

SEGURIDAD Y PROTECCIÓN DE SISTEMAS INFORMÁTICOS (2017-2018)
GRADO EN INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE GRANADA

Criptosistemas Asimétricos

Juan Ramón Gómez Berzosa

5 de noviembre de 2017

Índice

1	Generad, cada uno de vosotros, una clave RSA (que contiene el par de claves) de 768 bits. Para referirnos a ella supondré que se llama nombreRSAkey.pem. Esta clave no es necesario que esté protegida por contraseña.	4
2	"Extraed" la clave privada contenida en el archivo nombreRSAkey.pem a otro archivo que tenga por nombre nombreRSApriv.pem. Este archivo deberá estar protegido por contraseña cifrándolo con AES-128.	5
3	Extraed en nombreRSAPub.pem la clave pública contenida en el archivo nombreRSAkey.pem. Evidentemente nombreRSAPub.pem no debe estar cifrado ni protegido. Mostrar sus valores.	6
4	Reutilizaremos el archivo binario input.bin de 1024 bits, todos ellos con valor 0, de la práctica anterior.	6
5	Intentad cifrar input.bin con vuestras claves públicas. Explicad el resultado.	7
6	Diseñad un cifrado híbrido, con RSA como criptosistema simétrico. El modo de proceder será el siguiente:	8
6.1	El emisor debe seleccionar un sistema simétrico con su correspondiente modo de operación.	8
6.2	El emisor generará un archivo de texto, llamado por ejemplo sessionkey con dos líneas. La primera línea contendrá una cadena aleatoria hexadecimal cuya longitud sea la requerida por la clave. OpenSSL permite generar cadenas aleatorias con el comando openssl rand. La segunda línea contendrá la información del criptosistema simétrico seleccionado. Por ejemplo, si hemos decidido emplear el algoritmo Blowfish en modo ECB, la segunda línea debería contener -bf-ecb	8
6.3	El archivo sessionkey se cifrará con la clave pública del receptor.	9
6.4	El mensaje se cifrará utilizando el criptosistema simétrico, la clave se generará a partir del archivo anterior mediante la opción -pass file:sessionkey.	9
7	Utilizando el criptosistema híbrido diseñado, cada uno debe cifrar input.bin con su clave pública para, a continuación, descifrarlo con la clave privada. Comparar el resultado con el archivo original.	10
8	Generad un archivo stdECparam.pem que contenga los parámetros públicos de una de las curvas elípticas contenidas en las transparencias de teoría. Si no lográis localizarlas haced el resto de la práctica con una curva cualquiera a vuestra elección de las disponibles en OpenSSL. Mostrad los valores.	11

9	Generad cada uno de vosotros una clave para los parámetros anteriores. La clave se almacenará en nombreECkey.pem y no es necesario protegerla por contraseña.	14
10	"Extraed" la clave privada contenida en el archivo nombreECkey.pem. Este archivo deberá estar protegido por contraseña cifrándolo con 3DES. Mostrad sus valores.	15
11	Extraed en nombreECpub.pem la clave pública contenida en el archivo nombreECkey.pem. Como antes nombresECpub.pem no debe estar cifrado ni protegido. Mostrad sus valores.	16

Índice de figuras

1.1.	Clave RSA	4
1.2.	Generación clave RSA	4
2.1.	Clave privada RSA cifrada con AES-128	5
3.1.	Clave pública RSA	6
4.1.	Input.bin	7
5.1.	Cifrado del archivo input.bin con clave pública RSA	7
6.1.	Fichero sessionkey	9
6.2.	Fichero sessionkeyCypher	9
7.1.	Fichero inputCypher.bin	10
7.2.	Fichero sessionkeyDecrypt	11
7.3.	Fichero inputDecrypt.bin	11
8.1.	Lista de curvas - Parte 1	12
8.2.	Lista de curvas - Parte 2	13
8.3.	Archivo stdECparam.pem	13
8.4.	Información stdECparam.pem	14
9.1.	Archivo JuanRamonECkey.pem	14
10.1.	Archivo JuanRamonECprivate.pem	15
11.1.	Archivo JuanRamonECpublic.pem	16

Índice de tablas

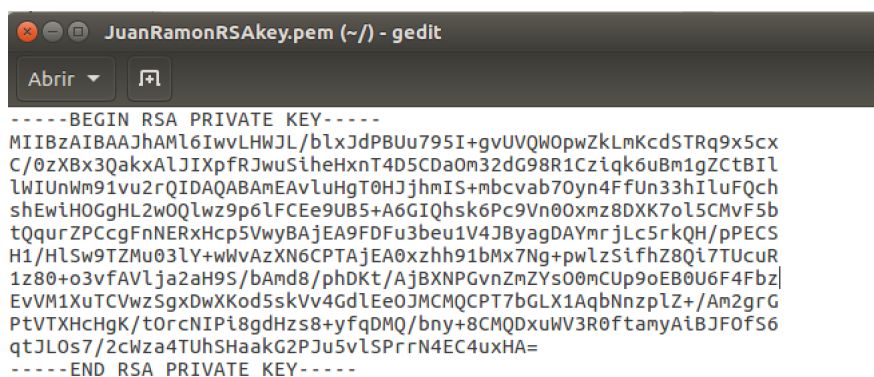
1. Generad, cada uno de vosotros, una clave RSA (que contiene el par de claves) de 768 bits. Para referirnos a ella supondré que se llama `nombbreRSAkey.pem`. Esta clave no es necesario que esté protegida por contraseña.

