Factorization Attack to RSA

Daniel Lerch Hostalot

Difficulty

●●●

RSA is, without any doubts, the most popular public key criptosystem which is being used and which has survived the analysis of the cripto-analysts for over a quarter of a century. This popular algorithm bases its security on the difficulty which is to factorize big numbers, considering as big numbers as those with over 100 decimal digits.

In this article, we will study the inner workings of RSA and the possibility of running factorization attacks. We will see how RSA keys and the attack procedure are used to obtain the private key out of the public key.

To follow this article and understand it, the reader needs to have a basic knowledge of C programming as well as some mathematical background. In the *On the Net* frame you will find additional material that will allow you to go further.

All the examples have been developed and tested on a GNU/Linux system.

Public key cryptography

On the contrary to private key cryptography, where a single key is used to encrypt and decrypt messages, public key cryptography uses two keys. These two keys are known as a public key and a private one. To cypher the communication, the user needs both of them. While the private key must remain secret, the public one can be available to anyone who wants to send cyphered messages to the user. A message cyphered with a public key can only be deciphered with its corresponding private key. To make this possible, we have to go through some mathematics problems. What we use RSA for is the factorization of the big numbers.

The beginning of the public key cryptography is connected with the publication by Diffie and Hellman from the year 1976. It introdced a protocol that allowed the exchange of certain information over an unsafe channel. Soon afterwards, in 1977, Rivest, Shamir and Adleman proposed the criptosystem RSA, the most widely-used criptosystem nowadays.

In 1997 appeared the documents proving that the cryptographers from the British Government Group for the Security of Electronic Communications (GSEC) had already known about this kind of cryptography in 1973.

What you will learn…

- how RSA works
- how to run factorization attacks

What you should know…

- basic knowledge of C programming

Mathematical concepts

Divisor or Factor:

An integer a is a divisor (or factor) of b when there is other integer c that complies with $b = a \cdot c$. Example: $21 = 7 \cdot 3$

Prime numbers and composed numbers:

An integer is prime if it can only be divided by one and by itself. An integer is composed if it's not a prime.
Example: $21 = 7 \cdot 3$ is a composed number, 7 and 3 are prime numbers.

Factorization:

Factorization of an integer n is the process of decomposing it on its prime factors: $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_i^{e_i}$ where p_i are prime numbers and e_i are positive integers.
Example: 84 is factorized as $21 = 2^2 \cdot 3 \cdot 7$.

Module:

We know as module and we represent it as $a \bmod b$ the rest of the entire division of a between b. Example: $5 \bmod 3 = 2$, $10 \bmod 7 = 3$, $983 \bmod 3 = 2$, $1400 \bmod 2 = 0$.
a and b are module of n: $b \equiv a (\bmod n)$ if their difference (a-b) is a multiple of n.

Maximum Common Divisor:

We call maximum common division of two integers a and b, represented as $\mathrm{mcd}(a,b)$, the bigger integer that can divide a and b.
Example: $\mathrm{mcd}(42,35) = \mathrm{mcd}(2 \cdot 3 \cdot 7, 5 \cdot 7) = 7$ Euclidean Algorithm:
The Euclidean Algorithm calculates the maximum common divisor of two numbers based on $\mathrm{mcd}(a,b) = \mathrm{mcd}(b,r)$, where $a > b > 0$ are integers and r rest of the division between a and b
Example: $\mathrm{mcd}(1470, 42)$
(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) $35 \bmod 7 = 0 \rightarrow \mathrm{mcd}(7,0) = 7$ Euler Indicator (Totient):
Given $n \geq 0$ we know as $\Phi(n)$ the number of integers in the interval $[1,n]$ that are prime* with n. For $n = p \cdot q$, $\Phi(n) = (p-1)(q-1)$.
*Two integers a and b are prime between themselves (or co-prime) if $\mathrm{mcd}(a,b) = 1$.

The RSA criptosystem

As we have said before, the security of RSA lies on the computational difficulty that represents the factorization of big numbers. Factorizing a number is to find the prime numbers (factors) that multiplied result on that number. If we want, for example, to factorize the number 12, we will obtain as a result 2·2·3. The simple way to find the factors of an *n* number is to divide it by all the prime numbers smaller than *n*. This procedure, although simple, is extremely slow if we want to factorize big numbers.

