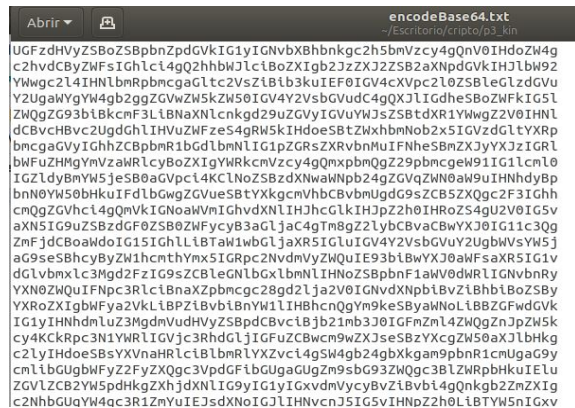


Memoria P3 Criptografía
OpenSSL y Criptografía Pública
By
David Quintana
Alfonso Carvajal

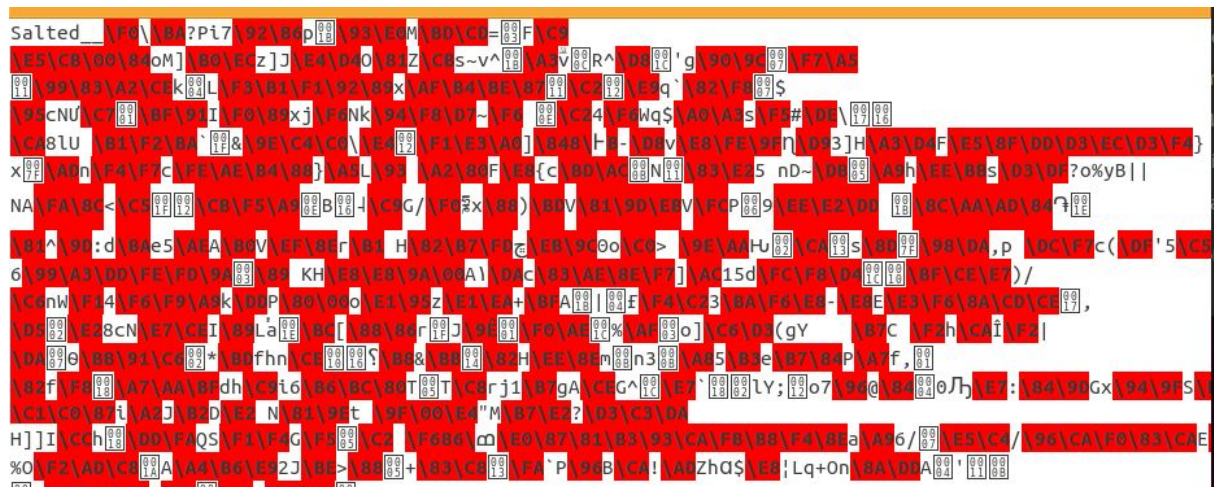
1 OpenSSL	3
1a Cifrados simétricos	3
1b Cifrados asimétricos	5
1c Generación de claves privada y pública	6
1d Diferencia entre velocidades cifrado simétrico/asimétrico	9
1e Certificados X.509	10
2 RSA	11
2a Potenciación de grandes números	11
2b Generación de números primos: Miller-Rabin	12
2c Factorización RSA conociendo d (usando A. las Vegas)	15



En el segundo ejemplo, hemos elegido cifrar el texto “short.txt” con un algoritmo que no conocemos como es el base64.



Por último, usamos el algoritmo RC4 para cifrar una imagen (don.png) con la password “hola”. El resultado es el siguiente:



Cabe destacar que para cada documento cifrado se ha comprobado que se descifra de manera correcta con la ejecución del comando -d para cada algoritmo ejecutado.

1b Cifrados asimétricos

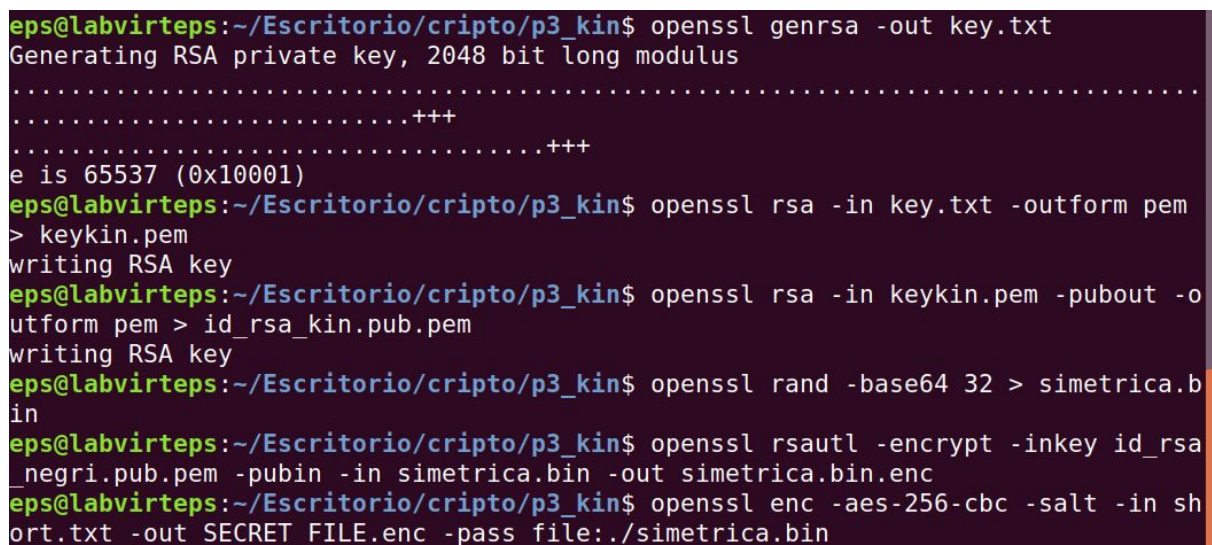
Lo primero que debemos hacer es generarnos una clave privada con el comando `openssl genrsa -out key.txt`, y la almacenamos en un fichero. Una vez hecho esto, debemos hacer que nuestro destinatario nos mande su clave pública, con la que ciframos la clave simétrica. Para ello debe ejecutar los comandos:

