CriptoCert

www.criptocert.com

¿Cómo se generan y/o eligen los dos números primos p y q empleados en RSA a la hora de generar el par de claves, pública y privada?... y otras cuestiones interesantes asociadas a la generación de las claves en RSA ;-)

Copyright: El presente artículo, y todos sus contenidos, son propiedad y copyright de CriptoCert S.L. (Copyright © 2019 CriptoCert S.L.) y fue proporcionado durante el mes de septiembre de 2019 en primicia inicialmente como contenido exclusivo a los candidatos registrados en la certificación **CriptoCert Certified Crypto Analyst**. No se permite su distribución, modificación, copia, traducción o utilización en ninguna otra actividad o medio sin la autorización expresa y por escrito de CriptoCert.

Autor: Raúl Siles (raul@criptocert.com) - CriptoCert - www.criptocert.com
Fecha: 17 de octubre de 2019
Versión: 1.1

Agradecimientos: Jorge Ramió (jorge@criptocert.com) y Alfonso Muñoz (alfonso@criptocert.com); ROCA: Tomas Hidalgo (Security Administrator en empresa del Sector Bancario)

La pregunta del título {P1} surgió en el grupo de candidatos que están preparando la certificación **CriptoCert Certified Crypto Analyst** (https://www.criptocert.com/#cert-analyst), a raíz de leer la documentación oficial de la certificación asociada al funcionamiento de RSA, concretamente el módulo 7 de criptografía asimétrica y específicamente el apartado del algoritmo RSA (el temario reducido está disponible en https://www.criptocert.com/CriptoCert_Analyst.html).

Otras preguntas relacionadas que también surgieron, y que es lógico que se plantee cualquier mente inquieta y ávida de sabiduría son, ***¿al generar un par de claves RSA de 2.048 bits, la longitud de los número primos p y q es exactamente de 1.024 bits para cada uno, o puede haber ligeras variaciones en su longitud, sin que sean necesariamente números exactamente de la misma longitud? {P2}***, ***¿cómo se generan y/o eligen los números primos p y q empleados en RSA por parte de un software de referencia en la industria como, por ejemplo, OpenSSL? {P3}*** *y, por último,* ***¿una vez generadas las dos claves, pública y privada, se podría elegir indistintamente una u otra para jugar el papel de clave pública o privada? {P4}***

A continuación vamos a intentar responder a todas estas preguntas y a proporcionar referencias adicionales para quienes estén interesados en profundizar aún más en las respuestas asociadas.

De manera breve e introductoria (para más información se recomienda la lectura del libro "Cifrado de las comunicaciones digitales. De la cifra clásica al algoritmo RSA" de Jorge Ramió y Alfonso Muñoz, cuya 2ª edición ha sido publicada en 2019 [0]), el algoritmo de cifra asimétrico RSA permite tanto cifrar la información (proporcionando confidencialidad), como firmarla (proporcionando autenticidad e integridad). Centrándonos en las operaciones de cifrado, es posible cifrar un mensaje M mediante la clave pública del destinatario *(n,e)* empleando la siguiente operación de exponenciación en aritmética modular, así como descifrar el mensaje cifrado C mediante la clave privada del destinatario *(n,d)* para obtener de nuevo el mensaje M original:

Cifrado: $C = M^e \bmod n$
Descifrado: $M = C^d \bmod n$

Una clave pública RSA está compuesta de un módulo *n* (que define un grupo finito o "grupo de cifra", el grupo multiplicativo de los enteros módulo *n*), que es el producto de dos números primos *p* y *q* ($n=p*q$) y de un exponente público *e*. Por tanto, la clave pública está compuesta por el par *(n,e)*, y la clave privada está compuesta por el par *(n,d)*, donde *d* es un exponente privado que depende de *n* y de *e* en base a la fórmula de inversos. El tamaño del par de claves RSA se considera habitualmente en base a la longitud del módulo *n* en bits, siendo habitual hoy en día (por ejemplo, en los certificados digitales empleados por multitud de sitios web que hacen uso de HTTPS; ver siguiente imagen con el certificado digital de https://www.criptocert.com) el uso de claves RSA de 2.048 bits (que tomaremos como referencia en el presente artículo), *n* es un número de 2.048 bits, donde *p* y *q* son números primos de 1.024 bits cada uno. Los valores de *p*, *q* y *d* deben permanecer en secreto (por tanto, los dos números primos *p* y *q* se mantienen en secreto), mientras que los valores de *n* y *e* son públicos.

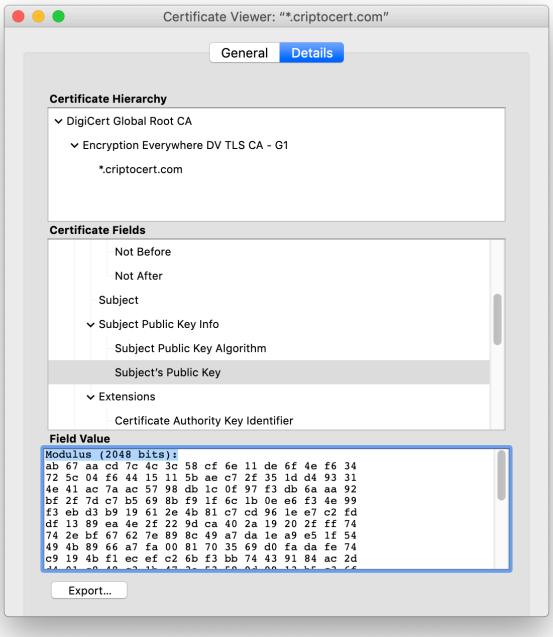


Imagen del certificado digital de la web de CriptoCert (https://www.criptocert.com), con módulo de 2.048 bits, obtenido mediante Firefox en macOS.

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

La fortaleza de RSA reside en la dificultad computacional de factorizar un número compuesto muy grande, es decir, el módulo *n*, compuesto por el producto de los dos números primos *p* y *q*.

Sin profundizar en los detalles matemáticos, una vez se dispone de *p* y *q*, y por tanto de *n*, se calcularán el resto de valores (*e* y *d*) en base a ciertas fórmulas matemáticas como, por ejemplo, el indicador de Euler del módulo *n*, $\phi(n) = (p-1)(q-1)$, un valor trampa secreto que permite elegir el exponente público *e* en el rango $2 < e < \phi(n)$, coprimo[1] con $\phi(n)$. Como detalle adicional, el indicador de Euler $\phi(n)$ proporciona el orden, es decir, la cantidad de números enteros positivos menores o iguales a *n* y coprimos con *n*, es decir, el tamaño del "grupo de cifra" definido por el módulo *n*. Teóricamente, cada usuario elegirá un valor de *e* para su clave pública asegurándose de que existe un inverso multiplicativo[2], es decir, un valor *d* para poder disponer de una clave privada asociada: $d = \mathrm{inv}(e,\phi(n))$ o $d = e^{-1} \bmod \phi(n)$.

En resumen, a la hora de generar sus claves RSA, cada usuario elige dos números primos *p* y *q* y los mantiene en secreto, obtiene el módulo $n=p*q$, que es público, genera la clave pública *e*, o *(n,e)*, y calcula el indicador de Euler $\phi(n)$ que actúa como valor trampa secreto para poder calcular la clave privada asociada *d*, o *(n,d)*[3]. El hacer públicos los valores *e* y *n* no pone en peligro la clave privada *d*, ya que para calcular *d* es necesario conocer el valor secreto trampa $\phi(n)$, es decir, los números primos *p* y *q*.

La pregunta inicial planteada {P1} centrada en cómo se eligen los números primos *p* y *q* es común y se ha repetido en foros y grupos de discusión de criptología a lo largo de los años. Conocer los detalles para elegir *p* y *q* correctamente es fundamental para cualquier desarrollador que quiera hacer uso de RSA en su software sin emplear librerías criptográficas o software de terceros, o que aun haciendo uso de éstas, quiera verificar que se hace un uso correcto de RSA. Tal como se describe en la nota 4.51 del capítulo 4 del libro "Handbook of Applied Cryptography" de A. Menezes, P. vanOorschot, y S. Vanstone, publicado en 1996 [2], y como se resume en [3], el algoritmo genérico empleado para la obtención de los números primos *p* y *q* en RSA es el siguiente:

1. Generar un número impar *p* (condición necesaria para que potencialmente sea un número primo[4]) aleatorio de 'n' bits, siendo 'n' = 1.024 bits para las claves RSA de 2.048 bits comúnmente empleadas hoy en día. Si el número generado no es impar, se le suma 1.
2. Comprobar si *p* es un número primo. Para llevar a cabo esta comprobación se pueden emplear diferentes métodos de verificación denominados **tests de primalidad**, descritos posteriormente.

