

Ghiribizzo's Cracking Tutorial

Killing off XOR Encryption for Good

Introduction to XOR decrypting

In this tutorial, I will go through a method of decrypting XOR encrypted messages where only the ciphertext is known. We will use methods to find the length of the key and then finally to decrypt the whole lot using only a calculator.

PGP and Signed Tutorials

My tutorials and programs should be signed electronically using PGP. PGP 5 supports DSS/Diffie-Hellman keys. These keys are not supported by previous versions of PGP.

You should check the signature to make sure that the tutorial and especially its program files have not been tampered with. All cracks, tutorials and zip files I release will be signed. This will prevent tampering and will hopefully reduce the chances of viral infection.

My signature will also be the only way you can identify me as my email address will often change.

My Web Site: <http://Ghiribizzo.home.ml.org>
My Email: Ghiribizzo@geocities.com
My Backup Email: Ghiribizzo@hotmail.com

This document is Copyright © 1997 by Ghiribizzo. This document may be distributed non-commercially, provided that it is not modified in any way (including change of format). This publication may not be sold or packaged, in whole or in part, as a service, or with a product for sale in any form without the prior written permission of the author. This document is presented with no warranties or guarantees of any kind including fitness for any particular purpose. If you use the information contained herein, you do so at your own risk.

"Given this fact about NPP, I consider [it] effectively uncrackable without the dongle. I hope that someone will take this as a challenge and try to figure out a way of cracking the envelope protection on a program that doesn't link hasp32b.obj..."

- Quine (HASP essay)

For those of you on the mailing list I alluded to a method of breaking XOR encryption given only ciphertext. Here I will outline how to do it by means of a worked example. The problem can be divided into 2 basic steps:

- 1) Finding the length of the key
- 2) Decrypting the message

Finding the length of the key

This isn't really a problem for crackers as the program will usually tell us the length of the key. But if it doesn't (perhaps using the method I suggested on the mailing list) then there is another way:

Counting Coincidences

XOR the ciphertext with itself displaced by different numbers of bytes. Then count the number of bytes that are equal. Now if the displacement is a multiple of the key length then around over 6 percent of the bytes will be equal. If not then less than 0.4 percent will be equal (if using ASCII text and random key – other plaintext will have a different percentage).

You should start from 1 byte shift and increase each time. You'll know when you've found the correct key length. You can double check by shifting by twice the key length and observing the increase. The smallest displacement which causes the jump in coincidences is the key length.

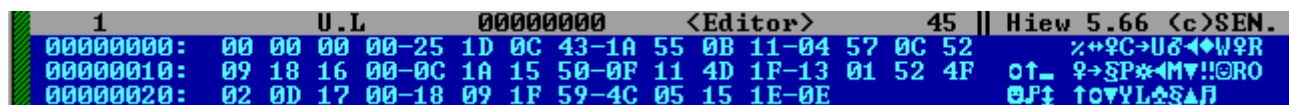
Decrypting the message

Shift the ciphertext by the key length and XOR it against itself. What you are left with is plaintext XORed with itself displaced by the length of the key. English has 1.3 bits of real information per byte so there's plenty of redundancy to determine a unique decryption.

The above method has been taken from Bruce Schneier's *Applied Cryptography* – I recommend it to all crackers.

Well, when I first read the method I just stared at it with a blank expression. However, the method isn't hard and you could work out how to do it by yourself, but I will give a short lesson. Finding the length of the key is quite straightforward, you should write your own utility to give the results, so I'll just talk you through the second part.

Firstly, create the following file using HIEW or your favourite hex editor:



```
1          U.L          00000000          <Editor>          45 || Hiew 5.66 (c)SEN.
00000000: 00 00 00 00-25 1D 0C 43-1A 55 0B 11-04 57 0C 52          %++9C->U8-4W9R
00000010: 09 18 16 00-0C 1A 15 50-0F 11 4D 1F-13 01 52 4F          0t_ 9->SP*4Mv!!0RO
00000020: 02 0D 17 00-18 09 1F 59-4C 05 15 1E-0E          0F± 10VYL4SΔJ
```

This is the plaintext XORed with itself (assume we have already discovered that the key is 4 bytes long). The first 4 bytes have been zeroed as I didn't bother to 'wrap' the ciphertext around when XORing (the last 4 bytes have also been truncated). You should ignore these 4 bytes.

If it seems impossible to get anything from this, don't worry – it is quite easy. The first thing I did was to think of an easy way to do it. I thought that the space character (20h) would appear quite often given that it is English text and that XORing with a standard letter would 'toggle' it's state of capitalisation. So I looked through for normal English letters (probably capitalised).