- `openssl rsa -in key_destinatario -outform pem > key_destinatario.pem`
- `openssl rsa -in key_destinatario -pubout -outform pem > key_destinatario.pub.pem`

Tras esto, generamos una clave simétrica que será guardada en `key.bin` (el comando para ello es `openssl rand -base64 32 > key.bin`). Una vez generada la clave simétrica, la ciframos con la clave pública de nuestro destinatario, que será el único que pueda descifrarla por ser el único que posee la clave privada asociada a dicha clave pública (`openssl rsautl -encrypt -inkey key_destinatario.pub.pem -pubin -in simetrica.bin -out simetrica.bin.enc`). Ahora ciframos el documento que queremos enviar a nuestro destinatario con cualquiera de los algoritmos simétricos que nos ofrece openssl (hemos decidido usar `aes-256-cbc` que como veremos, es de los más rápidos en ejecutar bloques de datos). Por último, tendremos que enviar nuestros ficheros `.enc` (el de la clave simétrica y el que acabamos de cifrar con AES) al receptor para que ejecutando los siguientes comandos, pueda leer el fichero enviado:

- `openssl rsautl -decrypt -inkey key_destinatario.pem -in simetrica.bin.enc -out simetrica.bin`
- `openssl enc -d -aes-256-cbc -in SECRET_FILE.enc -out SECRET_FILE -pass file:./simetrica.bin`

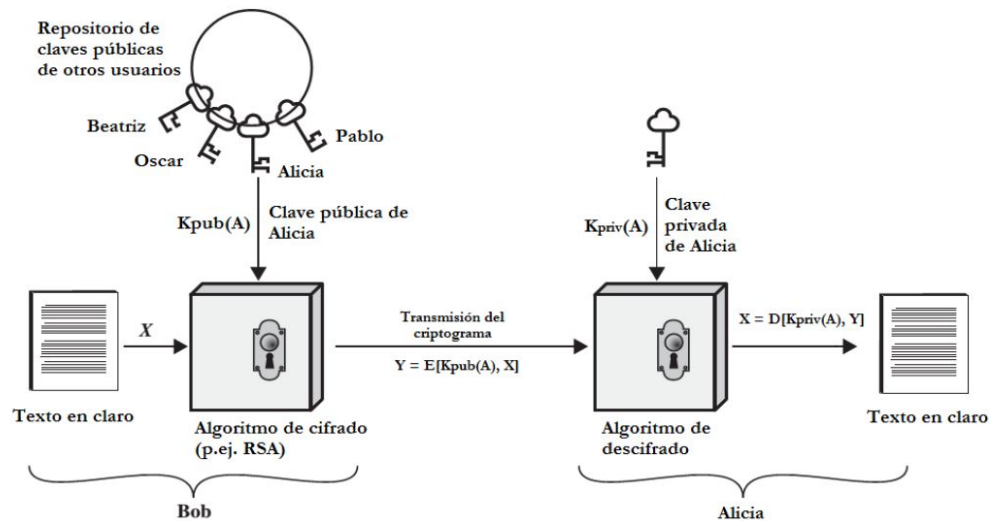
Aportamos una captura de pantalla con los comandos ejecutados:



```
eps@labvirteps:~/Escritorio/cripto/p3_kin$ openssl genrsa -out key.txt
Generating RSA private key, 2048 bit long modulus
.....
.....+++
.....+++
e is 65537 (0x10001)
eps@labvirteps:~/Escritorio/cripto/p3_kin$ openssl rsa -in key.txt -outform pem
> keykin.pem
writing RSA key
eps@labvirteps:~/Escritorio/cripto/p3_kin$ openssl rsa -in keykin.pem -pubout -outform pem > id_rsa_kin.pub.pem
writing RSA key
eps@labvirteps:~/Escritorio/cripto/p3_kin$ openssl rand -base64 32 > simetrica.bin
eps@labvirteps:~/Escritorio/cripto/p3_kin$ openssl rsautl -encrypt -inkey id_rsa_kin.pub.pem -pubin -in simetrica.bin -out simetrica.bin.enc
eps@labvirteps:~/Escritorio/cripto/p3_kin$ openssl enc -aes-256-cbc -salt -in short.txt -out SECRET_FILE.enc -pass file:./simetrica.bin
```


1c Generación de claves privada y pública

Para estudiar el esquema de cifrado público basado en RSA, hemos decidido ilustrarlo para una mejor comprensión del lector.



En primer lugar, Bob escribe un texto en claro, el cual se interpreta como un número entero, y cifra con la clave pública de Alicia (destinataria del mensaje) con un algoritmo de cifrado. En el caso de RSA se trata de tomar el mensaje X y hacer la operación $Y = X^e$, donde 'e' es la clave pública de Alice. El resultado de este cifrado es $Y = E(Púb(A), \text{texto plano})$. Y llega a Alicia, la cual aplica el algoritmo de descifrado al texto cifrado Y usando su clave privada (conocida únicamente por ella). Así el texto plano X se descifra acorde con la fórmula $X = D(Priv(A), Y)$. En el caso de RSA tenemos que $X = Y^d = (X^e)^d$ donde 'd' es la clave privada de Alice. Funciona porque 'e' ha sido elegido según los pasos de RSA y 'd' ha sido calculado a partir de 'e' de manera que $d \equiv e^{-1} \pmod{\lambda(n)}$. Aun sabiendo 'e' y 'n', (clave pública), es muy difícil (computacionalmente imposible) sacar 'd' porque se necesitaría factorizar los primos p, q ($pq=n$). Estos primos son enormes y no es posible factorizar 'n' sin saber alguno de los dos. Esto nos permite asegurar que el mensaje llega a nuestra destinataria y solamente es ella la que podrá descifrarlo puesto que es la única que conoce y posee su clave privada.

