:86:
 e8:d5:d9:8c:2d:4f:60:c1:e6:f1:b9:cd:41:f3:94:
 56:86:de:37:b9:fc:68:3d:8f:58:6c:bc:79:08:66:
 69:ef:47:3e:a4:3c:d6:0c:d7:a0:48:c5:41:80:20:
 13:f3:50:22:f9:dc:08:fc:47:bb:94:e3:22:4d:e6:
 96:14:35:ee:fe:66:20:88:e9:a7:45:cc:40:58:ab:
 d7:f8:da:f4:11:51:6c:aa:5f:1b:dd:7a:c4:db:1c:
 a4:ce:ce:7b:1a:70:08:ff:4b:4a:cc:68:c1:63:63:
 49:35
Exponent: 65537 (0x10001)
```

Figura 33. Clave pública del certificado

El algoritmo de criptografía asimétrico RSA permite tanto cifrar la información (proporcionando confidencialidad), como firmarla (proporcionando autenticidad e integridad).

Es posible cifrar un mensaje M mediante la clave pública (n, e) del destinatario utilizando operaciones de la aritmética modular. Este algoritmo, antes de cifrar, efectúa un cambio de base en el mensaje. Esto se debe principalmente a que RSA sólo trabaja con números.

Cuando se habla de criptografía moderna, las operaciones modulares se realizan con números muy grandes, siendo común el uso de claves de 2048 bits, algo que ralentiza bastante dichas operaciones. Una manera de optimizar ese cómputo es hacer uso del algoritmo de exponenciación rápida, el cual proporciona una manera de calcular grandes potencias de forma rápida.

Por este hecho, en los certificados digitales se utiliza como exponente el número 65537 como se muestra en la *Figura 33*. Esto se debe a que es un número con sólo dos unos y el resto son todos ceros, lo que facilita muchísimo los cálculos de la aritmética modular.

d. Algoritmo de criptografía simétrica utilizado. Longitud de la clave privada

El algoritmo de criptografía simétrica utilizado en este certificado se puede comprobar en el navegador con las opciones de desarrollador, dentro del apartado Seguridad.

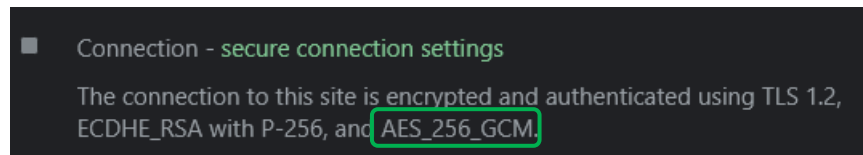


Figura 34. Algoritmo de criptografía simétrica utilizado en el certificado

Por lo tanto, la longitud de la clave privada sería 256 bits. El cifrado de 256 bits hasta día de hoy sigue siendo inexpugnable. Esto se debe a que intentar romper una clave así de larga por fuerza bruta requiere mucha más potencia de cómputo. En este caso, se está hablando de que existirían 2^{256} posibles valores para la clave.

e. Algoritmo de firma digital utilizado en el certificado digital

Gracias a la herramienta *ssllscan* se puede comprobar cómo se está utilizando el algoritmo SHA-384 para firmar el certificado, como se muestra en la siguiente imagen.


