Práctica

Ataque de factorización a RSA

Daniel Lerch Hostalot

Grado de dificultad

RSA es, sin lugar a dudas, el criptosistema de clave pública más utilizado en la actualidad, habiendo resistido al escrutinio de los criptoanalistas durante más de un cuarto de siglo. Este popular algoritmo basa su seguridad en la dificultad que supone factorizar números grandes, considerando como grandes los números de más de 100 dígitos decimales.

A diferencia de la criptografía de clave privada, donde se utiliza una única clave para cifrar y descifrar mensajes, en la criptografía de clave pública se utilizan dos claves. Estas dos claves son conocidas como clave pública y clave privada. Para realizar una comunicación cifrada, un usuario debe disponer de un par de claves. Mientras que la clave privada deberá permanecer secreta, la clave pública, como su nombre indica, estará a disposición de cualquiera que desee enviar mensajes cifrados al mencionado usuario. Pues un mensaje cifrado con una clave pública, únicamente podrá ser descifrado con su correspondiente clave privada. Para conseguir esta curiosa cualidad, es necesario recurrir a ciertos problemas matemáticos que lo permiten. En el caso de RSA, se recurre al problema de la factorización de números grandes.

El nacimiento de la criptografía de clave pública se produjo en 1976* con la publicación por parte de Diffie y Hellman de un protocolo que permitía intercambiar cierta información sobre un canal inseguro.

Poco después, en 1977, Rivest, Shamir y Adleman proponían el criptosistema RSA, el criptosistema de clave pública más usado en la actualidad.

* En 1997 se hicieron públicos ciertos documentos en los que se demostraba que en 1973, los criptógrafos del Grupo de Seguridad de Comunicaciones Electrónicas (CESG) del gobierno británico, ya disponían de conocimientos sobre este tipo de criptografía.

El criptosistema RSA

Como ya se ha comentado, la seguridad de RSA reside en la dificultad computacional

En este artículo aprenderás...

- cómo funciona RSA y cómo se realizan ataques de factorización,
- cómo se usan las claves RSA y el procedimiento de ataque para obtener la clave privada a partir de la pública, consiguiendo así, descifrar el mensaje.

Lo que deberías saber...

- cierta base matemática,
- conocimientos de programación en C,
- los ejemplos han sido desarrollados y probados en un sistema GNU/Linux.

Conceptos Matemáticos

Divisor o Factor:

Un número entero a es divisor (o factor) de b cuando existe otro entero c que cumple $b = a \cdot c$. Ejemplo: $21 = 7 \cdot 3$.

Números primos y Números compuestos:

Un número entero es primo si solo es divisible por uno y por sí mismo. Un número entero es compuesto si no es primo.

Ejemplo: $21 = 7 \cdot 3$ es un número compuesto, 7 y 3 son números primos

Factorización:

Se conoce como factorización de un número entero n el proceso de descomponerlo en sus factores primos: $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_i^{e_i}$, donde p_i son números primos i e_i son enteros positivos.

Ejemplo: 84 queda factorizado como $21 = 2^2 \cdot 3 \cdot 7$

Módulo:

Se conoce como módulo y se representa como $a \bmod b$ al resto de la división entera de a entre. b.

Ejemplo: $5 \bmod 3 = 2$, $10 \bmod 7 = 3$, $983 \bmod 3 = 2$, $1400 \bmod 2 = 0$.

a y b son congruentes módulo n: $b \equiv a \pmod n$ si su diferencia (a-b) es un múltiplo de n.

Máximo Común Divisor:

Llamamos Máximo Común Divisor de dos números enteros a y b, representado como $\mathrm{mcd}(a,b)$, al mayor número entero divisor de a y de b.

Ejemplo: $\mathrm{mcd}(42,35) = \mathrm{mcd}(2 \cdot 3 \cdot 7, 5 \cdot 7) = 7$

Algoritmo de Euclides:

El algoritmo de Euclides calcula el Máximo Común Divisor de dos números basándose en que $\mathrm{mcd}(a,b) = \mathrm{mcd}(b,r)$, donde $a > b > 0$ son enteros y r resto de la división entre a y b

Ejemplo: $\mathrm{mcd}(1470,42)$

(1) $1470 \bmod 42 = 35 \rightarrow \mathrm{mcd}(1470,42) = \mathrm{mcd}(42,35)$

(2) $42 \bmod 35 = 7 \rightarrow \mathrm{mcd}(42,35) = \mathrm{mcd}(35,7)$

(3) $35 \bmod 7 = 0 \rightarrow \mathrm{mcd}(7,0) = 7$

Indicador de Euler (Totient):

Dado $n \geq 0$ conocemos como $\Phi(n)$ el número de enteros en el intervalo $[1,n]$ que son primos* con n. Para $n = p \cdot q$, $\Phi(n) = (p-1)(q-1)$.

