Computerised One Time Pads

From Telecomix Crypto Munitions Bureau

The **one time pad** cipher is arguably one of the most simple ciphers. The cipher is completely unbreakable by means of logics. Computers can not crack it. The only known method to find the correct keys is through *torture*. As torture is one form of cryptanalysis (its called "rubber hose cryptanalysis") we could if we want say that this cipher is in fact breakable.

Notice that remote key exchange over an unsecure network is not possible. ONLY exchange cipherkeys directly with the person you wish to communicate with.

If you ever plan on using the OTP cipher, you really need to make sure that you completely understands it. It is simple to make mistakes!

Contents

One Time Pad (OTP)

"In cryptography, the one-time pad (OTP) is a type of encryption, which has been proven to be impossible to crack if used correctly." - wikipedia so it must be true!

One time pad is likely the most simple form of cryptoalgoritm. A key is generated and shared between two peers that wish to communicate with each other. The key must be equally long as the message that is sent. The key must also be transmitted *in person* or via a sneakernet of completely trusted couriers. (If you send the key to your peer with another crypto, OTP will be just as secure as the crypto used to transmit the key. Do not rely on other ciphers, the chain is just as secure as its weakest link. *Do not transmit the keys with any other means than sneakernet.*)

Known attacks against OTP

- Rubber hose cryptoanalysis

- Countermeasure: Usage of hidden channels to inform the receiving agent that you are under attack. A hidden channel can easily be constructed by carefully avoiding a cleartext symbol, such as "b". Whenever a message is found to contain the letter "b" after decryption, it should then be assumed that the sending agent is compromized. To save your friend from further subjection to the rubber hose, it is advised that you pretend that no hidden channel exist and play along with the capturers while you think of a plan to rescue your friend. More elaborate schemes can of course be constructed.

Software implementing OTP

- Plugin for Pidgin Pidgin-Paranoia

TCMB manuals for manual and semi-manual OTP-cryptography

Everything below is probably only interesting for computer nerds :-)

Generating keys for the OTP cipher

The following script (called `otpgrb`) can be used to generate keys.

```
#!/bin/sh
dd bs=1 count=$1 if=/dev/random of=$2
```

Usage:

otpgrb 512 my-key

will generate a 512 byte long random number and put it in the file my-key. Be sure to use a computer with an encrypted hard drive, or use a RAM file system, if you suspect that your computer could ever end up in the hands of others. If you are using OpenBSD you need to exchange /dev/random with /dev/srandom.

Encrypting with the OTP cipher

`xor` is a C program that xors the input from stdin with a given file, byte after byte, and presents the output to stdout. The syntax is: "`./xor key`". If you do not want strange stuff written to your screen, you could type something like "`./xor key < cleartext > ciphertext`".

```
#include <stdio.h>

int main(int argv, char *argc[]){
        FILE *seed;
        int c, d;
```

```
        seed = fopen(argc[1], "r");

        c = getchar();
        d = fgetc(seed);

        while( c != -1 && d != -1 ){
                putchar((char) c^d);
                d = fgetc(seed);
                c = getchar();
        }
}
```

To compile it: "gcc xor.c -o xor".

Example usage

Generating a key:

```
# ./otpgrb 128 key
128+0 records in
128+0 records out
128 bytes (128 B) copied, 0.00399996 s, 32.0 kB/s
# hd key
00000000  39 ba 42 2f 6d 05 f0 92  a7 6c 52 7b 95 eb 93 ac
|9.B/m....lR{....|
00000010  63 93 10 e8 d9 a1 6c 02  6c 76 40 7c 82 40 9c 7f
|c.....l.lv@|.@..|
00000020  3b a2 b0 25 32 aa 20 5c  fa c5 8e 36 c5 d9 b8 22  |;..%2. \...6..."|
00000030  a9 1c 36 28 40 89 f9 bb  d1 8c a9 06 eb 88 50 b1
|..6(@.........P.|
00000040  cf 88 4d 64 4d 85 b9 7f  1a c6 72 98 29 02 2c 3d
|..MdM.....r.).,=|
00000050  1c e5 46 c1 42 dc c8 21  60 12 f3 e5 e7 bb 0a d7
|..F.B..!`.......|
00000060  3f ca 6c c8 39 06 ce de  f8 b9 a8 27 92 eb c6 dc
|?.l.9......'....|
00000070  68 f2 c5 c0 44 a4 48 8f  f9 38 ce bc 84 05 73 0d
|h...D.H..8....s.|
00000080
#
```

After you have generated your key, you need to give it to the person whom which you wish to communicate with in the future. Put the key on an USB stick and deliver it to your friend.

