

Protocolos criptográficos

Práctica 3



Índice

[1. Generación de parámetros 3](#_Toc529264768)

[Comando 3](#_Toc529264769)

[Parámetros 3](#_Toc529264770)

[2. Generación de claves 3](#_Toc529264771)

[Comando 3](#_Toc529264772)

[3. Extracción de la clave privada 3](#_Toc529264773)

[Comando 3](#_Toc529264774)

[Clave privada 4](#_Toc529264775)

[Conclusiones 4](#_Toc529264776)

[4. Extracción de la clave pública 5](#_Toc529264777)

[Comando 5](#_Toc529264778)

[Clave pública 5](#_Toc529264779)

[5. Creamos un archivo de al menos 128 bytes 6](#_Toc529264780)

[6. Firmamos el mensaje 6](#_Toc529264781)

[Comando 6](#_Toc529264782)

[Contenido 7](#_Toc529264783)

[7. Dar un falso positivo 7](#_Toc529264784)

[Comando 8](#_Toc529264785)

[Conclusiones 8](#_Toc529264786)

[8. Generar hash 9](#_Toc529264787)

[Comando 9](#_Toc529264788)

[Archivo pedroDSApub.sha384 9](#_Toc529264789)

[9. Generar hash de 160 bits 9](#_Toc529264790)

[Comando 9](#_Toc529264791)

[Salida 9](#_Toc529264792)

[10. Firmar mensaje 2 usando dgst 9](#_Toc529264793)

[Comando 9](#_Toc529264794)

[Contenido 9](#_Toc529264795)

[11. Verificar los mensajes “mensaje” y “mensaje2” con “mensaje2.sign” 9](#_Toc529264796)

[Comando 9](#_Toc529264797)

[Explicación 10](#_Toc529264798)

[12. Verificación de “mensaje2” con “mensaje2.sign” usando la orden pkeyutl en vez de dgst 10](#_Toc529264799)

[Comando 10](#_Toc529264800)

[Explicación 10](#_Toc529264801)

[13. Valor HMAC de shared.pem 10](#_Toc529264802)

[Comando 10](#_Toc529264803)

[14. Protocolo estación a estación 10](#_Toc529264804)

[1. Alicia calcula una clave pública y privada en función de los parámetros compartidos de la curva elíptica y envía la clave pública a Bob 11](#_Toc529264805)

[2. Bob calcula una clave pública y privada 11](#_Toc529264806)

[3. Bob genera la clave derivada con la pública recibida por Alice y la privada generada por él 11](#_Toc529264807)

[4. Bob concatena la pública con la pública de Alice, las firma con su privada (No la generada), y las cifra con la clave derivada generada. Finalmente envía la firma cifrada y la pública generada a Alice 11](#_Toc529264808)

[5. Alice genera la clave derivada 11](#_Toc529264809)

[6. Alice descifra las firmas y las verifica con la clave pública de Bob 11](#_Toc529264810)

[7. Alice concatena la pública generada por ella y la de bob, las firma con su privada (no la generada), lo cifra con la clave derivada y se lo envía a Bob 11](#_Toc529264811)

[8. Bob desencripta con la clave derivada y verifica la firma con la pública de Alice 12](#_Toc529264812)

[9. Conclusiones 12](#_Toc529264813)

