Encryption
- An introduction to the concepts and solutions involved in the secure transmission of data in the modern world.

Stephen White

November 30, 1999

en·crypt (Ĕn-krĭpt)

v. tr. en·crypt·ed, en·crypt·ing, en·crypts.

1. To put into code or cipher.
2. *Computer Science.* To scramble access codes to (computerized information) so as to prevent unauthorized access.

1 Simple Codes

1.1 Letter Subsitution

I expect everyone will have seen simple codes, used to hide text from people. Prehaps the most simple of these are 'letter substitution codes'. Here we take the English alphabet and subsitute one letter for another.

In computing a commonly used letter subsitution code is **rot13**. This takes each letter in the alphabet and subsitutes it for the one 13 places forward or backward along the alphabet. Since the alphabet has 26 letters this arithmetic is modulo 26 and so rot13 is it's own inverse.

```
rot13('aliens') = 'nyvraf'
rot13('nyvraf') = 'aliens'
```

This is often used on usenet to hide text we want the reader to have to choose when, or even if, they want to read it. It is clearly far too simple to use and assume that others *can't* read it.

In fact if we really want to hide text from people any letter substitution code is not adequate, since they are all easy to crack. At first glace the $26! \approx 4 * 10^{26}$ possible arrangements seems too many to consider checking each one individually, but we don't have to. It is commonly known that 'E' is the most common letter in the english alphabet, and we can use patterns like this to 'crack the code'.

1.2 Using a keyword

A more complex approach would be to choose a word, and somehow combine each letter of this word with successive letters of the message. A trivial example of this that is used in the computing world is to convert the message and the word (or **key**) into a binary representation and then use XOR (exclusive OR) to combine the two streams of bits. To allow for the fact that the key is (presumably) shorter than the message we repeat it over and over again to provide enough bits to enocde the message.

However, there are still at least two ways we can attempt to crack this sort of message.

- If them message is many times longer than the key then we can use the same method as for the simple letter substitution code - since if we know that the key has length n then then letters in the message at positions n, $2n$, $3n \cdots$ will all get encrypted with the same letter from the codeword. Since the each letter in the coded message is the result of a 1-1 function between the corresponding letter in the plaintext (unencrypted) form of the message and the corresponding letter from the codeword it is easy to see that this is no better than a letter subsitution code.
- We can use a knowledge of what the message is likely to contain to obtain the key. We can then use the key to decrypt the rest of the message. For example, if we were decoding something we suspected to be a letter it would be reasonable to assume that it contained 'Dear, ' somewhere near the beginning. Since the encoding function for each letter was 1-1 it has an inverse - and we can use this to calculate the key given part of the origional message.

Of course by making the function more obscure (less easy to guess) we make the inverse harder to find, but this is obfuscation not encryption. If the algorithm is used by many people this wouldn't work anyway .. since they would all have a knowledge of the function. Even if the function was hidden in software we could reverse engineer the software to reveal the function.

1.3 A perfect code?

Anyone who was paying attention during the previous section will have realised that both the methods of cracking the message relied on the key being substaancially shorter than the message. This is true and leads us to a method of encryption called **the one time pad**. This, I believe, is the only method of encryption that has been proved to be impossible to crack and it is very simple: generate a key longer than the message and combine the two using a 1-1 function. It doesn't matter how simple our function is - since knowing it's inverse doesn't help.

Okay, it's slightly more complex than that. In order to ensure that it is impoosible to crack the message we must guarantee that the key is entirely unpredicatable at every stage (i.e. entirely random) and the the encoded message is also statisically indistingushable from random 'noise'. This is actually harder than it sounds, for large messages it is quite difficult to find anything with enough entropy to create an entirely random key. Having said that, this isn't the biggest problem with the one time pad. The big problem is that we need to have a secure data channel over which we can send the key. We have to transfer as much, if not more, data over this secure channel (i.e. the key itself) than we do over the insecure channel that we are sending the message over. This is clearly silly so we will stop and look at what we really want to use our code for.

2 The Real World

1. There is no secure channel over which we can send keys. Furthermore we can assume that since we know, and the person we are communicating with knows, the algorithm we are using then so does the rest of the world. There can be no obfuscation!

2. We would like to be able to authenticate the sender of the message, to be sure it is who we think it is.

3. We can assume that anyone who wishes to crack our encrypted message has access to a powerfull computer that can be used to crack small keys and to search for any patterns. Desktop computers are rapidly approaching the point where they can complete 1 billion calculations a second on floating point numbers, and this is before we even start to consider super-computers.

It is easy to see how 1 applies to sending messages (credit card numbers maybe) over the internet, but the same principles could be applied if the only method of communication is via radiowaves - or even the telephone.