Tras esta explicación, ejecutamos los comandos necesarios para generar nuestro par de claves pública-privada y que se guarda en el fichero `example.key`.

```
fons@Fonss-mb p3_kin % openssl genrsa -out eg_key.txt
Generating RSA private key, 2048 bit long modulus
....+++
.....+++
e is 65537 (0x10001)
fons@Fonss-mb p3_kin % cat eg_key.txt
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAACAQEAz4LLNBpwLcKHDjAiyutYIUFN099SXznptR9XvH6fpZ9IUWIB
YLkT/07/2034MC0aGLYyQq4d5b7NZycKNWzkfshDwFly5Ei8PG4t0WRaGaEQZ13E
NT+wn7RpE/83qd9Tj5gzva6RjZ8QlomIUI2lQJf0d8hVpt9c37RuY4G9HxAHcZvx
Bw0LFpEQ8098qga5edsJdm0yW1fBz+CULILj6Gv2QWH/7VyVMXS8pj0UJf0DVf8f
vW12w+5qJXvUcXLWj6ohha+F3hTSWFWRoAP3RuHBrEoi28iqaQB4pmGabXCXDKvd
m8qWjtzpk5S0jYnMLXfYLS6pAZd+zxYHBnV08wIDAQABAoIBAHzJVLpjRmSN+Ac
kgQaz9w2CQ03LPijoGRqJ3jExhCRqx7vqAa7HdL18yIppWmBMr+Wvi/LjwWkAvwN
6ybmpnuAk/i84ayMPcqFp0/gjWVQDmq5IPjp3j3D7BEDX34x0LQjgmKXILYqNEi2
C9qQQgr/ASWtFnmML0c5xcYQcm99mxjwdwLjuYQF5xVYQm7oKqDG/+W8RJ0m9jN0
K41l06zy3qUDBawz8YCpq0YT5sPXxNd4imV9LuZzWMTQqcZbaakRy2s6IKHLJSJL
GXg528U4QbZi5HND3Pi51JrQtUdin2FynQWamFVzC5TXj6oujSs/8+D0eoiv2kZ
k+Uk9rECgYEA6FG98lkxwQYN4nne3afMkRG0zkGfvFA7MGmW2PTL+1K+kuxsZ8kA
qhnyMYxMJa3p06kV5iG6pKnVLVkcTLdEidZyQ3FzRbCwTQuVzIhBgG2wMXFqeZrc
xyLpsVfIKRaU7/IkHVLL7nkVibgqcZ1j1wulopULSu2jHv4p9L3zrvcGgYEA5Lda
ABJUNo6mV6MQujSf0HX4W5RaeyVFcp3Y4rldj04tQM5upC8kjiJlfQXRQU1ccqgs
cdnJIXos8SSyUZLTWajfv2WaXE/yU5IJL4ZP02r5U+PwdMTa9rG5ER99jnf1Ds5k
gnK51w0cwcq7DK3nyf0LVpW9duM497060yFLOUCgYEAhYf+Wp7j1y0ItYUBEXs
04vmlaquoio+/BZubRcamvCUaPs6/nIP06vaQ2+PdHMcyoCM04EDAy8aGLlOKzZmF
PoJREmzSKdtd6lrFvjL3FL93R2P+JD1b7uXlVray6NL+9k/CbExbxQ05LTKzFT/
vuoPE40Qa1ndqYHDudYgi48CgYEAj+SHBoyB2GIhX/sGR2Nd0zk/L8BEvj9RVzDi
iqW/qzEh4Ckdj0Lsh+Yrn2LNq50Vkj8m1+wElswdiY1hddpAygU0WyV+3p6fBt5Y
UULdb1Vb0+EUJyTF8XsZyyBZTA9Gbi6HJ10weY2YUIdS8kNqGLPpUHHKUMd3zMm
P/dCBdECgYEAiH3Cv+l4j9oAZYnhlnwJVp9VX8oz+rn7jwf3+WQ1G0hwsbyYVc5j
3bwP5Z8e7DFvlv3b8yNJfy5yMccx8rrTXgsBBbbr9lk9kLK/pXxHYvYSQcetceKH
BQwF1WUbx60akn9AuERlwh6iQmftiEqqfyBUvqAM31Bc1fA6x1wvHUc=
-----END RSA PRIVATE KEY-----
```

Aquí podemos extraer la clave pública (que está guardada en el fichero junto con la privada).

```
fons@Fonss-mb p3_kin % openssl rsa -in eg_key.txt -pubout -out public_key.txt
writing RSA key
fons@Fonss-mb p3_kin % cat public_key.txt
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAz4LLNBpwLcKHDjAiyutY
IUFN099SXznptR9XvH6fpZ9IUWIBYLkT/07/2034MC0aGLYyQq4d5b7NZycKNWzk
fshDwFly5Ei8PG4t0WRaGaEQZ13ENT+wn7RpE/83qd9Tj5gzva6RjZ8QlomIUI2l
QJf0d8hVpt9c37RuY4G9HxAHcZvxBw0LFpEQ8098qga5edsJdm0yW1fBz+CULILj
6Gv2QWH/7VyVMXS8pj0UJf0DVf8fvW12w+5qJXvUcXLWj6ohha+F3hTSWFWRoAP3
RuHBrEoi28iqaQB4pmGabXCXDKvdm8qWjtzpk5S0jYnMLXfYLS6pAZd+zxYHBnV0
8wIDAQAB
-----END PUBLIC KEY-----
```

