Encryption and Data Security

Introduction

A wireless channel provides many benefits, mobility being one of the greatest, but it also has some major drawbacks. One is a complete lack of physical security. With a wired channel we can secure communications that go over the channel by securing the physical medium. For example, your (wired) phone conversations are fairly secure because someone who climbs a telephone pole to tap into your phone line is likely to get noticed and arrested. On the other hand, if you talk over an FM-radio link (as you do with an analog cellular phone), anyone nearby with an appropriate FM receiver can listen in without you knowing about it. Although in the U.S. it is illegal to snoop on cellular phone channels, it is nearly impossible to tell if someone is doing so because the channel is "on the air" for everyone to access.

With a digital communication system we can employ *encryption*. The problem we face is how to encode a sequence of data bits so that unauthorized snoopers cannot decode them. The original data is called the *plaintext* while the encrypted data is called the *ciphertext*.

One-Time Pad

The only encryption technique know to be perfectly secure is the so-called *one-time pad.* The basic idea is that the TX adds the plaintext to another message, the key (the "pad"), which has a length as long as the plaintext to create the ciphertext. The RX subtracts the key from the ciphertext to recover the plaintext. Provided the key is truly random, is only used once, and, of course, no one other than the TX and RX have access to it, this technique provides perfect security.

For example, let's say we wish to send messages consisting of the 26 letters and spaces. We can number the characters as follows:

_	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Let's say our plaintext is `secret` and our key is `aziwhq`. We add the plaintext and key character by character to get the ciphertext. The addition is performed "modulo 27," meaning that if we get a sum of 27 or more we subtract 27 so as to end up in the range 0 to 26. For the first character we have `s+a` which corresponds to `19+1=20` which in turn is `t`. Likewise, `e+z` corresponds to `5+26=31` and `31-27=4` which is `d`. In this manner we find the ciphertext `tdlnmj`.

```
 plaintext: secret
       key: aziwhq
ciphertext: tdlnmj
```

The receiver takes the ciphertext `tdlnmj` and subtracts the pad `aziwhq` to get the plaintext `secret`. Again, the arithmetic is done modulo 27, so if we get a difference less than 0, we add 27 to get back into the interval of 0 to 26. For example, `t-a` is `20-1=19` which is `s`, `d-z` is `4-26=-22` and `-22+27=5` which is `e`, and so on.

This method provides perfect security because for any ciphertext there is a key that will generate any plaintext (of the corresponding length). For example, applying the key `akrydf` to our same ciphertext recovers the message `stupid`. Since the key is completely random, the plaintext is just as likely to be `stupid` as to be `secret`, or indeed, any six-letter message.

```
ciphertext: tdlnmj
       key: akrydf
 plaintext: stupid
```

Because of the perfect security it provides, the one-time pad is the method of choice for highly sensitive government and military information.

However, there are some big drawbacks to the one-time pad. The key must be truly random. For example, you cannot use a computer algorithm, such as a "random number" function, to generate the key. Second, you can only use the key once. Third, the key has to be as long as the message you are encoding. The perfect security comes from the fact that there are many possible keys as there are messages.

Substitution Ciphers: DES

A generally more practical approach is the *substitution cipher*. The idea is very simple. List all your symbols in a column. In the next column list some permutation of these symbols. This is your *lookup table.* To encrypt, find the plaintext symbol you want to send in the first column and substitute the ciphertext symbol in the second column. To undo this find the ciphertext symbol in the second column and substitute the plaintext symbol in the first column. For example, if your symbols are letters then you might have something like

a	b
b	d
c	a
d	e
e	c
etc	etc

and the plaintext `bade` would become the ciphertext `dbec`. If you have N symbols, then there are $N!$ possible permutation tables of this sort. Applied in such a simple manner, this would not be very effective because you could use the know frequencies of certain letters to figure out the substitutions. In practice much larger groups of symbols are used and some sort of pre-randomization of the message, for instance through entropy coding, can be used.

The *Data Encryption Standard* is a digital substitution cipher developed for the U.S. government that operates on 64-bit blocks of data. A 64-bit key K generates the lookup table. Only 56 bits are independently chosen, the remaining bits are error-correcting parity bits. The encryption and decryption operations can be represented as

$$\begin{aligned} O &= E_K(I) \\ I &= D_K(O) \end{aligned} \tag{32.1}$$

The heart of the DES algorithm, of course, is the way in which the lookup table is generated from the key.

