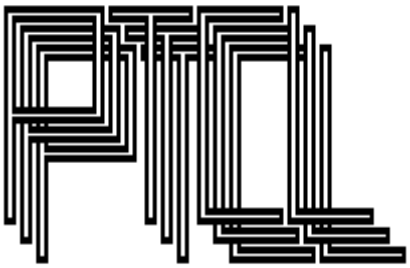


PlexusTCL Crypter

version 5.12 from January 20 2025

[this version of the program is not compatible with versions below 5.04]

IMPORTANT WARNINGS

1.) IF YOU ARE NOT SURE ABOUT THE STABILITY OF THE PROGRAM OR YOUR COMPUTER, THEN BEFORE ENCRYPTING A FILE, BE SURE TO MAKE A BACKUP COPY OF IT.

2.) THE PROGRAM DOES NOT GUARANTEE PRIVACY IF YOU ALLOW THE POSSIBILITY OF SIDE-CHANNEL ATTACKS, SUCH AS A RAM FOOTPRINT (DUMP) WHILE THE PROGRAM IS RUNNING, ENCRYPTION KEYS GETTING INTO THE SWAP FILE, INFECTION OF THE COMPUTER WITH MALICIOUS PROGRAM, YOUR LONG LANGUAGE OR YOUR SYSTEM ADMINISTRATOR, ETC.

3.) TO ENSURE THE MAXIMUM POSSIBLE LEVEL OF SECURITY WHEN USING THIS SOFTWARE, CONSULT WITH COMPUTER SECURITY EXPERTS, SINCE THE PROGRAM CONTAINS ORIGINAL DEVELOPMENTS.

4.) THE PROGRAM CREATES UNSIGNED ENCRYPTED FILES, WHICH MAKES IT IMPOSSIBLE TO IDENTIFY THEM AS ENCRYPTED, WHICH DOES NOT ALLOW THEM TO BE DISTINGUISHED FROM RANDOM DATA. IF THE PRESENCE OF SIGNATURES IS IMPORTANT TO YOU, USE ANOTHER CRYPTOGRAPHIC PROGRAM.

The program "PlexusTCL Crypter" is designed for cryptographic protection of information by encrypting files up to 8 EiB (Exbibytes) in size starting from version 5.10. Files can be processed (encrypted/decrypted) by five cryptographic algorithms at the user's choice, namely Rijndael-128 (AES), Serpent, Twofish, Blowfish and Threefish, using a key file or a string entered as a password .

Program and algorithms

The software is an executable file that implements encryption algorithms and an input/output control system in the form of called functions. The GUI (graphical user interface) is integrated into the executable file, so changing the interface will be problematic. The GUI is developed only for Windows OS and works starting with Windows XP SP3. If you are interested in changing the interface or algorithms, you can always rebuild the project from the source texts.

All implementations of cryptographic algorithms, namely AES, Serpent, Twofish, Blowfish and Threefish, are tested using test vectors published by their developers, and therefore fully comply with their mathematical descriptions or published standards.

When using the block cipher from the PlexusTCL Crypter program, it is important to take into account the fact (not so important for the average user) that the AES cipher differs from the Threefish cipher in that Threefish is designed as a 64-bit cipher, i.e. it operates on 64-bit (8-byte) numbers as independent units, while AES operates on data blocks

consisting of 8-bit (1-byte) values. This means that AES encrypts plaintext in blocks of bits, processing 8 bits (1 byte) at a time, while Threefish takes 64 bits (8 bytes) of plaintext as one large number, and operates on it as a whole. Therefore, the speed of Threefish on 64-bit processors is several times higher than the speed of its operation on 32-bit processors, but the speed of operation on 32-bit processors is 2-3 times lower than the speed of AES. These parameters may change depending on the compiler used to build the project, as well as the optimization parameters. Therefore, if the speed of data processing is important, it is recommended to use Threefish on 64-bit processors, and AES on 8, 16 and 32-bit ones. This applies only to the algorithms, but not to their implementations in this program.

Threefish, starting with version 4.91, has been optimized and extended with minor and major versions that accept 256-bit and 1024-bit keys, something that was not available in earlier versions of the program. This means that if you encrypted a file using version 4.91 or later, using the Threefish algorithm with a 256-bit or 1024-bit key, you will no longer be able to decrypt that file using version 4.90 or earlier. Threefish was developed by a team led by Bruce Schneier for use in the Skein hash function as an easy-to-implement, customizable block cipher built from basic arithmetic. It can be used to encrypt any data with any key length—it's a really good cipher.

AES cipher, starting with version 4.92, has also been optimized, and its operating speed on some computers is comparable to the operating speed of the Threefish cipher.

Serpent ciphers can be used on any 32 or 64-bit processor, as these ciphers show almost the same performance on both, and their implementations are very fast compared to Threefish and especially AES . Blowfish encrypts data in 32-bit blocks, encrypting 2 parts of the plaintext at once as a 64-bit data block, accepting the left and right 32-bit parts of the data separately as input, replacing them with ciphertext, like any other ciphers designed based on the Feistel network. Since Blowfish, Twofish and Serpent ciphers are designed as 32-bit, this may affect the performance when using the cipher on 64-bit platforms for the better, but the developer did not notice any difference. The speed of both ciphers on 32-bit and 64-bit processors is almost the same.

Please note that the ciphers implemented in the program, for which it is impossible to select the encryption key length, are implemented in such a way that they accept only keys of the maximum possible length, which applies only to the Blowfish cipher. Previously, this was also true for the ARC4 cipher, but starting with version 5.00, this cipher was cut from the program as hopelessly outdated, while the optimization of the AES cipher is such that it is not inferior to ARC4 in data processing speed, taking into account the time of writing to disk. With the update of the AES implementation, there is no longer any need to use ARC4 .

It is important to remember that processors, storage devices, operating systems, compilers, compiler optimizers, as well as the conscience, mental abilities and mental disorders of the authors of processors, storage devices, operating systems, compilers and optimizers, are as diverse and very interesting even for a non-specialist, as they can scare you to gray armpits. From the first models of Intel 80386 to AMD Ryzen 9 9950X (as of 2024), many engineers are constantly improving something, and these improvements can both harm the speed of the program and improve it. For example, the old computer of the author of this program running Windows XP SP3, with an Intel Pentium D 820 processor, overtook a computer with an Intel Core I3-3217U processor in encryption speed, and this despite the fact that the second one was running Windows 8.1. So the speed of operation can depend on almost everything that has a direct relation to computing technology, and that is why it is not specified in this README.

All five block ciphers, namely AES, Threefish, Blowfish, Twofish, and Serpent , operate in CFB mode (ciphertext feedback mode), which provides a fairly reliable level of security for using a block cipher, turning the block cipher into a stream cipher. There is no need to worry about this, since a properly used block cipher in the form of a stream cipher is in no way inferior in cryptographic strength to a block cipher. When encrypting data in CFB mode, the block cipher is not used at all to encrypt data, but is used to generate a pseudo-random sequence of bits by encrypting a constantly changing array, which is added modulo 2 with each bit of the encrypted data. By the way, using CFB mode allows you not to implement the decryption function, which simplifies the implementation and use of the cipher, but in the source texts of the program, the decryption functions are still implemented.

To encrypt something with a block cipher operating in CFB mode, you first generate a random or pseudo-random sequence called an IV (initialization vector), encrypt the IV with the block cipher, and add the resulting sequence bit-by-bit modulo 2 to the block of data you are encrypting to get the ciphertext. Once you have finished encrypting the first block of data, as I V when encrypting each subsequent block, the previous encrypted block is used, which ensures an avalanche change of the entire ciphertext when changing even 1 bit of the encryption key, the encrypted block or IV. If you express the CFB mode in symbols, you get something like this:

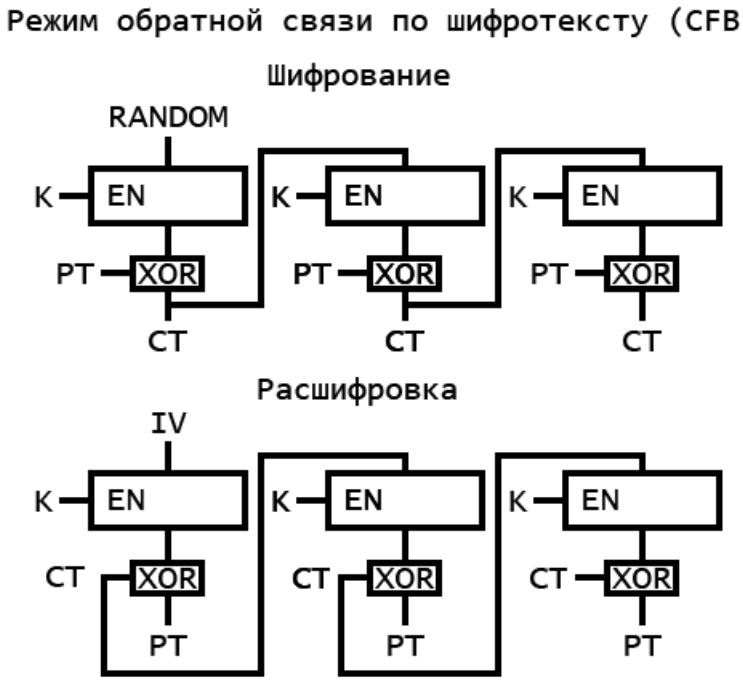
iv = r andom(t ime(n ow));
c [0] = e (iv , key);
c [i] = p [i] xor e (c [i-1], key); from i=1 to i=n;