Profundizando más, podemos ver más información sobre las claves:

'n' = pq


```
fons@Fonss-mb p3_kin % openssl rsa -text -in eg_key.txt -noout
Private-Key: (2048 bit)
modulus:
 00:cf:89:4b:34:1a:70:2d:c2:87:0e:30:22:ca:eb:
 58:21:41:4d:3b:df:52:5f:39:e9:b5:1f:57:bc:7e:
 9f:a5:9f:48:51:62:01:60:b9:13:fc:ee:ff:db:4d:
 f8:30:23:9a:1a:56:32:42:ae:1d:e5:be:cd:67:27:
 0a:35:6c:e4:7e:c8:43:c0:59:72:e4:48:bc:3c:6e:
 2d:39:64:5a:19:a1:10:67:5d:c4:35:3f:b0:9f:b4:
 69:13:ff:37:a9:df:53:8f:98:33:bd:ae:91:8d:9f:
 10:96:89:88:50:8d:a5:40:97:ce:77:c8:55:a6:df:
 5c:df:b4:6e:63:81:bd:1f:10:07:71:9b:f1:07:03:
 a5:16:91:10:f0:ef:7c:aa:06:b9:79:db:09:76:63:
 b2:5b:57:c1:cf:e0:94:2c:89:63:e8:6b:f6:41:61:
 ff:ed:5c:95:31:74:bc:a6:3d:14:25:f3:83:55:ff:
 1f:bd:6d:76:c3:ee:6a:25:7b:d4:71:72:f0:8f:aa:
 21:85:af:85:de:14:d2:58:55:91:a1:a3:f7:46:e1:
 c1:ad:ea:22:db:c8:aa:69:00:78:a6:61:9a:6d:70:
 97:0e:4b:dd:9b:ca:96:8e:dc:e9:93:94:90:8d:89:
 cc:2d:77:d8:2d:2e:a9:01:97:7e:cf:16:07:06:75:
 4e:f3
```

'e' = 65537

'd' = privateExponent

'p' = prime1

```
publicExponent: 65537 (0x10001)
privateExponent:
 7b:c9:56:53:e3:99:19:92:37:e0:1c:92:04:1a:cf:
 dc:36:09:0d:37:2c:f8:a3:a0:64:6a:27:78:c4:c6:
 10:91:ab:1e:ef:a8:06:bb:1d:d2:f5:f3:22:29:a5:
 69:81:32:bf:96:be:2f:cb:8f:05:a4:02:fc:0d:eb:
 26:e6:a6:7b:80:93:f8:bc:e1:ac:8c:3d:ca:85:a4:
 ef:e0:8d:65:50:0e:6a:b9:20:f8:e9:de:3d:c3:ec:
 11:03:5f:7e:31:38:b4:23:82:69:17:20:b6:2a:34:
 48:b6:0b:da:90:42:0a:ff:01:25:ad:16:79:8c:2c:
 e7:39:c5:c6:10:72:6f:7d:9b:18:f0:77:02:e3:b9:
 84:05:e7:15:58:42:6e:e8:2a:a0:c6:ff:e5:bc:44:
 93:a6:f6:33:74:2b:8d:65:3b:ac:f2:de:a5:03:05:
 ac:33:f1:80:a9:a8:e6:13:e6:c3:d7:c4:d7:78:8a:
 65:7d:2e:e6:73:58:c4:d0:a9:c6:5b:69:a9:11:cb:
 6b:3a:20:a1:cb:25:22:4b:19:78:39:db:c5:38:41:
 b6:48:e4:73:43:dc:f8:b9:d4:9a:d0:b5:47:62:9f:
 61:72:9d:05:9a:d2:61:55:cc:2e:53:5e:3e:a8:ba:
 34:ac:ff:cf:83:d1:ea:22:bf:69:19:93:e5:24:f6:
 b1
prime1:
 00:e8:51:bd:f2:59:31:c1:06:0d:e2:79:de:dd:a7:
 cc:91:11:b4:ce:41:9f:bc:50:3b:30:69:96:d8:f4:
 cb:fb:52:be:92:ec:6c:67:c9:00:aa:19:f2:31:8c:
 4c:25:ad:e9:d3:a9:15:e6:21:ba:a4:a9:d5:2d:59:
 1c:4c:b7:44:89:d6:72:43:71:73:45:b0:b0:4d:0b:
 95:cc:88:41:80:6d:b0:31:71:6a:79:9a:dc:c7:22:
 e9:b1:57:c8:29:16:94:ef:f2:24:1d:52:cb:ee:79:
 15:21:b8:2a:71:9d:63:d7:0b:b5:a2:95:25:4a:ed:
 a3:1e:fe:29:f4:bd:f3:ae:f7
```

'q' = prime2


```

prime2:
00:e4:b0:da:00:12:54:36:8e:a6:57:a3:10:ba:34:
9f:d0:75:f8:5b:94:5a:7b:25:45:72:9d:d8:e2:b9:
5d:8c:ee:2d:40:ce:6e:a4:2f:24:26:28:e5:7d:05:
d1:41:4d:5c:72:a8:2c:71:d9:c9:23:1a:2c:f1:24:
b2:51:92:d3:c0:02:5f:bf:65:9a:5c:4f:f2:53:92:
09:2f:86:4f:d3:6a:f9:53:e3:f0:74:c4:da:f6:b1:
b9:11:1f:7d:8e:77:f5:0e:ce:64:82:72:b9:d7:03:
9c:c1:ca:bb:0c:ad:e7:c9:f3:a5:56:95:96:f5:db:
8c:e3:de:f4:eb:4c:85:94:e5