[1] Los números coprimos son números enteros (positivos), por ejemplo, a y b, que no tienen ningún factor primo en común, es decir, que su único divisor común es el 1. Dos números son coprimos si su máximo común divisor (MCD) es igual a 1 (mcd [a, b] = 1).
[2] mcd $[e, \phi(n)] = 1$ (es decir, e y $\phi(n)$ son números coprimos)
[3] La clave privada *d* se calcula mediante el algoritmo extendido de Euclides.
[4] Todos los números primos (números enteros positivos) mayores que 2 son impares.

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

3. Si *p* no es primo, se puede tomar *p* = *p*+2 (búsqueda incremental) y volver a realizar la verificación del paso 2, o tomar un nuevo valor de *p* aleatorio (paso 1) y realizar de nuevo la verificación del paso 2.

Los **tests de primalidad** [9] utilizados actualmente para números grandes (de cientos o miles de bits) son test probabilísticos[5], es decir, comprueban con una muy alta probabilidad si un número grande es primo o no. Para ello tienen en cuenta ciertas propiedades conocidas que cumplen los números primos en base a la teoría de números. Si se ejecutan varias veces sobre un mismo número candidato con distintos valores semilla de entrada, y todas las ejecuciones ratifican que el número parece ser primo, se dispondrá de un mayor nivel de certeza. A un mayor número de ejecuciones del test, mayor certeza de que el número candidato realmente es primo.

El test de primalidad más comúnmente empleado en la actualidad para verificar si un número es primo es el de Miller-Rabin[6]. Este tipo de tests son bastante rápidos de calcular y, de media, a la hora de comprobar si un número de 1.024 bits es primo, sólo tienen que verificar unos cientos de candidatos posibles, ya que la densidad de números primos en un rango de números cualquiera es bastante elevada (definida por aproximadamente 1/ln(n)). El teorema de los números primos [6] refleja que la distancia entre números primos, por ejemplo de 1.024 bits, es aproximadamente 1.024*ln(2), es decir, 710. Dado que sólo hay que comprobar los números impares como posibles números primos, la verificación o test de primalidad necesitaría comprobar unos 355 candidatos (710/2) antes de encontrar el siguiente número primo de manera incremental. Como referencia, la probabilidad de que Miller-Rabin detecte si un número es primo en una ejecución o ronda es del 75% [7], por lo que tras 64 ejecuciones, la probabilidad de error es muy pequeña, del orden de 2^{-128}.

En el caso de OpenSSL [7], respondiendo a la pregunta {P3}, es posible consultar su código fuente[7] [8] (al final del presente artículo se proporcionan más detalles del código fuente de OpenSSL) para ver que hace uso de numerosos tests, encontrándose referencias a un primer filtrado mediante un algoritmo determinista de criba rápida (cuyo código está basado en la implementación y código de PGP [8]), donde se divide al número candidato por algunos números primos pequeños (los primeros 2.048 números primos[8], para descartar rápidamente si el candidato es primo o no), y si el número candidato pasa ese filtro, se emplean principalmente varias ejecuciones[9] del test de Miller-Rabin y de su versión mejorada (mencionada posteriormente en el estándar FIPS 186-4, C.3.1 y C.3.2), así como finalmente ciertas comprobaciones para hacer uso de

[5] A diferencia de las comprobaciones o tests determinísticos, como la criba de Eratóstenes, mucho más lento de calcular, o como el test AKS: https://es.wikipedia.org/wiki/Test_de_primalidad_AKS
[6] Complementariamente, existen algoritmos para la generación de números primos, como el algoritmo de Maurer (apartado 4.4.4 [2]), empleado para la generación de primos *provables*: https://pdfs.semanticscholar.org/3df0/e64897ebbed46a6d034196f9b3962ca45b07.pdf
[7] La versión actual de OpenSSL disponible en GitHub en el momento de elaboración del presente artículo era la 3.0.0-dev, mientras que la versión estable principal era la 1.1.1c.
[8] https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.h
[9] El número de ejecuciones o iteraciones del test de primalidad de Miller-Rabin depende de la longitud en bits del número primo (tal y como también establece el estándar FIPS 186-4, descrito posteriormente).

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

números primos seguros (en inglés, *safe primes*), es decir, donde (p-1)/2 también es un número primo[10] [10]. La utilización de primos seguros puede mitigar ciertos ataques cuando estos números son utilizados con el algoritmo de intercambio de clave de Diffie-Hellman, así como minimizar el número de Claves Privadas Parejas (CPP) asociadas a las claves RSA.

A modo de curiosidad, ciertas versiones de OpenSSL para claves con un reducido número de bits (128 bits), no utilizadas en la práctica actualmente, no verifican adecuadamente los números primos p y q empleados para la generación de las claves, lo que da lugar a claves RSA incorrectas, tal como se detalla en el artículo publicado en 2018 por Jorge Ramió: "¿Son correctas las claves RSA que generamos con Win64 OpenSSL1.1.0g? Intentando cifrar con RSA cuando p y q no son primos" [14].

Algunos estándares de la industria para la generación de claves RSA robustas, como ANSI X9.31[11], requerían que los números primos p y q empleados fuesen además primos fuertes (en inglés, *strong primes*), es decir, en teoría de números, un número primo que está más cerca del siguiente número primo que del número primo que le precede. En criptografía, un primo fuerte cumple ciertas condiciones relacionadas con el número primo que le precede y con el siguiente número primo [11]. El uso de primos fuertes en RSA se recomendó para minimizar los ataques de factorización mediante el algoritmo de Pollard p-1, aunque no protegen frente a otros métodos de factorización del módulo n, como Lenstra (basado en curvas elípticas) o la criba numérica o NFS (*Number Field Sieve*, o GNFS, *General Number Field Sieve*), por lo que algunos criptógrafos como Ron Rivest desaconsejaron su utilización ya en 2001 [12], debido también al mayor tiempo de búsqueda necesario para encontrar estos números primos y los limitados beneficios que aportan. En la actualidad, las implementaciones hacen uso de primos fuertes sólo si deben cumplir una certificación que lo impone como requisito (por ejemplo, IEEE 1363).

A modo de curiosidad, el record público actual de factorización de un módulo n de RSA es de 768 bits, conseguido mediante el método NFS a finales del año 2009 y documentado en "Factorization of a 768-bit RSA modulus" [19], asociado al RSA Factoring Challenge [20] (desafío que fue retirado previamente, en el año 2007).

La idea general empleada por las implementaciones reales se basa en que verificar si un número es primo con una muy alta probabilidad es mucho más rápido que factorizar dicho número, y dado que la existencia de números primos no es infrecuente cuando los números son grandes, es decir, es fácil encontrar números primos incrementando el número candidato y probando de nuevo, los tests de primalidad son muy eficientes (sin entrar en los detalles de cómo funcionan estos tests internamente, por ejemplo, verificando si un número es compuesto mediante el teorema de Fermat[12], junto a otras comprobaciones adicionales). A modo de referencia, y sin entrar en sus detalles, existen otros tests de primalidad adicionalmente al de Miller–Rabin, como Lucas (mencionado posteriormente en FIPS 186-4), Baillie-PSW, o Solovay-Strassen.

