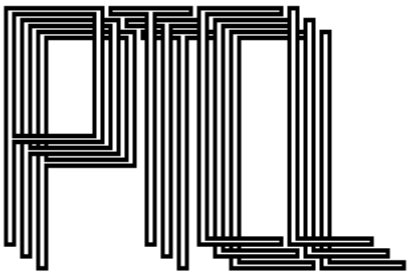


# PlexusTCL Crypter

версия 5.00 от 08 августа 2021 года

[данная версия программы несовместима со всеми предыдущими версиями]

**ВАЖНЫЕ ПРЕДУПРЕЖДЕНИЯ**

**1.) ЕСЛИ ВЫ НЕ УВЕРЕНЫ В СТАБИЛЬНОСТИ РАБОТЫ ПРОГРАММЫ ИЛИ ВАШЕГО КОМПЬЮТЕРА, ТО ПЕРЕД ШИФРОВАНИЕМ ФАЙЛА, ОБЯЗАТЕЛЬНО СДЕЛАЙТЕ ЕГО РЕЗЕРВНУЮ КОПИЮ.**

**2.) ПРОГРАММА НЕ ГАРАНТИРУЕТ ТАЙНУ, ЕСЛИ ВЫ ДОПУСКАЕТЕ ВОЗМОЖНОСТЬ АТАК ПО СТОРОННИМ КАНАЛАМ, ТАКИХ КАК ОТПЕЧАТОК (ДАМП) ОПЕРАТИВНОЙ ПАМЯТИ ВО ВРЕМЯ РАБОТЫ ПРОГРАММЫ, ПОПАДАНИЕ КЛЮЧЕЙ ШИФРОВАНИЯ В ФАЙЛ ПОДКАЧКИ, ЗАРАЖЕНИЕ КОМПЬЮТЕРА ВРЕДОНОСНОЙ ПРОГРАММОЙ, ДЛИННЫЙ ЯЗЫК ВАШЕГО СИСТ. АДМИНИСТРАТОРА И Т.Д.**

**3.) ЧТОБЫ ОБЕСПЕЧИТЬ МАКСИМАЛЬНО ВОЗМОЖНЫЙ УРОВЕНЬ БЕЗОПАСНОСТИ ПРИ ПРИМЕНЕНИИ НАСТОЯЩЕГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, ПРОКОНСУЛЬТИРУЙТЕСЬ СО СПЕЦИАЛИСТАМИ В ОБЛАСТИ КОМПЬЮТЕРНОЙ БЕЗОПАСНОСТИ, ТАК КАК В ПРОГРАММЕ ПРИСУТСТВУЮТ ОРИГИНАЛЬНЫЕ РАЗРАБОТКИ.**

**4.) ПРОГРАММА СОЗДАЕТ БЕЗСИГНАТУРНЫЕ ШИФРОВАННЫЕ ФАЙЛЫ, ЧТО ДЕЛАЕТ НЕВОЗМОЖНЫМ ИХ ИДЕНТИФИКАЦИЮ КАК ЗАШИФРОВАННЫХ, ЧТО НЕ ПОЗВОЛЯЕТ ОТЛИЧИТЬ ИХ ОТ СЛУЧАЙНЫХ ДАННЫХ. ЕСЛИ ВАМ ВАЖНО НАЛИЧИЕ СИГНАТУР, ИСПОЛЬЗУЙТЕ ДРУГУЮ КРИПТОГРАФИЧЕСКУЮ ПРОГРАММУ.**

Программа "PlexusTCL Crypter" предназначена для криптографической защиты информации, путем шифрования файлов, размером до (2 Гб — 40 байт) включительно. Файлы могут быть обработаны (зашифрованы/расшифрованы) пятью криптографическими алгоритмами по выбору пользователя, а именно Rijndael-128 (AES), Serpent, Twofish, Blowfish и Threefish, с использованием ключевого файла или введенной в качестве пароля строки.

## Программа и алгоритмы

Программное обеспечение представляет собой исполняемый файл, в котором реализованы алгоритмы шифрования и система управления вводом/выводом информации в виде вызываемых функций. GUI (графический интерфейс пользователя) интегрирован в исполняемый файл, так что сменить интерфейс будет проблематично. Если вас интересует смена интерфейса или алгоритмов, вы всегда можете пересобрать проект из исходных текстов.