```

1d Diferencia entre velocidades cifrado simétrico/asimétrico

Vamos a comparar el tiempo de procesamiento de los algoritmos AES y DES (cifrados simétricos) con el RSA (cifrado asimétrico). Para ello, ejecutamos el comando speed de SSL y analizamos primero los resultados de AES y RSA en la terminal:

```

The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes    64 bytes    256 bytes    1024 bytes    8192 bytes
aes-128 cbc    123741.85k    140212.91k    139557.42k    323429.72k    317158.74k
aes-192 cbc    112304.35k    117452.14k    119863.98k    269147.48k    261332.99k
aes-256 cbc    96056.28k     103765.03k    103130.28k    230524.59k    239372.97k

          sign    verify    sign/s    verify/s
rsa  512 bits 0.000050s 0.000003s 19900.9 289896.1
rsa 1024 bits 0.000150s 0.000009s 6670.6 108084.6
rsa 2048 bits 0.000706s 0.000032s 1415.6 30938.3
rsa 4096 bits 0.007770s 0.000116s 128.7 8609.7

```

Podemos observar que AES cifra los bits mucho más rápido que RSA, pero como sabemos RSA es un sistema de clave pública que se basa en generar “agreeing” entre dos usuarios más que en cifrar grandes cantidades de datos, siendo la velocidad en que estos se cifran verdaderamente irrelevante. AES por su parte puede cifrar flujos de datos bastante rápida pero solo si la clave simétrica ha sido comprobada previamente. Por tanto, tiene poco sentido esta comparativa.

Hemos decidido también comparar la velocidad de procesamiento del DES, que como observa la imagen, es considerablemente más lento que el AES (aproximadamente la mitad):

```

The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes    64 bytes    256 bytes    1024 bytes    8192 bytes
des cbc        69493.64k    70756.02k    71077.72k    70896.64k    65901.91k
des ede3       24865.71k    24883.16k    24628.74k    26778.28k    26146.13k

```

Por tanto, como hemos comentado anteriormente, no tiene mucho sentido comparar las velocidades entre los cifrados simétricos y asimétricos, puesto que no se está haciendo una comparativa real. De hecho, lo que se hace normalmente es cifrar la clave con RSA (suele contener pocos bits) y luego utilizar un cifrado simétrico, por ejemplo AES, para cifrar las grandes cantidades de datos almacenados en los ficheros a enviar.

1e Certificados X.509

Los certificados digitales son el equivalente digital del DNI, en lo que a la autenticación de individuos se refiere, ya que permiten que un individuo demuestre que es quien dice ser, es decir, que está en posesión de la clave secreta asociada a su certificado.

Para los usuarios proporcionan un mecanismo para verificar la autenticidad de programas y documentos obtenidos a través de la red, el envío de correo encriptado y/o firmado digitalmente, el control de acceso a recursos, el uso de estos es Agencia Estatal de Administración Tributaria, Banco de España, Boletín Oficial del Estado, Catastro, Instituto Nacional de Estadística, Loterías y Apuestas del Estado, Ministerios del Estado entre otros muchos.

Ahora vamos a generar nuestro certificado gracias a SSL. Lo primero es ejecutar el comando y tras esto, se contestan unas preguntas con información que será incorporada al certificado.

```

ps@Labvirtips:~/Escritorio/crtp/p3_kin$ openssl req -nodes -newkey rsa:2048 -keyout example.key -out example.crt -x509 -days 365
generating a 2048 bit RSA private key
.....+++
writing new private key to 'example.key'
-----
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]:SP
State or Province Name (full name) [Some-State]:Madrid
Locality Name (eg, city) []:Madrid
Organization Name (eg, company) [Internet Widgits Pty Ltd]:UAM
Organizational Unit Name (eg, section) []:Crypto
Common Name (e.g. server FQDN or YOUR name) []:David Quintana Ruiz
Email Address []:david.quintana@estudiante.uam.es

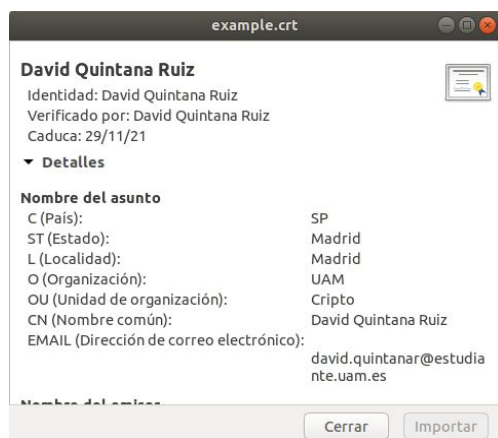
```