[10] Un número primo seguro se define habitualmente como un número de la forma 2p+1, donde p también es primo como, por ejemplo, 23 = 2*11+1, al ser 11 también un número primo.
[11] https://x9.org
[12] https://en.wikipedia.org/wiki/Fermat_primality_test

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

Adicionalmente, otras implementaciones basadas en el estándar FIPS 186-4 [4] (analizado posteriormente), concretamente en el apartado B.3.1 (centrado en la generación del par de claves empleadas por RSA; requisito 2(d)), una vez generados los números *p* y *q*, comprueban que la diferencia entre ambos números primos sea mayor a *m* bits, siendo *m* igual a 2 elevado a la mitad de la longitud del módulo *n* ("*nlen*") menos 100 ($|p-q|>2^{(nlen/2)-100}$), para que no se trate de números primos "muy" cercanos, lo que facilitaría factorizar el módulo *n*[13] [13]. Por otro lado, los números primos *p* y *q* empleados no pueden ser iguales, ya que si $p=q$ sería trivial factorizar *n* calculando la raíz cuadrada de *n* ($n=p*q=p^2=q^2$), lo que permitiría conocer *p* y *q*, y por tanto poder calcular $\phi(n) = (p-1)(q-1)$, y como resultado obtener la clave privada $d = \text{inv}(e,\phi(n))$.

La documentación oficial de la certificación CriptoCert Certified Crypto Analyst (https://www.criptocert.com/#cert-analyst) profundiza en otros aspectos del algoritmo RSA, incluyendo la existencia de Claves Privadas Parejas (CPP) y Claves Públicas Parejas y sus cálculos asociados, los números o mensajes no cifrables, el Teorema Chino del Resto (TCR), etc. En relación con las preguntas que nos ocupan, es interesante conocer que si ambos números primos *p* y *q* son seguros, sólo existirá una Clave Privada Pareja (el valor mínimo posible y más recomendable desde un punto de vista criptográfico).

Un breve artículo introductorio centrado en la elección de los números primos utilizados en la criptografía de clave pública es "Prime Numbers in Public Key Cryptography" [5], de Gerald Crow, publicado en 2003 en la SANS Reading Room, que describe ciertas propiedades asociadas a los números primos, ciertos tipos de números primos como los de Mersenne (2^n-1) o el concepto de números primos relativos o coprimos[1] (o números primos entre sí), y algunos teoremas relacionados, como el pequeño teorema de Fermat, que permite excluir si un número es o no primo, o el teorema de Euler.

La tesis "The properties of RSA key generation process in software libraries" de Matúš Nemec (Masaryk University - Faculty of Informatics) [1], publicada en 2016[14], realiza un análisis muy completo y exhaustivo de las diferentes aproximaciones y algoritmos empleados por las implementaciones de múltiples librerías criptográficas a la hora de generar las claves del algoritmo RSA (comparándolas con los requisitos definidos por estándares internacionales, como NIST FIPS 186-4, detallado posteriormente, ANSI X9.31 y X9.44, IEEE 1363-2000, RSAES-OAEP y otros), la selección de los números primos *p* y *q* asociados, y los tests de primalidad empleados para su verificación. La tesis profundiza en las peculiaridades de las distintas implementaciones analizadas, comparándolas entre sí, y en algunas optimizaciones existentes que intentan, por ejemplo, minimizar el número de invocaciones al generador de números (pseudo)aleatorios. Adicionalmente, describe trabajos previos, potenciales ataques sobre las claves RSA, algoritmos y métodos de factorización, e identifica propiedades comunes a claves débiles. Como conclusión, algunas implementaciones aumentan su

[13] Si los números primos *p* y *q* son "muy" cercanos, sería trivial factorizar *n* simplemente obteniendo su raíz cuadrada y buscando números impares, por arriba y por debajo de ese valor, que fuesen primos.
[14] El autor de la tesis estuvo involucrado en el descubrimiento y publicación de la vulnerabilidad ROCA [18] en el año 2017, descrita posteriormente en el presente artículo.

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

eficiencia en el proceso de generación de las claves sacrificando unos pocos bits de entropía, pero sin un impacto significativo en la seguridad de las claves generadas.

En el caso de OpenSSL, la tesis indica (en el apartado 4.2) lo comentado anteriormente tras realizar un breve análisis de su código fuente, es decir, que hace uso de una búsqueda incremental basada en la división del número candidato por algunos números primos pequeños (ya que llegar hasta la raíz cuadrada del número candidato no es práctico, por la cantidad de cálculos requeridos), teniendo en cuenta que los cálculos de división se realizan tanto para p como para $p/2$ (para emplear primos seguros). Sin embargo, los tests de primalidad no se realizan sobre $p/2$, lo que parece un bug, hecho que permite identificar claves RSA generadas por OpenSSL (v1.0.2g). La tesis también detalla el uso de OpenSSL con el módulo de compatibilidad FIPS.

Dentro de los estándares internacionales de referencia más relevantes para la generación de claves RSA, con carácter público y oficial, se encuentra la publicación del NIST conocida como FIPS 186-4, "Digital Signature Standard (DSS)", de julio de 2013 [4], y asociada a la firma digital (la certificación CriptoCert Certified Crypto Analyst cubre esta temática en el módulo 9, tal como se puede consultar en su temario: https://www.criptocert.com/CriptoCert_Analyst.html). En concreto, la sección 5.1 de FIPS 186-4, y especialmente el apéndice B.3.1, detallan los requisitos formales y más restrictivos (en comparación a otras implementaciones) para la generación del par de claves RSA y de la selección de los números primos p y q, y las relaciones entre estos, en base a FIPS 186-4 (referenciadas como IFC, *Integer Factorization Cryptography*). En resumen, se debe hacer uso de un generador de números aleatorios, o bits, de confianza, y los números primos p y q de 1.024 bits deben ser primos *provables* o *probables*[15] (en inglés), o satisfacer un conjunto de condiciones (primos con condiciones), asociadas a factores primos denominados primos auxiliares, que deben cumplir también unos requisitos de longitud mínima (> 140 bits) y máxima (su sumatorio debe ser, según si son *provables*, <494 bits, o *probables*, <1.007 bits). El estándar también define los requisitos del exponente público e y del exponente privado d, y su relación respecto a los valores de p y q. La generación de números primos *provables* y *probables*, y de primos de ambos tipos con condiciones, se describe en los apéndices B.3.2, B.3.3, B.3.4, B.3.5 y B.3.6 respectivamente (en base a los requisitos de A.1 y A.2), empleando una semilla aleatoria o tests de primalidad probabilísticos (según el apéndice C.3). Los tests de primalidad probabilísticos referenciados en FIPS 184-6 son Miller-Rabin (C.3.1), una versión mejorada de éste (C.3.2), y Lucas (C.3.3), sugiriéndose la utilización de un número mínimo de iteraciones de Miller-Rabin (según la longitud en bits de p y q y la probabilidad de error que se considera aceptable), o para más certeza (por ejemplo en DSA frente a RSA), su uso combinado, aplicando un test de Lucas adicional. El motivo es que no se conocen valores de números no primos que satisfagan el número de iteraciones del test Miller-Rabin indicadas, y también satisfagan el test de Lucas. En el caso de RSA, y como referencia, el número de iteraciones del test Miller-Rabin es de 5 siendo p y q números primos de 1.024 bits (con una probabilidad de error de 2^{-112}).

[15] En teoría de números, los números primos *provables* son aquellos que se ha calculado que son primos mediante algoritmos que permiten comprobar que no son números compuestos (garantizan que son primos), mientras que los números primos *probables* son aquellos que han pasado tests de primalidad probabilísticos y que muy probablemente (pero no con total certeza) son primos [1].

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

Los dos números primos *p* y *q* son habitualmente generados aleatoriamente con una longitud de 1.024 bits y posteriormente se comprueba su primalidad (tal y como se ha descrito anteriormente). Algunas implementaciones, incluyendo OpenSSL, generan un número aleatorio de 1.024 bits, asegurándose de que es un número impar (es decir, su bit menos significativo debe ser 1) y, por tanto, potencialmente primo. Adicionalmente se aseguran de que su bit más significativo también es 1 (el número comienza por 1 en binario) para estar seguros de que el número es de 1.024 bits, y no de 1.023 bits, asegurándose así también de que el módulo *n* será de 2.048 bits (o al menos de 2.047 bits[16]). Respondiendo a la pregunta {P2}, teóricamente no necesariamente ambos números *p* y *q* deben ser de 1.024 bits, para que su producto *n* sea de 2.048 bits, sino que uno de ellos podría ser algo más pequeño (por ejemplo, *p* de 1.020 bits), y el otro algo más grande (por ejemplo, *q* de 1.028 bits), siendo el producto *n* de ambos también de 2.048 bits. El conocer de antemano la longitud (1.024 bits) de ambos números primos *p* y *q* no facilita los ataques por factorización del módulo *n*. Lo importante es que el módulo *n* sea de 2.048 bits. De hecho ocurriría lo contrario, si uno de los dos números primos *p* o *q* fuera significativamente más pequeño, se facilitarían los ataques de factorización, que se centrarían en ese número más pequeño o débil. Se debe tener en cuenta que se está trabajando con números de gran tamaño y que la cantidad de números primos de 1.024 bits que existe es muy elevada. Además, por el teorema de Dirichlet, los números primos están uniformemente distribuidos en el módulo *n*.