The transmission of Credit Card numbers can again be used as an example for why we might wish for 2.

equirement 3 really considers the problems in achieving the first 2. It is, however, important since we are reminded of the work that went on at Bletchley Park during World War II. Here early computers were used, in combination with 'traditional' code-breaking techniques to crack the Lorenz Ciper used by the Nazi Enigma Machines. Each Enigma machine had 3 wheels which could each be put in any starting position and came with tables showing which starting positions should be used on which dates (i.e. the starting positions became the key). However the use of early computers meant that this key-space (the number of possible keys) was not big enough to prevent the code being cracked. The Nazi U Boats (submarines) had 4 wheel machines, increasing the number of keys and making their transmissions much harder to crack. Unfortunately for them, sloppy use combined with the Allies correctly guessing the content of some messages (such as weather reports) allowed even these transmissions to be cracked.

3 PGP

PGP stands for 'Pretty Good Privacy' and describes itself as "A suite of tools for encrypting, decrypting and verifying messages." It is widely used on computers for encrypting files and email as well as to provide a means of adding 'digital signatures' to things so that people can be sure it was you who wrote it. PGP also demonstrates techniques and algorithms that are used by many other pieces of software.

3.1 Public/Private Key Cryptography

This is a system for communicating accross a channel that ensures that only the intended recipient can read the message. We start by generating a **public key** and a **private** or **secret key**. We will consider a communication between two people. It is traditional to call these two people "Alice" and "Bob". It doesn't take a great deal of thought to realise that this is so we can call their public keys P_A and P_A and their secret keys S_A and S_B.

Now assume that all permissable messages form a set (prehaps the set of all finite-length bit sequences). We now define 4 one-one functions on this set (i.e. permutations). The function corresponding to Alice's public key P_A we will call $P_A()$ and the one corresponding to her secret key S_A we will call $S_A()$. These functions are also inverses of each other. So given a message M:

$$M = S_A(P_A(M))$$

(The functions relating to Bob's keys are predictably named $P_B()$ and $S_B()$ and are also inverses).

It is now clear to see how the system can work. Suppose Bob has message M and he wants to pass it securely to Alice. Since he (along with the rest of the world) knows Alice's public key P_A he is able to calculate $C = P_A(M)$. He can now safely pass C to Alice across the insecure channel. Once she has got C Alice can calculate $M = S_A(C)$ without any difficulty since she knows S_A (the inverse function to P_A). No-one else (Bob included) can do this calculation since they don't possess S_A.

Of course there is still the problem of choosing suitable functions. Remember we must be able to happily reveal P_A and this $P_A()$ while being more than happy that it is sufficiently impractical to calculate it's inverse function $S_A()$ (or equally S_A) from it.

3.2 RSA

I will look at one solution to this problem of finding suitable functions $S_A()$ and $P_A()$ that is used by a system called the Rivest-Shamir-Adleman (RSA) System. RSA, and several other public/private key systems, are supported by PGP.

The problem clearly revolves around finding a 'one way' function. A function that is easy to calculate yet it's inverse is not easy to calcualate.

> Given any two numbers, we may by a simple and infallible process obtain their product, but it is quite another matter when a large number is given to determine it's factors. Can the reader say what two numbers multiplied together will produce the number 8616460799?
> I think it is unlikely that anyone but myself will ever know.
> W. S. Jevons, 'The Principles of Science', Macmillan, 1874.

It doesn't take much time for a Pentium III to tell us that the two numbers Jevons spoke about are 96079 and 89681. However it is also true that we don't have any algorithms that are much better at calculating factors of n than one that simply tries every (prime) number up to $\sqrt{n}$. I've put 'prime' in brackets because it is often quicker to test to see if a number is a factor of n than it is to check if it is a prime. If we choose a large n, say 100 digits then it will clearly take even a computer a long time to find it's factors. Let's assume we check every prime number up to $\sqrt{10^{100}}$, and that we can check 1 billion numbers a second. We'll ignore the fact that finding the complete list of primes less than n is also non-trivial!

We refer to the prime number theorem that suggests that the number of primes less than x is approximately $x/\ln x$.

$\sqrt{10^{100}} = 10^{50}$
Apply prime number theorem
$8.69 * 10^{47}$
Assuming 1 billion a second
$8.69 * 10^{38}$ seconds
$2.41 * 10^{35}$ hours
$1.01 * 10^{34}$ days
$2.75 * 10^{31}$ years
Which is around 100 billion billion times the expected lifetime of the universe!

