An Introduction to Encryption, Part I

Introduction.

Encryption can be as simple or complicated as you need it to be and there are many advantages and disadvantages to each system. The trick to encryption (IMHO) is to use the type and complexity of encryption that best suits your needs, and not just the strongest encryption you can manage. First, a brief description of some of the principles involved in encryption that you should know before we start. After the principles follows a brief discussion of a few more important topics, then examples of the encryption types mentioned here. Part II will discuss these principles in more detail, and then some more advanced techniques.

Substitution (Simple) Encryption.

The original form of encryption, which is just a substitution cipher. Each character has a value added (or deducted, rotated etc) to it to make a new character, ie, A becomes B, B becomes C, C becomes D etc. To decrypt this, you need only save the key (the value added) and then reverse the encryption, ie, subtract instead of add. This encryption is only really effective against the casual reader. When talking in computer terms, XOR is generally used, as it is reversible (ie, value XOR valueII = valueIII, then valueIII XOR value II = value). This means you can use the same routine to decrypt as to encrypt.

Sliding Key Encryption.

A sliding key is a key that changes after each character is encrypted (for example just increased by one), ie, XXYZ is the code and the key starts with A. The first X is encrypted with A, second X with B, Y with C, Z with D etc. You'll notice that the two X's that were next to each other were encrypted with different values, meaning that the encrypted values of these two characters will be different. If somebody looks at this encrypted code, they won't be able to tell that the two characters are the same when unencrypted.

Long Key Encryption.

This is slightly more complex, using a key that is longer than the piece of code that is encrypted in each step. The key's values are used one after another to encrypt the code, ie, if the key was ABC and the code was VWXXZ then V would be encrypted with A, W with B, first X with C, second X with A and Z with B. This has two advantages. First, the key is much harder to guess or brute force. (given each value has a range of 255, a key of one character has (obviously) got 255 possible values, a key of only three characters has 16,581,375 (255 x 255 x 255) and a key of five characters has over 1,000,000,000,000 (255 x 255 x 255 x 255 x 255)). Secondly, this method also has the advantage of a sliding key.

Transposition (Order) Encryption.

This is normally used in conjunction with another method, as by itself it is pretty weak. All you do is basically scramble the order of all the characters in the target code. There are several methods to do this, but most require a lot of data or code to reconstruct the original if they are to be very random.

Multiple Encryption.

This is simply encrypting a piece of code, then encrypting the code again (usually with a different method and/or key). This drastically improves the strength of the encryption, as any patterns that show through the original encryption (see cryptanalysis below) will be obscured by the next level.

Cryptanalysis.

This only really applies to encrypted text, but it is possible that it could be applied to code in the hands of an experienced cryptographer who has assembly language knowledge. I'm only mentioning it so that you know that it may be possible. Cryptanalysis is basically the study of encrypted code for clues, and with the right sample and experience, it is possible to 'guess' the decryption method. For example, if simple encryption was used, you could count all the instances of each encrypted character, then rank them in numerical order. The list of encrypted characters would probably then read 'e,t,a,o,n,i,s,r,h,d'... etc, because this is the most common order of instances in the English language (I'm assuming you're decrypting English text) ie, if the most common letter you had in the encrypted example was Y, this would more than likely be the letter E when decrypted. Also some pairs of characters appear more often than others, for example 'th' or 'er'. After you have this information, it may be possible to guess some words. Once you start guessing words, the rest comes quite quickly. There are also other rules such as most three letter words will be 'the', which help speed things up. Of course, stronger (or more) encryption means less of these patterns show through. For example, if you also encrypt the space character, you make it ten times harder to decrypt encrypted text with cryptanalysis. All the word boundaries are lost, which means half the rules are of no use. A space is also more common than the letter 'e' which means you have to add this to your frequency chart. Simply adding another type of encryption will render most of these rules useless, as they are only really applicable to substitution encryption.

Encryption strength.

This is a common cause of mis-understanding when talking in the context of computers. Encrypted code is very vulnarable to deciphering if it is encrypted simply. Increasing the complexity of the encryption increases the strength of the encryption. However, even the strongest of encryption can easily be broken when used on a computer with an algorythm to decrypt and run the code included in the program. For the encrypted code to be able to run, you must include step by step instructions on how to decode it along with the code (the

decryption routine) and the key required. Anyone who knows machine code can read this like a book. This is where armouring comes in. When you talk about encryption strength in the context of self- decrypting code, it is generally assumed to mean how well armoured the code is as opposed to the strength of the encryption method used.