* Dos números enteros a y b son primos entre sí (o coprimos) si $\mathrm{mcd}(a,b) = 1$.

que supone la factorización de números grandes. Factorizar un número consiste en encontrar los números primos (factores) que multiplicados dan como resultado dicho número. De manera que si, por ejemplo, queremos factorizar el número 12, obtendremos como resultado 2·2·3. La forma sencilla de encontrar los factores de un número n consiste en ir dividiendo por todos los números primos inferiores a n. Este procedimiento, aunque sencillo, es extremadamente lento cuando se trata de factorizar números grandes.

Realicemos algunos cálculos sencillos para hacernos una idea. Una clave de 256 bits (como la que romperemos en un ejemplo posterior) tiene unos 78 dígitos decimales aproximadamente (1078). Dado que en las claves RSA este número suele tener únicamente dos factores primos, cada uno de ellos tendrá más o menos 39 dígitos. Esto significa que para factorizar el número tendríamos que dividir por todos los números primos de 39 dígitos o menos (1039). Suponiendo que solo el 0.1% de los números son primos, tendríamos que realizar unas 1036 divisiones. Imaginemos que disponemos de un sistema capaz de realizar 1020 divisiones por segundo. En este caso tardaríamos 1016 segundos en romper la clave, es decir, más de 300 millones de años. Un millón de veces la edad del universo. Por suerte, nosotros tardaremos un poco menos.

Veamos como funciona RSA. Empecemos generando la clave pública y la clave privada (en el cuadro *Conceptos Matemáticos* están disponibles algunos conceptos matemáticos de interés). Para este propósito es necesario seguir los siguientes pasos.

Paso 1:

Elegiremos aleatoriamente dos números primos p y q y los multiplicaremos, obteniendo n: $n = p \cdot q$. Si elegimos, por ejemplo, $p = 3$ y $q = 11$ obtenemos $n = 33$

Paso 2:

Calcularemos el indicador de Euler (Totient) con la siguiente fórmula: $\Phi(n) = \Phi(p \cdot q) = (p-1) \cdot (q-1)$. En nuestro ejemplo obtenemos $\Phi(n) = 20$

Paso 3:

Buscaremos un exponente de cifrado (posteriormente lo utilizaremos para cifrar) al que llamaremos e. El exponente de cifrado debe cumplir $\mathrm{mcd}(e, \Phi(n)) = 1$ por lo que nos serviría, por ejemplo $e = 3$, dado que no dispone de ningún factor común con 20 (factores 2 y 5).

Paso 4:

Calculamos un exponente de descifrado al que llamaremos d (posteriormente lo utilizaremos para descifrar). El exponente de descifrado debe verificar $1 < d < \Phi(n)$ de manera que $e \cdot d \equiv 1 \pmod{\Phi(n)}$. Lo que significa que d corresponderá a un número entre 1 y 20 que al multiplicarlo por 3 y dividirlo por 20 de resto 1. Por lo que d puede ser 7.

Las claves:

La clave pública del usuario corresponde a la pareja (n, e), en nuestro ejemplo (33,3) mientras que su clave

privada es d, es decir 7. Lógicamente los números p, q i $\Phi(n)$ deberán permanecer en secreto.

El (des)cifrado:

Llegados a este punto, solo es necesario cifrar con $C = M^e \bmod n$ y descifrar con $M = C^d \bmod n$. Si consideramos que nuestro mensaje es $M = 5$ el cifrado correspondería a $C = 5^3 \bmod 33 = 26$. Para descifrar solo tendríamos que aplicar $M = 26^7 \bmod 33 = 5$.

Como se comentaba al principio y puede verse en el procedimiento anterior, la seguridad del criptosistema reside en n. Es decir, si un atacante con acceso a la clave pública, consigue factorizar n obteniendo p y q, no tiene más que utilizar las fórmulas anteriores para obtener la clave privada.

The RSA Factoring Challenge

The RSA Factoring Challenge es una competición financiada por RSA Laboratories en la que se ofrecen suculentos premios al que consiga factorizar ciertos números grandes. Ésto les permite conocer el estado del arte de los sistemas de factorización, conociendo así cuál es el tamaño de clave adecuado para mantener RSA seguro.

Listado 1. *Pasar de Hexadecimal a Decimal*

```
#include <stdio.h>
#include <openssl/bn.h>
int main (int argc, char **argv)
{
   BIGNUM *n = BN_new();
   if (argc!=2)
   {
      printf ("%s <hex>\n",
       argv[0]);
      return 0;
   }
   if(!BN_hex2bn(&n, argv[1]))
   {
      printf("error:
      BN_hex2bn()");
      return 0;
   }
   printf("%s\n", BN_bn2dec(n));
   BN_free(n);
}
```

En el momento de escribir este artículo el récord de factorización es el RSA-640, un número de 193 dígitos que fue factorizado el 2 de noviembre del 2005 por F. Bahr et al. El próximo reto es el RSA-704, premiado con 30.000$.