## Generación de parámetros

### Comando

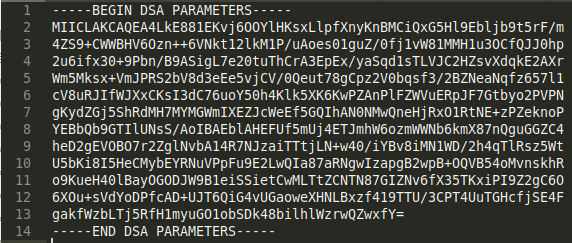


Dsaparam: indica que se van a crear parámetros de generación

-out: nombre del archivo de salida

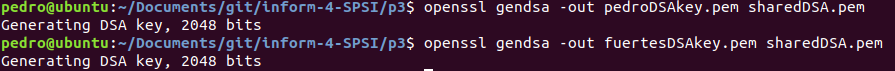
2048: para que sea de 2048 bits

### Parámetros



## Generación de claves

### Comando



Gendsa: indica que se va a crear una clave DSA

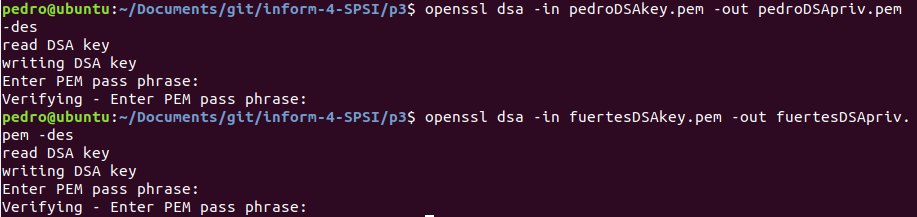
-in: indica el fichero donde están los parámetros, (Se han generado con el comando anterior)

-out: el nombre del archivo con la firma

Al final del todo se pone el archivo de donde obtener los parámetros, en este caso sharedDSA.pem

## Extracción de la clave privada

### Comando



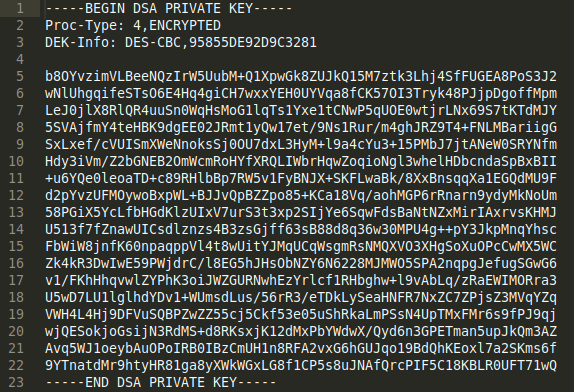
Contraseña: “0123456789”

Dsa: para extraer la clave privada

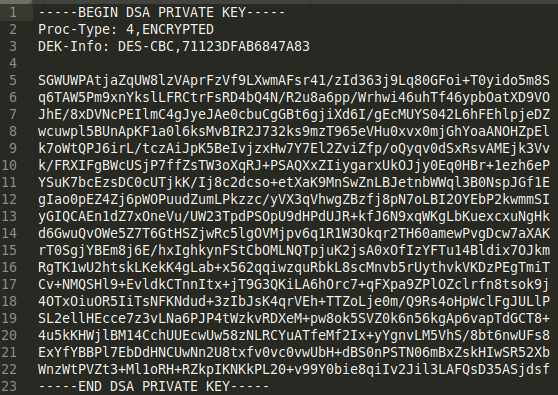
-des: para cifrarlo

### Clave privada

pedroDSApriv.pem



fuertesDSApriv.pem

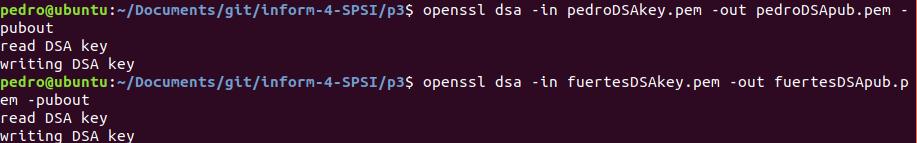


### Conclusiones

Se puede ver que los archivos resultantes son diferentes, esto puede ser debido al salt del cifrado, que mete número aleatorios, por lo que pese a tener ambos archivos la misma contraseña, el archivo resultante es distinto.