So it would appear that a suitable one way function is the multiplication of two large prime numbers. It is easy to multiply such large numbers together, but clearly very hard to factor their product if that is the only information that we are given.

It will now come as no surprise that this is the function that RSA relies on. The procedure is as follows:

1. Find two large prime numbers p and q. These should be around 100 digits in length. Define nn by $n = pq$.
2. Find a large random integer d that is relatively prime to $(p-1)(q-1)$.
3. Computer e (where $1 \leq e \leq (p-1)(q-1)$) from the formula $ed = 1(\text{mod } (p-1)(q-1))$
4. Publish the pair $P = (e, n)$ as your public key.
   Keep $S = (d, n)$ as your secret key.

We can now encrypt M using the function

$$C = M^e (\text{mod } n)$$

and we can decrypte C using the function

$$D = C^d (\text{mod } n)$$

Lets consider encrypting the string 'Why broccoli?' In ASCII this is represented by the following numbers.

87,104,121,32,98,114,111,99,99,111,108,105,63

Now suppose we chose prime $p = 47$ and $q = 59$. $n = pq = 2773$. A d coprime to $(p-1)(q-1)$ could be $d = 157$ which gives us $e = 17$. Now lets encrypt our string.

$87^{17} = 652$ (mod 2773)
$104^{17} = 193$ (mod 2773)
$121^{17} = 196$ (mod 2773)
...

If we continue like this we get an encrypted string of:

652,193,196,2227,1860,1684,131,2576,2576,131,169,1020,2092,2073

This clearly isn't great because our key was short and by encrypting one letter at a time we're not doing much better than a simple letter substitution code - but it shows the basic principle. Now to show that we really can decrypt the string:

$652^{157} = 87$ (mod 2773)
$193^{157} = 104$ (mod 2773)
$196^{157} = 121$ (mod 2773)
...

Which matches our origional data, and so we are happy. Except that one example that works hardly constitutes a complete proof. In order to show that RSA works we must prove that $M = D$ in the general case. Recall

$$C = M^e \ (\text{mod } n) \equiv C = M^e - sn \ (\text{mod } n)$$

$(s \in Z, s \geq 0)$. Now consider substituting this into the decrypted message.

$$D = C^d \ (\text{mod } n)$$

to give

$$D = (M^e - sn)^d \ (\text{mod } n)$$

which (by expanding by the binomial theorem) is

$$D = M^{ed} \ (\text{mod } n)$$

By definition $ed = 1(\text{mod } (p-1)(q-1))$ so, for some non-negative integer k we get

$$M^{ed} = M^{1+k(p-1)(q-1)}$$

$$M^{ed} = M * (M^{p-1})^{k(q-1)}$$

By Fermat's Theorem since p is prime if $a^{p-1} \equiv 1 \ (\text{mod } p)$ and if $M \neq 0 \ (\text{mod } p)$ we get

$$M^{ed} = M * 1^{k(q-1)} \pmod p$$

thus

$$M^{ed} = M \pmod p$$

(the case $M = 0 \pmod p$) is trivial)
We can similary show

$$M^{ed} = M \pmod q$$

So (by a simple lemma not written out here)

$$M^{ed} = M \pmod n$$

which means we now have

$$D = M \pmod n$$

and we are done.

3.3 Digital Signatures

We the first equation we considered when looking at public/private key cryptography was

$$M = S_A(P_A(M))$$

(Recall $S_A()$ and $P_A()$ are inverses)
And we looked at how this enabled private messages to be sent across an insecure channel. Now consider the equation

$$M = P_A(S_A(M))$$

This represents Alice encrypting a message (M) using her secret key. Anybody can now decrypt the message using her public key, but since only Alice can use $S_A()$ they can be sure it was her who encrypted the message. Suppose Alice didn't want to hide the message, merely provide a means for people to verify it was her who sent it. She could send both the origional message, M, and $S_A(M)$. Clearly anybody can read M and anyone who bothers to find Alice's public key can apply P_A to $S_A(M)$. If the result of this matches M they can be reasonably sure it was Alice who sent the message, so by providing $S_A(M)$ with the message Alice has 'signed' it.

Of course transmitting $S_A(M)$ and M is a waste, since we are now sending a message that is twice as long as M.

3.4 Message Digest Functions

To save of the length of data we need to append to a message to 'sign' it we want another one-way function, but this one doesn't want to be one-one. It needs to take a long message M and convert it to a shorter message (it's fingerprint or message digest). This message digest must be generated in such a way that it is nearly impossible to obtain any hint as to the content of the origional message from it's digest. The function must be entirely deterministic .. given the same input it must give the same output.