Armouring. (in brief)

Armouring (AKA anti-debugging) is a method of trying to prevent the decryption routine running in any circumstances that aren't acceptable to the decryptor (ie, in a debugger or emulator). A simple armouring routine will try to detect the debugger/emulator and quit if found. There are many ways to armour code, and more are being found all the time. It's an ongoing battle. For instance, some emulators/debuggers will use the stack for their own purposes (although they simulate the information going on and off the stack when pushed or popped). If we set the stack pointer to a vital part of our code, any data pushed by the debugger/emulator will overwrite this, causing an unpredictable change to the instructions in our decryption routine. This will at the least cause the decryption to fail, thus protecting the encrypted data. In the best case it will crash the computer entirely, wasting time for whoever is trying to crack your encryption. There are many tutorials around on armouring etc, so i will leave it to those more qualified to teach this subject. At least you know why it's important, and that you are basically wasting your time with anything other than simple encryption if your decryptor is not armoured. Anyone with debug (that's about 99% of all PC users) and a bit of knowledge (no comment) will be able to watch as your precious secrets are displayed on their screens, piece by piece. Remember, armouring is a completely different skill to encryption, but you should know at least a little if you intend to have your code decrypt itself *and* remain hidden to observers. There are also techniques that will defeat an emulator but not a debugger, such as calling an obsolete or rare interrupt that returns a predictable value and using this value as part of your key. In an emulated environment the value will be different and ruin the decryption. You might want to research RDK (Random Decryption Key) and RDA (Random Decryption Algorythm) techniques (aka throw-away key/algorithm). This is a technique in which the virus is encrypted with a key/algorithm that it is not recorded (ie, thrown away). to decrypt itself, the virus must therefor brute-force attack it's own encrypted code looking for a piece of fixed data at a fixed point.

Some Examples.

(These examples are only code snippets to demonstrate certain principles, obviously you will need to set up the sections within a program yourself. These routines can be used to both encrypt and decrypt the code sections, as the operations chosen are reversable. If you substitute a non- reversable operation, you will need to adjust/repeat the routines accordingly. Please note these are very simple examples of each type of encryption, much more can be done with these. All examples are shown in a clear manner, not necessarily the most optimised. LOD and STO are used instead of indexing to help in readability. For further examples, there are many articles and tutorials in VDAT.

Substitution (Simple) Encryption.

Sliding Key Encryption. (Pad encrypted area to an even number of bytes)

Long Key Encryption.

(Uses long key. Pad encrypted area to an even number of bytes)

```
setup:
   mov cx,(offset end_of_encryption - start_of_encryption) / 2
                                        ;length of encrypted code in words
   mov si,offset start_of_encryption    ;source=start of encrypted code
   mov di,si                                    ;destination=same as source
   mov bx,offset start_of_key                   ;bx=key indexing register
   mov dx,offset end_of_key - start_of_key      ;length of key (even sized key)

start_loop:
   lodsw                ;MOV's word from [si] to ax, and increases si by 2
   xor ax,[bx]          ;the actual decryption
   add bx,2             ;moves key register to next word in the key
   cmp bx,dx                    ;compare current index to length of key
   jb next:                     ;skip next instruction if not yet reached
   mov bx,offset start_of_key   ;bx=key indexing register

   next:
   stosw                ;MOV's word from ax to [di], and increase di by 2
   loop start_loop      ;DEC'c cx, and jumps to start_loop if CX > 0

done:
```

```
setup:
   mov cx,(offset end_of_encryption - offset start_of_encryption) / 2
   ;length of encrypted code in words
   mov si,offset start_of_encryption            ;source=start of encrypted code
   mov di,si                                    ;destination=same as source
   mov bx,02828h                                ;bx=decryption key

start_loop:
   lodsw                ;MOV's word from [si] to ax, and increases si by 2
   xor ax,bx            ;the actual decryption
   stosw                ;MOV's word from ax to [di], and increases di by 2
   loop start_loop      ;DEC'c cx, and jumps to start_loop if CX > 0

done:
```

```
setup:
   mov cx,(offset end_of_encryption - start_of_encryption) / 2
                                        ;length of encrypted code in words
   mov si,offset start_of_encryption    ;source=start of encrypted code
   mov di,si                            ;destination=same as source
   mov bx,02828h                        ;bx=decryption key

start_loop:
   lodsw                ;MOV's word from [si] to ax, and inreases si by 2
   xor ax,bx            ;the actual decryption
   inc bx               ;adds 1 to the key in each loop
   stosw                ;MOV's word from ax to [di], and increase di by 2
   loop start_loop      ;DEC'c cx, and jumps to start_loop if CX > 0

done:
```

```
setup:
   mov cx,(offset end_of_encryption - start_of_encryption) / 2
                                        ;length of encrypted code in words
   mov si,offset start_of_encryption    ;source=start of encrypted code
   mov di,si                            ;destination=same as source

start_loop:
   lodsw                ;loads first word from source
   mov bx,ax            ;stores first word in bx
   lodsw                ;loads second word from source
   stosw                ;puts second word into first word's place in destination
   mov ax,bx            ;restores first word from bx to ax
   stosw                ;puts first word in second word's place in destination
   loop start_loop      ;DEC'c cx, and jumps to start_loop if CX > 0

done:
```

Transposition (Order) Encryption.

(Swaps every pair of words. Encrypted area must be padded to a multiple of four bytes)

Conclusion.

As always, I welcome ANY feedback, good or bad, as long as it is reasonable.