En primer lugar usaremos el comando `genrsa`, el cual nos permite generar una clave RSA que contiene el par de claves pública/privada. Las claves pueden ser protegidas por contraseña pero para esta parte no lo usaremos.

La orden sería la siguiente:

```
# openssl genrsa -out JuanRamonRSAkey.pem 768
```

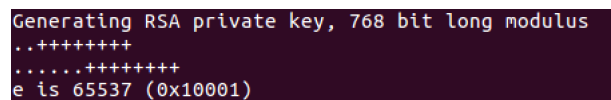
El resultado será la generación de nuestra clave RSA de 768 bits ya que se lo hemos indicado en el comando (parámetro `numbits`). El tamaño por defecto es de 2048 bits.



```
JuanRamonRSAkey.pem (~/) - gedit
Abrir
-----BEGIN RSA PRIVATE KEY-----
MIIBZAIBAAJhAMl6IwvLHWJL/blxJdPBuU795I+gvUVQWOpwZkLmKcdSTRq9xScx
C/0zXBx3QakxAlJIXpFRJwusIheHxnt4D5CDa0m32dG98R1Cziqk6uBm1gZCtBIL
lWIUnWm91vu2rQIDAQABAmEAvluHgT0HJjhmIS+mbcvab70yn4FFun33hIlufQch
shEwiH0GgHL2wOQlwz9p6lFCEe9UB5+A6GIQhsk6Pc9Vn00xmz8DXK7oL5CMvF5b
tQqurZPCcgFnNERxHcp5VwyBAjEA9FDfu3beu1V4JBByagDAYmrjLc5rkQH/pPECS
H1/HLSw9TZMu03lY+wWvAzXN6CPTAjeA0xzhh91bMx7Ng+pwLzSifhZ8Qi7TUcuR
1z80+o3vfAVlja2aH9S/bAmd8/phDKt/AjBXNPGvnZmZYs00mCU9p0EB0U6F4Fbz|
EvVM1XuTCVwzSgxDwXKod5skVv4GdlEeOJMCMQCPT7bGLX1AqbNnzplZ+/Am2grG
PtVTXHcHgK/t0rcNIPi8gdHzs8+yfqDMQ/bny+8CMQDxuWV3R0ftamyAiBJF0fS6
qtJL0s7/2cwza4TUhSHAakG2PJ5vLSPrrN4EC4uxHA=
-----END RSA PRIVATE KEY-----
```

Figura 1.1: Clave RSA

Si observamos la consola cuando ejecutamos el comando obtendremos algo así:



```
Generating RSA private key, 768 bit long modulus
..++++++
.....++++++
e is 65537 (0x10001)
```

Figura 1.2: Generación clave RSA

Si nos fijamos, aparecen una serie de “.” y símbolos “+” . Esto es una indicación del programa para decirnos que está intentando encontrar un primo p y luego un primo q , y comprobando que ambos son primos. Este proceso será más largo cuanto mayor queramos el tamaño de nuestra clave RSA. Además, si no especificamos nada al respecto, OpenSSL usa como exponente público estándar el número 4 de Fermat: 65.537 ó 0x10001. [1]

2. "Extraer" la clave privada contenida en el archivo nombreRSAkey.pem a otro archivo que tenga por nombre nombreRSApriv.pem. Este archivo deberá estar protegido por contraseña cifrándolo con AES-128.

Se dice "extraer" porque en realidad lo que hemos generado anteriormente es la clave privada, por lo que no es que vayamos a extraer de la la clave anterior la parte privada. Lo que se puede extraer como tal de la clave privada es la clave pública, ya que está contenida en ella. Por tanto, lo que realizaremos ahora será guardar en un fichero llamado **JuanRamonRSApriv.pem** la clave privada protegida por contraseña y cifrándola con el criptosistema simétrico AES-128.

Para ello usaremos el comando **rsa**, el cual permite manipular las claves generadas (extraer claves, añadir o suprimir claves.. etc). La orden sería el siguiente:

```
# openssl rsa -aes128 -in JuanRamonRSAkey.pem -out JuanRamonRSApriv.pem
```