Where:
r andom - pseudo-random number generator for your OS
t ime - the system time of your OS in seconds
n ow - real time
xor - bitwise exclusive OR operation
key - encryption key
iv - initialization vector
n - length of ciphertext in blocks
i - a counter increasing by one
p - plain text
c - ciphertext
e - encryption function

Even if you encrypt a 2 GiB file consisting entirely of zero bits using a sequence of zeros as the encryption key and IV , the encrypted file will appear to be a jumble of random data with no pattern. In the program "PlexusTCL Crypter" , the CFB mode is implemented in such a way that **as IV, an encrypted pseudorandom sequence is used** , written to the file before the first encrypted block, which is necessary for decryption and does not affect the cryptographic strength in any way, since knowledge of the IV will not help in any way when cracking the ciphertext without knowing the key, and **the key should always be the most guarded secret** . Knowing that the IV is written to the file as the first encrypted block will not help in any way to crack the ciphertext if the key is not known.

When encrypting two plaintexts whose first encrypted blocks are the same, the initialization vector unique to each plaintext is is used in CFB mode to hide this coincidence if the same key is used for encryption. A pseudo-random IV for each plaintext allows using a **"long-term key"**, i.e. one key to encrypt multiple plaintexts, even if they are the same. Since the IV in this program is an encrypted pseudo-random sequence generated by the underlying OS, this only poses a threat if the PRNG (pseudo-random number generator) is attacked. For example, if the system time is stopped in the OS, the pseudo-random number generator will endlessly generate the same sequence of numbers from 0 to 255 (from 0x00 to 0xFF), since the "seed" for the generator is the actual system time in seconds. If an

attacker knows the plaintext whose encryption yields an IV , the attacker can try to recover the key and decrypt all messages encrypted with the same key. If the PRNG generates the same numbers, then encryption of the same blocks of plaintext using the same key will produce the same ciphertexts, which makes it possible to get closer to cracking all ciphertexts obtained from one key. To avoid this, you need to monitor what other software on the computer you are using is doing (whether it is replacing anything), avoid using "pirated" copies of software that may be infected with malicious code. To increase randomness and at the same time reduce the probability of repeating values in IV, its first two bytes are added modulo two with the coordinates of the mouse cursor along the X and Y axes, which are calculated at the moment the "Start" button is pressed , which introduces an element of uncertainty into IV (the attacker cannot say in what position the mouse cursor was at the moment the button was pressed). Also, the first four bytes of IV are added modulo two with a random value from the stack, which introduces even more uncertainty into IV (the attacker, of course, will not be able to determine what value the stack frame variable had, for example, at address 0x00408A47). Of course, the value can be selected by a complete enumeration of all options, but this makes no sense without a key and does not affect cryptographic strength in any way.



Key generation function

A key derivation function (KDF) is designed to "stretch" a user's password and generate an encryption key from it. All KDFs are designed to make brute-force attacks more difficult, even if the brute-force attack involves a server park of thousands of machines, by making the attack relatively slow.

All KDFs are built the same way. It is always a function that takes some X as input, and from this X, upon completion of the work, initializes K, which is the encryption key for a block or stream cipher. A typical KDF takes as input the algorithm that will process the data, the user's password, its length, salt, the length of the salt, the number of iterations of data processing, and the desired key length. If you write down in symbols what any KDF does, you get something like this:

K = KD F(alg, pass, plen, salt, slen, count, klen);

Where:

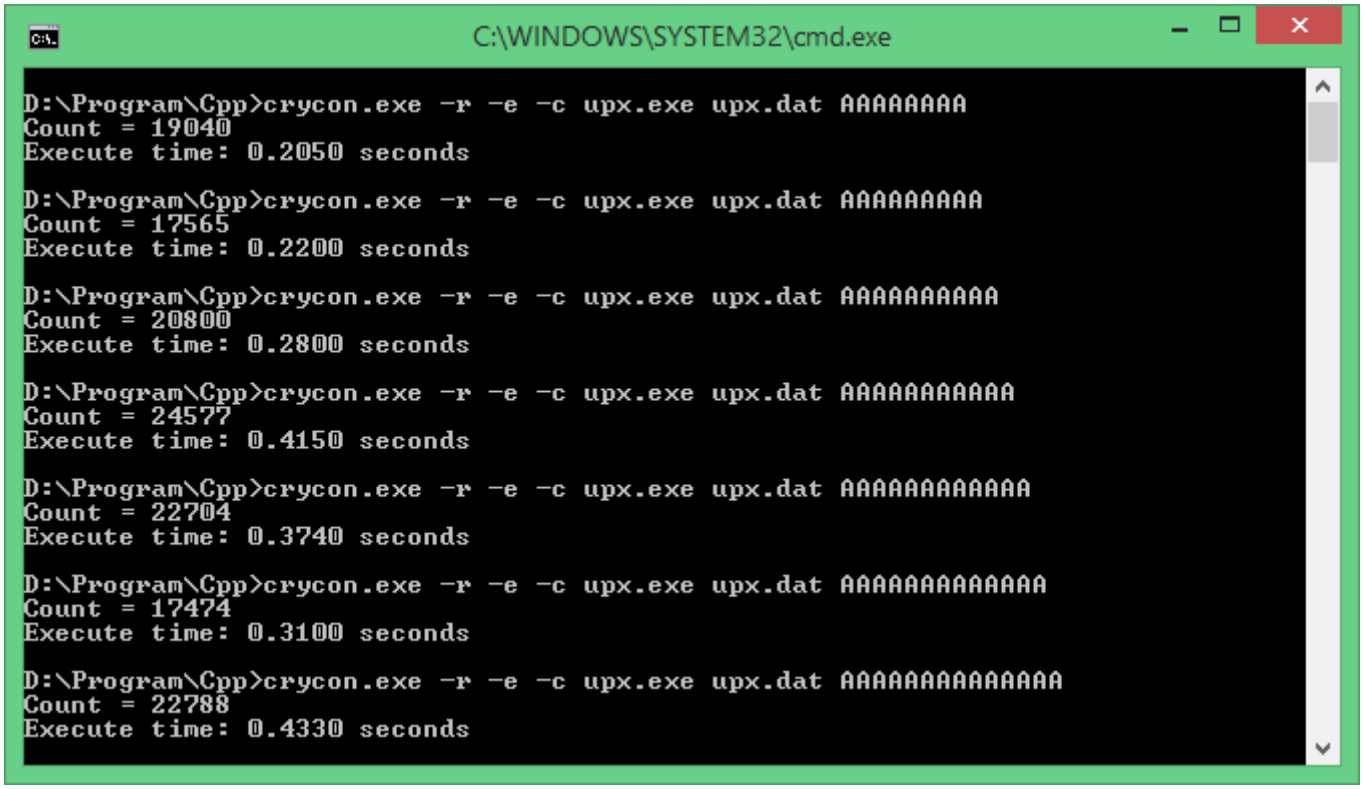
K - encryption key;
alg - algorithm for processing password and salt;
pass - user password;
plen - password length;
salt - pseudo-random constant (salt);
slen - salt length;
count - the number of iterations of password and salt processing;
klen - desired length of the encryption key;

As a result, K will contain the encryption key, which is fed to the input of the encryption/decryption function to process any data. Let's analyze a typical KDF in order.

First, a KDF takes a data processing algorithm that will process the user's password and salt as many times as specified in the count argument. Typically, this is a hashing algorithm such as MD5, SHA-2-512, Keccak, or an encryption algorithm such as Salsa20/20, DES, Rijndael, Serpent, IDEA, etc. The number of iterations of processing the password and salt is usually set statically, which is sometimes increased depending on the speed of the computer park of a potential attacker brute-forcing the password. But the count value should increase sooner or later, since computer performance is constantly increasing, according to Gordon Moore's observation. The count value for 2021 is usually from 150,000 to 2,000,000 and always depends on the performance of the encryption or hashing function processing the password and salt. All KDFs are usually designed to generate an encryption key in about one or two seconds on different processors. To generate a key to decrypt, for example, a record in a database, a user needs to call the KDF once and wait only one or two seconds, while an attacker will have to call the KDF billions of times to try all possible passwords. Trying many variants of a relatively small password, only 8-10 characters long, typed on a standard keyboard, an attacker will spend amount of time that simply won't survive.