## Extracción de la clave pública

### Comando



-pubout: para indicar que queremos la clave pública

### Clave pública

pedroDSApub.pem



fuertesDSApub.pem



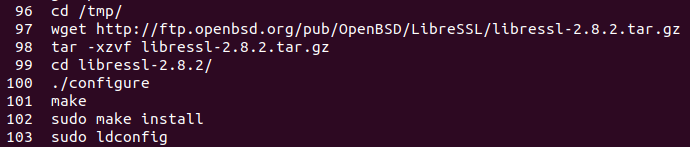
## Creamos un archivo de al menos 128 bytes



## Firmamos el mensaje

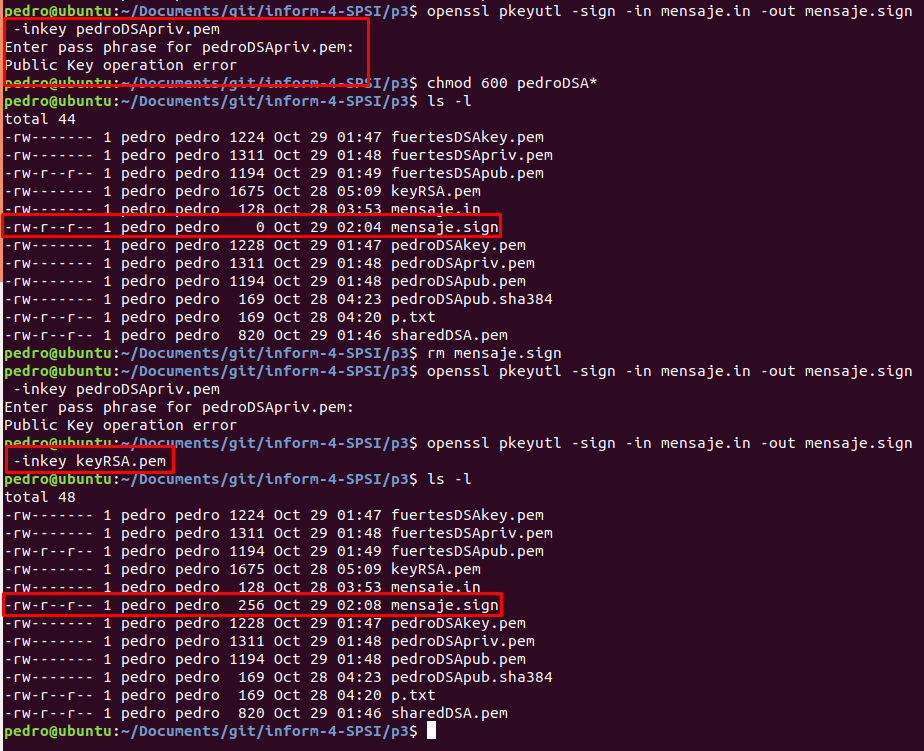
**Llegado a este punto, se encuentra un error al usar la clave privada generada “pedroDSApriv.pem”, podríamos generar una clave RSA para solventar el problema, sin embargo, para evitar problemas futuros, se decide instalar Libre SSl.**







### Comando



Pkeyutl: indica que se van a usar las claves

-sign: indica que se va a firmar

-in: archivo a firmar

-out: nombre del archivo que tendrá la firma

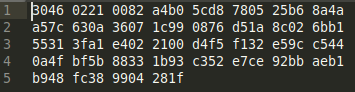
-inkey: la privada del usuario que quiere firmar

### Contenido

mensaje.in



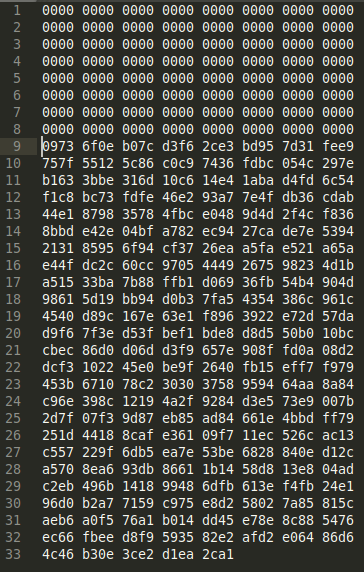
mensaje.sign



## Dar un falso positivo

A continuación intentaremos verificar dos mensajes distintos y que openssl nos diga que son iguales.