Una vez que ejecutemos el comando nos pedirá una contraseña, de la cual el criptosistema simétrico derivará la clave que usará para el cifrado. En nuestro caso usaremos la recomendada en la práctica: **0123456789**, como resultado obtendremos algo así:



```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: AES-128-CBC,3A14B84EF4EF1F37398A7C98C5462572

blzBVeLCbIXldF8kxgscLuTWqMGQrFMPfEv3f5zY6y5AloDHsgkDZ6KYsYuSuxR3
eD+wuppbPt8x4zZaZ/KXBKKkF4LXlMi9DxC9nVKbBK/QN5SrZ4bDTct5t/DVBKKv
8zOQzm/+5u/RQKQj6Dhl4BJ5G72CTwl4bTjNx6jyXE5xsFQKna5aHCisdSor49lf
rXy9Xbxno8u8XtsInKUPPXFN7AxhNuWyX4zx6EXxR+kycrrTiPQLXu3iygJ5oYNy
iLHT5zIjkiuLH3siGvoYaWxRgK1yJQxvsF19mymL93dFM4QE1Cq2n4Zie77U4Ajl
clE8eKCcs+Scci5ivWuFPngLuKrUZjjSkoy986+EUXQ+zzq0JWxeESp3bzlJy4sA
x0jGOZakTDLMuYjyb6yqEstGmcst3wEHLV/WGi2aw/G36Tt+fGiAWGMv7z0PxrF
cK95vKjKLe3T5qW63uAAj6t5n/XdJfSSgh5uSd0qCdSyrhhlzBfJARWbTWxXLOPq
csHgN89Bk7R4P0sFihC+FRXAE+P4C6uKnUU+uiqsZaD+xkuPLULvY6BfzroNax8n
8chwd/l2S8QWKCCduvnfn6mm3CUMpkol/H0PIfEcUU0qnk2jS08nNIAxwKGVvg17
-----END RSA PRIVATE KEY-----
```

Figura 2.1: Clave privada RSA cifrada con AES-128

Cabe destacar, que podríamos haber obtenido el mismo resultado si hubiéramos empleado el comando **openssl genrsa** indicando la contraseña y el criptosistema simétrico de cifrado.

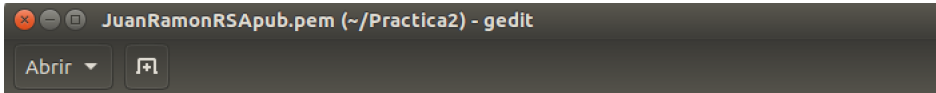
3. Extraed en nombreRSApub.pem la clave pública contenida en el archivo nombreRSAkey.pem. Evidentemente nombreRSApub.pem no debe estar cifrado ni protegido. Mostrar sus valores.

En este caso si que vamos a realizar una extracción del fichero generado en el primer punto, JuanRamonRSAkey.pem. En este caso no vamos a usar ningún cifrado ni protección, como es lógico, ya que esta clave será la que compartamos para que cualquier persona con la que queramos intercambiar información cifre el contenido de lo que quiera enviar y sólo el poseedor de la clave privada asociada a dicha clave pública pueda abrir (descifrar).

Para ello usaremos la siguiente orden:

```
# openssl rsa -in JuanRamonRSAkey.pem -pubout -out JuanRamonRSApub.pem
```

Este comando tomará el archivo JuanRamonRSAkey.pem que hemos generado anteriormente y extraerá la clave pública asociada, para ello usamos la opción **-pubout** que indica que la salida de la operación será la clave pública. Por defecto, si no indicamos nada, sacará la clave privada como ha pasado en el apartado anterior. Finalmente el archivo generado será:



```
-----BEGIN PUBLIC KEY-----
MHwwDQYJKoZIhvcNAQEBBQADAwAwAJhAMl6IwvLHWJL/blxJdPBUu795I+gvUVQ
W0pwZkLmKcdSTRq9x5cxC/0zXBx3QakxALJIXpfRJwuSiheHxnT4D5CDa0m32dG9
8R1Cziqk6uBm1gZCtBIllWIUnWm91vu2rQIDAQAB
-----END PUBLIC KEY-----
```

Figura 3.1: Clave pública RSA

4. Reutilizaremos el archivo binario input.bin de 1024 bits, todos ellos con valor 0, de la práctica anterior.

Vamos a usar el archivo que creamos en la práctica anterior, input.bin. Este archivo era de 1024 bits todos rellenos con 0.

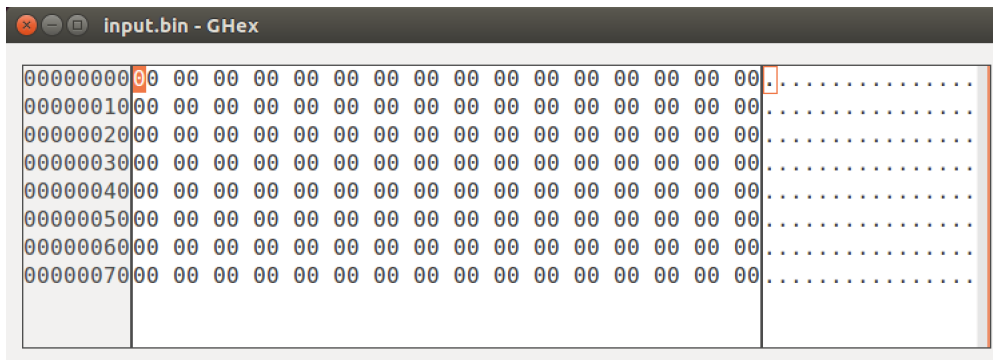


Figura 4.1: Input.bin

5. Intentad cifrar input.bin con vuestras claves públicas. Explicad el resultado.

Para ello usaremos el comando **rsautl** de OpenSSL. Nos permite utilizar varias acciones asociadas a las claves generadas, de las cuáles por ahora nos serán útiles cifrar y descifrar. Como entrada tendremos el fichero input.bin y como clave de entrada usaremos nuestra clave RSA pública generada anteriormente. La orden sería la siguiente:

```
# openssl rsautl -encrypt -in input.bin -out inputCypher.bin -inkey JuanRamonRSAPub.pem -pubin
```