Salt, or pseudo-random constant, is needed to somehow mix it with the password, multiplying the number of password variants, which greatly complicates the search, making it a very long and extremely unprofitable undertaking. The developer excluded salt from his KDF, because he did not like the word "salt".

In this program, KDF is an original development, since the author of the program is none other than a "paranoid suffering from spy mania". This means that the encryption key is generated by a cryptographically strong hash function SHA-2-256, the source code of which in the C language was prepared by NIST (National Institute of Standards and Technology of the USA), the password length is limited only by the bit depth of the user's processor register, and today is at least 4294967295 characters, and **the number of hashing iterations is dynamic** , and is generated by a tricky interweaving of bit operations and calculating the 32-bit cyclic redundancy code (CRC32) of various parts of the password . The figure below shows that when increasing a static password by just 1 character, not only the number of hashing iterations changes, but also the key generation time.



Why is the number of hashing iterations not made static, like in humans? To completely eliminate the possibility of password brute-force by tying not only the password with its length to the number of hashing iterations, but also the length of the key, making password brute-force very difficult. But not only the password length affects the number of iterations, but also each byte of the password, which turns KDF from an ordinary function into an "impenetrable taiga forest" of hashing. If the complexity (time) of a brute-force attack increases with the number of hashing iterations, and this number depends not only on the password length but also on each of its byte, then to find out the number of hashing iterations and try to create a password brute-force scheme, you need to know not only the password length but also each of its symbols, which reduces password cracking to knowing the password itself. The number of hashing iterations can, of course, be brute-forced, but such a brute-force makes life very difficult for an intruder (the brute-force still needs to be carried out). The author of the program admits with a heavy heart that a large static number of iterations (500,000-1,500,000) is almost always better than a dynamic one, but he somehow doesn't want to redo everything, and besides, when replacing the key generation function, the old versions of the program will become incompatible with the new and future versions for the second time (all encrypted files will have to be re-encrypted).

This KDF is called KDFCLOMUL, from the constant CLOMUL_CONST (Clock multipler), which still allows you to configure how large the number of hashing iterations will be, and therefore the time it takes to calculate the key. It is described in the header file "clomul.h", which can be easily found in the "src" directory, and the description is translated into English using Google Translate (it is almost the same as the description below).

The CLOMUL_CONST constant denotes the so-called "clock multiplier" used in the KDFCLOMUL function as one of the operands when forming the number of hashing iterations. If we describe the constant in simple terms, the higher its value, the longer it takes to generate a key from a password, since generating a key requires many more processor cycles. It is necessary for generating an encryption key from a password, since it

affects the number of iterations of the password hash calculation cycle. The cycle itself uses the SHA-2-256 hashing algorithm. By default, the "clock multiplier" is 1, but increasing it to 12, 16, 38 or 67 makes any equipment generate a key very slowly, which makes a brute-force attack even on a short 8-character password impossible in an acceptable time. Be careful with this constant, since an incorrectly chosen value can lead to you waiting for a minute or even tens of minutes for the key to be generated from a relatively short password.

Choose the "clock multiplier" value based on the estimated total power of your opponent's computer park, since a defense adequate to an attack is built only if the attacker's capabilities are known. For 2023, the value can be made from 5, this will be enough for another 5-10 years. The maximum value is 57344. Never exceed it, otherwise the number of password hashing iterations may overflow and become very small, since when unsigned numbers overflow, they are simply reset to zero and begin to grow again from zero.

A "clock multiplier" value of 1 ensures that the encryption key is generated from the password "password" on an Intel Core I3-3217U processor with a clock frequency of 1.8 GHz somewhere around 0 . 18 seconds, which already turns a full search of all 8 character passwords into a hellish nightmare, since to search through all passwords consisting only of lowercase Latin letters, you will need to search through 208827064576 password options. Since This processor is capable of sorting 5 passwords per second, then a complete search of all passwords on the above processor will take 1324 years. But keep in mind that graphic processors are tens of times faster than conventional ones, and password search can be performed in parallel on tens of thousands of machines, so use long passphrases (20 - 30 characters). But do not make passwords too long (40, 50 or 100 characters), as the key generation time is directly proportional to the password length - the longer the password, the longer the key generation takes. Note that a $1000 GPU, as of early 2020, can calculate over 10 billion SHA-1 hashes in one second, so not everything is as rosy as it usually seems from the numbers.

In case the base "clock multiplier" equal to 1 is used, then the number of password hashing iterations is about 20000 per byte of encryption key, and can have a maximum value of 32767, and the password hashing time "password" on the above processor is about 0.18 seconds, which is an ideal compromise between security and performance today. If the password is hashed about 100000 times to generate only 5 bytes of encryption key, then when generating a 256-bit (32-byte) encryption key, the password will be hashed about 640000 times. Not a bad result.

As the test on the above-mentioned processor showed, with a "clock multiplier" equal to 64, the generation of a 256-bit encryption key for the Rijndael algorithm, when using the password "password", lasts as long as 11 seconds. In order to save the user from waiting, the default value was reduced to the minimum, in order to unload the user's processor and comfortable work with the program, which does not affect security in any way. Programs that have different " clock multiplier " , **INCOMPATIBLE** !

Remember! The weak point of any program that works with passwords is the frivolity of its user! The password should be long, illogical and generally a little idiotic. In their book "Practical Cryptography", Dutch and American cryptographers Niels Ferguson and Bruce Schneier recommend using as a password phrase what they called "shocking nonsense", i.e. something terribly indecent, vile; something that in your right mind you would never write on paper and let anyone read. Passwords like " *Epihantozhuevsky shtruldik smacked the chipuzhku* "
or " *Sly+Bublik+ Sp&zdil +Rublik* " are quite good for use by Russian-speaking users, but the password " *123456* " can only be chosen by someone who knows nothing about password protection of information, since it has been one of the most popular passwords in the world for about 30 years. Store the password only in a good password manager with a free source

code, the use of which is approved by the general public and the international expert community, if you do not trust your memory.

The developer also recommends that the program user read the instructions for creating secure passwords from the Electronic Frontier Foundation (EFF).

When encrypting data with a block cipher with one initialization vector, the key can be used to encrypt only a limited amount of information, which is highly inadvisable to exceed. This amount of information is called the "key load", and the larger it is, the less often the encryption key should be changed to avoid attacks on the cryptosystem. In this program, the key load is set to 2 GiB, which is very small and much less than the actual key load of the Blowfish cipher, the cipher with the smallest block of all the other ciphers used in the program. If the set key load is exceeded, i.e. if the number of encrypted bytes exceeded or became equal to 2 GiB when encrypting a file, the program generates a new key by hashing the old key with the SHA-2-256 algorithm. This does not interfere with encryption/decryption, does not introduce a vulnerability, does not require any action from the user, and is done to enhance protection, since in all versions of the program before version 5.10, the program could only process files up to 2 GiB in size.

Authenticity of encrypted file and confirmation of authorship

In order to verify the authorship and integrity of files or texts, in the modern world they use elliptic curve cryptography, but since the author of the program is a "paranoid suffering from spy mania", he decided not to trust something that he himself understands little about and not to take the word of companies like Microsoft, Google , or organizations like NIST, as do many software developers around the world.

The author decided to go the most conservative way, which was known to him from books on cryptography and information security: authenticated password-cryptographic protection. Authentication (authenticity verification) will be discussed below.

To detect changes made to a file, the file is usually hashed using a checksum function, publishing the file and its hash where the file and especially its hash will be difficult to change. Authors of free software often use this approach: the program is published on the authors' website, along with a text file containing the hash of the published program, and access to change both the program and the file with the hash is set only for site administrators.

A text file with a checksum, used by the program user to check the file's checksum. The program user calculates its hash sum using the same hashing algorithm as the program developers, and then compares the resulting checksum with the one that the developers kindly published on their website.

It is quite clear that a program whose checksum does not match the one published by its developers cannot be called genuine. When checking the checksum of a file, we trust the site where the developers, as we think, published the checksum of their program. But what if the site of the program developers is attacked, the program is replaced by a malicious one and the hash sum is substituted? The question of author authentication arises, i.e. checking who exactly generated the published hash sum of the published program, the author or the attacker?
The HMAC (Hash-based message authentication code) algorithm comes to the rescue. This is the same data hashing algorithm, but using a secret. The author allowed himself to make a small editorial change to the original algorithm, calling it HMAC-SHA-2-256-Uf, in order, as he believes, to

complicate the life of a potential attacker even more than the developers of the original algorithm did.