Hecho esto, podemos mostrar la información del certificado con el siguiente comando:

```

eps@labvirt:~/Escritorio/cripto/p3_kin$ openssl x509 -in example.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            bd:6b:56:a1:e6:20:e1:fb
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=SP, ST=Madrid, L=Madrid, O=UAM, OU=Cripto, CN=David Quintana Ruiz/emailAddress=david.quintanar@estudiante.uam.es
        Validity
            Not Before: Nov 29 21:41:53 2020 GMT
            Not After : Nov 29 21:41:53 2021 GMT
        Subject: C=SP, ST=Madrid, L=Madrid, O=UAM, OU=Cripto, CN=David Quintana Ruiz/emailAddress=david.quintanar@estudiante.uam.es
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
            Modulus:
                00:eb:34:c2:67:89:f5:4c:49:fa:3d:6b:ee:8e:88:
                f9:87:3c:9a:d0:0f:6c:9f:cb:2d:b3:2a:7d:0c:de:
                5b:35:b0:dc:aa:52:af:b3:e4:70:7c:31:be:71:9d:
                4e:e2:70:58:0f:d4:ed:dc:df:63:9c:c3:22:0a:e2:
                4e:36:4f:22:46:84:e9:74:a5:49:94:fe:81:fc:a0:
                0e:4c:ac:21:c0:81:d4:e3:fa:07:1c:ed:a5:61:29:
                f0:1b:5b:e1:20:3c:8d:3b:46:18:44:80:a6:d6:3b:
                96:74:b7:73:4a:bb:72:d7:f1:66:f1:2a:31:81:47:
                99:50:d6:a1:ee:e5:44:23:e8:8c:1f:23:8c:2b:f0:
                4d:a6:95:c7:44:5e:6d:2a:73:ca:50:7f:7f:6d:e8:
                ec:ff:a8:97:6b:02:e6:b9:4a:43:10:d9:08:4a:ca:
                e4:98:bf:aa:24:7b:28:2f:d6:e3:17:88:4d:d5:ee:
                eb:1a:ba:7a:5e:42:09:36:4a:1b:2a:95:69:2d:09:
                3e:3f:ac:6d:ba:fa:db:0d:2c:be:d9:1c:19:4c:7f:
                a0:52:9e:f2:ce:08:04:d9:c5:27:c7:a3:ea:f2:71:
                ca:66:7d:2d:d1:b4:2f:61:9a:f1:e4:8f:eb:83:83:
    
```

El certificado se mantendrá almacenado en un archivo .crt (en nuestro caso example) para poder ser transportado y consultado cuando se desee.



2 RSA

2a Potenciación de grandes números

Vamos a generar un programa llamado “potencia” que sea capaz de calcular la potencia de números muy grandes de manera rápida y eficiente. El pseudocódigo encontrado en los apuntes ha sido de gran ayuda para la implementación del mismo. Para ello, lo primero es preguntar al usuario por pantalla que escoja una base, un exponente y un módulo con el que se realizará la operación y guardamos la cadena leída por pantalla en un `mpz_t`. Hecho esto, llamamos a la función que se encarga de realizar el algoritmo, `modularExponentiation(mpz_t result, mpz_t base, mpz_t exponent, mpz_t module)`, el cual se detalla a continuación:

- Crea unas estructuras `mpz_t` auxiliares para no modificar las variables que se pasan como argumento y se imprimen bien en la terminal. También se generan los `mpz` 0, 1 y 2 que nos servirán para hacer comparaciones.
- Si el módulo es 1, el resultado será 0.
- En caso contrario, antes de comenzar el bucle, hacemos que `base = base % módulo`.
- Aquí comienza el bucle, en el cual, mientras que el exponente sea mayor que 0, se comprueba si el exponente % 2 es igual a 1, y en tal caso nuestra variable `result` se modificará acorde con la fórmula $(result * base) \% \text{módulo}$. A continuación, habría que hacer un right shift del exponente (esto es dividir entre 2). Por último, modificamos la base acorde con la fórmula $(base * base) \% \text{módulo}$.
- El valor de `result` es el producto de todas las bases² para todas las potencias de 2 que constituyen el exponente.
- Liberamos la memoria y salimos.

Al final del programa, se comprueba que el algoritmo ejecuta correctamente las peticiones del usuario, mediante la función `mpz_powm(result_gmp, base_gmp, exponent_gmp, mod_gmp)`, incluida en la librería GMP de C. Simplemente, inicializamos una estructura `mpz_t` para cada variable recogida por pantalla. A continuación, mostramos un ejemplo del correcto funcionamiento de nuestro programa:


```

eps@labvirteps:~/Escritorio/cripto/p3_mpz$ ./potencia
This program called Potencia computes base to the power exponent module m
Enter base:
3762687462987346298734629873462987634
Enter exponent:
4786349875634358769837465
Enter module:
8378932792847
3762687462987346298734629873462987634 to the power 4786349875634358769837465 = 8162309041711 (mod8378932792847)
You can compare the results. Here is the result using the GMP library. RESULT = 8162309041711

```

2b Generación de números primos: Miller-Rabin

Para esta parte hemos realizado 2 apartados distintos. Uno para la generación de números potencialmente primos de hasta n bits (el primer apartado). El segundo, para la generación de números primos de exactamente n bits.

Son bastante parecidos, solo cambia la función de generación aleatoria del número (pues en el segundo caso es necesario poner el bit n a 1 para asegurar que el número tiene n bits significativos).

```

272 void gen_odd_num_bits(mpz_t num, int bits){
273     gen_odd_num(num, bits);
274     // /*set msb to 1*/
275     mpz_setbit(num, bits - 1);
276 }

```

La primera clave de este apartado es sacar el número de rondas necesarias para el test de Miller-Rabin. Sabiendo la fórmula para la probabilidad de que sea primo el número después de n rondas: $sec = 1 - \frac{1}{1 + \frac{4^n}{N \ln 2}}$ donde N es el número de bits que va a tener el número que estamos generando. Simplemente hay que despejar la n (ya que recibimos sec) quedando que $n \geq \log_4\left(\frac{N \ln 2 - (1 - sec) N \ln 2}{1 - sec}\right)$.

```

70
71 rounds = ceil((log((b*log(2) - (1-p_d)*b*log(2))/(1-p_d))/log(4)));

```

Además de esto, también es importante tener en cuenta el número de intentos que queremos realizar hasta encontrar el primo. Se tiene por Gauss que para N bits, hemos de realizar $\ln(2^N)/2$ intentos. Por ejemplo: para 1024 bits tenemos $\ln(2^{1024})/2 \approx 355$. Se va probando por el test de Miller-Rabin el número aleatorio. Cada vez que no pase el test, volvemos a generar otro número aleatorio hasta agotar el número de intentos.

```

83 for(i = 0; i < max_tries; i++){
84     isPrime = millerRabin(num, rounds);
85     // gmp_fprintf(F_OUT, "N: %Zd\n", num);
86     // printf("\n%d/%d", i, MAX_TRIES);
87     if(isPrime == FALSE)
88         gen_odd_num_bits(num, b);
89     else break;
90 }

```

Por la naturaleza del método es posible que a veces el programa termine sin haber generado ningún número, pero no suele ser el caso. Para intentar evitar esto más, hemos sumado 10 al número de intentos máximo cada vez.