Para esto, se ha creado un nuevo archivo llamado mensaje2.in.



Como puede verse, los primeros bits del archivo son iguales a los de mensaje.in.

### Comando



-verify: indica que se quiere verificar una firma

-pubin: porque se va a usar la pública del autor (Normal, se quiere verificar que el autor es el autor)

-inkey: para indicarle la clave a uar

-in: mensaje a verificar

-sigfile: firma del mensaje

### Conclusiones

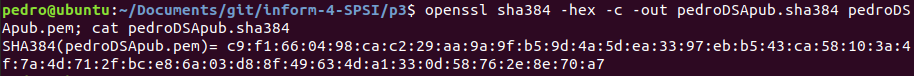
Se puede ver como al verificar el mensaje “mensaje2.sig” con la firma de “mensaje.in”, openssl nos dice que la firma es correcta.

Esto es debido a que al usar la orden **pkeyutl**, openssl **espera que se trabaje con el Hash** de los archivos y no con los archivos en sí, debido a esto, openssl trunca la entrada y como mensaje.in y mensaje2.in tienen los primeros bits iguales, el resultado es que openssl verifica una firma para un archivo diferente (¡Que, por ser, es hasta de mayor tamaño!)

**Si se quiere trabajar con archivos** directamente, se debe usar la orden **dgst**.

## Generar hash

### Comando



Sha384: Es el tipo de hash a usar

-hex: porque se quiere en hexadecimal

-c: Para que el texto esté separado por dos puntos (:)

-out: nombre del archivo con el hash

Finalmente, el nombre del archivo al que se le quiere calcular el hash.

### Archivo pedroDSApub.sha384



## Generar hash de 160 bits

Para generar un hash de 160 bits se va a usar el algoritmo hash sha1, el cual da un tamaño de 160 bits

### Comando



Sha1: para obtener una salida fijada a 128 bits

-binary: porque se quiere vinario

### Salida



## Firmar mensaje 2 usando dgst

### Comando



Dgst: indica que voy a firmar o verificar firmas de **archivos**, por lo que no es necesario hacer un Hash previo.

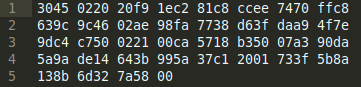
-sha1: para usar la función hash sha1

-sign pedroDSApriv.pem: para indicar que vamos a firmar con la clave pedroDSApriv.pem

-out: nombre del archivo de salida

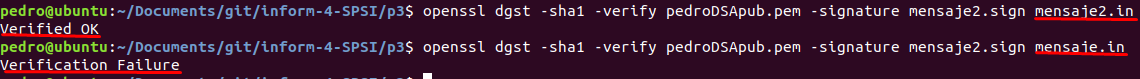
Y finalmente el archivo a firmar.

### Contenido



## Verificar los mensajes “mensaje” y “mensaje2” con “mensaje2.sign”

### Comando



-sha1: función hash a utilizar

-verify: clave pública con la que se quiere verificar

-signature: firma del mensaje a verificar

Finalmente se pasa el mensaje original .

### Explicación

Como se puede ver, ahora **el resultado es correcto**, verificando la firma únicamente el mensaje2.

Esto es debido a que el comando **dgst está pensado para trabajar con archivos** y no con funciones hash, por eso tal y como se puede ver a la hora de introducir el comando, se hace un hash previo al archivo, de modo que la firma se hace al hash del archivo y no a los primeros bits de este como ocurría con **pkeyutl,** ya que dicho comando **está pensado para trabajar con funciones hash** directamente.

## Verificación de “mensaje2” con “mensaje2.sign” usando la orden pkeyutl en vez de dgst

### Comando



### Explicación

Nuevamente nos da error de verificación, esto es debido a que como ya hemos dicho pkeyutl esperas hash y se le ha pasado un archivo.

Si se quiere verificar, había que hacerle el mismo hash usado en el comando dgst a mensaje2 y hacer la verificación.