If we describe the HMAC-SHA-2-256-Uf authentication algorithm in characters, we get something like this:

HMAC = h((key xor alpha) || h((key xor beta) || h(message)));

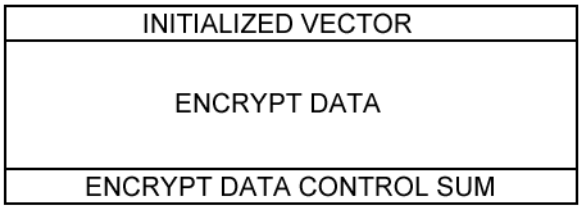
Where:

HMAC – message and key checksum;
h – function for calculating the h hash sum;
key – encryption key known to the sender and recipient;
alpha – the first secret number 0x66;
beta – the second secret number 0x55;
message – a message whose authenticity must be proven;
xor – addition modulo 2;
|| – concatenation ("gluing" of data);

How does HMAC allow us to verify not only the integrity of data, but also its
authorship? The sender's algorithm of actions:

1.) Add the encryption key and the second secret number modulo 2.
2.) Combine with the message to be protected before encrypting it.
3.) Calculate the hash sum.
4.) Add the encryption key and the first secret number modulo 2.
5.) Compare with the previously calculated checksum.
6.) Calculate the hash sum.
7.) Encrypt the message and send it along with the hash sum.

To make it easier to understand what a file looks like after encryption, the diagram of any file encrypted by this program is shown on the right. As you can see in the figure, depending on the cipher used, the first 128, 256, 512 or 1024 bits of the encrypted file are the initialization vector, followed by the encrypted data, and after 256 bits of the checksum of the plaintext and the encryption key.



An attacker who intercepts such an encrypted message without the encryption key is not only unable to read the plaintext, but also unable to forge the message without the recipient noticing. One of the most common mistakes in cryptography is to think of an encrypted message as one that cannot be manipulated. The HMAC algorithm was invented to protect against manipulation. Without knowing the encryption key, which is known only to the sender and recipient, an attacker cannot obtain the correct hash sum for the data he has modified, which means that the recipient of the message, having the correct encryption key, can easily find out whether the message has been forged or not. An attacker can still corrupt one or more messages, interrupt the transmission of messages, endlessly record messages being sent, redirect a message "to the wrong place", but this is where his capabilities as an attacker end.

This program uses
the HMAC function based on the SHA-2-256 algorithm, developed by the author of the program, because he is too lazy
to understand how other key authentication functions are arranged, and even more so he is too lazy to write complex and long functions in the C language, for the implementation of which he will not receive not only money, but even feedback from the community. The author decided to use something simple, fast and understandable, while maintaining the strength of the original HMAC function from 1997, and, as the author believes,

even complicate the life of a potential hacker, without introducing complex and difficult to understand public key cryptography.

Configuration file

The configuration file exists to store program settings, usually in text form. Configuration file support in the program "PlexusTCL Crypter" has been implemented since version 5.12, the configuration file itself is located in the file "settings.cry" and must be located in the same directory with the program being launched. If the configuration file is not in the directory with the program, or the parameter read from the file is incorrect, the program will apply the default settings.

The configuration file is structured as follows.

The program accepts as useful configuration file content only the underscore sign, capital English letters, numbers from 0 to 9, the separator in the form of the equal sign, the # sign and the semicolon. The program ignores other characters (all other bytes).

The length of any line in the file should not exceed 127 bytes. If the line length is 128 bytes or more, it will be read from the file in parts of 127 bytes. The program will read no more than 10,000 lines from the file. A line that begins with the # sign and ends with a line feed is a single-line comment. The program does not support multi-line comments like in the C language. If the # sign is encountered at the beginning or in the middle of a line, the program considers all subsequent characters before the line feed or the semicolon sign to be a comment (such a comment can be inserted even in the middle of a parameter).

In the configuration file, you can describe the parameters specified as a line of the form KEY=VALUE. The key must be followed by an equal sign, the value must be after the equal sign. If you want to describe the parameters in one line, they must be separated by a semicolon.

TOP_COLOR - sets the color of the program header. It, like other colors in the program, can only be set by a 32-bit number in the form of a HEX string, for example 00623E00. The senior 8 bits of the number (in the example 00) do not mean anything, only the junior 24 bits matter, since they contain the RGB color (Red, Green, Blue). In the example 00623E10, byte 62 means a shade of red, byte 3E means a shade of green and byte 10 a shade of blue.

TOP_TEXT_B_COLOR - sets the background color of the text in the program header.

TOP_TEXT_COLOR - sets the color of the text in the program header.

PROG_BAR_COLOR - sets the color of the progress bar.

CIPHER - sets the default cipher. Possible parameter values: AES, BLOWFISH, SERPENT, THREEFISH, TWOFISH.

KEY_SIZE - sets the key length of the specified cipher. The parameter values for the AES, SERPENT and TWOFISH ciphers are: 128, 192, 256. For the THREEFISH cipher: 256, 512, 1024. For the BLOWFISH cipher, this parameter is not used, it can be left at any value.

OPERATION - sets the operation to be performed by the program. Only ENCRYPT or DECRYPT values can be specified.

PASS_GEN_SIZE – sets the length of the generated password. You can set a value from 8 to 256.

ERASED – specifies whether to erase the file after its encryption/decryption is complete or leave it alone. Can only be set to TRUE or FALSE.

sha256sum

Also, in the "PlexusTCL Crypter" package, starting with version 2.73, there is a bonus, namely the sha256sum.exe utility, which calculates the SHA- 2-256 checksum of both texts and files up to 2 GiB in size. The utility, like its source code, is distributed free of charge, and it is present in the archive in case the user does not have a program for calculating checksums of lines and files. The program accepts three arguments as input, and calculating the checksum of the "PlexusTCL " line and printing it in a table will look like this:

[user@machine]~$:./ sha256sum – t – s " PlexusTCL "

2D 92 4A CF 99 37 74 AC 55 3D C7 A7 6C AD 3D DD
64 4D 93 91 E3 24 58 24 C1 21 FD 66 EE F8 0F EC

The sha256sum utility takes three arguments, namely "-s/t", "-s/f" and a simple string. The string equivalents of the first two arguments are "--string/table" and "--string/file". The first argument allows you to choose whether you want to get the checksum of a string or file, in string or table format. The "-s/--string" argument specifies that the checksum will be printed as a string, and the "-t/--table" argument specifies that the checksum will be printed as a table, as in the example above. The second argument allows you to explicitly specify what to checksum, the string entered as the third argument, or the file whose name is the third argument. To calculate the checksum of a file, you need to use the "-f/--file" argument, and to calculate the checksum of a string, the "-s/--string" argument. The source code of the SHA-2-256 algorithm itself is taken from the implementation by USA NIST (US National Institute of Standards and Technology), which has been tested using official test vectors and is completely safe to use. The algorithm has been repeatedly checked by many cryptanalysts from all over the world, and no vulnerabilities or loopholes have been found in it. The SHA-2-256 algorithm itself was developed and patented by USA NSA (US National Security Agency), which limits its use, but does not prohibit its use for home purposes.

CryCon

With ry pter for **C on** sole – a console program for Linux, an analogue of the graphical program " PlexusTCL Crypter ", from which the cryptographically strong password generator and the destroyer of the processed file have been removed, since these operations can be performed by standard tools of most Linux distributions. Everything written earlier about the program "PlexusTCL Crypter" and its algorithms, in addition to integration into initialization vector coordinates of the mouse cursor and configuration file support, is also true for the CryCon program. Like any console program, CryCon accepts arguments that are interpreted by the program, and depending on them, the program performs certain operations. The length of all arguments is limited to 2047 characters.

The first argument of the CryCon program is always a string indicating the encryption algorithm to be used, unless this argument is "- h " or "-- help ", which indicates that a short help message should be displayed. The user specifies which encryption algorithm should be used to encrypt or decrypt a file in the form of a short (letter) or long (string) argument. The arguments corresponding to the encryption algorithms are listed in the table below and can be passed to the program only in small (string) register.

encryption algorithm	literal argument	string argument
AES (Rijndael)	-r	--aes
Serpent	-s	--serpent
Twofish	-w	--twofish
Blowfish	-b	--blowfish
Threefish	-t	--threefish

The second argument always specifies what operation to perform, encryption or decryption. This argument can be short (alphabetic) or long (string), and looks like literal arguments like "- e " and "- d ", and string arguments like "-- encrypt " and "-- decrypt ".