Sin duda, The RSA Factoring Challenge, es una muy buena forma de conocer el estado actual de los sistemas de factorización.

En el cuadro *The RSA Factoring Challenge* puedes ver los desafíos propuestos actualmente.

Ataque de factorización

En los siguientes apartados se realizará un ataque de ejemplo a una clave RSA. Para agilizar los cálculos se utilizará una clave bastante más pequeña de lo normal, facilitando así su factorización. Aunque no se trata de un ejemplo real, nos servirá para ver cómo se realiza un ataque completo.

Primero prepararemos un entorno de trabajo con la herramienta OpenSSL, generando las claves necesarias y cifrando un mensaje que utilizaremos como objetivo de ataque. A continuación factorizaremos el módulo n y obtendremos la clave privada, descifrando finalmente el mensaje.

OpenSSL y RSA

OpenSSL es una herramienta criptográfica muy útil de fuente abierta. En la sección de referencias encontrarás dónde descargarla, aunque la mayoría de dsitribuciones GNU/Linux la llevan instalada por defecto.

En esta sección la utilizaremos para configurar un entorno de pruebas sobre el cual realizaremos el ataque.

El primer paso consiste en generar un par de claves para cifrar y descifrar. Generaremos claves de 256 bits, demasiado pequeñas para mantener sus comunicaciones seguras, pero suficientes para nuestro ejercicio.

Generamos el par de claves, manteniendo en secreto nuestra clave privada.

```
# Generar un par de claves RSA
de 256 bits
openssl genrsa -out rsa_privkey
.pem 256
cat rsa_privkey.pem
-----BEGIN RSA PRIVATE KEY-----
MIGqAgEAAiEA26dbqzGRt31
   qincXxy4jjZMMOId/DVT8aT
   cq8aamDiMCAwEAAQIh
AMvT1oXa/rxF3mrVLrR/RS7vK
   1WTsQ5CWl/+37wztZOpAhE
   A+4jgEkfalFH+0S+1
IPKD5wIRAN+NmMH4AF0B8jz
   MAXHHXGUCEGRpRZnGmVk
   wSlrTgqj+Zu0CEA7v7CQR
yRxt09zCGNqcYo0CEDEW7mvoz
   MYYLC5o+zgfV4U=
-----END RSA PRIVATE KEY-----
```

A continuación guardamos la clave pública en un fichero. Ésta es la que publicaríamos para que cualquiera pudiese enviarnos mensajes cifrados.

```
# Guarda la clave publica en
   un fichero
openssl rsa -in rsa_privkey.pem
   -pubout -out rsa_pubkey.pem
cat rsa_pubkey.pem
-----BEGIN PUBLIC KEY-----
MDwwDQYJKoZIhvcNAQEBBQA
   DKwAwKAIhANunW6sxkbd9a
   op3F8cuI42TDDiHfw1U
/Gk3KvGmpg4jAgMBAAE=
-----END PUBLIC KEY-----
```

Generado el par de claves, ya podemos cifrar y descifrar. Trabajaremos con el siguiente mensaje en claro:

```
echo "Cuarenta y dos" > plain.txt
```