Una vez elegidos los números primos *p* y *q*, es posible calcular *n*, junto a $\phi(n)$, y se pueden por tanto generar las claves pública y privada, *e* y *d*, o (*e*,*n*) y (*d*,*n*). Se podría pensar que dado que los datos que se cifran con la clave pública se pueden descifrar con la clave privada, y teniendo en cuenta que ambas claves permiten realizar operaciones matemáticas inversas, una vez generadas ambas claves, se podría indistintamente seleccionar inicialmente cualquiera de ellas para que actuase como la clave pública y/o como la clave privada, siendo la otra clave su inversa. Es decir, cuando un software con soporte para RSA, como por ejemplo OpenSSL, genera las dos claves, se podría pensar que es posible elegir indistintamente una u otra como clave pública o privada, aunque en realidad no es así. Planteado de otra forma, se podría pensar que el exponente público *e* (de la clave pública) y el exponente privado *d* (de la clave privada) son de un tamaño en bits similar, tal y como ocurría con *p* y *q*.

En realidad, y respondiendo a la pregunta {P4}, se recomienda elegir un valor de clave pública *e* pequeño en relación con el módulo *n* [0], ya que esto asegura que el valor de la clave privada *d* será muy grande, y potencialmente más segura, al dificultarse su obtención mediante fuerza bruta. En la práctica, se toma como valor único y conocido del exponente público *e* el número 65.537, conocido como número 4 de Fermat o F_4.

Tomando los dos números primos *p* y *q* de 1.024 bits, y el módulo *n* de 2.048 bits, al igual que $\phi(n)$, y dado que $d = \text{inv}(e, \phi(n))$ y que *e* es el valor 65.537 (compuesto por 17

[16] Posteriormente se describirá como OpenSSL se asegura en realidad de que los dos primeros bits de los números primos *p* y *q* tienen el valor 1, por lo que el módulo *n* será del tamaño esperado, 2.048 bits.

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

bits), por concepto de inversos, el valor de *d* será como mínimo de 2.031 bits (2.048-17 bits).

Adicionalmente, el número 4 de Fermat, empleado en *e*, presenta propiedades muy interesantes para las operaciones de cifrado de RSA. Debido a que el número 65.537 sólo tiene dos bits con valor 1 (y 15 bits con valor 0), 0x10001 en hexadecimal, al aplicar el algoritmo de exponenciación rápida en las operaciones de cifrado, donde *e* actúa de exponente, $C = M^e \bmod n$, debido a que está compuesto de muchos ceros[17], permite agilizar las operaciones de cálculo [0], mejorando la eficiencia del proceso de cifrado de RSA. Por tanto, las operaciones de cifrado con la clave pública *e* en RSA son mucho más eficientes y rápidas que las operaciones de descifrado con la clave privada *d*.

En resumen, la clave pública basada en el exponente público *e* es de reducido tamaño, los 17 bits del número 4 de Fermat, mientras que la clave privada basada en el exponente privado *d* tiene un tamaño cercano al módulo *n* (2.048 bits).

Para agilizar las costosas operaciones de exponenciación con números grandes involucradas en el proceso de descifrado de RSA empleando la clave privada *d* se hace uso del Teorema del Resto Chino (TRC), descrito en detalle en la documentación de la certificación CriptoCert Certified Crypto Analyst. El destinatario de un mensaje cifrado, al disponer tanto de *d*, como de los números primos *p* y *q*, podrá realizar los cálculos de aritmética modular en los grupos de *p* y *q*, de 1.024 bits, en lugar de en el grupo del módulo *n*, de 2.048 bits, reduciendo el tiempo de cómputo (en unas 4 veces [0]). Como se verá posteriormente de forma práctica, OpenSSL proporciona ciertos valores adicionales a la hora de generar las claves RSA que permiten hacer uso del TRC en el proceso de descifrado.

Para finalizar con el análisis teórico que nos ocupa, y antes de mostrar algunos ejemplos prácticos con OpenSSL, un caso real muy interesante relacionado con la incorrecta selección de los números primos *p* y *q* empleados para la generación de las claves RSA es el de la vulnerabilidad **ROCA** (The Return of Coppersmith's Attack) [18], también identificada con el CVE-2017-15361, publicada en el año 2017. Un grupo de investigadores del departamento CRoCS de la Facultad de Informática de Masaryk University (en la República Checa) descubrieron que la librería criptográfica de RSA empleada por el fabricante Infineon Technologies AG en sus numerosos productos hardware criptográficos de seguridad (smartcards, tokens, TPMs, chips, etc.), desde el año 2012 (incluso en dispositivos certificados NIST FIPS 140-2 y CC EAL 5+), sufría de una debilidad a la hora de elegir los números primos *p* y *q*. Como resultado, era posible factorizar de manera práctica el módulo *n* de las claves RSA incluso de hasta 1.024 y 2.048 bits (en el orden de días, si se dispone de suficiente potencia de cálculo). Es suficiente con disponer de la clave pública, ya que la misma incluye el módulo *n*, para factorizar *n* y obtener los números primos *p* y *q*, debido a que la generación de los números primos hace uso de una estructura muy específica[18] (que no proporciona la suficiente entropía o aleatoriedad), lo que permite aplicar una variante del método de

[17] Técnicamente se indica que F4 tiene un peso de Hamming bajo: https://en.wikipedia.org/wiki/Hamming_weight

[18] $p=k*M+(65537^a \bmod M)$

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

factorización de Coppersmith sobre el módulo *n* y, como resultado, poder obtener la clave privada *d* y suplantar al propietario de la clave. El método original de factorización de Coppersmith permite encontrar *p* y *q* a partir del módulo *n* si se conocen un número suficiente ($1/4*\log_2 n$) de los bits más significativos de *p*. Desde un punto de vista práctico, si la mitad de los bits de *p* (o de *q*) son conocidos, es posible factorizar *n* en tiempo polinomial[19]. Asimismo, los investigadores proporcionaron herramientas online y offline para verificar de forma inmediata si una clave pública hace uso en su módulo *n* de primos que son vulnerables a ROCA [18]. En concreto, algunas longitudes de clave son más fácilmente factorizables de manera práctica, dentro de las que se encuentra el rango de 1.984 a 2.144 bits, es decir, las claves de 2.048 bits utilizadas comúnmente hoy en día.

Un ejemplo práctico del impacto de esta vulnerabilidad fue el asociado al DNI electrónico (DNIe) empleado en España para la identificación digital y la firma electrónica de sus ciudadanos, lo que obligó al Cuerpo Nacional de Policía a revocar el certificado digital de los DNIe emitidos desde abril de 2015[20] [21] [22]. La vulnerabilidad ROCA enfatiza, de nuevo, la importancia de que los números primos *p* y *q* sean realmente aleatorios en RSA.

Mediante los siguientes ejemplos se analiza la vulnerabilidad ROCA sobre tres certificados digitales de autentificación asociados (y extraídos) de tres DNIe, cada uno de ellos emitido en un periodo temporal concreto alrededor de la publicación de la vulnerabilidad ROCA. Los certificados digitales incluyen las claves públicas correspondientes con distintas longitudes de clave (en bits), tal como se detalla a continuación:

- Certificado digital del DNIe emitido con una clave de 2.048 bits estándar en febrero de 2017, y por tanto vulnerable a ROCA, antes del anuncio público de la vulnerabilidad: cert_2048bits_20170223.der.
- Certificado digital del DNIe emitido con una clave de 1.920 bits, no estándar, al ser renovado en enero de 2018, empleando una versión de la librería RSA vulnerable a ROCA: cert_1920bits_20180102.der.
- Certificado digital del DNIe emitido con una clave de 2.048 bits estándar en febrero de 2019, y por tanto, no vulnerable a ROCA, una vez actualizada la versión de la librería RSA vulnerable en los nuevos DNIe: cert_2048bits_20190201.der.