Let's make some calculations to get the idea. A 256 bits key (as the one we will break later) has around 78 decimal digits (1078). As on the RSA keys this number usually has only two prime factors, each of them will have more or less 39 digits. This means that to factorize the number we will have to divide it by all the prime numbers of 39 digits or less (1039). Supposing that only 0.1% of the numbers are prime numbers, we will have to make around 1036 divisions. Let's imagine we have a system capable of making 1020 divisions per second. In this case, we will spend 1016 seconds on breaking the key. In other words, more than 300 million years, a million times the age of the universe. Luckily, we will spend a little bit less.

Let's see how RSA works. We will begin by generating the public and the private keys (in the table on the side we have some interesting mathematical concepts shown). For this purpose it is necessary to proceed the following steps.

Step 1

We randomly choose two prime numbers p and q, and we multiply them, obtaining n: $n = p \cdot q$. If we choose, for example, that $p = 3$ and $q = 11$ we obtain $n = 33$.

Step 2

We calculate the Euler (Totient) indicator with the following formula: $\Phi(n) = \Phi(p \cdot q) = (p-1) \cdot (q-1)$. In this example we will get $\Phi(n) = 20$.

Step 3

We find a cypher exponent (later we will use it to cypher) and we call it e. This number must be compatible to $\mathrm{mcd}(e, \Phi(n)) = 1$ A good example could be: $e = 3$ as it doesn't have any common factor with 20 (factors 2 and 5).

Step 4

We calculate a de-cypher exponent and we call it d (later we will use it for de-cyphering). This number must tally with $1 < d < \Phi(n)$ so that $e \cdot d \equiv 1(\bmod \Phi(n))$. This means that d will be a number between 1 and 20 which multiplied by 3 and divided by 20 will be 1. d, can be 7 then.

The keys

The user's public key belongs to the couple (n,e), in our example (33, 3), and the private key is d, so it's 7. Logically, the numbers p, q and $\Phi(n)$ should remain secret.

(De)cyphering

At this point, we only need to cypher with $C = M^e \bmod n$ and decipher with $M = C^d \bmod n$. If we consider that our message is $M = 5$, the corresponding cyphering will be $C = 5^3 \bmod 33 = 26$.

To de-cypher we would only have to apply $M = 26^7 \bmod 33 = 5$.

As we said in the beginning and as we can see from the previous procedure, the security of the criptosystem resides in n. This means that if an attacker who can access the public key manages to factorize n obtaining p and q, he only has to use the previous formulas to obtain the private key.

The RSA factoring challenge

The RSA Factoring Challenge is a contest financed by RSA Laboratories in which great prices are awarded to those who can factorize certain very large numbers. What allows them to know the state of the art of the factorization systems is being aware which key length is necessary to keep RSA safe.

While we write this article, the factorization record is RSA-640, a 193-digit number that was factorized on the 2nd of November 2005 by F. Bahr et al. The next challenge is RSA-704, with a 30.000$ award.

Without a doubt, The RSA Factoring Challenge is a great way to know the actual situation of the factorization systems.

You can see the current challenges in a table situated at the end of this article.

Factorization attack

In the following lines, we will make an example attack to an RSA key. To make the calculations faster we will use a key much shorter than normal, simplifying its factorization. Even if it's not a real example, it will be good to know how a complete attack is made.

First of all, we will create a work environment with OpenSSL, generating the necessary keys and cyphering a message that we will use as an objective for our attack. Later we'll factorize the n module and obtain the private key, finally deciphering the message.

OpenSSL and RSA

OpenSSL is a very useful open source cryptographic tool. In the reference section you will find where to download it, but most of the GNU/Linux distributions include it by default. In this section we will use it to configure a test environment in which we will run the attack.

The first step is to generate a couple of keys to cypher and decipher. We will generate 256-bits keys, too short to keep our communications safe, but enough for our exercise.