```
SSL Certificate:
Signature Algorithm: sha384WithRSAEncryption
RSA Key Strength: 2048
```

Figura 35. Algoritmo de firma digital

3. Diagrama de secuencia de la criptografía de clave pública y privada en certificados

¿Qué es lo que le va a proporcionar el servidor web al cliente para que la información vaya protegida? La respuesta es muy fácil, un certificado digital.

En seguridad informática un certificado digital sirve para proporcionar confidencialidad, integridad, disponibilidad, autenticidad y no repudio. En otras palabras, sirve para verificar que el servidor es quien dice ser. Además, asegura que la información que fluye entre cliente y servidor va a ser protegida. Por otro lado, si se tiene un certificado nunca se va a poder negar que has sido tú el que ha mandado la información que se va a utilizar para que la comunicación sea segura.

Cuando se hace una redirección, el servidor le tiene que mandar al cliente su certificado digital. Para poder ver dicho certificado en un navegador se hace click en el candado y posteriormente en “El certificado es válido”.

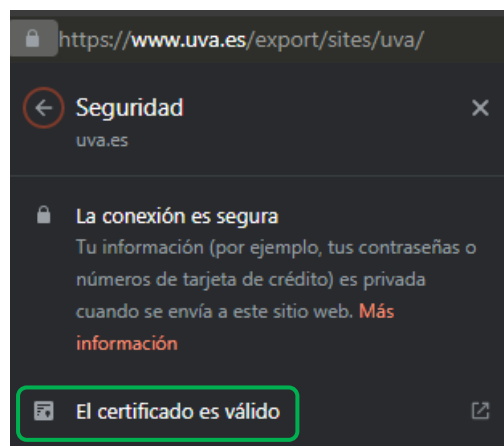


Figura 36. Comprobación del certificado

Un certificado digital contiene diferentes datos como por ejemplo, a quien pertenece o quien es la autoridad oficial que ha firmado dicho certificado.

Visor de certificados: *.uva.es	
General	Detalles
Enviado a	
Nombre común (CN)	*.uva.es
Organización (O)	Universidad de Valladolid
Unidad organizativa (OU)	<No incluido en el certificado>
Emitido por	
Nombre común (CN)	GEANT OV RSA CA 4
Organización (O)	GEANT Vereniging
Unidad organizativa (OU)	<No incluido en el certificado>
Período de validez	
Emitido el	miércoles, 4 de mayo de 2022, 2:00:00
Vencimiento el	viernes, 5 de mayo de 2023, 1:59:59
Huellas digitales	
Huella digital SHA-256	CD FD D1 AE 13 E2 66 78 8B 70 F8 52 2A CE 03 D7 E5 6C B3 53 94 5A 02 E2 B7 DD 6E 2D 52 21 20 25
Huella digital SHA-1	C2 7E 68 22 21 86 C6 EC BA 3E 54 51 01 D5 21 A4 30 A9 E2 6F

Figura 37. Certificado digital de la UVa

En el certificado digital aparecen temas relacionados con la criptografía. Por un lado, aparecen el termino RSA y por otro lado aparece SHA-256.

Cuando una página web proporciona su certificado digital aparece la clave pública relacionada con el algoritmo RSA. La clave publica de RSA sirve para cifrar la información. Únicamente la persona que tiene la clave privada puede descifrar el criptograma generado por RSA.

¿Para qué quiere el cliente la clave pública de RSA? Esto es debido a que el intercambio de claves en la red tiene que ser seguro. El cliente generará una clave privada con AES. A continuación, se tiene que garantizar que dicho intercambio es infalible. Para ello, el cliente coge el algoritmo RSA con la clave pública y la clave privada de AES para generar un criptograma cifrado. Ese criptograma se lo enviará al servidor. Posteriormente, el servidor utiliza RSA para aplicar la clave privada al criptograma, obteniendo así la clave privada de AES.