Es necesario aclarar que el segundo certificado corresponde a la renovación del certificado digital en un DNIe vulnerable a ROCA en enero de 2018, es decir, que dispone en su chip de la versión vulnerable de la librería RSA. Para mitigar el impacto de la vulnerabilidad ROCA, la Policía procedió a la renovación de los certificados digitales

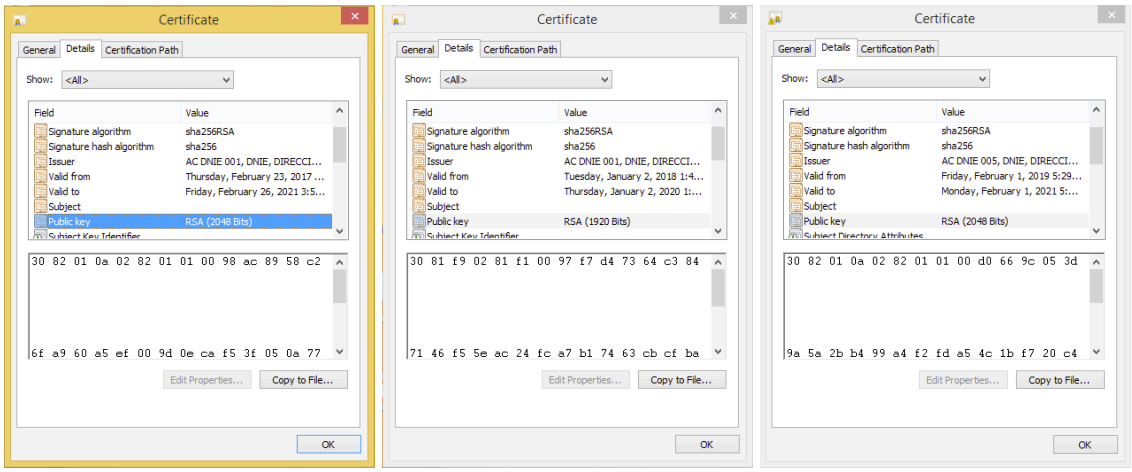
[19] https://nitaj.users.lmno.cnrs.fr/rsa28final.pdf
[20] https://unaaldia.hispasec.com/2017/11/nueva-victima-del-ataque-r-o-c-a-el-dnie-espanol.html
[21] https://www.xataka.com/seguridad/la-seguridad-del-dni-electronico-comprometida-a-quien-afecta-por-que-y-como-solucionarlo y https://www.xataka.com/seguridad/los-problemas-crecen-para-un-dni-electronico-que-es-un-fracaso-como-metodo-de-autenticacion
[22] https://elpais.com/politica/2017/11/09/actualidad/1510217634_470836.html

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

empleando una longitud de clave de 1.920 bits[23], es decir, con un módulo *n* dentro del rango teóricamente no factorizable de manera práctica a través de la vulnerabilidad ROCA (rango comprendido entre 1.216 y 1.984 bits [18]).

Los siguientes ejemplos, de Windows y Linux, muestran la longitud de la clave pública (o módulo *n*) de cada uno de los tres certificados (en orden cronológico):

```
$ openssl x509 -in cert_2048bits_20170223.der -inform DER -text -
noout | grep "Public-Key"
                RSA Public-Key: (2048 bit)
$ openssl x509 -in cert_1920bits_20180102.der -inform DER -text -
noout | grep "Public-Key"
                RSA Public-Key: (1920 bit)
$ openssl x509 -in cert_2048bits_20190201.der -inform DER -text -
noout | grep "Public-Key"
                RSA Public-Key: (2048 bit)
```



Imágenes y comandos OpenSSL mostrando el tamaño en bits del módulo *n* de la clave pública de tres certificados digitales obtenidos de tres DNIe, algunos de ellos vulnerables a ROCA.

Las tres siguientes imágenes muestran ejemplos con el resultado de ejecutar la herramienta de verificación de la vulnerabilidad ROCA sobre cada uno de los tres certificados digitales asociados a los tres DNIe mostrados previamente:

[23] https://twitter.com/tommy_hs/status/943943806759653376

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

```
root@kali: ~/ROCA
File  Edit  View  Search  Terminal  Help
root@kali:~/ROCA# roca-detect cert_2048bits_20170223.der
2019-09-06 21:03:52 [588] WARNING Fingerprint found in the Certificate cert_2048bits_20170223.der idx None
{"type": "der-cert", "fname": "cert_2048bits_20170223.der", "idx": null, "fprint": "4b5986aaedc22f7ce180126c
3                                    4906", "subject": "C: ES, SN:
                                                    "issuer": "C: ES, O: DIRECCION GENERAL DE LA POLICIA, OU: DNI
E, CN: AC DNIE 001", "issuer_org": "DIRECCION GENERAL DE LA POLICIA", "created_at": "2017-02-23", "created_a
t_utc": 1487855392.0, "not_valid_after_utc": 1614380399.0, "pem": null, "aux": null, "e": "0x10001", "n": "0
x98ac8958c2291c606df2b78cbfb23a8a164869e1840bff033b90a9c239fc42b165e1be9f4d953d6c1e19b4f2535cebaf3863d7cc955

325f2c694959118592bb00e0a5503e60032c9bba636757342efdfb1fbe39458403ea334668bf8de47", "marked": true, "time_ye
ars": 85.2510075035263 2, "price_aws_c4": 37365.516588795595}
2019-09-06 21:03:52 [588] INFO ### SUMMARY ####################
2019-09-06 21:03:52 [588] INFO Records tested: 1
2019-09-06 21:03:52 [588] INFO .. PEM certs: . . . 0
2019-09-06 21:03:52 [588] INFO .. DER certs: . . . 1
2019-09-06 21:03:52 [588] INFO .. RSA key files: . 0
2019-09-06 21:03:52 [588] INFO .. PGP master keys: 0
2019-09-06 21:03:52 [588] INFO .. PGP total keys:  0
2019-09-06 21:03:52 [588] INFO .. SSH keys:  . . . 0
2019-09-06 21:03:52 [588] INFO .. APK keys:  . . . 0
2019-09-06 21:03:52 [588] INFO .. JSON keys: . . . 0
2019-09-06 21:03:52 [588] INFO .. LDIFF certs: . . 0
2019-09-06 21:03:52 [588] INFO .. JKS certs: . . . 0
2019-09-06 21:03:52 [588] INFO .. PKCS7: . . . . . 0
2019-09-06 21:03:52 [588] INFO Fingerprinted keys found: 1
2019-09-06 21:03:52 [588] INFO WARNING: Potential vulnerability
2019-09-06 21:03:52 [588] INFO ################################
root@kali:~/ROCA#
```

```
root@kali: ~/ROCA
File  Edit  View  Search  Terminal  Help
root@kali:~/ROCA#
root@kali:~/ROCA# roca-detect cert_1920bits_20180102.der
2019-09-06 21:07:13 [805] WARNING Fingerprint found in the Certificate cert_1920bits_20180102.der idx None
{"type": "der-cert", "fname": "cert_1920bits_20180102.der", "idx": null, "fprint": "0fadc7565820222c87aca3b5
58                                    97c6", "subject": "C: ES, SN:
                                                    "issuer": "C: ES, O: DIRECCION GENERAL DE LA POLICIA
, OU: DNIE, CN: AC DNIE 001", "issuer_org": "DIRECCION GENERAL DE LA POLICIA", "created_at": "2018-01-02", "
created_at_utc": 1514882996.0, "not_valid_after_utc": 1577954994.0, "pem": null, "aux": null, "e": "0x10001"
, "n": "0x97f7d47364c38446dbe32b8155008077ce808d8e0844d6aaa7fa05dff687dfbc6eb99bd8d0aff17df0e7cbee5b0573056e

7f844be9df308d5037102fe3b94c814f5e2f6d013719a3263528242397", "marked": true, "time_years": -1, "price_aws_c4
": -1}
2019-09-06 21:07:13 [805] INFO ### SUMMARY ################################
2019-09-06 21:07:13 [805] INFO Records tested: 1
2019-09-06 21:07:13 [805] INFO .. PEM certs: . . . 0
2019-09-06 21:07:13 [805] INFO .. DER certs: . . . 1
2019-09-06 21:07:13 [805] INFO .. RSA key files: . 0
2019-09-06 21:07:13 [805] INFO .. PGP master keys: 0
2019-09-06 21:07:13 [805] INFO .. PGP total keys:  0
2019-09-06 21:07:13 [805] INFO .. SSH keys:  . . . 0
2019-09-06 21:07:13 [805] INFO .. APK keys:  . . . 0
2019-09-06 21:07:13 [805] INFO .. JSON keys: . . . 0
2019-09-06 21:07:13 [805] INFO .. LDIFF certs: . . 0
2019-09-06 21:07:13 [805] INFO .. JKS certs: . . . 0
2019-09-06 21:07:13 [805] INFO .. PKCS7: . . . . . 0
2019-09-06 21:07:13 [805] INFO Fingerprinted keys found: 1
2019-09-06 21:07:13 [805] INFO WARNING: Potential vulnerability
2019-09-06 21:07:13 [805] INFO ################################
root@kali:~/ROCA#
```

```
root@kali: ~/ROCA
File  Edit  View  Search  Terminal  Help
root@kali:~/ROCA#
root@kali:~/ROCA# roca-detect cert_2048bits_20190201.der
2019-09-06 21:07:44 [814] INFO ### SUMMARY ####################
2019-09-06 21:07:44 [814] INFO Records tested: 1
2019-09-06 21:07:44 [814] INFO .. PEM certs: . . . 0
2019-09-06 21:07:44 [814] INFO .. DER certs: . . . 1
2019-09-06 21:07:44 [814] INFO .. RSA key files: . 0
2019-09-06 21:07:44 [814] INFO .. PGP master keys: 0
2019-09-06 21:07:44 [814] INFO .. PGP total keys:  0
2019-09-06 21:07:44 [814] INFO .. SSH keys:  . . . 0
2019-09-06 21:07:44 [814] INFO .. APK keys:  . . . 0
2019-09-06 21:07:44 [814] INFO .. JSON keys: . . . 0
2019-09-06 21:07:44 [814] INFO .. LDIFF certs: . . 0
2019-09-06 21:07:44 [814] INFO .. JKS certs: . . . 0
2019-09-06 21:07:44 [814] INFO .. PKCS7: . . . . . 0
2019-09-06 21:07:44 [814] INFO No fingerprinted keys found (OK)
2019-09-06 21:07:44 [814] INFO ################################
root@kali:~/ROCA#
```