Ahora vamos a explicar las opciones introducidas:

- **-encrypt:** Esta opción la empleamos para indicar que vamos a encriptar un fichero.
- **-in:** Con este se indica el fichero que queremos cifrar.
- **-out:** Con esta opción se indica el nombre del fichero que queremos que tenga nuestro archivo cifrado.
- **-inkey:** Con ella indicamos que clave queremos emplear para cifrar el archivo, en nuestro caso la pública.
- **-pubin:** Le indicamos al comando que queremos utilizar como entrada una clave pública para cifrar el archivo, ya que por defecto la toma como privada.

El resultado de la ejecución de dicho comando es el siguiente:

```
jramongomez@jramongomez-Parallels-Virtual-Platform:~$ openssl rsautl -encrypt
-in input.bin -out inputCypher.bin -inkey JuanRamonRSAPub.pem -pubin
RSA operation error
139801560393368:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data
too large for key size:rsa_pk1.c:153:
```

Figura 5.1: Cifrado del archivo input.bin con clave pública RSA

Este error se debe a que el tamaño del fichero que queremos cifrar es de 1024 bits, mientras que nuestra clave es una clave de 768 bits. Al no tratarse de un cifrado que utilice un cifrado por bloques, no podemos tener un mensaje a cifrar de mayor tamaño que la clave, por lo tanto vamos a implementar un cifrado híbrido que mezcle cifrado simétrico con bloques y cifrado asimétrico para cifrar el archivo.

6. Diseñad un cifrado híbrido, con RSA como criptosistema simétrico. El modo de proceder será el siguiente:

El esquema del cifrado híbrido sería el siguiente, dónde el azul hace referencia al uso de claves y cifrados simétricos, y el verde y el rojo hacen referencia a las claves públicas y privadas del cifrado asimétrico (RSA) respectivamente.

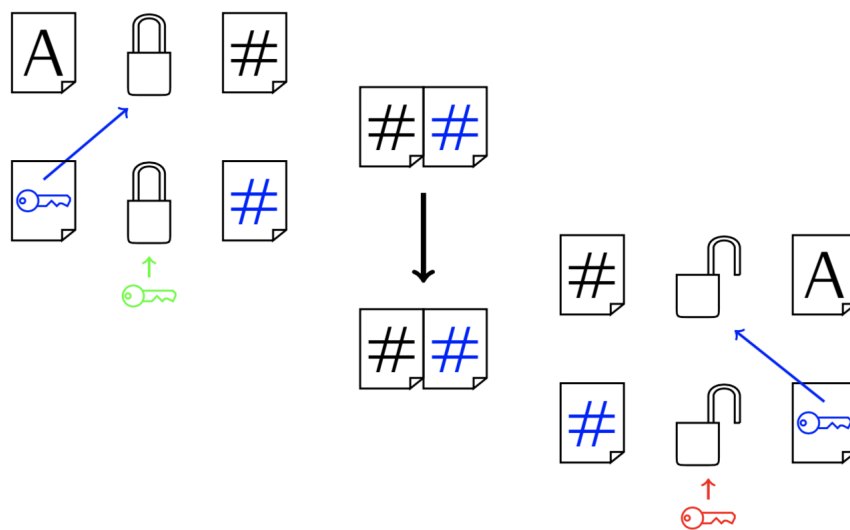


Figura 6.1: Cifrado híbrido.

6.1. El emisor debe seleccionar un sistema simétrico con su correspondiente modo de operación.

Para el cifrado híbrido voy a utilizar el criptosistema simétrico AES con tamaño de clave de 128 bits (AES-128) en modo de operación ECB.

6.2. El emisor generará un archivo de texto, llamado por ejemplo sessionkey con dos líneas. La primera línea contendrá una cadena aleatoria hexadecimal cuya longitud sea la requerida por la clave. OpenSSL permite generar cadenas aleatorias con el comando openssl rand. La segunda línea contendrá la información del criposistema simétrico seleccionado. Por ejemplo, si hemos decidido emplear el algoritmo Blowfish en modo ECB, la segunda línea debería contener -bf-ecb

Para generar la clave aleatoria de sesión usaremos el comando **rand** de openssl:

```
# openssl rand -hex 16 >sessionkey
```

Usaremos la opción **-hex** para que nos genere una cadena en hexadecimal y le introducimos el 16 para indicarle que queremos una cadena de 16 bytes, ya que 16 bytes son 128 bits y el tamaño de clave que utilizaremos para el cifrado simétrico de aes será de 128 bits. También le indicamos que lo vuelque en un fichero llamado sessionkey.