Gracias tanto a las matemáticas discretas como a la teoría de los números primos, se ha conseguido intercambiar una clave fundamental, como es la clave de cifrado y descifrado con AES, que con criptografía asimétrica es la misma, para que la puedan compartir el cliente y el servidor.

A partir de aquí cada vez que el cliente quiera intercambiar información con el servidor coge el algoritmo AES-256. Al mensaje que quiera mandar le aplica la clave privada de AES para obtener un mensaje cifrado. A continuación, se lo manda al servidor.

Posteriormente, el servidor coge AES, al criptograma se le aplica la clave privada que le ha mandado el cliente para obtener el mensaje en claro. Este proceso se repite durante toda la comunicación.

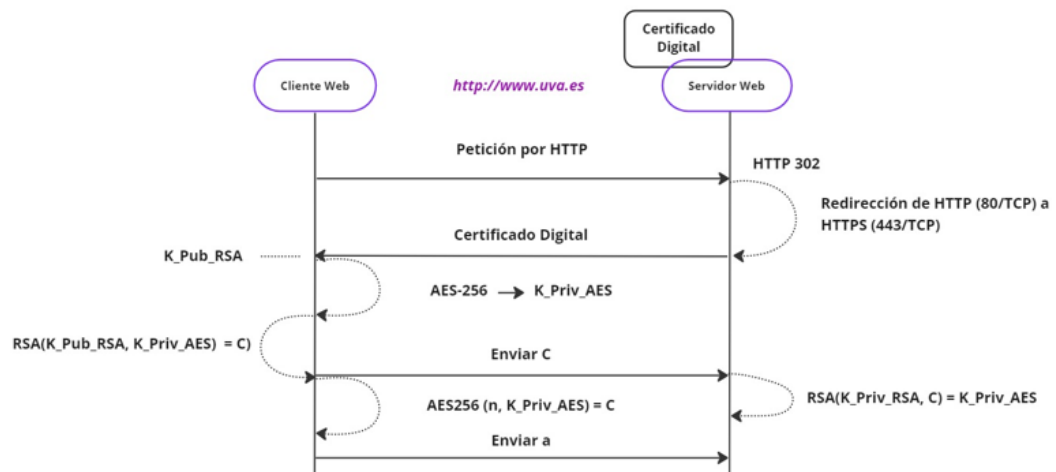


Figura 38. Diagrama de secuencia de la criptografía usada de un certificado digital

RSA es un algoritmo de cifrado asimétrico de clave pública. Esto significa que va a haber dos claves distintas, una clave para cifrar y otra para descifrar. Con este algoritmo se necesita una clave pública (n, e) y otra clave privada que va a ser (n, d).

Para cifrar, si se quiere obtener un criptograma, se coge el mensaje, se eleva a el numero e y se calcula el módulo con n . Sin embargo para descifrar y obtener el texto en claro, se coge el criptograma, se eleva al número d y se calcula el módulo con n . ¿En este proceso quién es n ? El número n es la multiplicación de p por q , donde p y q son dos números primos grandes distintos. Finalmente, la clave privada de RSA sería (p, q, d).

- **Clave Pública:** $(n, e) \longrightarrow C \equiv M^e \mod n$

- **Clave Privada:** $(n, d) \longrightarrow M \equiv C^d \mod n$

$$n = p \cdot q \quad / \quad p \wedge q \text{ son números primos grandes}$$

Figura 39. Explicación algoritmo RSA

¿Qué sentido tiene esto? Si se genera una clave con un algoritmo simétrico, la misma clave se utiliza para cifrar y descifrar el mensaje. Si se quiere que esa clave que se va a utilizar también la tenga el servidor se la tienes que mandar de forma segura. Entonces por un lado se va a utilizar RSA y por otro lado la clave pública.

El algoritmo RSA se utiliza principalmente para cifrar y descifrar mensajes. Además, otro de sus usos es para la firma digital tanto para la generación como para la verificación.

4. Uso de la herramienta OpenSSL para la generación de un certificado digital de 1024 bits. Analizarlo con ssllscan y ssltest.

Como anotación antes de empezar a explicar el procedimiento, para la generación de un certificado digital se va a usar una máquina Kali Linux que ya tiene incorporado *OpenSSL* como se puede comprobar a continuación.