Все реализации криптографических алгоритмов, а именно AES, Serpent, Twofish, Blowfish и Threefish, протестированы с помощью тестовых векторов, опубликованных их разработчиками, а значит полностью соответствуют своим математическим описаниям или опубликованным стандартам.

При использовании блочного шифра из программы "PlexusTCL Crypter", важно учитывать тот факт, что шифр AES отличается от шифра Threefish тем, что Threefish спроектирован как 64-битный шифр, т.е оперирующий 64-битными (8 байтными) числами как самостоятельными единицами, в то время как AES

оперирует блоками данных, состоящими из 8-битных (1 байтных) значений. Это значит, что AES шифрует открытый текст блоками битов, обрабатывая по 8 битов (1 байту) за операцию, в то время как Threefish принимает 64 бита (8 байтов) открытого текста за одно большое число, и оперирует им как одним целым. По этому, скорость работы Threefish на 64-битных процессорах, в разы больше скорости его работы на 32-битных процессорах, но скорость работы на 32-битных процессорах, в 2 — 3 раза ниже, чем скорость работы AES. Эти параметры могут изменятся в зависимости от используемого при сборке проекта компилятора, а так-же параметров оптимизации кода. По этому, в случае если важна скорость шифрования, рекомендуется использовать Threefish на 64-битных процессорах, а AES на 8, 16 и 32-битных. Это относится только к алгоритмам, но не к их реализациям в настоящей программе.

Шифр Threefish, начиная с версии программы 4.91, оптимизирован и дополнен своими младшей и старшей версиями, принимающими 256-битные и 1024-битные ключи, чего не было в ранних версиях программы. Это значит, что если вы зашифровали файл, используя программу версии 4.91 или старше, выбрав для шифрования алгоритм Threefish с 256-битным или 1024-битным ключом, расшифровать такой файл программой версии 4.90 или младше, не получится. Команда под руководством Брюса Шнайера разработала шифр Threefish для использования в хэш-функции Skein как легкий в реализации, настраиваемый блочный шифр, построенный из простых арифметических операциях. Его можно использовать для шифрования любых данных с ключами любой длины — этот шифр действительно очень хорош.

Шифр AES начиная с версии 4.92, так-же оптимизирован, а скорость его работы на некоторых компьютерах, сравнима со скоростью работы шифра Threefish (около 4.5 МеБ/сек).

Использовать шифры Blowfish, Twofish и Serpent можно на любом 32 или 64-битном процессоре, так как эти шифры показывают почти одинаковую производительность на обоих, к тому же их реализации очень быстрые по сравнению с Threefish и тем более AES. Blowfish шифрует данные блоками по 32 бита, шифруя сразу 2 части открытого текста в виде 64-битного блока данных, принимая на вход левую и правую 32-битные части данных по отдельности, заменяя их шифротекстом, как и любые другие шифры спроектированные на основе сети Фейстеля. Так как шифры Blowfish, Twofish и Serpent спроектированы 32-битным, это может сказаться на производительности при использовании шифра на 64-битных платформах в лучшую сторону, но разработчик не заметил разницы. Скорость работы обоих шифров на 32-битном и 64-битном процессорах почти одинаковая.

Знайте, что шифры реализованные в программе, для которых нельзя выбрать длину ключа шифрования, реализованы так, что принимают только ключи максимально возможной длины, что относится только к шифру Blowfish. Ранее это было справедливо и для шифра ARC4, но начиная с версии 5.00 этот шифр был вырезан из программы как безнадежно устаревший, оптимизация же шифра AES такова, что не уступает ARC4 в скорости обработки данных. С обновлением реализации AES, использовать ARC4 стало незачем.

Все пять блочных шифров, а именно AES, Threefish, Blowfish, Twofish, и Serpent, работают в режиме CFB (режим обратной связи по шифротексту), что обеспечивает достаточно надежный уровень безопасности применения блочного шифра, превращая блочный шифр в поточный. Не стоит беспокоится на этот счет, так как правильно использованный блочный шифр в виде поточного, ничем не уступает в криптостойкости блочному шифру. При шифровании данных в режиме CFB, блочный шифр вообще не используется для шифрования данных а используется для генерации псевдослучайной последовательности битов, путем шифрования постоянно меняющегося массива, которая складывается по модулю 2 с каждым битом шифруемых данных. Кстати, использование режима CFB позволяет не реализовывать функцию расшифровки, что упрощает реализацию и использование шифра, но в исходных текстах программы, функции расшифровки все равно реализованы.