Una vez hecho eso, abrimos el fichero sessionkey y en la siguiente línea introducimos el cifrado simétrico que vamos a emplear junto a su modo de operación, en nuestro caso: **aes-128-ecb**.

El fichero quedará tal que así:

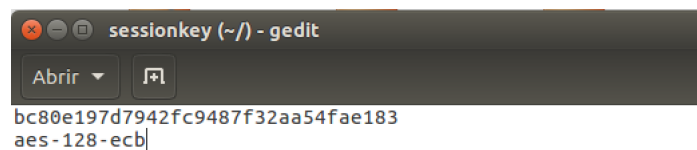


Figura 6.2: Fichero sessionkey

6.3. El archivo sessionkey se cifrará con la clave pública del receptor.

Para ello vamos a emplear nuestra clave pública RSA generada anteriormente, ya que en este caso emisor y receptor van a ser la misma persona para comprobar que el cifrado y descifrado posteriormente funciona correctamente.

Para cifrarlo emplearemos de nuevo el comando del ejercicio anterior, rsautl. La orden será la siguiente:

```
# openssl rsautl -encrypt -in sessionkey -out sessionkeyCipher -inkey JuanRamonRSAPub.pem -pubin
```

En este caso, recibirá como archivo de entrada el fichero sessionkey, con los datos que va a emplear el cifrado simétrico para cifrar el fichero posteriormente. Como salida generará el sessionkeyCipher, el cual se enviará al receptor junto al posterior mensaje cifrado. El fichero cifrado quedará tal que así:

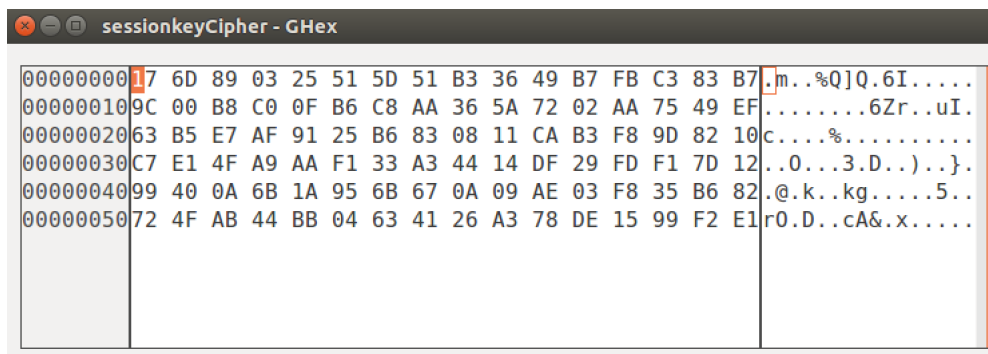


Figura 6.3: Fichero sessionkeyCypher

- 6.4. El mensaje se cifrará utilizando el criptosistema simétrico, la clave se generará a partir del archivo anterior mediante la opción `-pass file:sessionkey`.

Esta parte no la realizaremos ahora ya que la llevaremos a cabo en el siguiente apartado.

7. Utilizando el criptosistema híbrido diseñado, cada uno debe cifrar input.bin con su clave pública para, a continuación, descifrarlo con la clave privada. Comparar el resultado con el archivo original.

Ahora vamos a cifrar usando el criptosistema simétrico que hemos decidido utilizar a la hora de generar el fichero sessionkey, en nuestro caso, aes-128. Para ello ejecutaremos la siguiente orden:

```
# openssl aes-128-ecb -pass file:sessionkey -in input.bin -out inputEncrypt.bin
```

Le pasamos el archivo sessionkey con el argumento `-pass file:sessionkey`, con ello se utilizará la clave de sesión generada anteriormente. Como fichero a cifrar le indicamos el input.bin

El fichero cifrado quedará tal que así:

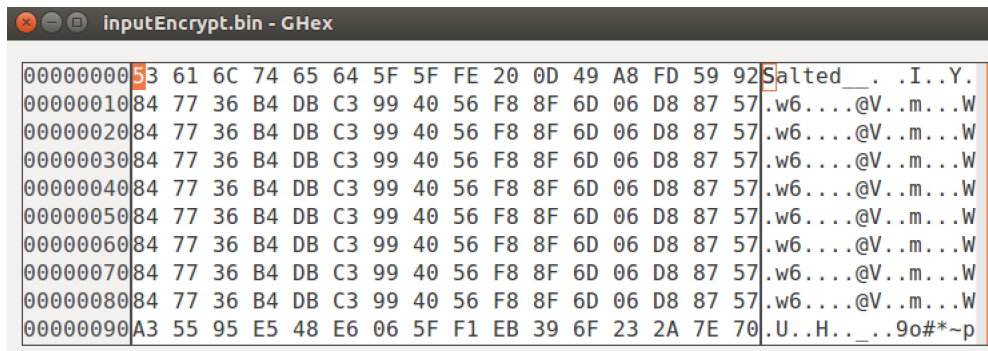


Figura 7.1: Fichero inputCypher.bin

Una vez hemos completado el cifrado, pasaremos al descifrado. Cuando utilizamos un cifrado híbrido, se le pasa al receptor el sessionkey cifrado y el mensaje cifrado. Posteriormente, el receptor descifrará con su clave privada el sessionkey y una vez hecho esto aplicará el descifrado del criptosistema simétrico encontrado en el sessionkey con la clave de sesión del mismo archivo. Para ello, en primer lugar usaremos el comando de openssl **resautl** para descifrar el sessionkey:

```
# openssl rsautl -decrypt -in sessionkeyCipher -out sessionkeyDecryp -inkey JuanRamonRSPriv.pem
```

En este caso no indicamos la opción **-pubin** ya que por defecto coge la clave privada, que es la que nos interesa ahora. También le indicamos que la clave está en el archivo: **JuanRamonRSPriv.pem**.