We generate a pair of keys, keeping our private key secret.

```
# Generate a pair of RSA 256-bit
keys
openssl genrsa -out rsa_privkey
.pem 256
cat rsa_privkey.pem
-----BEGIN RSA PRIVATE KEY-----
MIGqAgEAAiEA26dbqzGRt31qincXxy
   4jjZMMOId/DVT8aTcq8aam
   DiMCAwEAAQIh
AmvT1oXa/rxF3mrVLrR/RS7vK1WT
   sQ5CWl/+37wztZOpAhEA+4jg
   EkfalFH+0S+1
IPKD5wIRAN+NmMH4AF0B8jz
   MAXHHXGUCEGRpRZnGmV
   kwSlrTgqj+Zu0CEA7v7CQR
yRxt09zCGNqcYo0CEDEW7mvoz
   MYYLC5o+zgfV4U=
 -----END RSA PRIVATE KEY-----
```

Following this step, we save the public key in a file. This is the key we will publish to allow anyone to send us cyphered messages.

```
# Saving the public key on a file
openssl rsa -in rsa_privkey.pem
   -pubout -out rsa_pubkey.pem
cat rsa_pubkey.pem
-----BEGIN PUBLIC KEY-----
MdwwDQYJKoZIhvcNAQEBB
   QADKwAwKAIhANunW6sxkbd
   9aop3F8cuI42TDDiHfw1U
/Gk3KvGmpg4jAgMBAAE=
-----END PUBLIC KEY-----
```

After generating this pair of keys, we can cypher and decipher. We will work with the following message:

```
echo "Forty-two" > plain.txt
```

This message could be easily cyphered by the use of the following command and the public key:

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

To de-cypher we will use the private key:

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

Once we have seen how to use OpenSSL with RSA and knowing the need to have the private key to decipher the messages, our objective is to obtain this private key without accessing the original. In other words, how to obtain the private key using the public key. The first thing we need to do this is to obtain the n module and the cypher exponent. This can be done with the following command and the public key:

```
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=DBA75BAB3191B77D
   6A8A7717C72E238D930C38877
   F0D54FC69372AF1A6A60E23
writing RSA key
-----BEGIN PUBLIC KEY-----
MdwwDQYJKoZIhvcNAQEBBQ
   ADKwAwKAIhANunW6sxkbd9
   aop3F8cuI42TDDiHfw1U
/Gk3KvGmpg4jAgMBAAE=
-----END PUBLIC KEY-----
```

The module is represented in hexadecimal. In order to convert it to decimal you can use the program shown in Listing 1.

```
gcc hex2dec.c -lssl
./a.out DBA75BAB3191B77D6
```

Listing 1. *Transform Hexadecimal to 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);
}
```

```
   A8A7717C72E238D930C388
   77F0D54FC69372AF1A6A60E23
99352209973842013949736850170
   18576999826711908906333939 6
   5755672874269775007 07
```

Once obtained the module in decimal, the next step is to factorize it.

Factorization of the n module

As the number we are factorizing is not too great, it's faster to apply the QS factorization algorithm. This algorithm is implemented by *msieve,* a program that you can download looking at the reference table. *Msieve* has enough documentation to install and use it, which is not at all complicated. It's enough with the following command to factorize the proposed number:

```
/msieve -v
993522099738420139497368 5
   0170185769998267119089063 3
   3939657556728742697750070 7
```

A modern computer can factorize this number in around ten minutes, depending on the hardware. The result follows:

```
factor: 2971530552111374923 11
   771648517932014693
factor: 33434692402287044583 6
   047493827484877799
```

At this point, once factorized the n module and with the cypher exponent 65537 obtained with the previous step, we have all the necessary data to obtain the private key.

Obtaining the private key and de-cyphering the message

Because of the difficulties of this process when using common tools, we will develop a program that can do it for us. You will find the sources in Listing 3.

To do the calculations we have used the OpenSSL library. The BIGNUM variables are used by this library to work with big numbers. These have their own API to make

Listing 2. *A private key*

```
#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;
   }
   BN_mul(n, p, q, ctx);
   // We calculate 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
   // We calculate d mod (p-1)
   BN_mod(dmp1, d, r1, ctx);
   // We calculate d mod (q-1)
   BN_mod(dmq1, d, r2, ctx);
   // We calculate the reverse of q mod p
   BN_mod_inverse(iqmp, q, p, ctx);
   // RSA keys
   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;
}
```

operations such as addition, subtraction, modular operations, etc.

The example program begins putting in BIGNUM variables the parameters p, q and e. Following this, if you look at the code in detail and with help of the commentaries, you can see the procedure of generating the private key. The same procedure we previously explained in theory. In fact, the only difference between the test and real generation of the key lies in the fact that p and q would be randomly chosen. In our case we obtained them from the factorization of the module. Finally, helped by PEM_write_RSAPrivateKey() we write the private key used in the examples in PEM format. If we compare the generated key with the original private key we can see that we have achieved our objective, as we have made it to the private key from the public key.

If we keep our new private key on a text file, for example rsa_hacked_privkey.pem, we can decipher the message:

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

Listing 3. *Changing a private key into a public key*