```
(thegoodhacker@kali)-[~]
$ openssl
help:

Standard commands
asn1parse          ca                ciphers           cmp
cms                 crl               crl2pkcs7         dgst
dhparam            dsa               dsaparam          ec
ecparam            enc               engine            errstr
fipsinstall         gendsa           genpkey           genrsa
```

Figura 40. Comprobación para ver si la máquina tiene OpenSSL

El certificado digital que se va a crear va a tener que usar el algoritmo RSA con una clave tanto pública como privada. Por lo tanto, si alguien te roba la clave privada va a poder suplantar la identidad digital.

El primer paso sería crear un directorio nuevo dentro del directorio */opt*, el cual se usa para almacenar paquetes de software, que servirá para guardar todos los archivos que se vayan creando a lo largo de la explicación. Esta acción hay que hacerla con permisos de administrador, ya que sino no se podría escribir dentro de dicho directorio.

```
(thegoodhacker@kali)-[~]
$ sudo mkdir /opt/rsa
[sudo] password for thegoodhacker:
```

Figura 41. Creación del directorio de trabajo

```
(thegoodhacker@kali)-[~]
$ cd /opt/

(thegoodhacker@kali)-[/opt]
$ ls
containerd  microsoft  rsa
```

Figura 42. Comprobación de que se ha creado correctamente

Dentro de ese nuevo directorio lo primero que se va a generar es la clave privada de RSA, porque sin esa clave no se puede descifrar la clave privada de AES que proporciona el cliente al servidor. Esta acción hay que hacerla como *root* porque se va a escribir en un directorio que no es mío.

```
(thegoodhacker@kali)-[/opt/rsa]
$ sudo openssl genrsa -out alice.key 1024

(thegoodhacker@kali)-[/opt/rsa]
$ ls
alice.key
```

Figura 43. Generación de la clave privada de alice

Lo que hace este comando es generar con RSA la clave privada de *alice* con una longitud de 1024 bits, ya que esto es un ejemplo académico. En una situación real estas claves tendrían una longitud de 2048 o 4096 bits. Una vez se tiene creada la clave, habría que comprobar cuál es su contenido de la siguiente forma.

```
(thegoodhacker@kali)-[/opt/rsa]
$ sudo cat alice.key
-----BEGIN PRIVATE KEY-----
MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJcAgEAAoGBA0JXvLWtQJYP/TPJ
uSBz5h+K57M25QdTI0mo0biSe7bGrMf8LRVIXXcK7K2J9vNlKE8X+tE23SnPrVhX
3a1d3VJjPF+Z+G7yHsQb0mX1yU0QqilUntwhiN+PYza0S8NwvLrLZHTjjJvHR9wW
HQ2QswSIqH+bQdJJ+92BX5Hx1ttXAgMBAAECgYEAo8JGzksjR28aCrYQMuKnXhQd
WK6G1d3Y0hncaSXSTFNB/w78r0FEHjTROF1xRngRyefXmDbRnMac7CXmROB8lxqV
8mPIcexI3geLQzwgMCy4zDLwk8ip3XF2+4C8qK8ExRZWP4ODG8A0SsB0h6SbvrgV
N09IACzu0++dA0PLNkECQQD50MbV6XcfURzSjt/Vsh10ZyVhX8pLM0Q07FJWB0+s
VUsLOhnTNU8F3Ik1mRjkebKb3ivVAU0s/R2v+eoN6RUjAkeA5/IzKySTL5JhnXA2
e3dx3DgW7z80q+QQmiYjdGPYYpgITgiiXVan6sLLdGjjj5G3Z+j0KnYmMVCPSgm
WdoGPQJBANIJC7P8V7BX5TD0+TjVSTiLFm2ZOTT+9oC8LLJ6l3NViv67I2ci3U1
J5ONxdUK0PWlxRqQtKy0YyKRXoulxsCQC5d12a/eBWruIHNhcamuQcp20PWYcQ
e4ka+awYA06hZ9gcLaNG52fVwafOKXiiru3s1MWIvoJboWK0SDW4EzIxAKAK0x0p
JngE/yb0cSigZ/N/Rmh/6KZkmI2Y0AMsHHou/nxXpR/a2/uLNNy0Jay4k0Q4HquQ
ZDU2DWohadbnrFNJ
-----END PRIVATE KEY-----
```

Figura 44. Contenido de la clave privada

Otra forma de visualizar su contenido sería con OpenSSL, mandándole una petición para que muestre en un texto que se pueda comprender el contenido de la clave privada.