Una vez tenemos el número, la probabilidad de que sea primo es: $1 - \frac{1}{1 + \frac{4^a}{N \ln 2}}$, que es lo que indicamos en la salida. Esto ocurre porque el test de Miller-Rabin confirma si es un número compuesto, pero solo puede decir con probabilidad (alta probabilidad) que es un número primo.

Aquí unos ejemplos de ejecución con diversos valores:

```
fons@Fonss-mb p3_mpz % ./primo_b -b 10 -p 0.99994

Setting F_OUT to stdout
RAND: 903

max_tries == 13

NUMBER FOUND in 4 tries: 617
MY TEST: Prime with probability: 0.999973559232589089867000000000
MY TEST: ROUNDS Passed = 9
Here is the comprobatation using GMP:
Definitely prime
```

```
fons@Fonss-mb p3_mpz % ./primo_b -b 100 -p 0.99994

Setting F_OUT to stdout
RAND: 1050074070587313019218193886417

max_tries == 44

NUMBER FOUND in 0 tries: 1050074070587313019218193886417
MY TEST: Prime with probability: 0.999983474356511670245000000000
MY TEST: ROUNDS Passed = 11
Here is the comprobatation using GMP:
Probably prime
```

```
fons@Fonss-mb p3_mpz % ./primo_b -b 700 -p 0.99994

Setting F_OUT to stdout
RAND: 2709103242291441238396120209249932155671952912339874386543
0730468205320629641011827582196565827239639646391296644939795200

max_tries == 252
PRIME NUMBER NOT FOUND after 252 tries
```

```
PRIME NUMBER NOT FOUND after 252 tries
fons@Fonss-mb p3_mpz % ./primo_b -b 700 -p 0.99994

Setting F_OUT to stdout
RAND: 390240506894364740915465946546430960649905390686896332353854595503007941341303254492165811252745807471368157612259271915431490212354562328430333
7542460318641000801738379480315573616500284311917834573738921515875

max_tries == 252

NUMBER FOUND in 130 tries: 311113455605622727424807974014047390453624504988410795910555310142198740545051219110177988767091286178182768342106929088187
0860802568613322777394331963665297720956009486702071313588670277154657645573502867722677
MY TEST: Prime with probability: 0.999971080482330652073000000000
MY TEST: ROUNDS Passed = 12
Here is the comprobatation using GMP:
Probably prime
```


14

2c Factorización RSA conociendo d (usando A. las Vegas)

Siguiendo paso a paso el algoritmo de Vegas, es posible a veces, obtener los primos p, q a partir de la clave pública (n, e) y la clave privada (n, d) .

Recordamos RSA:

1. Elegimos primos $p, q \rightarrow n = pq$.
2. $\phi(n) = (p-1)(q-1)$
3. Elegimos $e : 1 < e < \phi(n) : \text{mcd}(e, \phi(n)) = 1$
4. Elegimos $d : d = e^{-1} \text{mod}(\phi(n))$
5. La clave pública es (n, e) y la privada (n, d)
6. $y = C(m) = m^e \text{mod}(n)$, $m = D(y) = y^d \text{mod}(n)$

Utilizando los pasos del algoritmo, vemos que Vegas predice correctamente los primos que componen n mediante el conocimiento de la clave privada.

```
fons@Fonss-mb p3_mpz % ./primo_rsa -n 77 -e 7 -d 43

P = 0
Q = 0
N = 77
E = 7
D = 43
Tam en bits de N: 7
Comenzando VEGAS:

P o Q es mcd(y+1,n)
P_GUESS = 11
Q_GUESS = 7

Your Guesses are correct%
```

Hemos realizado la prueba de manera que si no se meten argumentos, genera primos aleatorios:

```
fons@Fonss-mb p3_mpz % ./primo_rsa

BITS == 79
P = 510644941162555734866983
Q = 844043
N = 431006288073667030124332932269
E = 264125352128094221799772486627
D = 231246256087308133084405995235
Tam en bits de N: 99
Comenzando VEGAS:

P o Q es mcd(y+1,n)
P_GUESS = 844043
Q_GUESS = 510644941162555734866983

Your Guesses are correct%
```

Para $N = 99$.

```
fons@Fonss-mb p3_mpz % ./primo_rsa
BITS == 883
P = 6195138461853927174207159084057833572959268693182820192091379710841441678460856526185523201729744435042971676109145848302532808532725571033697195
368864785613116843584313086637339957114821953467638231689294917922110713142437163814048286706837234514372421619818594697
Q = 254181998727659737242311952340762729
N = 1574698593611597302595237054506881704266365691498961140672963663298443200020588340605368844892963619673272875205034994884510705917864016084228310
003970560025593538728624127768692617046643246044990184383365261463537385802624620988390080981618737036502923464597632472852164524396369434903357894
648113
E = 71761976920517202617511872265118257949584464679323730239126774173075171497438537655883370028237320359314050742200426004842563350719824486906528344
524976770689593682444939197287087822050959825383687957205605420508508054192025964492797483961815378071529615067812356763536662086214432423348063224330
22131
D = 98334898008779063864033089850269831127093732547529758195573745072382227470326402141986056819905379767528904598560253867656730442657869933074303713
8587357606901741204960023757172303695282852376523387433231864172397895338510870447712523253331706512877919939737011120409073085822150104351484593371847
15195
Tam en bits de N: 1001
Comenzando VEGAS:
P o Q es mcd(y+1,n)
P_GUESS = 6195138461853927174207159084057833572959268693182820192091379710841441678460856526185523201729744435042971676109145848302532808532725571033
697195368864785613116843584313086637339957114821953467638231689294917922110713142437163814048286706837234514372421619818594697
Q_GUESS = 254181998727659737242311952340762729
Your Guesses are correct
```