The technique that is commonly used, and that is used by PGP, is called MD5. The algorithm isn't very exciting mathematically, and if you want to read all about it it is fully explained in RFC1312. Given any length of message it computes a 128 bit message digest. This gives around $3.4 * 10^{38}$ possible message digests .. so it's unlikely we will ever find two messages that result in the same digest even if everyone on the planet spends their entire lives trying! The UNIX program md5sum calculates the message digest of an arbitary message baed on the MD5 algorithm. Here are some examples.

```
[stephen@eddie encrypt]$ echo "There is $1500 in the blue box" | md5sum
11bee12a9acd083ce990667488d99ca2  -
[stephen@eddie encrypt]$ echo "There is $1100 in the blue box" | md5sum
d1b2f167ca464a1dd789f0740c6b356c  -
[stephen@eddie encrypt]$ echo "This is a very different message" | md5sum
2c46d4d4f539754c97cbf5919eee09b4  -
[stephen@eddie encrypt]$ echo "There is $1100 in the blue box." | md5sum
dd7d6842aef63ba104b884d9a53d4ebf  -
[stephen@eddie encrypt]$ echo "There is $1500 in the blue box" | md5sum
11bee12a9acd083ce990667488d99ca2  -
```

We can see that even messages that vary only very slightly result in radically different message digests.

It is easy to see how this is useful in the context of public/private key cryptography. Rather than sign a message by including $S_A(M)$ with it Alice can merely provide $S_A(MD5(M))$. We can still verify it was her who sent it in the same way as we did before since

$$P_A(S_A(MD5(M))) = MD5(M)$$

and since it is conjectured to be computationally infeaible to produce two messages having the same message digest we have lost little by using this technique.

3.5 But how do we know Alice really is Alice

This is a problem, since however good the theories above are they all rely on us being sure that the P_A that we have really is Alice's public key and not someone else's who we have been given on the pretence that it's Alice's.

We can get round this problem by using fingerprints .. calculating the message digest of our public keys. It is a pain to quote public keys over the phone or even write them on bits of paper, but the fingerprint is ideal for this since it is much shorter. If we meet then we can give each other a copy of the fingerprint of our public key. We can then check this fingerprint with the copy of the public key we have. If the fingerprints match we can assume we have the person's real public key and we sign it with our own one. If we obtain someones public key from an untrusted source we can see who has signed this key. If someone we know or trust has signed it then we know we can trust the the public key really belongs to the person that it claims it does.

I use 'gpg' which is a program that understands how PGP works. You can download it from http://www.gnupg.org. If you run windows you might prefer pgp itself. This is available from http://www.pgpl.org.

Here is my public key.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: Key for stephen@earth.li

mQGiBDhD7AARBADqXdex1iEqztnftxEEGoCsLE0DEVy0i/s92x8FChiIr993a767
z8QMNXF6tU479ic5RoLEivxzWiw45mmlAsMSqPcj3CICergi0L4GGpoovvcUnMGq
2UjNfyv6fupFgcqb6DRKa44kX0l3edvM0AzqrbZ7roCX0ICsFkJL1im/TwCg+WiK
okrsB6kJ90pIz1ifajCeG30EAMzi7DMdGuyjpSCMg6YPCdA9I6QAb9c7k32naZyy
0nxlxb9wDWtfr8HJXnmqevZV2o5Cz86LjxkREkVFKRSieg/NY1CCLNuOTzvNX/f0
kwFiWc9rgJjXlX7EuieYLxBPuqfK9+DzcYUOJtuFlJ2TirKwQ/FT8O0+NQPWw/we
5gKCA/wLCxTlU+o2MrAptnu6k3hCg6P5jNSaA0P4N0ODWz0LXry1zpZlNvK/jjTk
KO0CBXNQWdtXJNNybiepr0wuja2lqZ5R1Lgpu8vX42bbYE1EhwL9nM7Eb+VQL/fG
hjVyj4bMEDbjH1s5jLTZpM1k4JVwv5dh0uKzCJDu6IuySuXT37QgU3RlcGhlbiBX
aGl0ZSA8c3RlcGhlbkBlYXJ0aC5saT6IVQQTEQIAFQUC0EPsAAMLCgMDFQMCAxYC
AQIXgAAKCRA7Ok8Kx55ban78AKCL0oc5kxcECZtSyTfHSLeZdTGJuwCguhLkEJn/
PO6GwvUQj0uAwAcrpXmJARUDBRA4Q/ktHUtPijd4cyMBAYpaCACPrhkGv2fgVSmm
9HEPfgeaR9CTLPVFqK7O+309H2oR/BCNRUc4HapbKtlxfdofYx630u/736sNGH2G
wstB2oGyb0ac7taJHkia0lYhJwgpZQiWMwsUcdi2iKOg7DWVV42Leh7elVcr18Ef
gGU/mOS5m1XQvKhrIYfStXdxYQRDAoHrZNq2E5E/4ZHlzlI6mlI7GITA0uhLVmJn
HBN5IWMxawla/3wV1r4qrAvN+cWnoIiaUFTRHjAMfkSXaExwsvYILdVW6VwALWkq
KJFUCvc78Z4/flQImQZGUsTZC+ZeV6yJZAret0Rbd1eQxwZnxpB7mQ2BgJWjIgR9
8DjuKLqjuQINBDhD7WQQCAD4nLAv9Lo+uxKgkiNY2q0ge56xPWMkX+YbCpStLxsI
j+B/3ODRXohVOjc6O8ic78mhHsT6eajCP0hLJKgjVECE1QKKlB2Q4J1vUwZoOw+t
FGqrrQhdHAh+V0mIkwCFufJyf7SOu3aiikMF9iQNmxJW9Moj0lZJHauf+VOU/C3q
XRxm9fJygmSeHJ/NH5xrtYTshRyDEM1cYqcUvdHe1Ji6GiUE/QeENQ9a8WO9IOPV
```