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

CriptoCert

Aunque el segundo certificado digital hace uso de una clave RSA que potencialmente no es factorizable de forma práctica mediante ROCA (1.920 bits), ha sido generada por un DNIe que dispone de la librería RSA vulnerable, por lo que el proceso de verificación empleado por la herramienta detecta que los números primos empleados para generar el módulo *n* siguen la estructura vulnerable identificada en ROCA (mediante un patrón o *fingerprint*).

De cara a finalizar el artículo, y con el objetivo de comprobar los contenidos previos, se debe tener en cuenta que todos los detalles descritos previamente a lo largo del presente artículo sobre los números *p* y *q* y las claves RSA pueden ser analizados y corroborados de forma práctica mediante la generación de un par de claves RSA, pública y privada, empleando **OpenSSL** [15] (los ejemplos hacen uso de macOS, pero los resultados serán muy similares en Windows o Linux). El siguiente comando permite la generación del par de claves RSA, pública y privada, de 2.048 bits mediante OpenSSL (sin proteger o cifrar el fichero resultante que contiene la clave privada):

```
$ openssl genrsa -out privada.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................
.................................................+++
............+++
e is 65537 (0x10001)
$ ls -l
-rw-r--r--  1 siles  staff  1679 Sep  1 00:00 privada.pem
$ file privada.pem
privada.pem: PEM RSA private key
```

NOTA: Si se procede a la generación de claves RSA de mayor tamaño, por ejemplo de 8.192 bits, se apreciará cómo el tiempo medio necesario para encontrar los números primos *p* y *q* aumenta significativamente (por ejemplo, de 0,14s a 14s), y cómo el número de iteraciones de los test de primalidad disminuye (ver FIPS 186-4).

Los signos, o símbolos, asociados al carácter "." y "+" que se muestran durante la generación de la clave indican el progreso en la selección de los números primos *p* y *q*. Tal como indica la página del manual (man) del comando genrsa de OpenSSL[24], el "." indica cada número candidato que ha pasado una criba inicial como posible número primo, y el "+" indica que el número candidato ha pasado una iteración del test de primalidad de Miller-Rabin. Un carácter de nueva línea indica que el número ha pasado todos los tests de primalidad. En base a estos detalles, parece que en la versión de OpenSSL empleada en los ejemplos con macOS Mojave 10.14.x (LibreSSL[25] 2.6.5[26]), únicamente se llevan a cabo 3 iteraciones del test de primalidad de Miller-Rabin para cada número primo *p* y *q* cuando estos son de 1.024 bits (a diferencia de las 5 iteraciones esperadas según el código fuente de OpenSSL, tal como se detalla posteriormente).

[24] https://www.openssl.org/docs/man1.1.1/man1/genrsa.html
[25] https://www.libressl.org/
[26] Obtenida mediante el comando "openssl version".

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

Si en su lugar se hace uso de la versión de OpenSSL disponible en Linux, por ejemplo 1.1.0h, es posible corroborar que se llevan a cabo las 5 iteraciones del test de primalidad de Miller-Rabin esperadas (indicadas en la salida del siguiente comando por los 5 signos "+"):

```
$ openssl genrsa -out privada.pem 2048
Generating RSA private key, 2048 bit long modulus
...+++++
...+++++
e is 65537 (0x010001)
```

Si se inspecciona el fichero "privada.pem" generado, es posible identificar que contiene una clave RSA privada (en formato base64) mediante el texto "-----BEGIN RSA PRIVATE KEY-----":

```
$ cat privada.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAtxNesQq1i5i/4BBWWe5JBc02+E6GaRLeVbSaYS3+OKWzPCKb
...
zTIN1f91C1/t2/rYQGWEumCqKnq/UIQTLVeOOg0+wcolnHLMoeVGKg==
-----END RSA PRIVATE KEY-----
```

El siguiente comando de OpenSSL permite obtener una versión en formato texto de la clave privada RSA generada, con todos los parámetros asociados:

```
$ openssl rsa -in privada.pem -text -out privada.txt
writing RSA key
$ ls -l
-rw-r--r--  1 siles  staff  1679 Sep  1 00:00 privada.pem
-rw-r--r--  1 siles  staff  5681 Sep  1 00:01 privada.txt
```

```
$ cat privada.txt
Private-Key: (2048 bit)
modulus:
    00:b7:13:5e:b1:0a:b5:8b:98:bf:e0:10:56:59:ee:
    <… +15 líneas …>
    39:00:4b:4a:65:3f:90:13:f3:11:2a:ce:48:61:f8:
    77:6b
publicExponent: 65537 (0x10001)
privateExponent:
    00:99:86:bd:de:fc:1b:18:b1:05:1f:72:b3:e7:80:
    <… +15 líneas …>
    5e:cf:6b:7e:11:7c:9d:ca:56:e2:6b:16:17:35:13:
    ca:31
prime1:
    00:df:bb:4c:7b:56:b1:71:9b:c8:10:9f:a8:65:8b:
    <… +6 líneas …>
    ab:15:56:ab:b5:af:8f:65:ef:63:bc:6a:a9:7b:82:
    26:ac:71:79:a1:71:aa:63:a9
```