The third argument is interpreted depending on the algorithm selected. If AES, Twofish, Serpent, or Threefish is selected, the third argument is interpreted as specifying whether the key length should be 128, 192, or 256 bits. If Threefish is selected, the key lengths will be 256, 512, or 1024 bits. This argument can be short (alphabetic), such as "- a ", "- b ", "- c ", or long (string), such as "--128", "--192", or "--256". If Threefish is selected, the alphanumeric arguments do not change, but the string arguments change to "--256", "--512", and "--1024". If Blowfish is selected, the third argument is interpreted as the name of the file being processed, because the key length is not specified when using this algorithm. The whole point is that Blowfish algorithm is a variable key length algorithm, which is always maximum in the program and is 448 bits. It follows that the arguments for running the program, when choosing the AES, Twofish algorithm, Serpent with a 256-bit key will be like this:

[user@machine]~$:./crycon --twofish --encrypt --256 secret.dat en.secret.dat

[user@machine]~$:./crycon --aes --encrypt --256 secret.dat en.secret.dat

[user@machine]~$:./crycon --serpent --encrypt --256 secret.dat en.secret.dat

when choosing the Blowfish algorithm such:

[user@machine]~$:./crycon --blowfish --encrypt secret.dat en.secret.dat

and when choosing the Threefish algorithm with a 512-bit key, like this:

[user@machine]~$:./crycon --threefish --encrypt --512 secret.dat en.secret.dat

If an error occurs during the program's operation, for example, there is no more disk space, the file for processing was not opened (meaning something is interfering), the entered argument is incorrect (written incorrectly), the length of the key in the key file is small, etc., the program will notify you of this by displaying a text message and will interrupt all operations.

In all versions of the program before version 5.10, the last command line argument always pointed to the name of the file containing the encryption key, and if the file could not be opened for reading or the file did not exist, its name was accepted by the program as a password, converted into an encryption key by the KDFCLOMUL function. Starting with version 5.10, the program asks the user to enter the password or the name of the key file on a separate line, inviting the user to enter a dialog line as in the example below:

```
[user@machine]~$:./crycon -r -e -c secret.dat en.secret.dat
[$] Enter password or name keyfile:
```

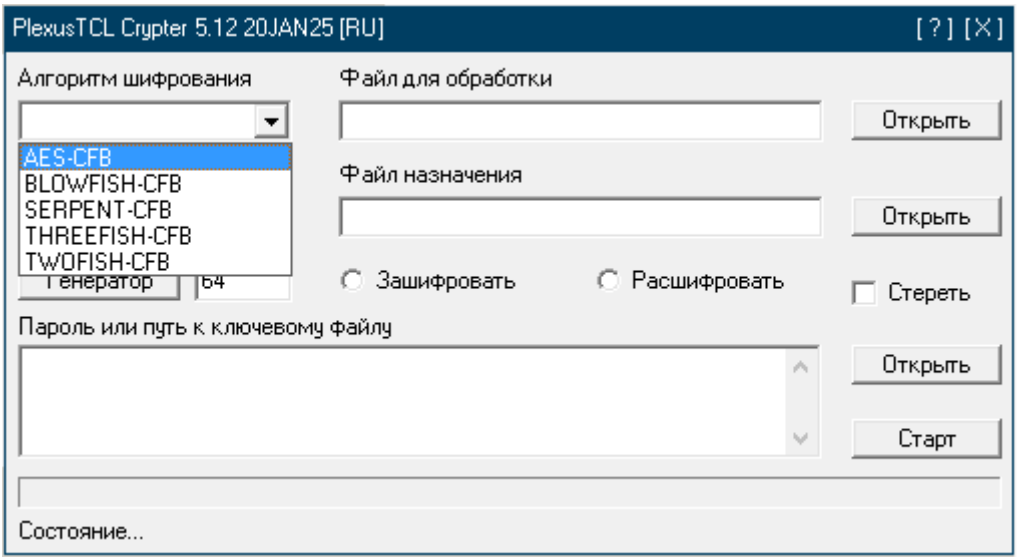
The data in the keyfile is considered the encryption key itself and is not subject to any transformations, so when using the keyfile, keep it in a safe place.

Graphical user interface

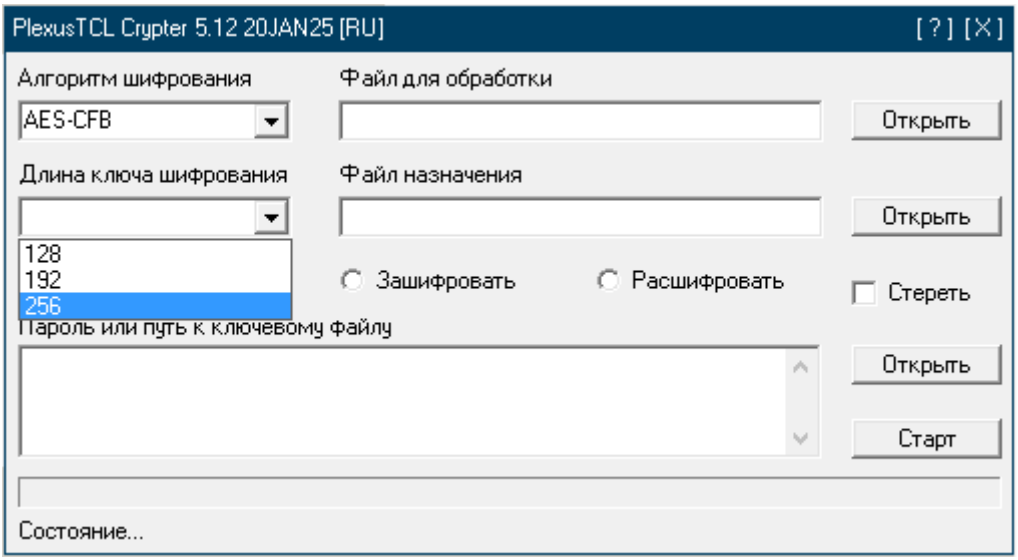
GUI exists to control the program with a computer mouse, since not all users successfully work in the command line, and many are simply irritated by the need to constantly type commands. In addition, GUI can be beautiful, elegant, strict, intuitive and simply pleasant to work with. To use the program "PlexusTCL Crypter" to process a file, you need to do the following.

1.) Run PlexusTCL Crypter 5.12.exe

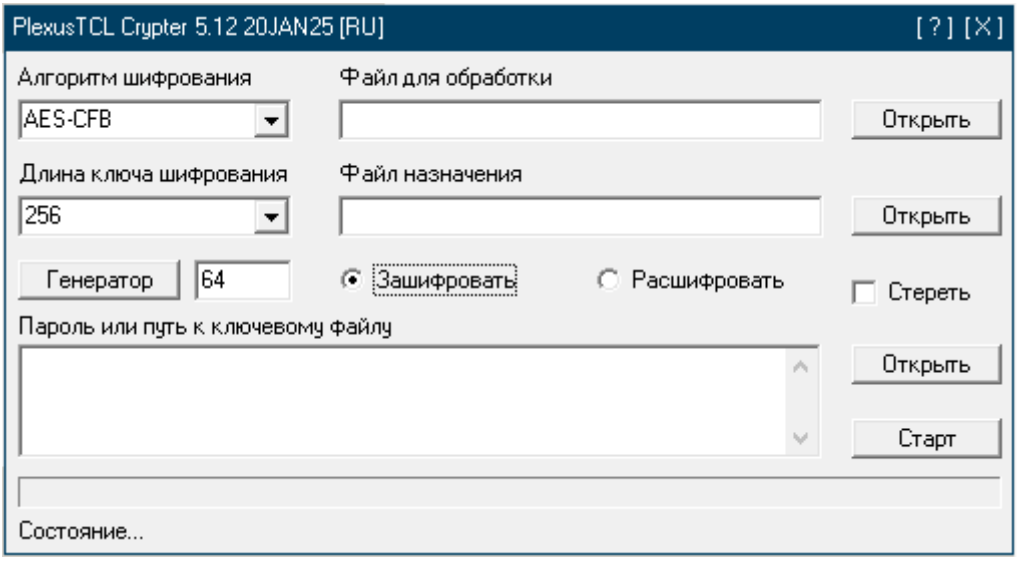
2.) Select the encryption algorithm you need from the drop-down list as shown in the figure below.



