# *“Rotor” Enigma+ encryption library*

*From the advanced projects division of Iozone.org*



 There are many available encryption algorithms however most of these are block oriented and non-symmetric . Rotor is a stream oriented, symmetric mechanism. This mechanism provides a capability for the user to control the strength of the encryption and have more control over the CPU consumption. It also is not based on any existing standard, and has not been weakened by any backdoors.

 The industry is starting to see a shift in where encryption is performed, and who has access to the keys. It appears that the trend is to move the location and access out of the middleware and all the way to the end user and the application layer. This is now being referred to as BYOE (Bring Your Own Encryption)
Rotor is such an implementation. The full source is covered with an open source license from MIT. The source is available for inspection and does not have any backdoors.

 The source code provides a library, with a well defined API so that other tools that need encryption can easily incorporate the “rotor” encryption mechanism.

 Also provided in the source code is an example program that uses the rotor encryption library. This example program can be used to encrypt, and decrypt any file.

**Description of operation**

 Think of a cylinder with letters 0 to 254 on its surface, in randomized order. Now stack a series of cylinders next to each other, such that they mechanically interact as described below. (There is a drawing in the next few pages that may help the reader follow along in the description)

 An input character is Xor’d with a random number (Mersenne Twister algorithm) and then this value is used to find the offset to that character on the cylinder. This offset is then used as the input for the next cylinder. At the end of the stack of cylinders out pops a value that is the offset on the last cylinder of a value. This value can later be used to decode the previous cylinder, and so on, until one arrives at the first cylinder. This can be XOR’d with the original random number. One will then find the original character that was input.

 Each cylinder contains 0 through 254 (all possible ascii characters). Each cylinder is randomized by a series of shuffles that are based on a Miller and Parks LC RNG (random number generator). Rand() gets it's seed from the user of this application. This provides a method where every message can have differently randomized set of characters on each of the cylinders.
As each character is processed, the seed is incremented, and one of the cylinders (randomly selected) is re-shuffled, at a user specified frequency (cog). Also, a partial shuffle is also performed per input character.

A reverse index is also created within each rotor so that a lookup for any given character is a direct index, not a linear search. This provides faster encryption/decryption and reduces CPU consumption.

All one needs to create an encrypted file is this program and:

* The key used to prime the random number generator.
* The number of rotors that will be used. ( or leave as default )
* The re-shuffle frequency. (cog)
* The input characters to encrypt.

 All that one needs to decode the original message is this program and:

* The key used to prime the random number generator.
* The number of rotors were used. ( or leave as default )
* The re-shuffle frequency. (cog)
* The encrypted characters to decrypt.

Library API:

* void \_start\_encryption( unsigned long long key, int rotors, int freq )
* void \_end\_encryption(void)
* unsigned char \_crypt\_char( unsigned char );
* unsigned char \_decrypt\_char( unsigned char);
* unsigned char \* \_crypt\_string( unsigned char \*dest, unsigned char \*src, int length );
* unsigned char \* \_decrypt\_string(unsigned char \*dest, unsigned char \*src, int length);

The input character is XOR’d with a random number and then input to the rotors, as shown in the example below. ( Simplfied by having only only 2 rotors shown. The actual number of rotors is user defined )



Note that the contents of the rotors above are reshuffled at the user specified frequency, and a partial shuffle is performed with each character that is processed.

This mechanism exceeds the original Enigma in many aspects.

* The position of the characters on each rotor is randomized.
* There are 255 characters on each rotor.
* The number of rotors is variable, and user defined.
* The rotors are shuffled at a user defined frequency.
* Partial shuffles are done as characters are processed.
* Multiple, different random number generators are used in the mechanism.
* The seeds for the random number generators are large numbers. 2^64
* The work to encrypt and decrypt is variable.