```
(thegoodhacker@kali)-[/opt/rsa]
$ sudo openssl rsa -text -in alice.key
Private-Key: (1024 bit, 2 primes)
modulus:
 00:e2:57:bc:b5:ad:40:96:0f:fd:33:c9:b9:20:73:
 e6:1f:8a:e7:b3:36:e5:07:53:23:49:a8:39:b8:92:
 7b:b6:c6:ac:c7:fc:2d:15:48:5d:77:0a:ec:ad:89:
 f6:f3:65:28:4f:17:fa:d1:36:dd:29:cf:ad:58:57:
 dd:ad:5d:dd:52:63:3c:5f:99:f8:6e:f2:1e:c4:1b:
 d2:65:f5:c9:43:90:42:29:54:9e:dc:21:88:df:8f:
 63:36:8e:4b:c3:70:be:5a:cb:64:74:e3:8c:9b:c7:
 47:dc:16:1d:0d:90:b3:04:88:a8:7f:9b:41:d2:49:
 fb:dd:81:5f:91:f1:d6:db:57
publicExponent: 65537 (0x10001)
privateExponent:
 00:a3:c2:46:ce:4b:23:47:6f:1a:0a:b6:10:32:e2:
 a7:c4:7a:83:58:ae:86:d5:dd:d8:d2:19:dc:69:25:
 d2:4c:53:41:ff:0e:fc:af:41:44:1e:34:d1:38:5d:
 71:46:78:11:c9:e7:d7:98:36:d1:9c:c6:9c:ec:25:
 e6:44:e0:7c:97:1a:95:f2:63:c8:71:ec:48:de:07:
 8b:43:3c:20:30:2c:b8:cc:32:f0:93:c8:a9:dd:71:
 76:fb:80:bc:a8:af:04:c5:16:56:3f:83:83:1b:c0:
 0e:4a:c0:74:87:a4:9b:be:b8:15:37:4f:48:01:c6:
 6e:d3:ef:9d:03:43:cb:36:41
```

Figura 45. Contenido extendido de la clave privada

Por un lado, aparece el exponente público que es un número de 17 bits. Por otro lado, se muestra el módulo. La fortaleza de RSA es que dado el módulo (el número n) saber que dos números primos lo han generado, desde el punto de vista computacional, es muy costoso.

Una vez se tiene la clave privada, hay que crear una solicitud de firma para dicha clave. Es decir, se va a firmar el certificado digital. La clave privada hay que especificarla con el parámetro `-key`. El fichero de salida va a tener extensión `.csr`, *Certificate Signing Request*, lo cual significa que es un fichero digital que un solicitante transmite a una entidad certificadora para obtener la firma de su certificado.

Durante el proceso se irá pidiendo información sobre la organización que quiere el certificado digital. Si hay algún campo que la empresa no quiera rellenar, se puede dejar en blanco presionando la tecla `ENTER`. En el apartado del correo electrónico hay que tener mucho cuidado porque los atacantes comprobarán si aparecen nombres de usuario en ellos.