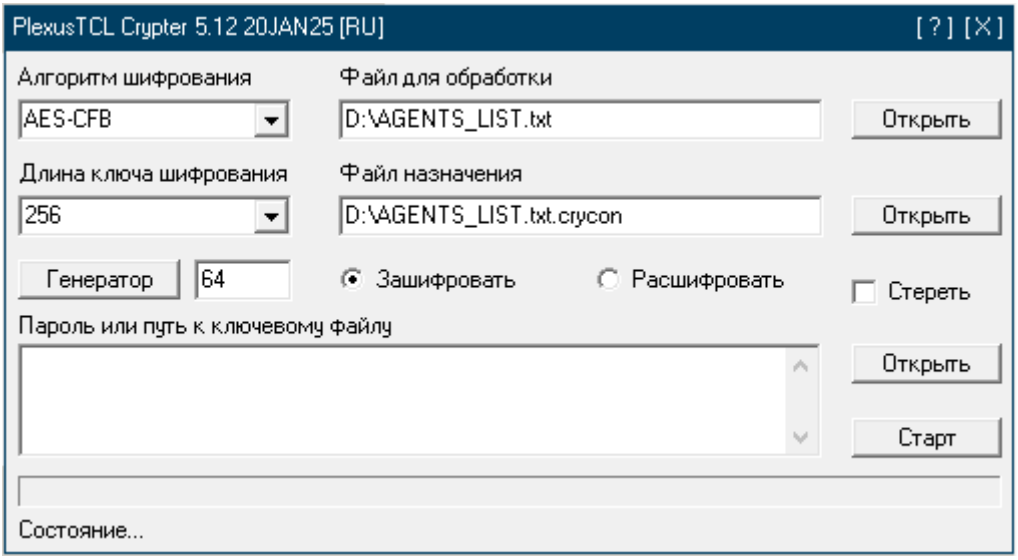
3.) If a field for selecting the encryption key length appears, select the desired key length from the drop-down list. The longer the encryption key length you select, the greater the cryptographic strength, but the generation of the encryption key, as well as the encryption itself, may take longer.



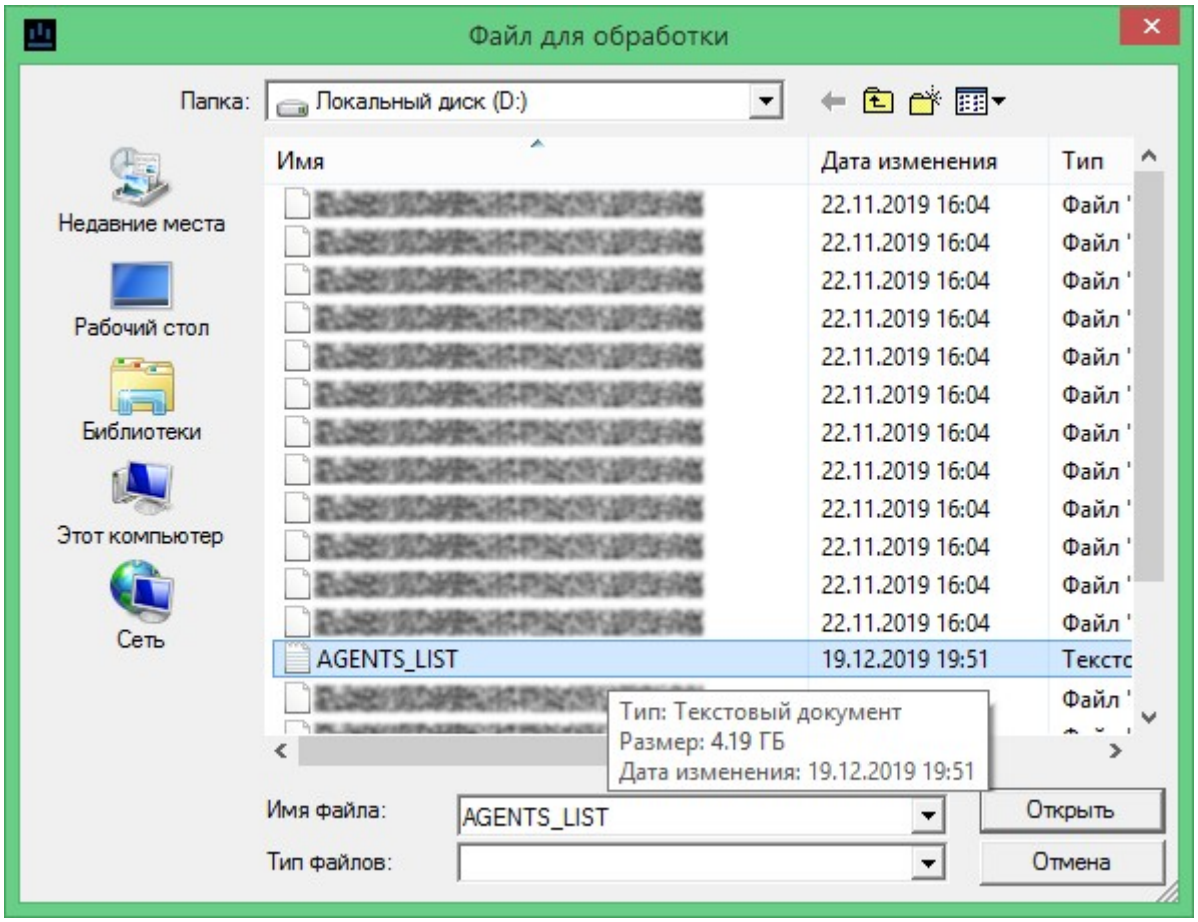
4.) Select the desired action (encrypt/decrypt). If you are using the Blowfish block cipher, the field for selecting the encryption key length will not appear.



5.) Enter the names of the files you want to process and the one to which the processed data will be written in the fields titled "File to process" and "Destination file". The lines in the fields should not match, which means that the destination file name should be different from the name of the file being processed and should not match the name of any file in the destination directory (unless you are going to overwrite an existing file).

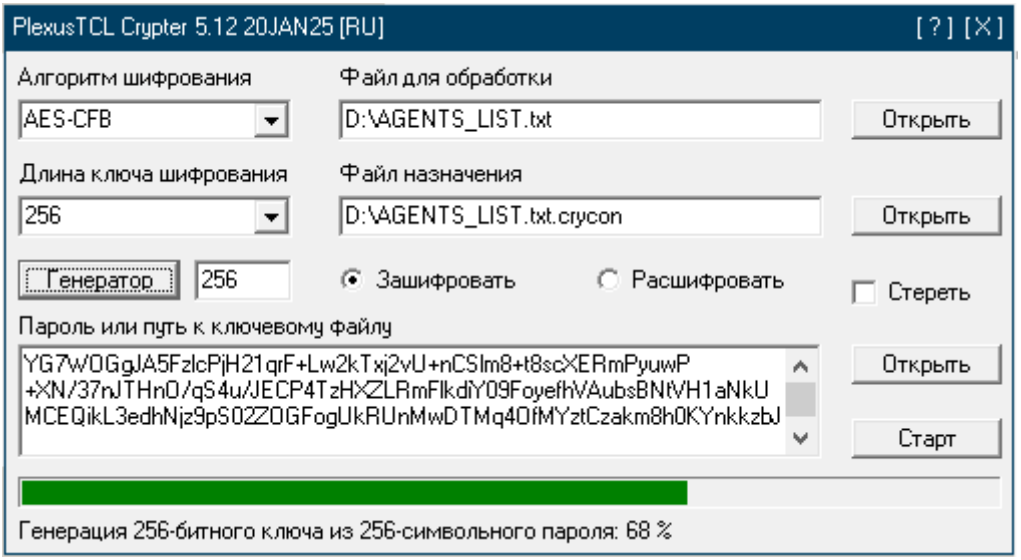


You need to enter the file name with its extension if the file being processed is located in the same directory as the program that will work with it, since the program cannot determine the extensions of the files it works with. If the file is located in another directory, you need to enter the file name together with the full path to it. If you do not want to enter the file name manually, you can call up the file selection dialog box by clicking on the button labeled "Open" to the right of the corresponding field.