```
gcc get_priv_key.c -lssl -o get_priv_key
   ./get_priv_key 297153055211137492311771648517932014693 \
   334346924022870445836047493827484877799 65537
   -----BEGIN RSA PRIVATE KEY-----
   MIGqAgEAAiEA26dbqzGRt31qincXxy4jjZMMOId/DVT8aTcq8aamDiMCAwEAAQIh
   AMvT1oXa/rxF3mrVLrR/RS7vK1WTsQ5CWl/+37wztZOpAhEA+4jgEkfalFH+0S+1
   IPKD5wIRAN+NmMH4AF0B8jzMAXHHXGUCEGRpRZnGmVkwSlrTgqj+Zu0CEA7v7CQR
   yRxt09zCGNqcYo0CEDEW7mvozMYYLC5o+zgfV4U=
   -----END RSA PRIVATE KEY-----
```

Listing 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;
      }
...
```

Modern factorization algorithms

Factorization algorithms have improved much in time, so that today we have such fast algorithms as the Elliptic Curve Method (ECM), the Quadratic Sieve (QS) or the Number Field Sieve (NFS). From this algorithms come also certain variations dependent on the type of number that we have to factorize or on the way to resolve certain parts of it. These algorithms are rather complex. They are normally divided into steps in which different calculations leading to factorizing the number are made. QS and NFS have a sieve step. At this stage some kind of relations are gathered, which finally construct a system of equations to obtain the result. The sieve step can be done by several machines working simultaneously, as it's normally the longest stage.

In the examples we have used the msieve program, an implementation of the Multiple Polynomial Quadratic Sieve (MPQS), a variation of QS. The QA algorithm is faster when you have to factorize numbers of less than 110 digits. But when we go beyond this limit, NFS should be applied. A variation of NFS used to factorize any type of number is GNFS or General Number Field Sieve. There is not much free software that implements GNFS, and the one that exists neither has a good documentation available nor is easy to use. At least, that is the case when we are writing this article. Anyway, we will see how GGNFS works. It is an implementation of GNFS that although not being completely stable, allows us to factorize without too many problems.

GGNFS is composed by a group of tools, that used one by one, can go through all the steps that compose this algorithm. For a newbie, factorizing a number through this procedure can be really complicated. That's why GGNFS includes a perl script which does all the job. The script allows us to use the program flawlessly however it's really not the best way of getting the most out of the tools that compose GGNFS.

The simplest way to try this program is to edit a file, that we can call test.n indicating the number that we want to factorize.

```
cat test.n
n: 1522605027922533360535618378132637429718068114961380688657908494580122963258952897654000350692006139
```

Later we run:

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

And this will factorize the number. Well, after some hours. The time depends on the hardware used. To make the most out of GGNFS it's necessary to forget about the script factLat.pl and to use the tools it has with the correct parameters. As GGNFS usage can take a whole article, I'm not going to explain it here. The best way to learn how to use it is to read the documentation available with the source code and to check the discussion forum (see On the Net frame). It is also advisable to read some documents about NFS. Nevertheless, we have to take into account the fact that we'll need some advanced knowledge on linear algebra and number theory.

The need for a distributed attack

The key factorized on this example is very small when compared to the length of the kind of key used nowadays. If right now we want to create an RSA key for our personal use, we should use a minimum of 1024 bits. If we want to be safer, we should use a key of 2048 or 4096 bits. When we try to factorize one of these keys with our home PC, no matter how fast it is, we will see how it stays doing endless calculations, not going anywhere. The truth is, we cannot break such a key. But the advances on computers and mathematics make the distance to this objective smaller and smaller every day. Under certain conditions, we can do distributed attacks using thousands of machines simultaneously helping with the process of factorization. There are many studies done on this field analyzing the possibilities of attacking a 1024 bits key (see the links table). At this point, this is beyond most people's reach, but not beyond the reach of certain governments and organizations.