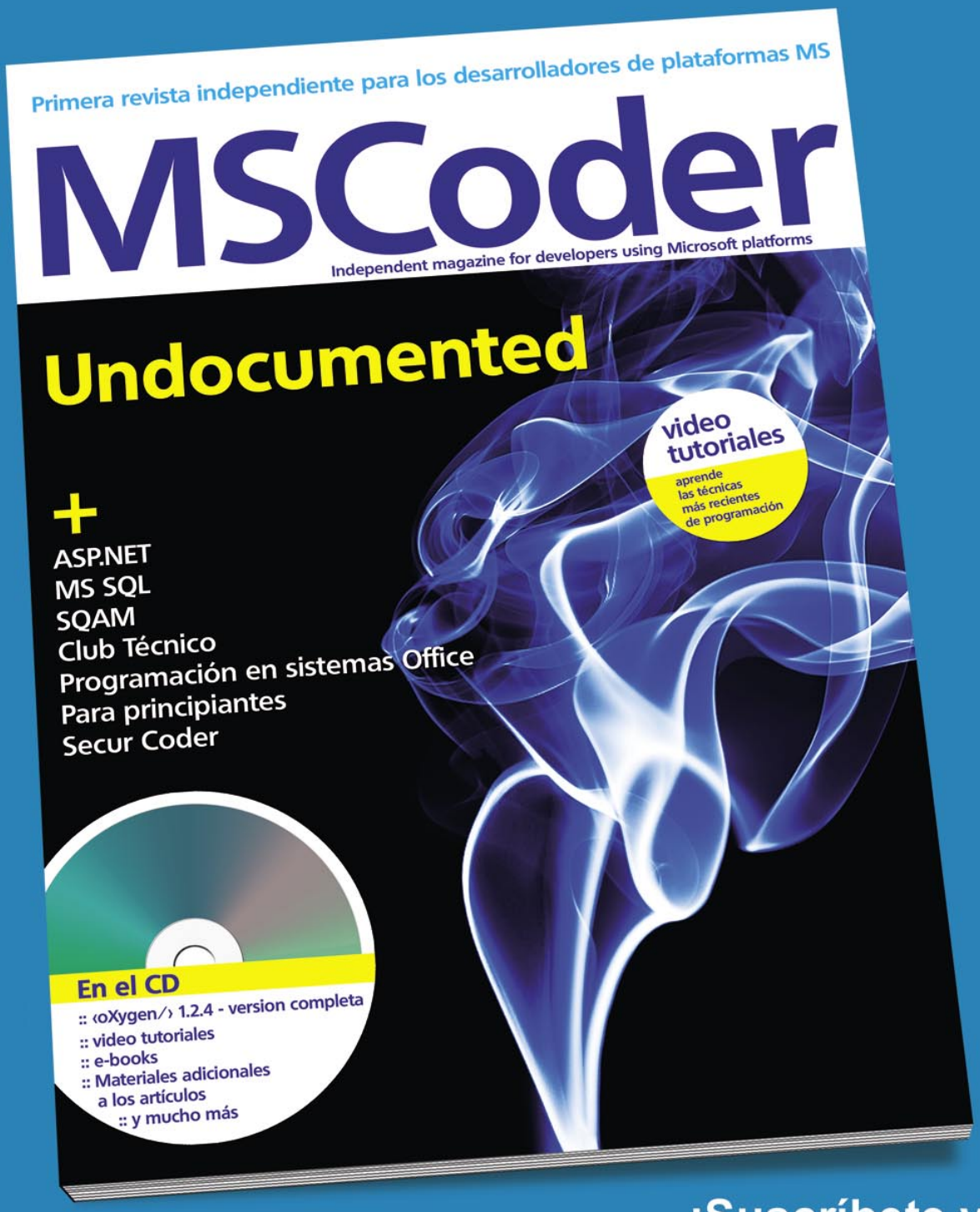
Este mensaje puede ser cifrado de forma sencilla con el siguiente comando y la clave pública:

```
openssl rsautl -encrypt -pubin
   -inkey rsa_pubkey.pem \
   -in plain.txt -out cipher.txt
```

Para descifrar utilizaremos la clave privada:

```
openssl rsautl -decrypt -inkey
   rsa_privkey.pem -in cipher.txt
Cuarenta y dos
```

Desde octubre en tu tienda



¡Suscríbete ya!
¡No te lo puedes perder!

En la página 33 encontrarás el formulario de suscripción

buyitpress.com

www.mscoder.org/es

Listado 2. *Clave privada*

```
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/rsa.h>
#include <openssl/engine.h>
#include <openssl/pem.h>
int main (int argc, char **argv)
{
   RSA *keypair = RSA_new();
   BN_CTX *ctx = BN_CTX_new();
   BN_CTX_start(ctx);
   BIGNUM *n = BN_new();
   BIGNUM *d = BN_new();
   BIGNUM *e = BN_new();
   BIGNUM *p = BN_new();
   BIGNUM *q = BN_new();
   BIGNUM *dmp1 = BN_new();
   BIGNUM *dmq1 = BN_new();
   BIGNUM *iqmp = BN_new();
   BIGNUM *r0 = BN_CTX_get(ctx);
   BIGNUM *r1 = BN_CTX_get(ctx);
   BIGNUM *r2 = BN_CTX_get(ctx);
   BIGNUM *r3 = BN_CTX_get(ctx);
   if (argc!=4)
   {
      printf ("%s [p] [q] [exp]\n", argv[0]);
      return 0;
   }
   BN_dec2bn(&p, argv[1]);
   BN_dec2bn(&q, argv[2]);
   BN_dec2bn(&e, argv[3]);
   if(BN_cmp(p, q)<0) {
      BIGNUM *tmp = p;
      p = q;
      q = tmp;
   }
   // Calculamos n
   BN_mul(n, p, q, ctx);
   // Calculamos d
   BN_sub(r1, p, BN_value_one()); // p-1
   BN_sub(r2, q, BN_value_one()); // q-1/
   BN_mul(r0, r1, r2, ctx);       // (p-1)(q-1)
   BN_mod_inverse(d, e, r0, ctx); // d
   // Calculamos d mod (p-1)
   BN_mod(dmp1, d, r1, ctx);
   // Calculamos d mod (q-1)
   BN_mod(dmq1, d, r2, ctx);
   // Calculamos el inverso de q mod p
   BN_mod_inverse(iqmp, q, p, ctx);
   // Claves RSA
   keypair->n = n;
   keypair->d = d;
   keypair->e = e;
   keypair->p = p;
   keypair->q = q;
   keypair->dmq1 = dmq1;
   keypair->dmp1 = dmp1;
   keypair->iqmp = iqmp;
   PEM_write_RSAPrivateKey(stdout, keypair,
      NULL, NULL, 0, NULL, NULL);
   BN_CTX_end(ctx);
   BN_CTX_free(ctx);
   RSA_free(keypair);
   return 0;
}
```