Para N = 1001.

```
fons@Fonss-mb p3_mpz % ./primo_rsa
BITS == 1557
P = 34938873488415165797208180408216193407554041120112800720645286491774244776333145932816207409106302875791798459663112632965608229635027206790038714
713855654118087824469026288383534715551392344790711975782454780038661089619603173576769789086408755029863850984558927889370464554280706638668967328
042266166772748182062471520050664324735501900477372811868609856965295549723176023455637203736611873632525740378231457148270592429681940975462666164987
86722621186504041328001
Q = 27894271185115447084632931090660471444767909223509931826310527639382658810977186517506031664069032452336379077218186464091883343039936473909670502
89
N = 974594441198829318169865803818332156229981059733662081007912754902659512254920501096476386714541744487017385605133478373954990471767451483967319183
47455000240884764034101999799910236458589048186026952489514289935116530341992769430996264221506921515819164625539445698606438191810825977007603286633
38172081548731024819892608634652356968792918044620155408062466952553844023908958758051073510566115821553704332047332031772375311532068031566632711575
9815521563980236651639639381344050778711391658821304079417020457058554912929143096962431780119678855097123948682567755773121029700573368283979667895
69583633301310842289
E = 457581067970882395386060901485558411213352250841362137118736980037837561087279032756864739793507546173581214801006134274527100368347044028762518
997820456029614122556690451253152269365035913137007645146434089259909926246047598279763446921971403753422463835843789397393641764921898819266293807679
042596001692374828151232680286057720579762474516481162285762941257606421469267322328639882742241503840478904555418170967554159936435033801829348279006
703857300432117388545767833700122801320623656349807183626332334180446295094700087984901446266710118019650885708020365095405145053846797213386915096166
63487649360238577367
D = 91638687170197043940749433809619914826912277150684116870474095216549556996681411391938581369707281135712770188547590963307476930691486309418246560
1984759021093654258161400645578364320065998571568186071296536252803227748124515052767014723503307382322521275950168635744304657395308631329900325862
79749680107261784425981995089074933111776428286780643455828021563730936793309856331254084150779094365056857308806982028241150214349412652521126020968
10675420757230728026502413161460931720172876548776565135804660725157044180981499939750156571666620499414724843294193812268797927064153709261344686110
01478373123700841703
Tam en bits de N: 2047
Comenzando VEGAS:
P o Q es mcd(y+1,n)
P_GUESS = 34938873488415165797208180408216193407554041120112800720645286491774244776333145932816207409106302875791798459663112632965608229635027206790
038714713855654118087824469026288383534715551392344790711975782454780038661089619603173576769789086408755029863850984558927889370464554280706638668
967328042266166772748182062471520050664324735501900477372811868609856965295549723176023455637203736611873632525740378231457148270592429681940975462666
16498786722621186504041328001
Q_GUESS = 27894271185115447084632931090660471444767909223509931826310527639382658810977186517506031664069032452336379077218186464091883343039936473909
67050289
Your Guesses are correct
```

Para N = 2047

```
881910540443780810033047336719505
898771329
Tam en bits de N: 4999
Comenzando VEGAS:
No Responde
```

Aquí un ejemplo en el que el algoritmo no funciona. Esto es por que el algoritmo de Vegas dice con seguridad si obtiene los primos, pero no siempre es capaz de asegurar obtenerlos.

```
4053537344331622540881748098706516326855499253108801148092329409703408919485391722044818515283925736207867402768245993583072149849101304132089383321
26137257442618777186990635242633600084766820294470656606504359896175734472052455298396019139934738465622117425445308272560213066717940784458444316346
076433628477917182515182768068368862208447186582654011790093950083975275678267486996458816886845656287712786071748941671371233855473262685490815049690
37793799
Tam en bits de N: 4998
Comenzando VEGAS:
P o Q es mcd(y+1,n)
P_GUESS = 802325081826871320063093986307810338420039587941273269304105428163348332863326606090776520728568304579872942310282615261424126673672390921751
605376886362490192936977341875434233013540960154470663160451407501487401855108026473629227283276987488248143741871855902622733155455090544916440574115
861826400382559633733676385092137251827168363781142798651176408870051540079960684263764920227473394004033460775471011769947231190131445692559946343442
30595571801132173839168299389052130283913637704953335421954358509220019870912277671872524272976106645549641988889294922411905510432418709350191620535
047055377660236544512964391737745421640492912835785109444032864384576627948497371203428334603437110433006281957091158465798108561088936951166230944207
490815078782025316356738476254336363449214219066071942657538274483192659837803155620755444059513452523434636149698618033710666086944883434209155642817
Q_GUESS = 225646650803224221920323803087509302821928192627513834225527492693927527322807946848523592053166059983492146593578999266892227704568259655
0599559989219539890051086507673346851684618390140151284068568432227537098871789455323836962056237701954589570922379328785301791503759562096042815995
2550418344600753646911902099187187045438112005960472395980775967646227234642399750105013851752550671735125506129943002364162760654053677154199166140
5297696610857053400961521657004583862290460784039608482637390003226066639193663866825974228193716425745346020659020246331411650519656698463894619662308
7750776197193787063130741
Your Guesses are correct
```

Pero finalmente aquí vemos para 4998 bits que sí ha funcionado.