To encrypt messages, you need to `xor` the plaintext with the OTP key:

```
# echo Lets kill the king\! The revolution is here\! > cleartext
# ./xor key < cleartext > ciphertext
# shred cleartext
# rm cleartext
# history -c
# hd ciphertext
00000000  75 df 36 5c 4d 6e 99 fe  cb 4c 26 13 f0 cb f8 c5
|u.6\Mn...L&.....|
```

```
00000010  0d f4 31 c8 8d c9 09 22  1e 13 36 13 ee 35 e8 16
|..1...."..6..5..|
00000020  54 cc 90 4c 41 8a 48 39  88 a0 af 3c
|T..LA.H9...<|
0000002c
#
```

Once your friend has received the key and the ciphertext, it is possible to decipher the message. To decrypt the ciphertext:

```
# ./xor key < ciphertext > cleartext
# cat cleartext
Lets kill the king! The revolution is here!
#
```

Once a key has been used, it should **NEVER** again be used. Both you and your friend should throw it away. If you reuse the key, the cipher is *extremely* vulnerable. However, if you never reuse the keys, it is not possible to crack the cipher with any known methods ;)

Performing OTP cryptography by hand

If the field agents memorize the coding scheme and keep numbered records of OTP-keys it is possible to perform the computations needed for perfectly secure communication *by hand*.

The cipher is very simple. Just generate a random sequence of [a-z][1-6]-symbols (32 symbols) and save them as your key. When you encrypt a message, just shift the clear text left equal to the position in the alphabet of the key (a = 0 shifts left, b = 1 shifts left, etc.). Do this symbol by symbol until you reach the end of the message. Decryption works exactly the same, but in reverse.

Generate randomness by collecting binary sequences from /dev/random (linux) or /dev/srandom (OpenBSD). Use the `otpgrb` script to do this. Then `xor` the random blocks that have been generated (possibly at different machines at different times) to generate a new binary random sequence. This sequence could then be fed to `alphabetize` (see below) to create an standard human readable form of the key. If the field agent writes this key down, it is possible to perform the rest of the calculations with only a brain, a paper and a pen.

Description of the code used

This table is very useful if you want to use the encryption scheme without a computer. TCMB field agents should have printouts of it in their cipherbooks, along with the numbered OTP-keys.

```
  abcdefghijklmnopqrstuvxyz123456

a abcdefghijklmnopqrstuvxyz123456
b bcdefghijklmnopqrstuvxyz123456a
c cdefghijklmnopqrstuvxyz123456ab
d defghijklmnopqrstuvxyz123456abc
```

```
e efghijklmnopqrstuvxyz123456abcd
f fghijklmnopqrstuvxyz123456abcde
g ghijklmnopqrstuvxyz123456abcdef
h hijklmnopqrstuvxyz123456abcdefg
i ijklmnopqrstuvxyz123456abcdefgh
j jklmnopqrstuvxyz123456abcdefghi
k klmnopqrstuvxyz123456abcdefghij
l lmnopqrstuvxyz123456abcdefghijk
m mnopqrstuvxyz123456abcdefghijkl
n nopqrstuvxyz123456abcdefghijklm
o opqrstuvxyz123456abcdefghijklmn
p pqrstuvxyz123456abcdefghijklmno
q qrstuvxyz123456abcdefghijklmnop
r rstuvxyz123456abcdefghijklmnopq
s stuvxyz123456abcdefghijklmnopqr
t tuvxyz123456abcdefghijklmnopqrs
u uvxyz123456abcdefghijklmnopqrst
v vxyz123456abcdefghijklmnopqrstu
x xyz123456abcdefghijklmnopqrstuv
y yz123456abcdefghijklmnopqrstuvx
z z123456abcdefghijklmnopqrstuvxy
1 123456abcdefghijklmnopqrstuvxyz
2 23456abcdefghijklmnopqrstuvxyz1
3 3456abcdefghijklmnopqrstuvxyz12
4 456abcdefghijklmnopqrstuvxyz123
5 56abcdefghijklmnopqrstuvxyz1234
6 6abcdefghijklmnopqrstuvxyz12345
```

It is a simple tabula recta for the 32 symbol long alphabet we use. The number 32 was chosen because it is a nice exponent of 2 and because it is probably the last such exponent that is easily usable by humans. A 64 symbol long alphabet would have generated a 4 times larger table; 2 times larger in both horizontal and vertical length. On the other hand, 16 symbols would add complexity because there are obviously more letters in most human alphabets.