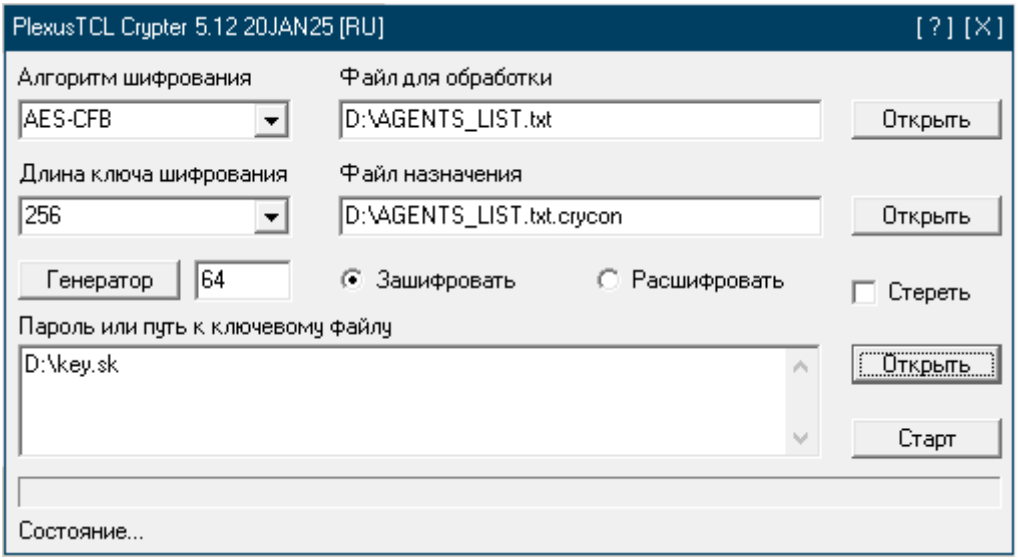


The file selected for processing is the file whose name appears in the "File name" field and which simultaneously becomes selected. To approve a file for processing, click "Open" and its name together with the full path to it will appear in the corresponding field. The name of the file to which the processed data will be saved is entered in the "Destination file" field or selected in the same way as the file for processing, after clicking the "Open" button to the right of the corresponding field. Each button corresponds to the field to the left of it. Starting with version 4.91, the program, having detected a match between the names of the file being processed and the destination file, automatically adds the string ".crycon" to the end of the destination file name to save the user from manually changing the destination file name.

6.) If you want to use a simple string or passphrase as the encryption key, enter it in the field labeled "Password or path to keyfile". There is no need to worry about the security of using a string as the encryption key, since the string is not used as the encryption key. The string will be converted into an encryption key by the key generation function KDFCLOMUL (original development) based on the SHA- 2-256 hashing algorithm . In the example below, the password is a string of 256 pseudo-random uppercase, lowercase Latin letters, Arabic numerals and special characters, obtained using the built-in generator, which can generate strings from 8 to 256 characters.

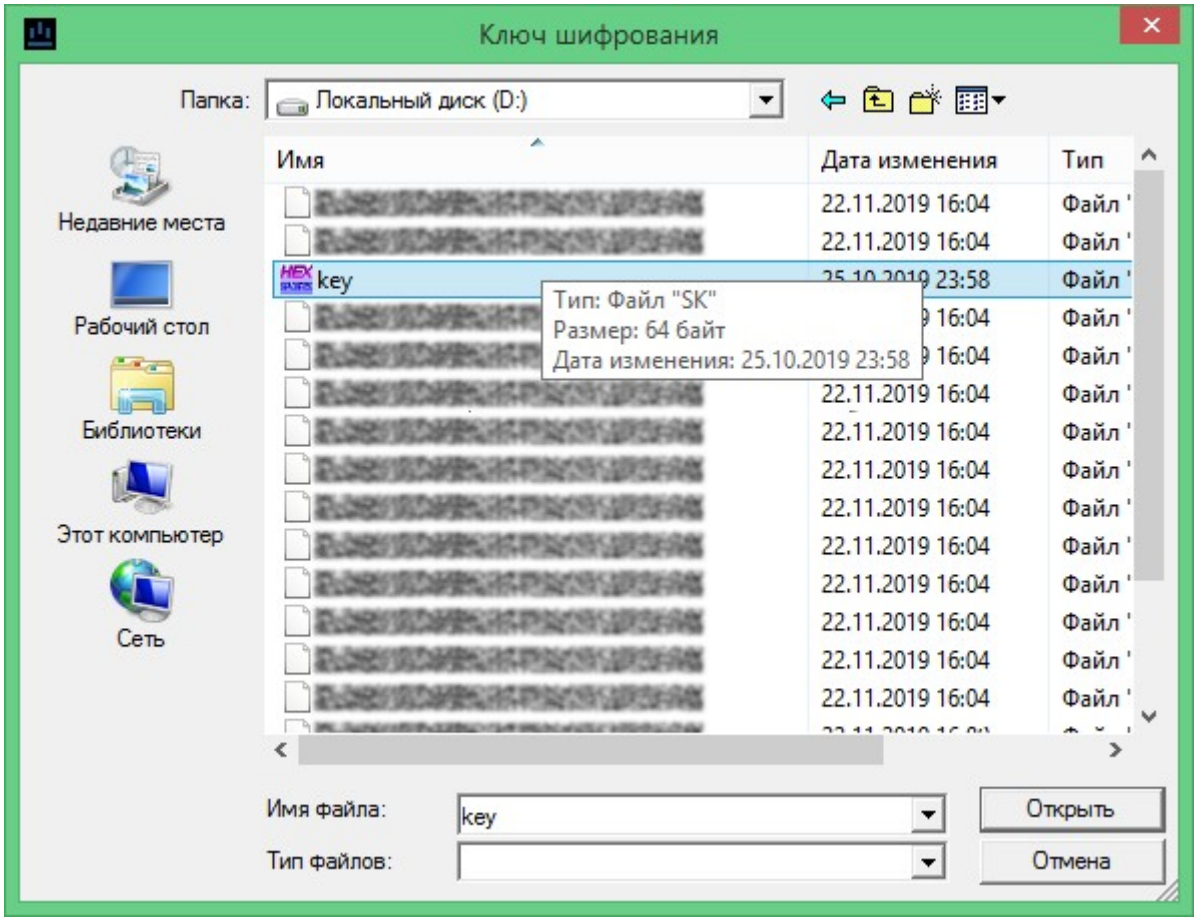


The example above shows the progress of generating an encryption key from a 256-character password. Generation starts only after pressing the "Start" button, when the password is entered, its length is from 8 to 256 characters and the program is completely ready to process the data.

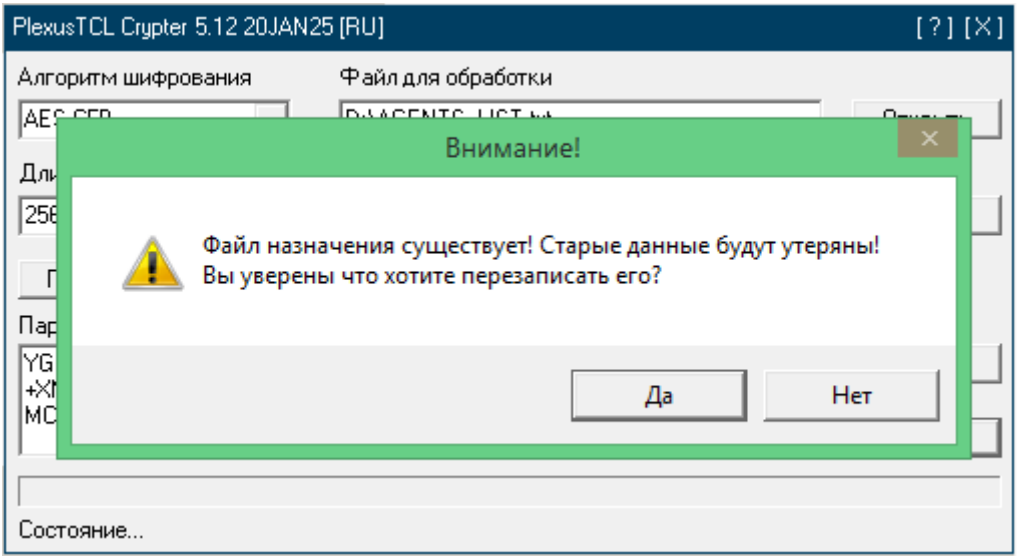


If you want to use data from a file as an encryption key (the file can be any file, but its size must be greater than or equal to the encryption key length in bytes), enter the file name in the field. It is important to note that the data from the key file is NOT subject to any changes - the data from the key file is considered the encryption key itself. This is done so that the user can use truly random data stored in the file as a key. If you decide to use a file as a source of the encryption key, hide this file away and do not tell anyone about it.

You can also call up the key file selection dialog box (as in the image below) if you want to select one of multiple files by clicking "Open" next to the key entry field. The file is considered approved in the same way as in the example with selecting the file to be processed.