```
OV2NHlbDqX+qLYO9chNk3hXJ/XBqFmZ2VU1XjmBfrTDZ/Zkr8szDqqeKhvDvOgc3
B7Kg/GE/2KYgTaF8lE9/hF8tZOl/rvKmox+kcESYbuL3AAMFCACt8Fv/J1sYYihP
ouVJfR5osmPBOn2hkohRRNUNlhBwQWgaYdUozGLrEj6bDpIrau1lp7qjLZmZeeNN
fNedMnhNkzA5FVrTtIvTrADngfXF1kbd1k6XVleRCH056/fLaNDYEViTDOSdgetm
z06+msuR3GpogRJYj4tTJ0C7fZ4Mf9H9YDbjUUhziUUdUMnHDf8tHSrs0d6JPBhW
tINPkN8YronHGICBnnE9a+lw2PCmVhdO6meSvw6O6P4CnscRcN73nk9E37y/OxEk
DCIQQwckAvjOYUVRhCGyRMcNj8absIqVPwu8Zf/YTgwZvdbF5K6ggY6Z0uxcKjuy
VAujKUzZiEYEGBECAAYFAjhD7WQACgkQOzpPCseeW2qkdQCfU5ClIDJAMViROTwt
vR+X8xlH1oQAnAumDhSY1WIKWBC34ztjapTCLEZy
=5WR3
-----END PGP PUBLIC KEY BLOCK-----
```

Now suppose you've just downloaded this from a public server .. you wouldn't know whether to trust it or not.

```
[stephen@eddie encrypt]$ gpg --list-sigs stephen@earth.li
pub  1024D/C79E5B6A 1999-11-30 Stephen White <stephen@earth.li>
sig        C79E5B6A 1999-11-30  Stephen White <stephen@earth.li>
sig        37787323 1999-11-30  Stephen White <swhite@ox.compsoc.net>
sig        80C83E8E 1999-11-30  Ian Lynagh (wibble) <ian@lynagh.demon.co.uk>
```

From this you can see that Ian Lynagh has signed my key. If you are happy that this is his signature (maybe because you've previously checked it's fingerprint) then you would know my key was valid.

Alternatively you could ask me directly for my key's fingerprint.

It is 622E 7813 45A9 C65B 4C41 DFD8 3B3A 4F0A C79E 5B6A

Now having written this down and gone back to your computer you type

```
[stephen@eddie encrypt]$ gpg --fingerprint stephen@earth.li
pub  1024D/C79E5B6A 1999-11-30 Stephen White <stephen@earth.li>
     Key fingerprint = 622E 7813 45A9 C65B 4C41  DFD8 3B3A 4F0A C79E 5B6A
sub  2048g/1B77E62B 1999-11-30
```

The fingerprints match .. so you know that the public keys match and anything that has been signed by that key you know really came from me.

4 References

- 'Introduction to Algorithms' by Thomas H. Cormen, Charles E. Leiserson and Ronald L. Riverest

  Excelent on the maths and proofs behind public-private key cryptography and the algorthims involved in working with [large] prime numbers.

- 'Codes and Cryptography' by Dominic Welsh.

  More algorithms and proofs.

- 'PGP: Pretty Good Privacy' by Simson Garfinkel - O'Reilly and Associates, Inc

- RFC1321

  Describes the MD5 algorithm.

- 'GnuPG'

  A free software implementation of PGP.