Чтобы зашифровать что-либо в помощью блочного шифра, работающего в режиме CFB, сначала необходимо сгенерировать случайную или псевдослучайную последовательность, называемую IV (вектором инициализации), зашифровать IV с помощью блочного шифра, чтобы получившуюся последовательность побитно сложить по модулю 2 с шифруемым блоком данных, получая шифротекст. После окончания шифрования первого блока данных, в качестве IV при шифровании каждого последующего блока, используется предыдущий зашифрованный блок, что обеспечивает лавинное изменение всего шифротекста при изменении даже 1 бита ключа шифрования, шифруемого блока или IV. Если выразить режим CFB символами, то получится что-то такое:

```
iv   = random(time(now));
c[0] = e(iv, key);
c[i] = p[i] xor e(c[i-1], key); от i=1 до i=n;
```

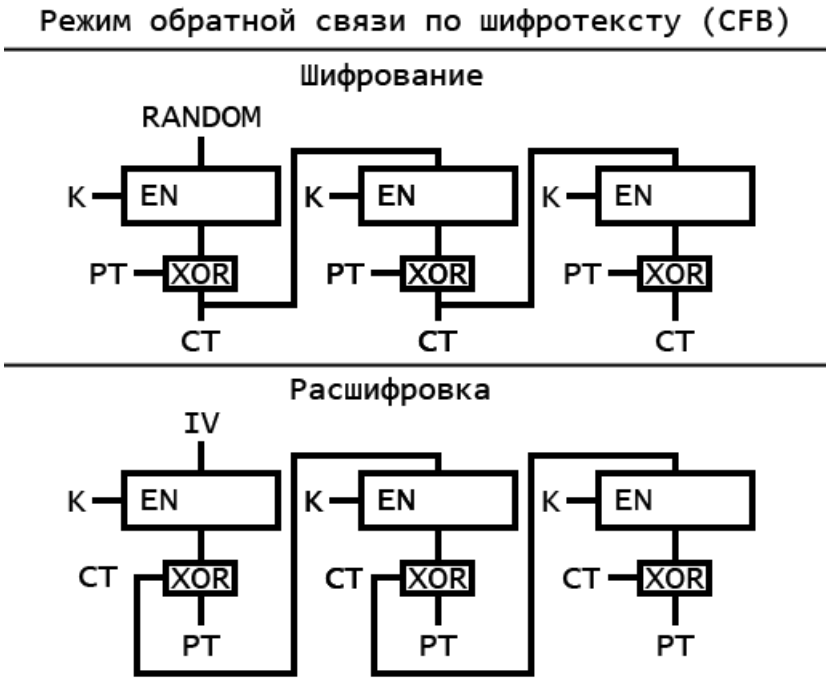
**где:**

**random – генератор псевдослучайных чисел вашей ОС**
**time – системное время вашей ОС в секундах**
**now – реальное время**
**xor – операция побитового исключающего ИЛИ**
**key – ключ шифрования**
**iv – вектор инициализации**
**n – длина шифротекста в блоках**
**i – увеличивающийся на единицу счетчик**
**p – открытый текст**
**c – шифротекст**
**e – функция шифрования**

Даже при шифровании файла размером 2 Гб, полностью состоящего из нулевых битов, используя в качестве ключа шифрования и IV последовательность нулей, зашифрованный файл будет выглядеть нагромождением случайных данных, не имеющих никакой закономерности. В программе "PlexusTCL Crypter", режим CFB реализован так, что **в качестве IV, используется зашифрованная псевдослучайная последовательность**, записываемая в файл перед первым зашифрованным блоком, что необходимо для расшифровки и нисколько не сказывается на криптостойкости, так как знание IV ничем не поможет при взломе шифротекста без знания ключа, а **ключ всегда должен быть самым охраняемым секретом.** Знание того, что IV записан в файл первым зашифрованным блоком, никак не поможет взломать шифротекст, если не известен ключ.