Como se dispone de dicho archivo, se prueba a verificar con el hash.



Y como se puede ver, ahora el resultado es correcto.

## Valor HMAC de shared.pem

### Comando

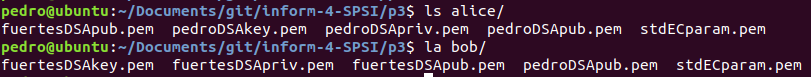


-hmac: para indicar que queremos calcular un hmac y la clave a usar

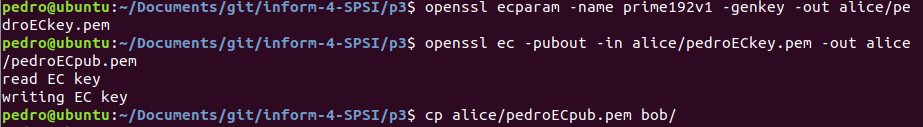
## Protocolo estación a estación

Suponemos que tanto Alice como Bob comparten los parámetros de la curva elíptica:

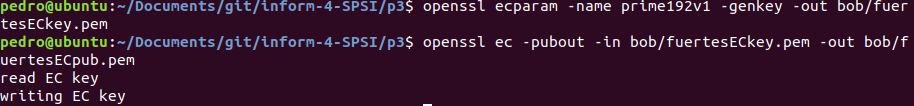
Archivos de partida:



### Alicia calcula una clave pública y privada en función de los parámetros compartidos de la curva elíptica y envía la clave pública a Bob



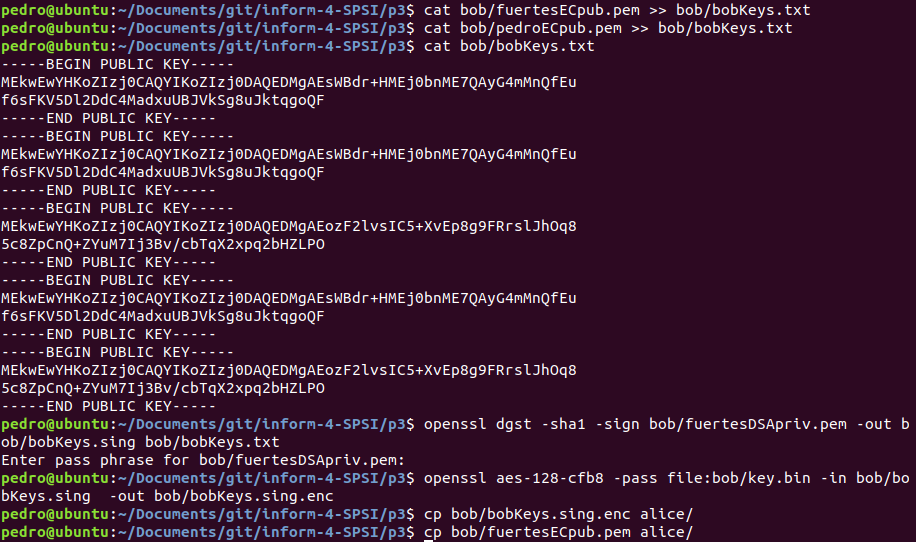
### Bob calcula una clave pública y privada