The convention is to replace the symbols 1-6 with whatever you think is more needed. Such as space, lol :)

The left-most column is the **key**. For each symbol that you are going to encrypt or decrypt, find the random key in this column. The top-most row is the **clear text**. When you encrypt, the clear text is known. When you decrypt, the clear text is what should appear when you decrypt a **crypto text.** All symbols in the table are used to garble the clear text into crypto text, or the other way around, using the key.

Encryption: Find the key in the left-most column. Then find the clear text-symbol in the top-most row. Go straight down from the clear text-symbol to the row where the key is. The letter you find here is the crypto text-symbol. Write it down and continue with the next symbol you want to encrypt. **Example:** You want to encrypt the symbol F with the key 2. A is then the crypto text-symbol.

Decryption: Find the key in the left-most column. Go straight to the right from this point to the crypto text-symbol. When you find it, go straight up. When you reach the "clear text"-row at the very top, you will find the clear text symbol. **Example:** The key is D, the ciphersymbol is K. Then the clear text-symbol is H.

Alphabetizer

Recodes binary input to our alphabet and outputs the ASCII.

Usage:

- **Syntax**: alphabetizer [human]
  - if you supply "human" as argument 1, it will write the output in nice 16-symbol wide rows and give an ending newline.
- **XF-OTP-STD0 compatible**: It throws away the 3 most significant bits in each byte during recoding. At output it maps the alphabet to the ASCII letters [a-z] and [1-6]. Reads from stdin until EOF and writes to stdout. (Notice that XF-OTP-STD0 is a deprecated coding scheme. This does however not weaken the security of the cipher.)

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

char alphabetize(int byte)
{
   /*
       The alphabet we are using is 32 characters long:
       abcdefghijklmnopqrstuvwxyz123456
                                                        */

   if( byte >= 32 )
   {
      fprintf(stderr, "Internal program error.\n");
      exit(-1);
   }

   if( byte < 26 )
   {
      return( (char) byte + 'a' );
   }
   else if( byte == 26 ) return('1');
   else if( byte == 27 ) return('2');
   else if( byte == 28 ) return('3');
   else if( byte == 29 ) return('4');
   else if( byte == 30 ) return('5');
   else if( byte == 31 ) return('6');

}

int main(int argv, char *argc[]){
   int byte;
   int i = 0;
   bool human = false;

   if( argv > 1 )
   {
      if( argv > 2 )
      {
         fprintf(stderr, "Too many arguments.\n");
         exit(1);
      }
      if( !strcmp( argc[1], "human" ) )
```

```
      {
         fprintf(stderr, "Using human readable format.\n");
         human = true;
      }
      else
      {
         fprintf(stderr, "Unrecognized argument.\n");
         exit(1);
      }
   }

   byte = getchar();

   while( byte != -1 )
   {

      if( human && i == 16 )
      {
         /* we want 16 letter long rows if we are humans */
         i=0;
         putchar('\n');
      }

      byte &= 0x1F;
      putchar( alphabetize( byte ) );
      i++;

