**F.E.D.S. – Flexible Encryption of Data Stores**

Alphabet:

Typically, the standard alphabet is used. This alphabet, [Space, A-Z, .!?] allows for comprehensive storage of data while minimizing the amount of characters used, thus allowing operation and modification by operators by hand. It is possible to use the ASCII alphabet, but only where the recipient either has lots of time, paper, and patience, or is known to be using the decryption program instead of processing by hand. Generally, the standard alphabet is preferred.

Key Structure/Encryption:

Key structure follows this simple layout. KeyA | KeyB. KeyA consists of the Operators used to generate the encryption table, used to convert the message into each letters respective values. KeyB is the string which is converted and is added to the converted string to be encrypted, or subtracted from the encrypted message to reveal the plaintext’s values, then fed through the table to receive the plaintext. Example of such encryption is:

Table: (Using A – O alone)

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15

Message: HELLO

Key: LED (Any key may be used, either message based [Key = HI] or random [Key=AGFDH])

H E L L O becomes 8,5,12,12,15

LED becomes 12,5,4

8+12,5+5,12+4,12+12, 15+5 (Note how, when the key reaches the end, it simply loops around and begins again)

20,10,16,24,20 – Pseudo encrypted values. Since some values extend beyond the table, the number of entries in the table must be subtracted from the exceeding values. That value is added back in decryption to any negative numbers, until a positive value is reached.

15 table entries – 15

5,10,1,9,5

Final message: EJAIE

Notice how the 3rd and 4th characters are different, even though in the original message they had the same values. This is another important point of FEDS.

(The above example defers closer to the FEDS predecessor NOMAD due to its lack of customization of the table. This point is addressed later.)

Operators/Tables:

All operators use the output of the operator before. An example would be, taking C3 and then R, would, for the string ABCDEF (in place of the actual table) become ABCDEF 🡪 DEFABC 🡪 CBAFED. The operators are used to generate the table from the alphabet, adding an extra layer of random-seeming data which is, as in the nature of cryptography, reproducible only with the exact steps taken to generate it in the first place, dictated by KeyA.

KeyA starts with either N or A, N symbolizing Normal (Lowercase included) or A (ASCII alphabet). You can leave off these characters (ex. C3ER….) however, it limits your character selection to [space] A-Z (a-z with N) .!?

C[1-25/126 for ASCII] – Caesar Cipher. Take the existing string (that which has been generated so far) and perform a Caesar Cipher on it. The number after C is the rot #. Example: Using string ABCDEF

C3 – ABCDEF 🡪 DEFABC

R – Reverse the string. ABCDEF 🡪 FEDCBA

S[ab] – Swap the values of the two characters in the string.

SCE – ABCDEF 🡪 ABEDCF

E – Every other letter. By hand, this is achieved by making two rows, and writing out the text, then adding the bottom to the top, like so.

A C E  
B D F 🡪 ACEBDF

Of course, any method that achieves the same result may be used.

While these operators may not seem like much, the various combinations of them allow for nearly infinite table combinations. KeyA sets may look anything from AC3SR to NREC14SALC2ERSXLC3E or any other combination therein. This allows an impossible-to-crack table, especially due to the random seeming element claimed by the S character.