Also the existence of competitions such as the previously mentioned RSA Factoring Challenge helps the

Listing 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);
...
```

Listing 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);
      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);
   }
}
```

experts on this field and gives them motivation to create distributed tools for the factorization of big numbers.

Distributed attack

In previous examples we have seen the software *msieve*. As we have learned, it's easy to use and the program is developed enough not to create too many problems to the user. In my opinion, this is the best software so far which implements the Quadratic Sieve algorithm. But the program is not more than a demo of the basic usage of the msieve library, and it can only be used on a single machine.

On the program's documentation there are a couple of recipes to use the demo program with different machines so that a distributed factorization can be done. It is, however, a manual and not very practical procedure. That is why I have decided to implement a small example program that introduces the usage of the msieve library to do distributed factorization. This program is called dfact and you can find it on the CD that comes with this magazine and on the links section. The program can be compiled with a make and it only requires a msieve library correctly installed. The path of this library has to be included in the Makefile. Once compiled we can find two binaries on the folder bin/ which corresponds to the client and server sides. The server (dfs) will be executed on a machine with enough memory (the bigger the number, the more memory is needed) and will be the one to distribute workload and coordinate the clients. The server gets four parameters: The number to factorize, the number of relations we want the client to recompile for every packet sent and the number of seconds for the server to check if it has enough data from the clients to finish the factorization successfully. In the next example we ask the clients to send the relations every 5000 and the server to verify the number of relations every 60 seconds.

```
bin/dfs 9935220997384201394
        973685017018576999826
        711908906333939657556
        7287426977500707 5000 60
```

We will run dmc in a couple of clients, giving it, as a parameter, the IP of the server and the path to a temporary file where the relations can be save. For example:

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

The program *dfact* has been developed using the msieve library as its base. This one has an example program called demo.c that shows it's usage in a simple way. If we observe the code we can see it's not too difficult to follow. In Listing 4 we can see a piece of code of the dfact client. Here we show the inner works of the main loop where the client gets the number to factorize from the server, then calculates the relations asked through msieve, and sends them to the server so that it can process them. Let's see how the server handles the situation (Listing 5). Every client asking for sending the list of relations to the server is managed by process_client() through a separate process.

Another separate procedure takes care of processing the relations that the clients send in regular time intervals (see Listing 6).

The RSA Ractoring 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*.

On the Net

- Factorization of big integers – *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*,
- The Shor algorithm – *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*.

About the author

Daniel Lerch Hostalot, C/C+ Software on GNU/Linux platforms engineer, MA in Wireless & Network Security from Cisco Networking Academy Program (CCNA), Technical Engineer for IT Systems graduated from Oberta University, Catalonia (UOC). Currently working for telecommunication sector. Knows following programming languages: C/C++, ShellScript, Java, Perl, PHP (C modules program).
e-mail: *dlerch@gmail.com*, url: *http://daniellerch.com*

The example program can let us factorize a number using several machines. Even though it could be achieved via Internet, the lack of authentication and/or cyphering mechanisms makes it not advisable. The improvement (this could be a good exercise for the readers by the way) could be the usage of SSL, strengthening of the security, performance optimizations, etc...

We have mentioned previously that GNFS is more efficient than MPQS when factorizing the numbers of over 110 digits. At this point, it seems that there is no open source software allowing to easily implement a distributed system of factorization with GNFS as we have done with msieve (QS). The author of msieve, however, is preparing the support for GNFS. Even he is currently half-way through this, it is possible that in the near future it will be available. If this happens, it wouldn't be very difficult to modify our example (dfact) to make distributed factorization with GNFS.

Anyway, GGNFS has the possibility of using several machines for factorization purposes. This can be done through the script *factLat.pl*, as shown previously, but it's a very unstable version and it only allows to use a few machines on a LAN.

Conclusion

To finish, I want to mention the repercussion that can have mathematical advances on this field. A number that today is impossible to factorize through computation, tomorrow could be factorized in a few minutes. Everything depends on that if someone has a revolutionary idea to tackle the problem. However, the experience of 20 years of working with RSA algorithms speaks for its security and makes this possibility quite remote.

Also, the imminent release of quantum computers will be a serious threat to the security of this known criptosystem. This is due to the Shor algorithm (see On the Net frame) that shows a way to tackle the problem with polynomial complexity. This will allow to factorize a key in a very reasonable time. ●