Introduction

Here at the Linux Supporters Group Adelaide we use `GnuPG` (the Gnu Privacy Guard suite of programs) as our encryption method of choice, because it is open source and has no arbitrary restrictions. As the revelations[1] about the N.S.A. promise no abatement, it behoves us to learn more than we ever thought we needed to know about encryption – and its vulnerabilities. This talk introduces you to the mathematics used in the RSA[2] implementation of public key encryption[3].

Appreciation of the RSA Procedure

In the old days it was sufficient for Edgar Allan Poe to sprinkle `"etaoin shrdlu"` over a cryptogram and the plaintext would spring out with open arms. Not any more. Modern algorithms remove all trace of letter-frequency bias, so there is no *cracking* using those clues.

RSA itself gives you some good things:

- You need not worry about the interception of a key transmitted to the recipient because RSA is *designed* to have public keys.
- You can be sure that the person claiming to have sent the message really did send it, and that it has not been tampered with.
- The security of the algorithm depends on the assumption that, knowing the public key, decryption requires discovering just how this key is a product of two very large prime numbers, the calculation of which would take too long on current computers.

On the other hand, you must keep the secret key *very* secret, and always bear in mind that a better algorithm for *cracking* RSA keys might be invented by anyone at any time. In fact, people regularly publish the latest factoring of a large non-prime number. A recent one is known by the name `RSA-768`. It is this 232-digit non-prime number

```
N = p*q = 1,230,186,684,530,117,755,130,494,958,384,962,720,772,853,569,595,334,
792,197,322,452,151,726,400,507,263,657,518,745,202,199,786,469,389,956,474,942,
774,063,845,925,192,557,326,303,453,731,548,268,507,917,026,122,142,913,461,670,
429,214,311,602,221,240,479,274,737,794,080,665,351,419,597,459,856,902,143,413
```

which is the product of this 116-digit prime number

```
 p = 33,478,071,698,956,898,786,044,169,848,212,690,817,704,794,983,713,768,568,
912,431,388,982,883,793,878,002,287,614,711,652,531,743,087,737,814,467,999,489
```

and this 116-digit prime number

```
 q = 36,746,043,666,799,590,428,244,633,799,627,952,632,279,158,164,343,087,642,
676,032,283,815,739,666,511,279,233,373,417,143,396,810,270,092,798,736,308,917
```

I've shown these numbers to demonstrate the practical scale of these matters and to show that factoring is difficult but not impossible. Check the result for yourself (!) *Hint:* Abandon pencil, take up the Linux `'dc'` app., which processes arithmetic of practically unlimited precision, define the `p,q` strings above (no commas) and invoke something like: `echo "$p $q * p"|dc`.

[1]http://www.theguardian.com/world/the-nsa-files
[2]http://en.wikipedia.org/wiki/RSA_%28algorithm%29
[3]http://en.wikipedia.org/wiki/Public-key_cryptography

Summary of the RSA Algorithm

RSA is an algorithm that is best suited to encrypting relatively small messages. It is perfect for exchanging encryption keys that are destined to be used by various symmetric and block ciphers such as AES, Serpent, Twofish etc., to encrypt data more efficiently, with a smaller key length and much faster. Without going into the theory[4] behind this, you take these steps to use RSA:

- You and your computer create two numbers <N,e> which comprise your *public key.*
- You and your computer create three numbers <p,q,d> which comprise your *private key.*
- You publish your public key <N,e> so that other people can encrypt a message M and send it to you using the mathematical equation $C = \mathit{encrypt}(M) = M^e(\mathit{mod})N$.
- You keep your private key <p,q,d> very secret, and you can receive their cipher C and decrypt it using the mathematical equation $M = \mathit{decrypt}(C) = C^d(\mathit{mod})N$.

Your Public Key

Choose (if you can find them!) two rather large prime numbers with the same number of digits. When I say rather large I mean greater than 308 decimal digits or 2^{1024} — and as time goes on, the necessary minimum key lengths will only increase as computing power increases — seriously long numbers, if you had to write them out by hand.