Visto como utilizar OpenSSL con RSA y conociendo la necesidad de disponer de la clave privada para descifrar los mensajes, nuestro objetivo a continuación será obtener dicha clave privada sin tener acceso a la original. Es decir, obtener la clave privada a partir de la clave pública. Para poder hacer esto lo primero que necesitamos es obtener el módulo n y el exponente de cifrado. Esto nos lo proporciona el siguiente comando y la clave pública:

```
openssl rsa -in rsa_pubkey.pem
      -pubin -text -modulus
Modulus (256 bit):
   00:db:a7:5b:ab:31:91:b7:7d:
      6a:8a:77:17:c7:2e:
   23:8d:93:0c:38:87:7f:0d:54:fc:
      69:37:2a:f1:a6:
   a6:0e:23
Exponent: 65537 (0x10001)
Modulus=DBA75BAB3191B77D6
      A8A7717C72E238D930C3887
      7F0D54FC69372AF1A6A60E23
writing RSA key
-----BEGIN PUBLIC KEY-----
MDwwDQYJKoZIhvcNAQEBBQAD
      KwAwKAIhANunW6sxkbd9aop
      3F8cuI42TDDiHfw1U
/Gk3KvGmpg4jAgMBAAE=
-----END PUBLIC KEY-----
```

El módulo está representado en hexadecimal. Para pasarlo a decimal puedes utilizar el programa listado en el Listado 1.

```
gcc hex2dec.c -lssl
./a.out DBA75BAB3191B77D6A8
      A7717C72E238D930C38877
      F0D54FC69372AF1A6A60E23
99352209973842013949736850 1
      70185769998267119089063 3
      39396575567287426977500707
```

Obtenido el módulo en decimal, el siguiente paso consiste en factorizarlo.

Factorización del módulo n

Dado que el número que vamos a factorizar no es excesivamente grande, resulta más rápido aplicar

el algoritmo de factorización QS. Este algoritmo lo implementa msieve, programa que puedes descargar a partir de la tabla de referencias. msieve lleva la documentación necesaria para instalarlo y utilizarlo, aunque no resulta nada complicado. Pues basta con el siguiente comando para factorizar el número propuesto:

```
/msieve -v
9935220997384201394973685O
      17018576999826711908906
      33393965755672874269775
      00707
```

Un ordenador actual puede factorizar este número en unos diez minutos, más o menos en función del hardware. El resultado es el siguiente:

```
factor: 297153055211137492311
      771648517932014693
factor: 334346924022870445836
      047493827484877799
```

En este punto, factorizado el módulo n y con el exponenete de cifrado 65537 obtenido en el apartado anterior, ya tenemos todos los datos necesarios para obtener la clave privada.

Obtención de la clave privada y descifrado del mensaje

Dada la dificultad del proceso utilizando herramientas comunes, desarrollaremos un programa que lo haga por nosotros. Encontrarás las fuentes en Listado 2.

Para realizar los cálculos se ha utilizado la librería OpenSSL. Las variables BIGNUM son las utilizadas por esta librería para trabajar con números grandes. Éstas tienen una API propia para realizar operaciones como sumas, restas, operaciones modulares, etc.

El programa de ejemplo empieza colocando en variables BIGNUM los parámetros p, q y e. A continuación, si se observa el código con detenimiento y con la ayuda de los comentarios, puede verse el procedimiento de generación de la clave privada. El mismo procedimiento indicado anteriormente de forma teórica. De hecho, la única diferencia con la generación de una clave *normal* consiste en que p y q se elegirían aleatoriamente, y en nuestro caso los obtenemos de la factorización del módulo. Finalmente, mediante la ayuda de `PEM_write_RSAPrivateKey()` escribimos la clave privada en formato PEM, el utilizado en los ejemplos. Si comparamos la clave generada con la clave privada original veremos que hemos conseguido nuestro objetivo, pues se ha llegado a la clave privada a partir de la clave pública (Listado 3.).

Listado 3. *Llegar a la clave privada a partir de la clave pública*

```
gcc get_priv_key.c -lssl -o get_priv_key
  ./get_priv_key 297153055211137492311771648517932014693 \
  33434692402287044583604749382748487779 9 65537
  -----BEGIN RSA PRIVATE KEY-----
  MIGqAgEAAiEA26dbqzGRt31qincXxy4jjZMMOId/DVT8aTcq8aamDiMCAwEAAQIh
  AMvT1oXa/rxF3mrVLrR/RS7vK1WTsQ5CWl/+37wztZOpAhEA+4jgEkfalFH+0S+1
  IPKD5wIRAN+NmMH4AF0B8jzMAXHHXGUCEGRpRZnGmVkwSlrTgqj+Zu0CEA7v7CQR
  yRxt09zCGNqcYo0CEDEW7mvozMYYLC5o+zgfV4U=
  -----END RSA PRIVATE KEY-----
```