      byte = getchar();
   }
   if( human ) putchar('\n');
}
```

ONE-TIME IMAGE

One-Time Image is a program written in Java to encrypt images using the principle of the one-time pad to create a pair of black and white images. Each appears as a 'snow' of black and white pixels, and no information can be extracted from either image on its own - unlike most cryptography, it is not merely 'very difficult' to extract information: an implementation of the one-time image principle with true random numbers is *perfectly* secure and unbreakable (in practise this program uses the default Java random number generator, and so isn't absolutely secure - information is provided in the source code on how to add your own random number generator if you want to make it so).

However, despite the fact that no information can be retrieved from either image on its own, if both are printed onto transparencies and one is laid directly over the other, the original image will immediately and clearly appear. One-Time Image is free and open-source - you can get the program and source code here.

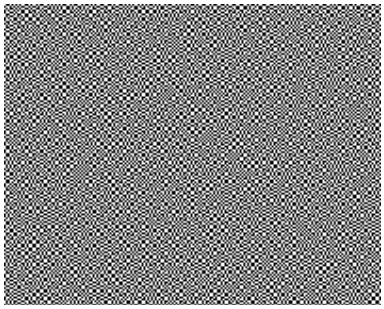
HOW DOES IT WORK?

The principle of the one-time pad was developed during World War One, though it was 25 years before a mathematician proved it was perfectly secure, as opposed to

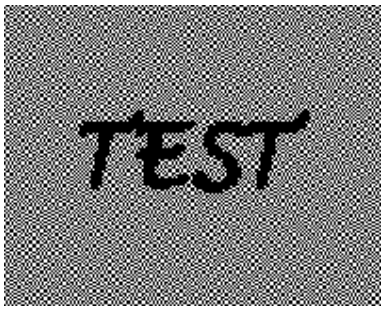
merely prohibitously difficult to break. It is a very simple substitution cypher, but with the twist that the key is the same length as the message. As a result each letter has its own unique (and random) rotation, making the encoded message proof against any sort of analysis without the key. The final result is a pair of random-looking message (one the encrypted message, one the key used to encrypt it) of the same length, either of which cannot be broken or analysed in any way on its own, but which can easily be decoded once the two are brought together.

One-Time Image extends the principle to that of images. First, the source image is converted to black and white (not greyscale, true black and white with pixels of only these two different colours).

Secondly a key image is generated, of the same dimensions, where each pixel is randomly set to white or black. Third, the original image is encrypted using this key - if the pixel in the key is white then the corresponding pixel in the original image is used in the encrypted image, whereas if the key pixel is black then the corresponding pixel in the original image is flipped (black to white, white to black) for the encrypted image. The result is two images of apparently random black and white pixels.



Finally, each image is then doubled in size - each pixel becomes a 2x2 square of pixels. Black pixels have black pixels in the top-left and bottom-right corners while the other two pixels are while, while a white pixel in the original image produces the opposite 2x2 square. These enlarged images are the final, encrypted ones - they have the appearance of random static, or snow, and neither one can be decrypted on its own no matter how powerful the computer or clever the analyst..



The trick is that, when printed onto transparencies and one is laid over the other (the order is irrelevant), the original image is suddenly revealed! This is because a black pixel in the original image produced pixels of different colour in the key and encrypted images (one black, one white). Since these black and white pixels became 2x2 squares with two black and two white pixels, when overlaid all four pixels in the square become black. However, a white pixel in the original produced pixels of matching colour in the key and encrypted images (both black, or both white). Hence the 2x2 squares in the final images are identical, and when overlaid half the pixels remain white. Hence, when you look at the image from anything but very short range, these 2x2 squares look grey while other look black.

If you don't have a printer and transparencies to hand, you can demonstrate it for yourself using Photoshop or some other image-manipulation program - take one of the output images and import/paste the other on top of it as a new layer and you'll see your original image.

THE PROGRAM

You can download the program with source code here. It is released under the GPL, so you're free to use, alter and redistribute it. The interface is pretty basic: Java's not my favorite language, though I can get by with it (I used it in this case for its image handling capabilities). A readme file with various information is included, along wtih makefiles, directions to run it, etc.

What's it useful for? In the world of serious cryptography - not much. In practise it makes a lot more sense to use the same principles to produce a pair of USB keys rather than messing around with transparent sheets of printed static. And even then the one-time pad is mostly a curiosity: its lack of authentication, need to transport the message securely, need to dispose of the keys securely after use and other problems means its usage is limited.

It is, however, a useful tool for demonstrating the principle behind the one-time pad and cryptographic principles in general, since the decoding is automatic, instantaneous and very visual. The fact that transparencies are involved also makes it perfectly suited for use with overhead projectors if being formally taught. It's also fun to just play around with - I find it fascinating, the way two nonsensical images produce a meaningful one when overlayed (but maybe that's just me). You could also use it for activities like orienteering, by giving one transparency to a team and leaving the other at the point they need to locate (it'd be like a high-tech equivalent of those hole-punches they normally use).