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

```
prime2:
    00:d1:7a:f4:da:fb:74:c1:1b:19:df:7d:cc:07:d1:
    <… +6 líneas …>
    c1:9d:ee:57:2f:22:9c:67:8f:59:82:bf:9b:9f:41:
    ee:fd:5b:a6:a1:c0:e7:ae:f3
exponent1:
    00:86:97:05:26:79:7b:9b:8d:8c:68:3b:b3:b1:0a:
    <… +6 líneas …>
    5d:9c:23:9c:7e:5a:d3:98:0d:cf:e0:ec:05:72:f0:
    53:d5:8f:1a:0d:7d:f4:73:a9
exponent2:
    0a:ec:d8:bc:5b:04:f9:d5:4a:02:27:f3:6e:2c:f0:
    <… +6 líneas …>
    92:b0:0d:87:fd:cc:1e:72:91:7e:8a:33:b9:98:9c:
    b8:46:01:68:ca:40:cb:15
coefficient:
    23:d5:3c:97:e5:8b:34:0f:4c:51:6e:b9:12:72:3c:
    <… +6 líneas …>
    aa:2a:7a:bf:50:84:13:2d:57:8e:3a:0d:3e:c1:ca:
    25:9c:72:cc:a1:e5:46:2a
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAtxNesQq1i5i/4BBWWe5JBc02+E6GaRLeVbSaYS3+OKWzPCKb
...
zTIN1f91C1/t2/rYQGWEumCqKnq/UIQTLVeOOg0+wcolnHLMoeVGKg==
-----END RSA PRIVATE KEY-----
```

NOTA: Los valores se muestran empleando 15 pares de caracteres hexadecimales (o 15 bytes) por cada fila.

Se puede ver como las claves RSA generadas por OpenSSL hacen uso del número 4 de Fermat para la clave pública *e* = 65.537 (0x10001), tanto en el proceso de generación de las claves, como al inspeccionar sus detalles.

Los detalles en formato texto de la clave privada permiten obtener el tamaño de la clave privada, 2.048 bits, junto al módulo *n* (*modulus*, compuesto por 257 bytes o 2.056 bits), los valores del exponente público *e* (65.537) y del exponente privado *d*, pudiendo apreciarse la diferencia en tamaño entre ambos (17 bits y 2.056 bits, o 257 bytes, respectivamente), los números primos *p* (*prime1*, compuesto por 129 bytes o 1.032 bits) y *q* (*prime2*, también compuesto por 129 bytes o 1.032 bits), y la clave privada RSA en formato base64 al final (al no hacer uso de la opción "-noout").

Adicionalmente, OpenSSL proporciona otros parámetros como *exponent1* (dp, compuesto por 129 bytes o 1.032 bits), *exponent2* (dq, compuesto por 128 bytes o 1.024 bits) y *coefficient* (qInv, también compuesto por 128 bytes o 1.024 bits) que son utilizados, junto a los números primos *p* y *q*, en la fórmula de Garner [0] como parte de las optimizaciones empleadas al aplicar el TRC en el proceso de descifrado de RSA.

Curiosamente, el tamaño en bits (o bytes) de los diferentes elementos involucrados en la clave privada RSA no coinciden con los esperados. Ello se debe a que OpenSSL, con el formato de fichero DER (Distinguished Encoding Rules, que es realmente el formato binario que es codificado en base64 en los ficheros ".pem", Privacy Enhanced Mail, PEM), añade un byte inicial con el valor 0x00 que debe ser eliminado. Como resultado, tanto el módulo *n* como el exponente privado *d* pasarían a estar formados por un byte

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

menos, es decir, 2.048 bits (o 256 bytes), los números primos *p* y *q* pasarían a estar compuestos por 1.024 bits (o 128 bytes), al igual que el *exponent1* (dp). De esta forma, el tamaño de todos los valores mostrados coincide con el esperado.

El motivo de este byte adicional se debe a que el formato de representación binaria ASN.1 que es utilizado por el formato de fichero DER codifica los números enteros (ASN.1 INTEGER) con signo, en formato *big endian*. Como todos los números empleados en RSA son enteros positivos, y dado que son números cuya longitud es múltiplo de 8 bits (o un byte), como 1.024 o 2.048 bits, OpenSSL los antecede por la izquierda con el valor 0x00. En caso contrario, la codificación ASN.1 de estos valores podría representar números enteros negativos (en codificación de complemento a 2, donde el bit más significativo representa el bit de signo, indicando un bit con valor 1 un número negativo). Se debe hacer notar que en el ejemplo previo no se ha añadido ese byte 0x00 adicional para los valores de *exponent2* y de *coefficient* (que presentaban el tamaño esperado de 1.024 bits), y se deja como ejercicio para el lector descubrir cuál es el motivo. Alternativamente, el siguiente comando de OpenSSL (asn1parse) permite obtener los valores numéricos sin el valor 0x00:

```
$ openssl asn1parse -in privada.pem
```

Por otro lado, es posible exportar únicamente la clave pública (opción "-pubout") a un fichero, mediante el siguiente comando de OpenSSL[27]:

```
$ openssl rsa -in privada.pem -outform PEM -pubout -out publica.pem
writing RSA key
$ ls -l
-rw-r--r--  1 siles  staff  1679 Sep  1 00:00 privada.pem
-rw-r--r--  1 siles  staff  5681 Sep  1 00:01 privada.txt
-rw-r--r--  1 siles  staff   451 Sep  1 00:02 publica.pem
$ file publica.pem
publica.pem: ASCII text
```

NOTA: Como se puede ver, el tamaño del fichero con la clave pública es significativamente más pequeño que el tamaño del fichero con la clave privada.

Si se inspecciona el fichero "publica.pem", es posible identificar que contiene una clave RSA pública (en formato base64) mediante el texto "-----BEGIN PUBLIC KEY-----":

```
$ cat publica.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtxNesQq1i5i/4BBWWe5J
...
awIDAQAB
-----END PUBLIC KEY-----
```

[27] Posteriormente se podría convertir la clave pública de formato PEM a formato DER: "openssl rsa -pubin -inform PEM -in publica.pem -outform DER -out publica.der".

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

Al igual que anteriormente se ha hecho con la clave privada, es posible mediante el siguiente comando de OpenSSL obtener una versión en formato texto de la clave pública RSA generada (opción "-pubin"), con todos los parámetros asociados:

```
$ openssl rsa -in publica.pem -text -pubin -out publica.txt
writing RSA key
$ ls -l
-rw-r--r--  1 siles  staff  1679 Sep  1 00:00 privada.pem
-rw-r--r--  1 siles  staff  5681 Sep  1 00:01 privada.txt
-rw-r--r--  1 siles  staff   451 Sep  1 00:02 publica.pem
-rw-r--r--  1 siles  staff  1369 Sep  1 00:03 publica.txt
```

```
$ cat publica.txt
Public-Key: (2048 bit)
Modulus:
    00:b7:13:5e:b1:0a:b5:8b:98:bf:e0:10:56:59:ee:
    <… +15 líneas …>
    39:00:4b:4a:65:3f:90:13:f3:11:2a:ce:48:61:f8:
    77:6b
Exponent: 65537 (0x10001)
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtxNesQq1i5i/4BBWWe5J
...
awIDAQAB
-----END PUBLIC KEY-----
```

Los detalles en formato texto de la clave pública permiten obtener el tamaño de la clave pública, 2.048 bits, junto al módulo *n*, con exactamente el mismo valor que el de la clave privada (*modulus*, compuesto por 256 bytes o 2.048 bits, eliminando el valor 0x00 inicial), junto al valor del exponente público *e* (65.537), y de la clave pública RSA en formato base64 al final (al no hacer uso de la opción "-noout").

Complementariamente se puede hacer uso del software educativo genRSA [16] (actualmente en su versión 2.1), ampliamente referenciado en la certificación CriptoCert Certified Cripto Analyst, para llevar a cabo el estudio y los cálculos relativos a los números primos *p* y *q* y a las claves RSA asociadas, tanto de manera manual como automática, para la obtención de las Claves Privadas Parejas (CPP), así como para la realización de diferentes ataques sobre las claves RSA. genRSA incluye los tests de primalidad de Miller-Rabin y de Fermat. Complementariamente, se puede seguir el cuaderno 4 del proyecto CLCript (junto a otros cuadernos también focalizados en RSA), centrado en la generación de claves RSA con genRSA v2.1 [17].