```
(thegoodhacker@kali)-[/opt/rsa]
$ sudo openssl req -new -key alice.key -out alice.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Valladolid
Locality Name (eg, city) []:Valladolid
Organization Name (eg, company) [Internet Widgits Pty Ltd]:4ck.com
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.4ck.com
Email Address []:admin@4ck.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Figura 46. Solicitud de firma del certificado digital

En este punto se tendrían dos ficheros, uno sería el certificado firmado y otro la clave privada que se ha usado para firmarlo. Antes de generar el certificado, habría que comprobar el contenido de lo que se acaba de generar como se muestra en la *Figura 47*.

```

(thegoodhacker@kali)-[/opt/rsa]
$ ls
alice.csr  alice.key

(thegoodhacker@kali)-[/opt/rsa]
$ sudo openssl req -noout -text -in alice.csr
Certificate Request:
Data:
  Version: 1 (0x0)
  Subject: C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www.4ck.com, emailAddress = admin@4ck.com
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    Public-Key: (1024 bit)
    Modulus:
      00:e2:57:bc:b5:ad:40:96:0f:fd:33:c9:b9:20:73:
      e6:1f:8a:e7:b3:36:e5:07:53:23:49:a8:39:b8:92:
      7b:b6:c6:ac:c7:fc:2d:15:48:5d:77:0a:ec:ad:89:
      f6:f3:65:28:4f:17:fa:d1:36:dd:29:cf:ad:58:57:
      dd:ad:5d:dd:52:63:3c:5f:99:f8:6e:f2:1e:c4:1b:
      d2:65:f5:c9:43:90:42:29:54:9e:dc:21:88:df:8f:
      63:36:8e:4b:c3:70:be:5a:cb:64:74:e3:8c:9b:c7:
      47:dc:16:1d:0d:90:b3:04:88:a8:7f:9b:41:d2:49:
      fb:dd:81:5f:91:f1:d6:db:57
    Exponent: 65537 (0x10001)
  Attributes:
    (none)
  Requested Extensions:
  Signature Algorithm: sha256WithRSAEncryption
  Signature Value:
    a0:40:4e:7e:17:a8:c1:8a:d3:ef:8f:00:ad:d4:c6:b1:f2:0f:
    f0:7a:b9:fc:9a:9b:e8:b0:cf:46:9a:b3:53:3d:9f:b4:17:1c:
    bd:f4:a9:d7:fa:93:3e:3d:b4:89:f3:ac:3b:17:1c:8d:44:83:
    36:b3:f2:53:fc:f1:f2:af:bf:bb:c9:d6:45:c4:2e:36:a6:67:
    45:07:64:df:03:26:b9:29:87:61:82:02:e1:27:28:22:9d:77:
    f4:a8:06:f1:b1:6b:8f:47:ad:38:8a:7e:14:31:d7:d8:ab:88:
    0e:12:8d:45:93:53:f9:4b:70:72:05:69:b7:f4:91:cb:a3:14:
    f8:3a

```

Figura 47. Contenido del fichero alice.csr

Finalmente, habría que generar el certificado compatible con el estándar x509. Para ello con la herramienta OpenSSL hay que mandar una solicitud de compatibilidad, como se muestra a continuación, especificando para cuantos días va a ser válido este certificado. El fichero de salida va a tener la extensión .crt de certificado.

```

(thegoodhacker@kali)-[/opt/rsa]
$ sudo openssl x509 -req -days 365 -in alice.csr -signkey alice.key -out alice.crt
Certificate request self-signature ok
subject=C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www.4ck.com, emailAddress = admin@4ck.com

(thegoodhacker@kali)-[/opt/rsa]
$ ls
alice.crt  alice.csr  alice.key

```

Figura 48. Generación de certificado compatible con el estándar x509

Una vez que se han generado todos los ficheros necesarios, hay que decidir cuales se dejan a disposición del público y cuales habría que guardar. El certificado digital se va a mover al directorio /etc/ssl/certs/ que es el directorio por defecto en Linux donde se almacenan los certificados digitales. Por otro lado, la clave pública se va a guardar en /etc/ssl/private.

```

(thegoodhacker@kali)-[/opt/rsa]
$ sudo cp alice.crt /etc/ssl/certs/

(thegoodhacker@kali)-[/opt/rsa]
$ sudo cp alice.key /etc/ssl/private/