The first one appears at 7h ('C') scanning along by 4 bytes you see a lot of other capital letters. This looks good. I replace the 'C' character with a 'c' character (XOR by 20h). This gives me 'c' (63h), I then XOR 63h with the byte 4 bytes further on (11h), this gives 'r' (72h) – another letter, this is looking good:

```

1          U.L          0000000C <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-25 1D 0C 63-1A 55 0B 72-04 57 0C 52      %++c->Udr*WQR
00000010: 09 18 16 00-0C 1A 15 50-0F 11 4D 1F-13 01 52 4F      0t_ ?->SP*!Mv!!@RO
00000020: 02 0D 17 00-18 09 1F 59-4C 05 15 1E-0E      0F‡ †oVYLΔSgfl

```

Keep on going, all of the bytes retrieved are either spaces or letters which is a very good sign:

```

1          U.L          0000002C <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-25 1D 0C 63-1A 55 0B 72-04 57 0C 20      %++c->Udr*WQ
00000010: 09 18 16 20-0C 1A 15 70-0F 11 4D 6F-13 01 52 20      0t_ ?->Sp*!Mo!!@R
00000020: 02 0D 17 20-18 09 1F 79-4C 05 15 67-0E      0F‡ †oVyLΔSgfl

```

We could have simply used a partial key to XOR each one, but doing it by hand here gives a better 'feel' as to how to do it. Let's try getting another one. There's a 'U' two bytes after our first letter. Let's try the same thing:

```

1          U.L          0000001E <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-25 1D 0C 63-1A 75 0B 72-04 22 0C 20      %++c->udr*WQ
00000010: 09 3A 16 20-0C 20 15 70-0F 31 4D 6F-13 30 52 20      0:_ ? Sp*dMo!!@R
00000020: 02 0D 17 20-18 09 1F 79-4C 05 15 67-0E      0F‡ †oVyLΔSgfl

```

Looks like our initial guess that it was a 'u' is wrong. Remember: we're assuming capital letters come from letter XORed with space. Perhaps the 'U' should be a space:

```

1          U.L          0000002A <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-25 1D 0C 63-1A 20 0B 72-04 77 0C 20      %++c-> dr*WQ
00000010: 09 6F 16 20-0C 75 15 70-0F 64 4D 6F-13 65 52 20      0o_ ?uSp*dMo!!eR
00000020: 02 68 17 20-18 61 1F 79-4C 64 15 67-0E      0h‡ †aVyLdSgfl

```

Yes, this looks better don't forget to work backwards as well to get the byte at 5h. This is easily done, using HIEW note down the byte after it (20h) press backspace to undo the 20h replacement. This will reveal that the byte was previously 55h. Now 55h XOR 20h is 75h (you'll get good at XORing after a few hours of this! ☺) Now we have:

```

1          U.L          00000006 <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-25 75 0C 63-1A 20 0B 72-04 77 0C 20      %u?c-> dr*WQ
00000010: 09 6F 16 20-0C 75 15 70-0F 64 4D 6F-13 65 52 20      0o_ ?uSp*dMo!!eR
00000020: 02 68 17 20-18 61 1F 79-4C 64 15 67-0E      0h‡ †aVyLdSgfl

```

If you can't see what it is yet carry on:

```

1          U.L          0000001F <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-25 75 69 63-1A 20 62 72-04 77 6E 20      %uic-> br*WQ
00000010: 09 6F 78 20-0C 75 6D 70-0F 64 20 6F-13 65 72 20      0ox ?ump*d o!ler
00000020: 02 68 65 20-18 61 7A 79-4C 64 6F 67-0E      0he †azyLdogfl

```

Then finally:

```

1          U.L          0000002D <Editor>          45 || Hiew 5.66 <c>SEN.
00000000: 00 00 00 00-71 75 69 63-6B 20 62 72-6F 77 6E 20      quick brown
00000010: 66 6F 78 20-6A 75 6D 70-65 64 20 6F-76 65 72 20      fox jumped over
00000020: 74 68 65 20-6C 61 7A 79-20 64 6F 67-2E      the lazy dog.

```

I hope that after this tutorial you have a good idea of how to go about decrypting XOR based protections. This method can be applied to code as well as English text. I'm sure you can work out how to do this yourself.

What you will need to do is to write your own little utilities to count coincidences and to XOR the ciphertext against itself. You may also want to write a utility to analyse different plaintexts to get some statistical information about it, for example, frequency analysis of win32 code. Some information may already be out there on the net somewhere – I haven't looked.

If you find useful info or write useful utilities, please email me – I'd love to see what you have done.

~~

Ghiribizzo