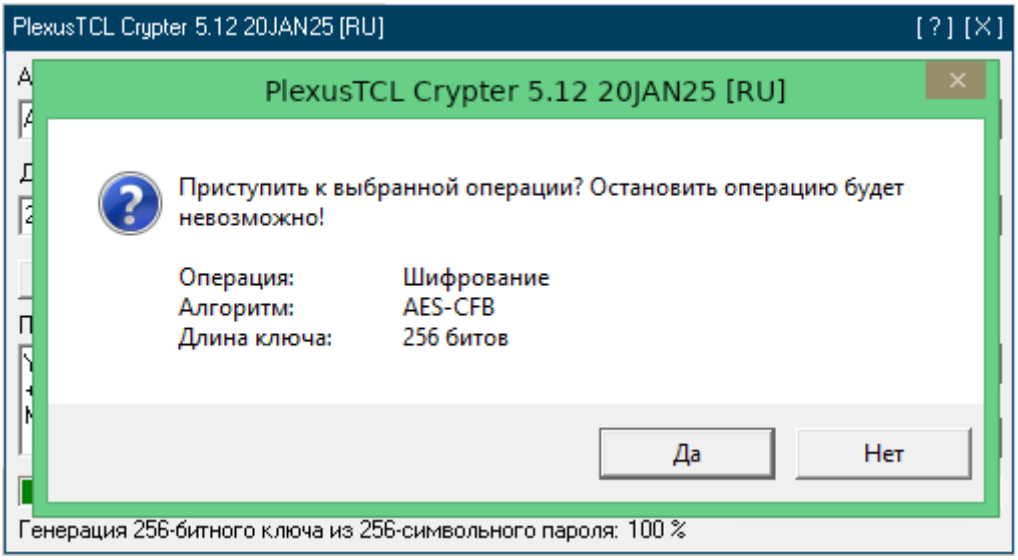


7.) Click "Start" to begin the encryption/decryption operation. If you make a mistake, the program will notify you of this by displaying a text message. The program will also notify you that an existing file has been selected as the destination file and will offer to overwrite it.

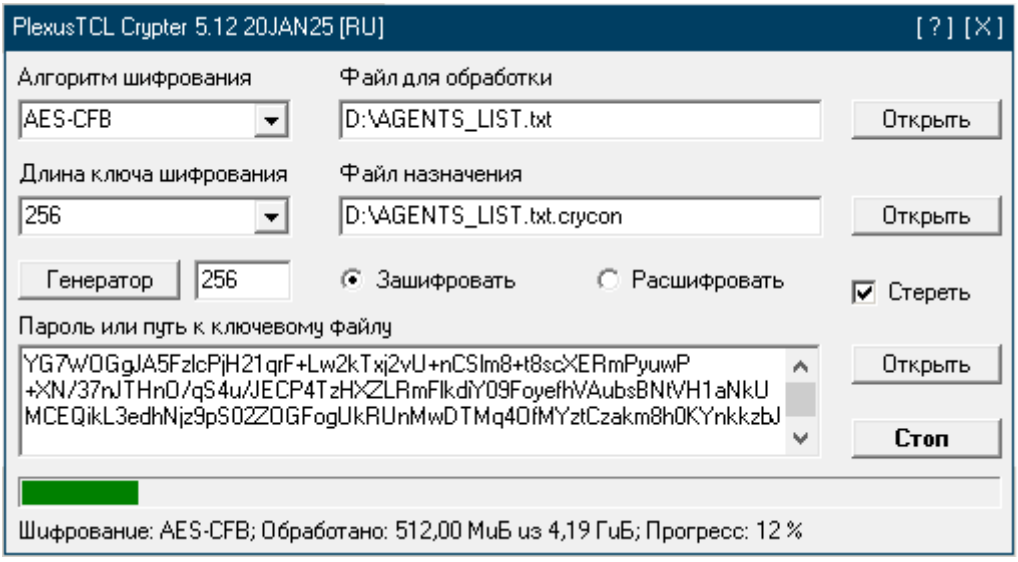


If you agree to overwrite an existing file, the program will immediately destroy all data in the destination file by writing the processed data from the file to be processed, over the old data. In case of matching strings in any two or more fields, namely in the fields "File to be processed", "Destination file" and "Encryption key or path to key file", the program will abort the operation and notify about the equality of strings, because it is not allowed.

If no errors were made and the program is ready to process the file, the program will notify you of this by displaying a dialog box.

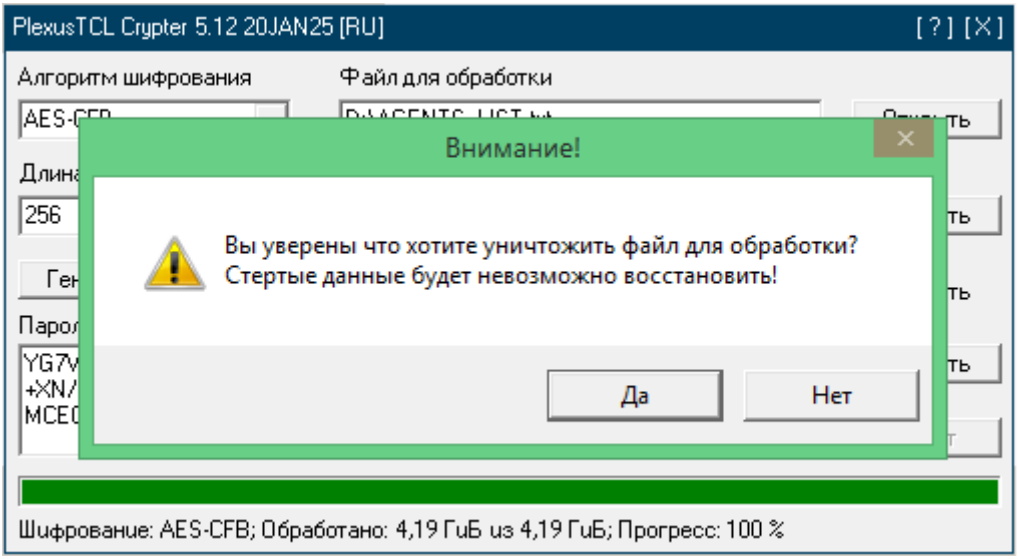


The operation being performed, the algorithm used, the amount of data processed and the percentage of progress will be visible in the bottom field, as in the figure below.



When the operation is completed, the program will notify you by displaying a message. Please note that during the encryption key generation operation, the program disables the "Start" button. In all programs below version 5.09, the button was disabled before the encryption operation began. By the release of version 5.09, the author of the program finally deigned to study mutexes, semaphores and finally got around to studying critical sections by data, so with the release of this version, encryption can not only be paused but also canceled. From the beginning of file encryption, the "Start" button changes to "Stop". Cancellation of the running operation is carried out by pressing the "Stop" button, but it does not delete the created destination file from the disk.

Before starting the encryption process, you can check the " Erase " box if you want to destroy the file for processing after it is processed. If the box is checked, then after the encryption is complete, you will be shown a dialog box in which you can choose whether to destroy the file for processing or not.



If the checkbox is checked and you agree to delete the file, the file to be processed will be overwritten with zeros, its size will be truncated to zero and it will be deleted by default. the DeleteFile function from the Kernel32.dll library .

Warning! Safe overwriting of the processed file (irreversible deletion) means that the file system on your computer's hard drive allows you to overwrite the file from its beginning to the end so that the information on the hard drive sectors is completely overwritten. This is usually true, but many file systems, such as JFS, ReiserFS, XFS, Ext3, RAID - based file systems , file systems that cache data during processing, do not allow you to completely overwrite a file due to their architecture. When using " PlexusTCL Crypter", avoid these file systems, since in case of their use, overwriting of data is not only not guaranteed, but may also be impossible. **If you do not trust this program to destroy sensitive data for you, then use the free BleachBit program or the SDelete utility by Mark Russinovich, which you can download from the official website of Microsoft Corporation.**

8.) To decrypt the file, fill in the fields the same way as for encryption, but select "Decrypt" before clicking the "Start" button.

Source codes

crycon and sha256sum programs are written in the C programming language and compiled into executable files using the GCC compiler version 10.2.1-6 20210110.

The GUI under Windows OS is written in C , C ++ and x86 assembly languages, and compiled into an executable file by the C++ Builder rapid application development environment compiler in version 6.0 .

All executable files in package " PlexusTCL Crypter" and their source codes, are distributed under the GNU GENERAL PUBLIC LICENSE version 3, dated June 29, 2007.

Below are the SHA -2-256 checksums of all programs included in "PlexusTCL Crypter":

File name	SHA-2-256 checksum
PlexusTCL Crypter 5.12[EN].exe	9f429f9d74d91fa52553cb9a63b38647 662be1ccfbf274c364aa00d767bc0234
PlexusTCL Crypter 5.12 [RU].exe	e 8252183323a724313fafb90f9792ade 5b244cca91dbcabddc9eeecc6ed37f4e
crycon	a 8e1786831f6f272756021fe386e44ad daae313133834a431b4d34e5b4700b74
sha256sum	e 4d6ab5c1cd09150b4c2ad9db602737c 8a52b9387fb9488928862ed19bea745e
sha256sum.exe	01ddbf8b37961cbac3b149536fca0c3e 319f4fcd34a887f0d775bc472c0b1cb5

DE, SE, DA, VE, Plexus Technology Cybernetic Laboratory, 2010 – 2024
[74526B00E85297526D979BA9D8DF9182C4C15FA63EAE1EBC667A88D86AE58228]