Listado 4. *dfact_client*

```
...
   for(;;)
   {
      get_random_seeds(&seed1, &seed2);
      switch(status)
      {
         case DF_CLIENT_STATUS_WAITING:
            N = recv_N_number(&rel_by_host, host);
            if(!N)
               sleep(DF_TIME_TO_RECV);
            else
               status = DF_CLIENT_STATUS_RUNNING;
         break;
         case DF_CLIENT_STATUS_RUNNING:
         {
            msieve_obj *obj = NULL;
            obj = msieve_obj_new(N, flags, relations, NULL,
                                 NULL, seed1, seed2, rel_by_host, 0, 0);
            if (obj == NULL)
            {
               syslog(LOG_ERR, "Factoring initialization failed");
               free(N);
               return 0;
            }
            msieve_run(obj);
            if(obj) msieve_obj_free(obj);
            while(!send_relations(N, host, relations))
               sleep(DF_TIME_TO_SEND);
            if(unlink(relations)==-1)
               syslog(LOG_ERR, "unlink(): %s: %s", relations,
                       strerror(errno));
            status = DF_CLIENT_STATUS_WAITING;
            free(N);
            }
         break;
         default:
         break;
      }
...
```

Si guardamos la nueva clave privada en un fichero de texto, por ejemplo en rsa_hacked_privkey.pem, ya podemos descifrar el mensaje como sigue:

```
openssl rsautl -decrypt -inkey
      rsa_hacked_privkey.pem -in
                        cipher.txt
Cuarenta y dos
```

Algoritmos modernos de factorización

Los algoritmos de factorización han ido mejorando con el tiempo hasta llegar a un punto en la actualidad, en que disponemos de algoritmos tan rápidos como el método de curva elíptica (ECM), el Quadratic Sieve (QS) o el Number Field Sieve(NFS). De estos algoritmos también existen diversas variaciones en función del tipo de número a factorizar o de la forma de resolver algunas partes del mismo. Dichos algoritmos son bastante complejos y suelen dividirse en varias etapas en las que se van realizando diferentes cálculos que servirán finalmente para factorizar el número. QS y NFS destacan por disponer de una fase de criba (sieve) en la que se recopilan una serie de relaciones que finalmente se utilizarán para construir un sistema de ecuaciones y obtener el resultado. La fase de criba es posible realizarla en paralelo con diferentes máquinas y suele ser la más larga.

En los ejemplos anteriores hemos usado el programa msieve que es una implementación del Multiple Polinomial Quadratic Sieve (MPQS), una variante del QS. El algoritmo QS es el más rápido cuando se trata de factorizar números de menos de 110 dígitos. Pero cuando sobrepasamos este límite, NFS toma el relevo.

Una variante del NFS utilizada para factorizar cualquier tipo de número es el GNFS o General Number Field Sieve. No hay mucho software libre que implemente GNFS, y el poco que existe, no destaca ni por su buena documentación ni por su facilidad de uso. Al menos, en el momento de escribir este artículo. En cualquier caso, vamos a ver como funciona ggnfs, una implementación de GNFS, que aunque no es del todo estable, permite factorizar sin demasiados problemas.

ggnfs está compuesto por un conjunto de herramientas, que utilizadas una a una, permiten cubrir todas las etapas en las que se divide este algoritmo. Para un novato, factorizar un número mediante este procedimiento puede resultar realmente complicado, por lo que ggnfs ofrece un script en perl que hace todo el trabajo.

Este script permite utilizar el programa sin quebraderos de cabeza, pero desde luego no es la mejor forma de sacarle partido a

Listado 5. *dfact_server*

```
...
   for(;;)
   {
      while(child_count >= DF_MAX_CLIENTS) sleep(1);
      sd_tmp = socket_server_accept(sd, client, sizeof(client));
      if((pid=fork())==0)
      {
         close(sd);
         process_client(sd_tmp, N, num_relations, rel_by_host, client);
      }
      else if (pid>0)
      {
         close(sd_tmp);
         child_count++;
      }
      else
      {
         perror("fork()");
      }
      close(sd_tmp);
   }
   close(sd);
...
```

Listado 6. *dfact_server (process_relations)*

```
void process_relations(char *N, int num_relations, int seconds)
{
   for(;;)
   {
      int n_sieves = get_num_relations_in_file(DF_FILE_RELATIONS);
      printf ("relations: %d, need: %d \n", n_sieves, num_relations);
      // Has enought relations?
      if(n_sieves>=num_relations)
      {
         printf("Factoring %s\n", N);
         kill(0, SIGUSR1);
         uint32 seed1;
         uint32 seed2;
         uint32 flags;
         flags |= MSIEVE_FLAG_USE_LOGFILE;
         get_random_seeds(&seed1, &seed2);
         factor_integer(N,flags,DF_FILE_RELATIONS,NULL,&seed1,&seed2);
         printf("Factoring Done\n");
         kill(getppid(), SIGKILL);
         exit(0);
      }
      sleep(seconds);
   }
}
```

¡Ya a la venta!