El fichero de sesión quedará tal que así:

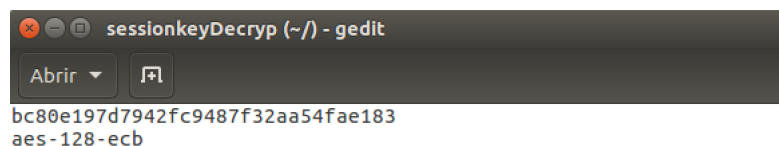


Figura 7.2: Fichero sessionkeyDecrypt

Como podemos comprobar se ha descifrado de forma correcta. Ahora usaremos el cifrado simétrico para descifrar el archivo **inputCipher.bin** que habíamos generado. Para ello usaremos la siguiente orden:

```
# openssl aes-128-ecb -d -pass file:sessionkey -in inputEncrypt.bin -out inputDecrypt.bin.
```

Como resultado obtenemos:

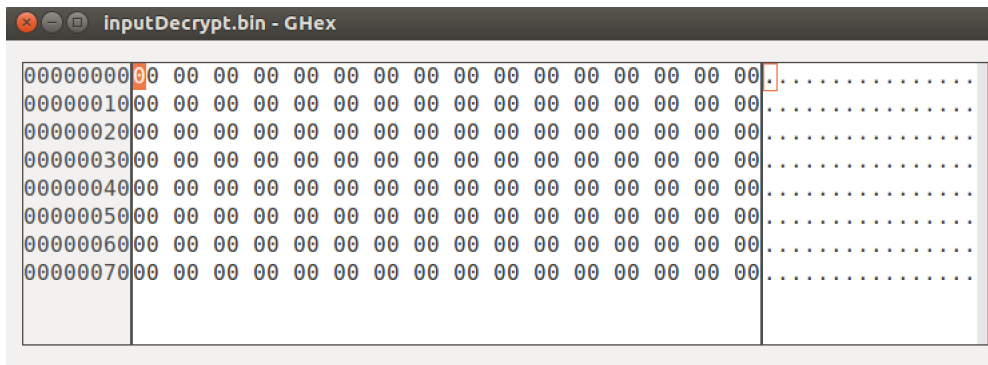


Figura 7.3: Fichero inputDecrypt.bin

Como podemos comprobar, el archivo se ha descifrado correctamente, 1024 bits rellenos de 0.

8. Generad un archivo `stdECparam.pem` que contenga los parámetros públicos de una de las curvas elípticas contenidas en las transparencias de teoría. Si no lográis localizarlas haced el resto de la práctica con una curva cualquiera a vuestra elección de las disponibles en OpenSSL. Mostrad los valores.

Para el desarrollo de este apartado usaré el comando `ecparam` de openssl. Este comando se utiliza para generar y manipular los parámetros asociados a las curvas elípticas y parejas de claves privadas/públicas.

En primer lugar he consultado los tipos de curvas que vienen predefinidos con el comando:

```
# openssl ecparam -list_curves
```

```
jramongomez@jramongomez-Parallels-Virtual-Platform: ~  
jramongomez@jramongomez-Parallels-Virtual-Platform:~$ openssl ecparam -list_curves  
secp112r1 : SECG/WTLS curve over a 112 bit prime field  
secp112r2 : SECG curve over a 112 bit prime field  
secp128r1 : SECG curve over a 128 bit prime field  
secp128r2 : SECG curve over a 128 bit prime field  
secp160k1 : SECG curve over a 160 bit prime field  
secp160r1 : SECG curve over a 160 bit prime field  
secp160r2 : SECG/WTLS curve over a 160 bit prime field  
secp192k1 : SECG curve over a 192 bit prime field  
secp224k1 : SECG curve over a 224 bit prime field  
secp224r1 : NIST/SECG curve over a 224 bit prime field  
secp256k1 : SECG curve over a 256 bit prime field  
secp384r1 : NIST/SECG curve over a 384 bit prime field  
secp521r1 : NIST/SECG curve over a 521 bit prime field  
prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field  
prime192v2: X9.62 curve over a 192 bit prime field  
prime192v3: X9.62 curve over a 192 bit prime field  
prime239v1: X9.62 curve over a 239 bit prime field  
prime239v2: X9.62 curve over a 239 bit prime field  
prime239v3: X9.62 curve over a 239 bit prime field  
prime256v1: X9.62/SECG curve over a 256 bit prime field  
sect113r1 : SECG curve over a 113 bit binary field  
sect113r2 : SECG curve over a 113 bit binary field  
sect131r1 : SECG/WTLS curve over a 131 bit binary field  
sect131r2 : SECG curve over a 131 bit binary field  
sect163k1 : NIST/SECG/WTLS curve over a 163 bit binary field  
sect163r1 : SECG curve over a 163 bit binary field  
sect163r2 : NIST/SECG curve over a 163 bit binary field  
sect193r1 : SECG curve over a 193 bit binary field  
sect193r2 : SECG curve over a 193 bit binary field  
sect233k1 : NIST/SECG/WTLS curve over a 233 bit binary field
```