Let's call these two prime numbers p and q. Multiply these two numbers to obtain what is called the modulus: $N = p * q$. This number N is the first number that people need in order to send you encrypted messages. It is believed to be very hard to factor[5] into p and q.

The other number that people need in order to send you encrypted messages is the encryption exponent, called e. If $S = (p-1)*(q-1)$ then e is the largest number between 3 and $N-1$ satisfying the condition $\mathit{gcd}(e, S) = 1$, which means the number e has no divisors in common with the divisors of S. You'll need to look further[6] to understand `gcd` (the greatest common divisor function).

These two numbers $<N,e>$ comprise your public key.

Your Private Key

You need to calculate the decryption exponent, called d. (This is going to sound more complicated than it is in practice.) Given our chosen p and q, we define $S = (p - 1) * (q - 1)$. Now look for an integer i so that the number $i * S + 1$ is exactly divisible by e. When this happens, you'll have an integer value of d which makes $d * e = i * S + 1$. That's the unique number we are after.

These three numbers $<p,q,d>$ comprise your private key.

Encrypting a Message to Send

A message might consist of a number called $M<N$, and your friend wants to encrypt that number M to a cipher called C. The method is quite simply to calculate $C = M^e(\mathit{mod})N$. In english, this means: first multiply M by itself e times, and then divide that result by N, and whatever is left over (the *remainder*) is the encoded *ciphertext*, which is sent to you.

[4]http://www.mathaware.org/mam/06/Kaliski.pdf
[5]http://www.crypto-world.com/FactorPapers.html
[6]'Basic Number-Theoretic Functions' in *Cryptography in C and C++* by Michael Welschenbach.

Decrypting a Message that is Received

You receive the encrypted *ciphertext* called C. It is only necessary to calculate $M = C^d(mod)N$. In english this means: multiply the number C by itself d times, and then divide the result by N. Whatever is left over (the *remainder*) is the decrypted number M, the retrieved message.

A Numeric Example of the RSA Algorithm

Let's do a complete cycle as an illustration, but with very small numbers so you can see what is happening without being overwhelmed by the calculations. Note that there are myriad theorems[7] that make the arithmetic of these calculations tractable when the numbers are realistically large.

you choose two primes $p = 7$ and $q = 13$ are two prime numbers.

you compute their product $N = p * q = 7 * 13 = 91$

you compute the sub-product $S = (p - 1) * (q - 1) = 6 * 12 = 72$

you choose the encryption exponent e so that $gcd(e, 72) = 1$ (e has no divisors that 72 has). Since gcd(3,72)=3 and gcd(4,72)=4 and gcd(5,72)=1, we therefore use $e = 5$.

you compute your private key d such that $e * d(mod)S = 1$, that is, there is a remainder of 1 when we divide $e * d$ by S. In this case, $5 * d(mod)72 = 1$ (see following).

```
we want some integer that makes   integer*72+1    exactly divisible by 5
we try integer=1   we get   1*72+1=73    which is not divisible by 5
we try integer=2   we get   2*72+1=145   and this is divisible by 5
so e*d=5*d=145
and so d=29
and checking   e*d(mod)S = 5*29(mod)72 = 145(mod)72 = 1
```

you publish your public key $N = 91$ and $e = 5$

they establish their message M=41 (say, the ASCII representation of the letter A)

they encrypt their message $C = M^e(mod)N = 41^5(mod)91 = 115856201(mod)91 = 6$

they send, and you receive, the cipher $C = 6$

you recall your public and private key $N = 91$ and $d = 29$

you decrypt the cipher $C^d(mod)N = 6^{29}(mod)91 = 36845653286788892983296(mod)91 = 41$ and indeed '41' is the message that was sent.

Conclusion

Really, this talk has not even scratched the surface[8]. Do further research! And good luck!

[7] In, for example, *Cryptography and E-Commerce* by Jon Graff.

[8] Try this → *Applied Cryptography: Protocols, Algorithms and Source Code in C* by Bruce Schneier