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

Referencias:

[0] https://0xword.com/es/libros/36-libro-cifrado-comunicaciones-rsa.html
[1] https://is.muni.cz/th/oba30/thesis-nemec.pdf
[2] http://cacr.uwaterloo.ca/hac/about/chap4.pdf
[3] https://crypto.stackexchange.com/questions/1970/how-are-primes-generated-for-rsa
[4] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
[5] https://www.sans.org/reading-room/whitepapers/vpns/prime-numbers-public-key-cryptography-969
[6] https://en.wikipedia.org/wiki/Prime_number_theorem
[7] https://security.stackexchange.com/questions/176394/how-does-openssl-generate-a-big-prime-number-so-fast
[8] https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c
[9] https://es.wikipedia.org/wiki/Test_de_primalidad
[10] https://en.wikipedia.org/wiki/Safe_prime
[11] https://en.wikipedia.org/wiki/Strong_prime
[12] https://eprint.iacr.org/2001/007
[13] https://crypto.stackexchange.com/questions/5262/rsa-and-prime-difference
[14] http://www.criptored.upm.es/descarga/Win64OpenSSL1.1.0g_No_Genera_Claves_Correctas_Hasta_128_Bits.pdf
[15] https://www.openssl.org
[16] http://www.criptored.upm.es/software/sw_m001d.htm
[17] http://www.criptored.upm.es/software/sw_m001s.htm
[18] https://crocs.fi.muni.cz/public/papers/rsa_ccs17, https://crocs.fi.muni.cz/_media/public/papers/nemec_roca_ccs17_preprint.pdf
[19] https://eprint.iacr.org/2010/006.pdf, https://hal.inria.fr/inria-00444693v2/document
[20] https://en.wikipedia.org/wiki/RSA_Factoring_Challenge

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

Breve análisis del código fuente de OpenSSL: (https://github.com/openssl/openssl/)

Si se analiza el código fuente de OpenSSL un poco más en detalle [8] (la versión actual disponible en GitHub en el momento de elaboración del presente artículo era la 3.0.0-dev, mientras que la versión estable principal era la 1.1.1c), complementando la descripción más genérica de cómo genera OpenSSL los números primos *p* y *q* proporcionada previamente a lo largo del artículo, se confirma concretamente que la función BN_generate_prime()[28] (*deprecated*: reemplazada por la siguiente función) invoca a la función BN_generate_prime_ex()[29], que hace uso de BN_generate_prime_ex2()[29] - la función asociada al estándar FIPS - para finalmente invocar a la función probable_prime()[29]. Ésta hace uso internamente de la función BN_priv_rand_ex()[30] para la generación del número candidato aleatorio mediante bnrand()[30] y, tras generar el número aleatorio con la función rand_bytes_ex()[31], procede a establecer el valor del bit más significativo (en realidad de los dos bits más significativos[32]) y del bit menos significativo de dicho número aleatorio (estos bits son fijados a 1 por las constantes BN_RAND_TOP_TWO y BN_RAND_BOTTOM_ODD respectivamente[33]). Se dispone de más documentación sobre la implementación de OpenSSL en la página man de BN_generate_prime[34].

La función BN_is_prime_fasttest_ex()[29] es la encargada de realizar una primera comprobación rápida de si un número candidato es o no primo verificando si es impar, y dividiéndolo por algunos números primos pequeños.

Dentro de la función BN_generate_prime_ex2()[29] se hace uso de la función BN_prime_checks_for_size(bits)[35] para determinar el número de iteraciones del test de primalidad de Miller-Rabin que deben llevarse a cabo en función de la longitud en bits del número primo candidato, siguiendo las recomendaciones de FIPS 186-4. En el caso de claves RSA de 2.048 bits, con números primos de 1.024 bits, se hace uso de 5 iteraciones.

Se debe tener en cuenta que las referencias a "BN" o "bn" dentro de OpenSSL corresponden a BIGNUM.

Por tanto, desde un punto de vista práctico, todos los números primos generados por OpenSSL tendrán su bit menos significativo igual a 1 (reflejando que se trata de un

[28] https://github.com/openssl/openssl/blob/master/crypto/bn/bn_depr.c
[29] https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c
[30] https://github.com/openssl/openssl/blob/master/crypto/bn/bn_rand.c
[31] https://github.com/openssl/openssl/blob/master/crypto/include/internal/rand_int.h (La utilización de un buen generador de números (pseudo)aleatorios es fundamental desde un punto de vista criptográfico, por lo que esta función es implementada mediante fragmentos de código específicos de OpenSSL según el sistema operativo: https://github.com/openssl/openssl/tree/master/crypto/rand)
[32] Con el objetivo de asegurarse de que el módulo *n* será de la longitud de bits correcta, y no inferior. Esta optimización no tiene un impacto significativo en la seguridad de los primos seleccionados [1].
[33] https://github.com/openssl/openssl/blob/master/include/openssl/bn.h
[34] https://www.openssl.org/docs/man1.1.1/man3/BN_generate_prime.html
[35] https://github.com/openssl/openssl/blob/master/include/openssl/bn.h

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.

número impar), y comenzarán (en hexadecimal) por el carácter 0xC, D, E o F, ya que sus dos primeros bits serán siempre iguales a 1:

```
$ for i in $(seq 100)
> do
> openssl prime -generate -bits 1024 -hex | head -c 1
> done
CDEEEEEEECDCEDDCEFCCCECFDDDEFCDDECDCEFDEEECCCFCDFDEDCDDEDEDFFDECEECD
CDDDDDEECFFCDCDFECCDCCCDFEFCDDCF$
```

NOTA: El script superior muestra sólo el primer carácter hexadecimal correspondiente a la generación de 100 números primos, es decir, los 100 caracteres hexadecimales.

Para analizar el coste computacional, medido en tiempo, que requiere OpenSSL para generar un número primo de 1.024 bits es posible hacer uso del siguiente comando que, de media, requiere de un tiempo de cómputo real de entre 0,02 y 0,05 segundos en un ordenador portátil estándar:

```
$ time openssl prime -generate -bits 1024 -hex
D8BC54E49221550A9CECE3AB46E4DC550626394771F3E656D381D49D30E453A81533
C37346F9CDEFBCC601BFF96ADF0FE448786280CE2710728915ED4EA1227EBECC3C2A
DDA08E39E9F45ABFAA22DD477E7E8DE4A2BEF876A8AD4507D0AB15DB4B449590BE9B
A5BFEF7D3C53D14129C5524AEAE76D8C4A79C584E51169DA56E1

real   0m0.041s
user   0m0.038s
sys    0m0.003s
```

NOTA: El tiempo requerido depende de la "suerte" a la hora de generar un número aleatorio que sea primo, reduciéndose el tiempo empleado en ocasiones (en pruebas reales realizadas) hasta 0,008 segundos.

Con respecto al número de bits empleados para la generación de los números primos aleatorios en todas las funciones descritas previamente (indicado habitualmente por el segundo argumento de las diferentes funciones, denominado "bits"), OpenSSL hace uso de exactamente el número de bits proporcionado mediante los parámetros de línea de comandos, tanto a la hora de generar un número primo mediante el comando "openssl prime -generate -bits 1024 -hex"[36] (que hace uso directamente de la función BN_generate_prime_ex()), como a la hora de generar una clave RSA mediante el comando "openssl genrsa -out privada.pem 2048"[37] (que hace uso de la función RSA_generate_multi_prime_key()[38]).

Finalmente, OpenSSL lleva a cabo la generación de las claves RSA mediante la función rsa_builtin_keygen()[38], que internamente hace uso de la función BN_generate_prime_ex()[29] ya mencionada anteriormente.

[36] https://github.com/openssl/openssl/blob/master/apps/prime.c
[37] https://github.com/openssl/openssl/blob/master/apps/genrsa.c
[38] https://github.com/openssl/openssl/blob/master/crypto/rsa/rsa_gen.c

Copyright © 2019 CriptoCert S.L. (www.criptocert.com)
Todos los derechos reservados.