También puedes comprarlo en nuestra tienda virtual:
www.buyitpress.com

www.lpmagazine.org

las herramientas que constituyen ggnfs.

La forma más sencilla de probar este programa consisten en editar un archivo, al que podemos llamar test.n indicando el número que queremos factorizar:

```
cat test.n
n: 1522605027922533360535
      61837813263742971806 8
      11496138068865790849 4
      58012296325895289765 4
      000350692006139
```

A continuación ejecutamos:

```
tests/factLat.pl test.n
```

Esto factorizará el número. Eso sí, en unas cuantas horas. Más o menos, en función del hardware empleado. Para sacarle más partido a ggnfs es necesario prescindir del script factLat.pl y utilizar las herramientas que proporciona con los parámetros correctos. Dado que la utilización de ggnfs da para un artículo entero, no lo explicaré aquí. Lo mejor para aprender a utilizarlo es leer la documentación proporcionada con el código fuente y acceder al foro de discusión (ver enlaces).

También es conveniente leer algunos documentos sobre el algoritmo NFS, aunque hay que tener en cuenta que se requieren conocimientos avanzados de teoría de números y álgebra lineal.

La necesidad del ataque distribuido

La clave factorizada en el ejemplo es muy pequeña si la comparamos con la longitud de una clave de las que se suelen usar actualmente. Si ahora mismo quisiéramos crear una clave RSA para nuestro uso personal, optaríamos por un mínimo de 1024 bits. Siendo cautelosos escogeríamos una de 2048 o 4096 bits. Si intentamos factorizar una de estas claves con nuestro PC de casa, por muy rápido que sea, veremos como se queda realizando cálculos indefinidamente, sin llegar a ninguna parte. Lo cierto es que no podremos con una clave de estas características. Sin embargo, los continuos avances tanto en computación como en el campo matemático, hacen que esta distancia se vaya reduciendo, y en determinadas condiciones sea posible realizar ataques distribuidos, es decir, utilizando miles de máquinas paralelamente en el proceso de factorización. Existen diversos estudios realizados por expertos en este campo, que analizan las posibilidades de atacar una clave de 1024 bits (ver tabla de enlaces). De momento, esto está fuera del alcance de la mayoría, aunque probablemente, no fuera del alcance de ciertos gobiernos u organizaciones.

Por otra parte, la existencia de retos como el RSA Factoring Challenge, comentado anteriormente, motivan todavía más los estudios en este campo, así como el desarrollo de herramientas distribuidas para la factorización de números grandes.

Ataque distribuido

En ejemplos anteriores hemos conocido el software msieve. Como hemos visto es de utilización sencilla y el programa está lo suficientemente maduro como para no dar demasiados problemas al usuario. Por lo que he visto hasta el momento, ésta es sin duda la mejor implementación abierta del algoritmo Quadratic Sieve. Sin embargo, el programa que se distribuye no es más que una demo del uso básico de la librería msieve. Y solo sirve para ser utilizada en una única máquina.

En la documentación del programa se muestran un par de recetas para utilizar el programa de demostración con diferentes máquinas y así poder hacer una factorización distribuida. Pero se trata de un procedimiento manual y poco práctico. Por este motivo he decidido implementar un pequeño programa de ejemplo que presenta el uso de la librería msieve para realizar factorización distribuida. He llamado a este programa dfact y puedes encontrarlo en el CD que acompaña la revista o en la sección de enlaces.

El programa se puede compilar con un make y solo requiere que la librería msive esté correctamente instalada. La ruta de la misma deberá incluirse en el Makefile. Una vez compilado podemos encontrar dos binarios en la carpeta bin/ correspondientes al servidor y el cliente. El servidor (dfs) se ejecutará en una máquina con bastante memoria (necesitará más cuanto más grande sea el número) y será el encargado de repartir trabajo y coordinar a todos los clientes. El servidor recibe cuatro parámetros: el número a factorizar, el número de relaciones que queremos que recopile el cliente para cada envío y el número de segundos cada cuando queremos que el servidor compruebe si tiene los suficientes datos de los clientes para terminar la factorización con éxito. En el ejemplo siguiente se pide a los clientes que envíen las relaciones de 5000 en 5000, y al servidor que verifique el número de relaciones de que dispone cada 60 segundos.

```
bin/dfs 99352209973842013 94
      97368501701857699982 67
      11908906333939657556 72
      87426977500707 5000 60
```

En unos cuantos clientes ejecutaremos dmc, pasándole como parámetro la IP del servidor y la ruta de un fichero temporal donde puede recopilar las relaciones. Por ejemplo:

```
bin/dfc /tmp/rel 192.168.1.7
```

El programa dfact se ha desarrollado tomando como base la librería msieve. Ésta ofrece un programa de ejemplo llamado demo.c que muestra su uso de manera sencilla. Si se observa el código se puede ver que no resulta complicado de seguir. En el Listado 5. puede verse un pedazo de código del cliente dfact. En él se muestra el funcionamiento del blucle principal en el que el cliente recibe el número a

factorizar del servidor, calcula las relaciones solicitadas mediante msieve, y se las envía al servidor para que las procese (Listado 4.).