Figura 8.1: Lista de curvas - Parte 1

```
jramongomez@jramongomez-Parallels-Virtual-Platform: ~
wap-wsg-idm-ecid-wtls5: X9.62 curve over a 163 bit binary field
wap-wsg-idm-ecid-wtls6: SECG/WTLS curve over a 112 bit prime field
wap-wsg-idm-ecid-wtls7: SECG/WTLS curve over a 160 bit prime field
wap-wsg-idm-ecid-wtls8: WTLS curve over a 112 bit prime field
wap-wsg-idm-ecid-wtls9: WTLS curve over a 160 bit prime field
wap-wsg-idm-ecid-wtls10: NIST/SECG/WTLS curve over a 233 bit binary field
wap-wsg-idm-ecid-wtls11: NIST/SECG/WTLS curve over a 233 bit binary field
wap-wsg-idm-ecid-wtls12: WTLS curve over a 224 bit prime field
Oakley-EC2N-3:
  IPsec/IKE/Oakley curve #3 over a 155 bit binary field.
  Not suitable for ECDSA.
  Questionable extension field!
Oakley-EC2N-4:
  IPsec/IKE/Oakley curve #4 over a 185 bit binary field.
  Not suitable for ECDSA.
  Questionable extension field!
brainpoolP160r1: RFC 5639 curve over a 160 bit prime field
brainpoolP160t1: RFC 5639 curve over a 160 bit prime field
brainpoolP192r1: RFC 5639 curve over a 192 bit prime field
brainpoolP192t1: RFC 5639 curve over a 192 bit prime field
brainpoolP224r1: RFC 5639 curve over a 224 bit prime field
brainpoolP224t1: RFC 5639 curve over a 224 bit prime field
brainpoolP256r1: RFC 5639 curve over a 256 bit prime field
brainpoolP256t1: RFC 5639 curve over a 256 bit prime field
brainpoolP320r1: RFC 5639 curve over a 320 bit prime field
brainpoolP320t1: RFC 5639 curve over a 320 bit prime field
brainpoolP384r1: RFC 5639 curve over a 384 bit prime field
brainpoolP384t1: RFC 5639 curve over a 384 bit prime field
brainpoolP512r1: RFC 5639 curve over a 512 bit prime field
brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
```

Figura 8.2: Lista de curvas - Parte 2

Por lo que he estado mirando, creo que la curva **P-192** de las transparencias de teoría se correspondería con la curva **prime192v1**.

Por tanto ejecutaremos la siguiente orden:

```
# openssl ecparam -out stdECparam.pem -name prime192v1
```

Con esto indicaremos que queremos generar el archivo `stdECparam.pem` con los parámetros por defecto del grupo `prime192v1`. El archivo que se ha generado sería el siguiente:

```
stdECparam.pem (~/) - gedit
Abrir
-----BEGIN EC PARAMETERS-----
BggqhkJOPQMBQ==
-----END EC PARAMETERS-----
```

Figura 8.3: Archivo `stdECparam.pem`

Como podemos comprobar, encontramos los parámetros públicos del grupo `prime192v1`. Ahora vamos a mostrar información sobre el fichero que hemos creado. Para ello usamos

la siguiente orden:

```
jramongomez@jramongomez-Parallels-Virtual-Platform:~$ openssl ecparam -in std
ECparam.pem -noout -text
ASN1 OID: prime192v1
NIST CURVE: P-192
```

Figura 8.4: Información stdECparam.pem

Como podemos observar, la primera línea nos indica que los parámetros de la curva elíptica se han codificado usando los parámetros del grupo prime 192v1. La segunda línea nos dice que se corresponde con la curva P-192, la cual es una de las que aparece en las transparencias de teoría.

9. Generad cada uno de vosotros una clave para los parámetros anteriores. La clave se almacenará en nombreECkey.pem y no es necesario protegerla por contraseña.

Para ello, usaremos de nuevo el comando ecparam de OpenSSL. La orden será la siguiente:

```
# openssl ecparam -in stdECparam.pem -out JuanRamonECkey.pem -genkey
```

A continuación vamos a comentar los parámetros introducidos:

- **-in:** Indicamos de donde queremos que se cojan los parámetros para generar la clave.
- **-out:** Indicamos el fichero dónde queremos que la clave se almacene.
- **-genkey:** Con esto le indicamos que queremos generar la clave privada asociada a la curva elíptica con los parámetros introducidos.