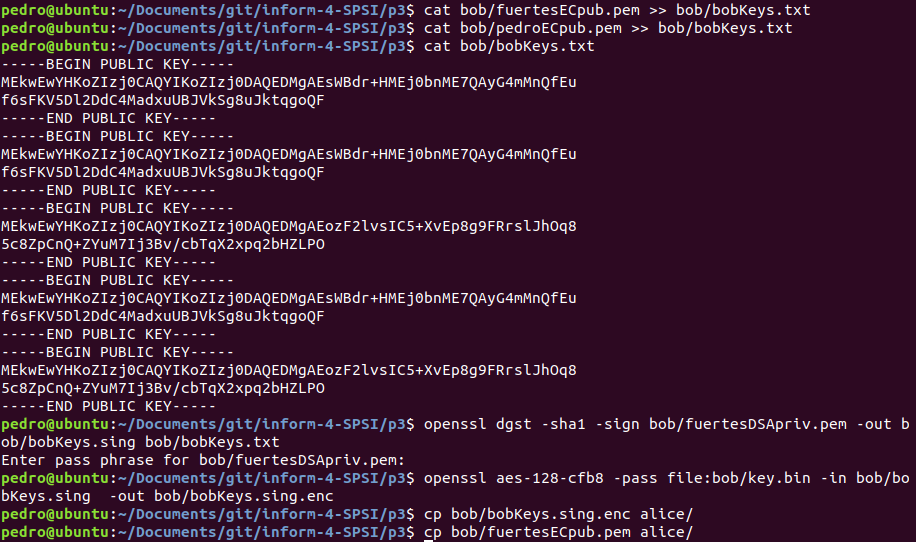


### Bob genera la clave derivada con la pública recibida por Alice y la privada generada por él



### Bob concatena la pública con la pública de Alice, las firma con su privada (No la generada), y las cifra con la clave derivada generada. Finalmente envía la firma cifrada y la pública generada a Alice

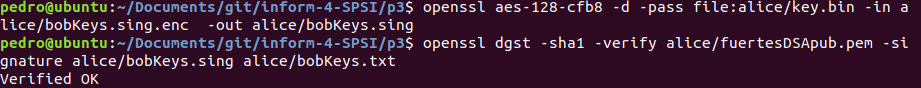




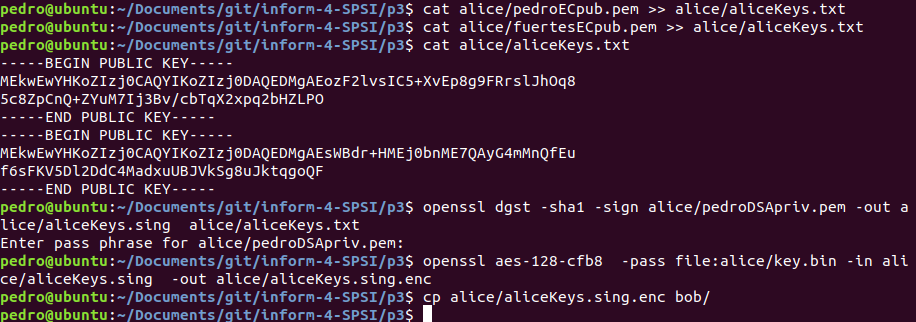
### Alice genera la clave derivada



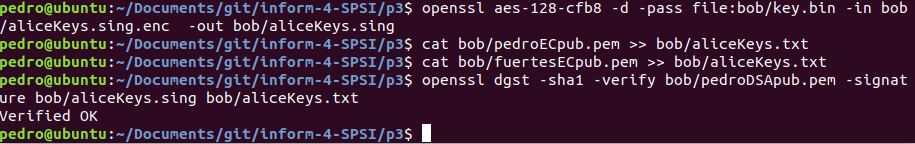
### Alice descifra las firmas y las verifica con la clave pública de Bob



### Alice concatena la pública generada por ella y la de bob, las firma con su privada (no la generada), lo cifra con la clave derivada y se lo envía a Bob



### Bob desencripta con la clave derivada y verifica la firma con la pública de Alice



### Conclusiones

Este sistema permite, por un lado, verificar que Alice y Bob, son quienes dicen ser, ya que en el paso 6 Alice comprueba la firma de Bob y en el paso 8 Bob verifica la de Alice.

Por otro lado, permite generar una clave de sesión que se puede usar para usar un cifrado simétrico, mucho más rápido y eficiente que el asimétrico, e intercambiarla en un canal no seguro.

Esto se consigue gracias a la generación de una clave derivada, la cual únicamente necesita la clave pública generada por el otro, de esta manera, aunque haya alguien escuchando en el canal, al no disponer de las privadas, no podrá obtener la clave derivada y por tanto no podrá descifrar la comunicación.