Veamos como maneja la situación el servidor (Listado 5.). Cada cliente que solicita enviar la lista de relaciones al servidor es gestionado por process_client() mediante un proceso separado.

Otro proceso separado se ocupa de procesar las relaciones que los clientes van enviando cada cierto tiempo (Listado 6.).

El programa de ejemplo nos permite factorizar un número utilizando varias decenas de máquinas y, aunque podría utilizarse en Internet, la ausencia de autentificación y/o cifrado, entre otros, lo hacen poco recomendable. Algunas mejoras que podrían ser de interés, y que sin duda representarán un buen ejercicio para el lector, son el uso de SSL, mejoras en la seguridad, mejoras en el rendimiento, etc.

Anteriormente hemos comentado que GNFS es más eficiente que MPQS en la factorización de números de más de 110 dígitos. En estos momentos, no parece que exista ningún software libre que permita implementar fácilmente un sistema distribuido de factorización con GNFS, al igual que hemos hecho con msieve (QS). Sin embargo, el autor de msieve lo está dotando de soporte para GNFS, y aunque éste se encuentra todavía a medias, quizás en no mucho tiempo lo tengamos disponible. Si eso sucediese, no sería muy difícil modificar nuestro ejemplo (dfact) para realizar factorización distribuida con GNFS.

En cualquier caso ggnfs dispone de la posibilidad de utilizar varias máquinas para realizar una factorización. Esto puede hacerse con el script factLat.pl, visto anteriormente, aunque es una versión muy inestable y solo permite utilizar unas pocas máquinas en una LAN.

Para finalizar

Para finalizar, quiero mencionar la repercusión que pueden tener los avances matemáticos que se realizan en este campo. Un número que hoy es computacionalmente imposible de factorizar, mañana podría ser factorizable en pocos minutos. Todo depende de que a alguien se le ocurra una forma revolucionaria de atacar el problema. Sin embargo los más de 20 años que lleva en funcionamiento el algoritmo RSA avalan la seguridad del mismo, y hacen poco probable este hecho.

Por otra parte, la quizás inminente llegada de la computación cuántica sí resulta una amenaza seria a la seguridad de este conocido criptosistema. Ésto es debido a la existencia del algoritmo de Shor (ver sección de enlaces), que muestra una forma de atacar el problema con complejidad polinómica. Es decir, que permitiría factorizar una clave en un tiempo razonable. ●

Sobre el autor:

Daniel Lerch Hostalot
Ingeniero de Software C/C++ en plataformas GNU/Linux,
Master: Cisco Networking Academy Program: CCNA, wireless & Network Security, Ingeniero Técnico en Informática de Sistemas por Universitat Oberta de Catalunya UOC, actualmente trabaja en sector de telecomunicaciones.
Conoce los siguientes lenguajes de programación: C/C++, ShellScript, Java, Perl, PHP (prog módulos en C).
e-mail: *dlerch@gmail.com*, url: *http://daniellerch.com*

The RSA Factoring Challenge

- RSA-704 (30.000$): *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA704*
- RSA-768 (50.000$): *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA768*
- RSA-896(75.000$):*http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA896*
- RSA-1024 (100.000$): *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA1024*
- RSA-1536 (150.000$): *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA1536*
- RSA-2048 (200.000$): *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA2048*

Referencias

- Factorización de números enteros grandes - *http://factorizacion.blogspot.com*
- DFACT – *http://daniellerch.com/sources/projects/dfact/dfact-hakin9.tar.gz*
- GGNFS - A Number Field Sieve implementation: *http://www.math.ttu.edu/~cmonico/software/ggnfs/*
- Yahoo! Group for GGNFS - *http://www.groups.yahoo.com/group/ggnfs*
- MSIEVE - Integer Factorization - *http://www.boo.net/~jasonp/qs.html*
- The RSA Factoring Challenge - *http://www.rsasecurity.com/rsalabs/node.asp?id=2092*
- OpenSSL - *http://www.openssl.org*
- Algoritmo de Shor - *http://es.wikipedia.org/wiki/Algoritmo_de_Shor*
- On the cost of factoring RSA 1024 - *http://www.wisdom.weizmann.ac.il/%7Etromer/papers/cbtwirl.pdf*
- Factoring estimates for a 1024 bit RSA modulus - *http://www.wisdom.weizmann.ac.il/%7Etromer/papers/factorest.pdf*