```

Figura 49. Guardar los ficheros generados

En este punto, se van a analizar las propiedades de los directorios `/etc/ssl/`. Por un lado, el directorio donde se almacenan las claves privadas el único que tiene permiso de lectura es el usuario `root`. Sin embargo, el directorio donde se almacenan los certificados digitales cualquiera puede leerlos.

```
(thegoodhacker@kali)-[/opt/rsa]
$ sudo ls -l /etc/ssl/
total 68
drwxr-xr-x 3 root root    16384 Nov 10 07:23 certs
-rw-r--r-- 1 root root      653 Jul 14 04:18 kali.cnf
-rw-r--r-- 1 root root   12550 Jul 20 08:05 openssl.cnf
-rw-r--r-- 1 root root   11118 Mar 15 2022 openssl.cnf.dpkg-new
-rw-r--r-- 1 root root   12449 Jun 13 16:16 openssl.cnf.original
drwx--x-- 2 root ssl-cert 4096 Nov 10 07:23 private
```

Figura 50. Permisos del directorio `/etc/ssl`

Finalmente, para poder analizar un certificado con las herramientas `sslsn` y `ssltest`, dicho certificado tiene que estar almacenado en un servidor. Como en este caso específico esto no sucede, ya que es un ejemplo de aprendizaje, el análisis del certificado se va a realizar con la aplicación `OpenSSL`.

Para verificar los datos de una solicitud de firma (`.csr`) se va a utilizar el siguiente comando, donde se puede observar cómo verifica que todo esté correcto.

```
(thegoodhacker@kali)-[/opt/rsa]
$ sudo openssl req -text -noout -verify -in alice.csr
Certificate request self-signature verify OK
Certificate Request:
Data:
  Version: 1 (0x0)
  Subject: C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www.4ck.com, emailAddress = admin@4ck.com
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    Public-Key: (1024 bit)
    Modulus:
      00:e2:57:bc:b5:ad:40:96:0f:fd:33:c9:b9:20:73:
      e6:1f:8a:e7:b3:36:e5:07:53:23:49:a8:39:b8:92:
      7b:b6:c6:ac:c7:fc:2d:15:48:5d:77:0a:ec:ad:89:
      f6:f3:65:28:4f:17:fa:d1:36:dd:29:cf:ad:58:57:
      dd:ad:5d:dd:52:63:3c:5f:99:f0:6e:f2:1e:c4:1b:
      d2:65:f5:c9:43:90:42:29:54:9e:dc:21:88:df:8f:
      63:36:8e:4b:c3:70:be:5a:cb:64:74:e3:8c:9b:c7:
      47:dc:16:1d:0d:90:b3:04:88:a8:7f:9b:41:d2:49:
      fb:dd:81:5f:91:f1:d6:db:57
    Exponent: 65537 (0x10001)
  Attributes:
    (none)
  Requested Extensions:
  Signature Algorithm: sha256WithRSAEncryption
  Signature Value:
    a0:40:4e:7e:17:a8:c1:8a:d3:ef:8f:00:ad:d4:c6:b1:f2:0f:
    f0:7a:b9:fc:9a:9b:e8:b0:cf:46:9a:b3:53:3d:9f:b4:17:1c:
    bd:f4:a9:d7:fa:93:3e:3d:b4:89:f3:ac:3b:17:1c:8d:44:83:
    36:b3:f2:53:fc:f1:f2:af:bf:bb:c9:d6:45:c4:2e:36:a6:67:
    45:07:64:df:03:26:b9:29:87:61:82:02:e1:27:28:22:9d:77:
    f4:a8:06:f1:b1:6b:8f:47:ad:38:8a:7e:14:31:d7:d8:ab:88:
    0e:12:8d:45:93:53:f9:4b:70:72:05:69:b7:f4:91:cb:a3:14:
    f8:3a
```

Figura 51. Análisis del nuevo certificado creado