El archivo generado será el siguiente:



```
JuanRamonECkey.pem (~/) - gedit
Abrir
-----BEGIN EC PARAMETERS-----
BgqghkjOPQMBAQ==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MF8CAQEEGKtF74dzUfcEwOVsxz0n/jhgWlADC11b6AKBgqghkjOPQMBAaE0AzIA
BIT6+dQP9JulDhULX+4quRAXF5s5maDSWpRfLQVDizimPqlhDoB8vkeZJNDDb2bM
Lg==
-----END EC PRIVATE KEY-----|
```

Figura 9.1: Archivo JuanRamonECkey.pem

Cabe destacar que podríamos haber generado tanto los parámetros como las claves en la misma orden, ya que si nos damos cuenta es prácticamente idéntica. Hacerlo en dos pasos tiene utilidad cuando varios usuarios quieren emplear claves asociadas a los mismos parámetros.

10. "Extraed" la clave privada contenida en el archivo **nombreECkey.pem**. Este archivo deberá estar protegido por contraseña cifrándolo con 3DES. Mostrad sus valores.

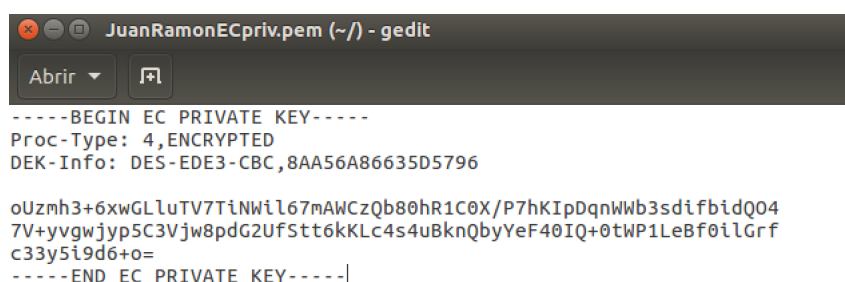
Al igual que pasaba antes con RSA, ahora decimos también "extraer" porque en realidad lo que hemos generado antes es la clave privada, lo que haremos ahora es guardarla en un archivo protegido con contraseña y cifrándolo con el criptosistema simétrico 3DES.

Para ello, ahora vamos a emplear el comando **ec** de OpenSSL. Este comando es similar a **rsa** y nos ayudará a manejar las claves. Usaremos la siguiente orden:

```
# openssl ec -in JuanRamonECkey.pem -des3 -out JuanRamonECpriv.pem
```

Al igual que con RSA, cuando utilicemos este comando nos pedirá una contraseña para que criptosistema simétrico genere una clave a partir de esta. La contraseña empleada será la recomendada en la práctica: 0123456789.

Finalmente el contenido del fichero que almacenará la clave privada cifrada será este:



```
-----BEGIN EC PRIVATE KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: DES-EDE3-CBC,8AA56A86635D5796

oUzmh3+6xwGLluTV7TiNWl67mAWCzQb80hR1C0X/P7hKIpdqnWb3sdifbidQ04
7V+yvgwjyp5C3Vjw8pdG2UfStt6kKLc4s4uBknQbyYef40IQ+0tWP1LeBf0ilGrf
c33y5i9d6+o=
-----END EC PRIVATE KEY-----
```

Figura 10.1: Archivo JuanRamonECprivate.pem

Como podemos observar en la cabecera "DEK-Info", el criptosistema de cifrado es DES3 en modo CBC. El resto de la cabecera, es la información necesaria que junto a la contraseña introducida anteriormente sirva para descifrar el archivo.

11. Extraed en nombreECpub.pem la clave pública contenida en el archivo nombreECkey.pem. Como antes nombresECpub.pem no debe estar cifrado ni protegido. Mostrad sus valores.

Ahora vamos a extraer de la clave generada anteriormente sin cifrar la clave pública asociada a ella. Para ello usaremos la siguiente orden:

```
# openssl ec -in JuanRamonECkey.pem -pubout -out JuanRamonECpub.pem
```

En este caso, no indicamos ningún método de cifrado ya que no tiene sentido cifrar la clave pública ya que es la que publicaremos para que el receptor nos cifre mensajes que sólo el poseedor de la clave privada asociada a esta clave pública pueda leer (descifrar). Indicamos "-pubout" para indicar que queremos que nos extraiga la clave pública en el fichero que hemos indicado de salida.

El contenido del fichero que almacena la clave pública será el siguiente:



```
-----BEGIN PUBLIC KEY-----
MEkwEwYHKOZIZj0CAQYIKoZIZj0DAQEDMgAEhPr51A/0m6U0FQtf7iq5EBcXmzmZ
oNJa1EWVBUOLOKY+qWE0gHy+R5kk0MNVZswu
-----END PUBLIC KEY-----
```

Figura 11.1: Archivo JuanRamonECpublic.pem

Con esto ya tendríamos la pareja de clave pública y privada asociada a nuestra curva elíptica.

Referencias

- [1] Generación de claves rsa. <http://www.criptored.upm.es/crypt4you/temas/RSA/leccion7/leccion07.html>.