The security of DES comes from two facts. First, there is no known algorithm for breaking it other than a brute force search over all possible keys. Second, since $2^{56} \approx 7 \cdot 10^{16}$, it should be practically impossible to implement a brute force search. Actually, while this was true in the 1970's, when DES was developed, state-of-the-art (circa 2000) parallel computers have the capability of breaking DES by brute force in about 24 hours. As an interim fix, *triple DES* can be implemented. Typically this involves choosing two 64-bit keys K_1 and K_2, which is the equivalent of a single 128-bit key. We then implement a look up table as a cascade of DES operation. The encryption and decryption operations are

$$\begin{aligned} O &= E_{K_1}\left(D_{K_2}\left[E_{K_1}(I)\right]\right) \\ I &= E_{K_1}\left(E_{K_2}\left[D_{K_1}(O)\right]\right) \end{aligned} \tag{32.2}$$

Since the extra 56 free bits imply a factor of about $7 \cdot 10^{16}$ times as many possible look up tables.

In light of the shortcomings of DES, a new *Advanced Encryption Standard* (AES) has been devised. This operates on 128-bit blocks of data and allows 128-, 192-, or 256-bit keys.

Public Key Encryption

Substitution ciphers such as DES are very powerful and convenient to implement, but they still retain the problem of requiring secure delivery of the key. This requires a secure communication channel, but how do you set up that secure channel in the first place? Public key encryption is an ingenious way to get around this problem.

Public key encryption exploits the concept of a one-way function. This is a function that is easy to compute but very difficulty to invert. As a somewhat trivial example, with pencil and paper it is relatively easy to square a number, but difficult to calculate square roots.

Exponentiation is central to public key algorithms. It is relatively easy to calculate p^q where p and q are two integers. Even for huge integers this can be accomplished efficiently by the *square-and-multiply* technique. q is expressed as a binary number, i.e., as a sum of powers of 2. For example, 9 is binary 1001, that is, $9 = 8 + 1 = 2^3 + 1$. Then $p^9 = p^1 \cdot p^{2^3}$. Since $p^{2^3} = \left[\left[p^2\right]^2\right]^2$ this involves only taking squares of numbers and multiplying. The technique also works when the operations are performed modulo some integer.

Diffie-Hellmen

The one-way function in the Diffie-Hellmen algorithm is exponentiation modulo some integer. As we've seen exponentiation is relatively simple. The inverse – taking a logarithm modulo some integer – is very difficult.

- Public integers α and $q > \alpha$ are chosen. α must be a "primitive root" of q which implies that for any $0 \le i \le q-1$ there is a unique $0 \le b \le q-1$ such that $b = \alpha^i \bmod q$. i is the *discrete logarithm* of b.

- User *k* selects a *private key* x_k and calculates a *public key* $y_k = \alpha^{x_k} \bmod q$.

Then User 1 and User 2 can set up a secure channel as follows.

- User 1 looks up User 2's public key to calculate $K = y_2^{x_1} \bmod q$.
- User 2 looks up User 1's public key to calculate the same key $K = y_1^{x_2} \bmod q$.

They are guaranteed to get the same key *K* because $\left(\alpha^{x_1} \bmod q\right)^{x_2} \bmod q = \left(\alpha^{x_2} \bmod q\right)^{x_1} \bmod q$. This is the discrete version of $\left(\alpha^{x_1}\right)^{x_2} = \left(\alpha^{x_2}\right)^{x_1}$. A third user cannot obtain *K* because he does not have access to the private keys. To break the cipher you would have to invert $y_k = \alpha^{x_k} \bmod q$. There is no known algorithm for doing this other than a brute force search. If α and *q* are really large, then this is practically impossible.

RSA

The RSA public key algorithm was developed by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1977. The one-way function here is the representation of an integer by prime factors. Each user generates public and private keys as follows:

- Choose two large prime numbers *p* and *q* known only to yourself.
- Calculate $n = pq$.
- Choose a number $e < n$ such that *e* and $(p-1)(q-1)$ have no common factors.
- Find a number *d* such that $\dfrac{ed-1}{(p-1)(q-1)}$ is an integer.
- Your *public key* is (n, e). Your *private key* is (n, d).

Then User 1 can send a secure message to User 2 using the following steps

- To send a message *m*, User 1 calculates the cipher $c = m^{e_2} \bmod n_2$ where (n_2, e_2) is User 2's public key.
- The cipher *c* is sent over a public channel.
- User 2 calculates $m = c^{d_2} \bmod n$ where (n_2, d_2) is his private key.

The relationship between *e* and *d* insures that $m = \left(m^{e_2} \bmod n_2\right)^{d_2} \bmod n_2$. In general for $d_3 \neq d_2$, $\left(m^{e_2} \bmod n\right)^{d_3} \bmod n \neq m$. To break the cipher you'd need to factor $n_2 = p_2 q_2$, so the security of RSA comes from the fact that there are no known algorithms for prime factorization, other than a brute force search. By choose *really large* numbers *p* and *q* this can be made practically impossible.

References

1. http://www.rsasecurity.com/ (2002-6-17)

2. http://www.odci.gov/csi/books/venona/venona.htm (2002-7-3)
3. http://csrc.nist.gov/cryptval/des.htm