При шифровании двух открытых текстов, первые шифруемые блоки которых совпадают, уникальный для каждого открытого текста вектор инициализации, используется в режиме CFB для сокрытия этого совпадения, если для шифрования используется один и тот-же ключ. Псевдослучайный для каждого открытого текста IV, позволяет использовать **"долгосрочный ключ"**, т.е один ключ для шифрования множества открытых текстов, даже если они совпадают. Так как в настоящей программе в качестве IV используется зашифрованная псевдослучайная последовательность, генерируемая используемой ОС, то это представляет угрозу только в том случае, если будет атакован ГПСЧ (генератор псевдослучайных чисел). К примеру, если в ОС остановлено системное время, то генератор псевдослучайных чисел будет бесконечно генерировать одну и ту же последовательность чисел от 0 до 255 (от 0x00 до 0xFF), так как "зерном" для генератора выступает реальное системное время в секундах. Если злоумышленник знает открытый текст, шифрование которого дает IV, то злоумышленник может попытаться восстановить ключ, и расшифровать все сообщения, зашифрованные тем же ключом. Если ГПСЧ будет генерировать одни и те же числа, то шифрование одинаковых блоков открытого текста при использовании одного и того же ключа, будет давать одинаковые шифротексты, что дает возможность приблизиться к взлому всех шифротекстов. Чтобы этого избежать, нужно следить за тем, что делает прочее программное обеспечение на используемом компьютере (не подменяет ли оно что-нибудь), избегать использования "пиратских" копий программного обеспечения которые могут быть заражены вредоносным кодом и никогда ничего не шифровать в случае получения сообщения "Критическая ошибка ГПСЧ!", что означает некорректную работу ГПСЧ (три псевдослучайных байта

IV равны, а такого никогда не должно быть). Чтобы увеличить случайность и одновременно уменьшить вероятность повторения значений в IV, его первые два байта складываются по модулю два с координатами курсора мыши по осям X и Y, которые вычисляются в момент нажатия кнопки "Старт", что вносит в IV элемент неопределенности (злоумышленник не может сказать, в каком именно положении был курсор мыши в момент нажатия кнопки). Так-же, первые четыре байта IV складываются по модулю два со случайным значением из программного стека, что вносит в IV еще больше неопределенности (какое значение имел кадр стека, к примеру по адресу 0x00408A4E, злоумышленник конечно-же установить не сможет).




**Функция формирования ключа**

Функция формирования ключа (KDF) создана для того чтобы "растянуть" пароль пользователя и сгенерировать из него ключ шифрования. Все KDF устроены так, чтобы усложнить задачу перебора паролей, даже если для перебора используется целый серверный парк из тысяч машин, сделав перебор относительно долгим.

Все KDF устроенны одинаково. Это всегда функция, которая принимает на вход некое X (Икс), и из этого X, по завершении работы, инициализирует K (Ка), которое является ключом шифрования для блочного или поточного шифра. Типичная KDF принимает на вход алгоритм, которым будут обработаны данные, пароль пользователя, его длину, соль, длину соли, количество итераций обработки данных а так-же желаемую длину ключа. Если записать символами то, что делает любая KDF, то получится что-то подобное:

**K = KDF(alg, pass, plen, salt, slen, count, klen);**

**Где:**

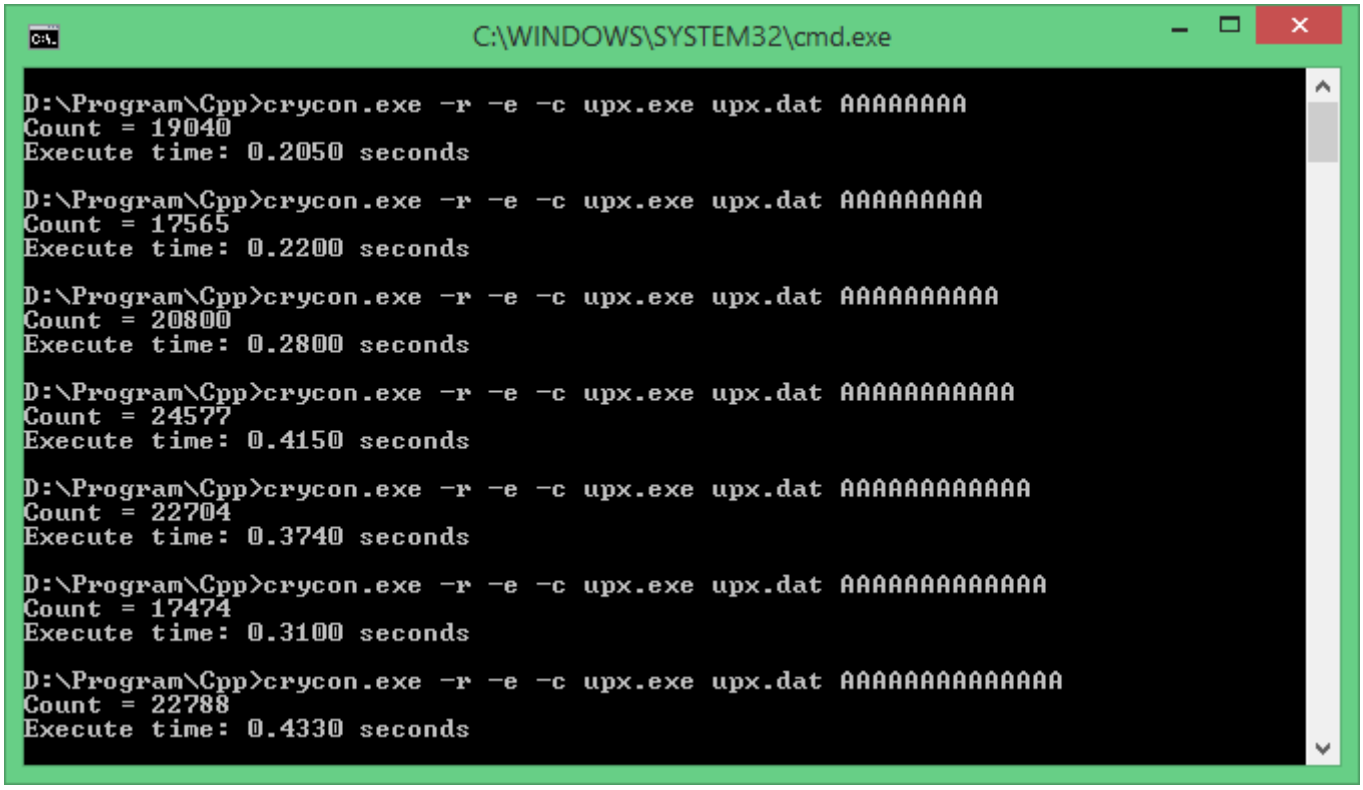
**K – ключ шифрования;**
**alg – алгоритм обработки пароля и соли;**
**pass – пароль пользователя;**
**plen – длина пароля;**
**salt – псевдослучайная константа (соль);**
**slen – длина соли;**
**count – количество итераций обработки пароля и соли;**
**klen – желаемая длина ключа шифрования;**

в итоге, K будет содержать ключ шифрования, который поступает на вход функции шифрования/расшифровки для обработки каких-либо данных. Разберем типичную KDF по порядку.

Сначала KDF принимает алгоритм обработки данных, который будет обрабатывать пароль пользователя и соль столько раз, сколько указано в аргументе count. Обычно это алгоритм хэширования, такой как MD5, SHA-2-512, Keccak (кечак), или алгоритм шифрования, такой как Salsa20/20, DES, Rijndael, Serpent, IDEA и т. д. Количество итераций обработки пароля и соли обычно задают статичным, которое иногда увеличивают в зависимости от быстродействия компьютерного парка потенциального злоумышленника, перебирающего пароль. Но значение count рано или поздно должно вырасти, так как производительность компьютеров все время растет, согласно наблюдению Гордона Мура. Значение count на 2021 год обычно от 150,000 до 2,000,000, и всегда зависит от производительности функции шифрования или хэширования, обрабатывающей пароль и соль. Все KDF обычно устроены так, что на разных процессорах генерируют ключ шифрования около одной или двух секунд. Чтобы сгенерировать ключ для расшифровки, к примеру записи в базе данных, пользователю нужно вызвать KDF один раз и подождать всего одну или две секунды, в то же время злоумышленник вынужден будет вызвать KDF миллиарды раз, чтобы перебрать все возможные пароли. Перебирая множество вариантов относительно небольшого пароля, длиной всего 8 – 10 символов, печатаемых на стандартной клавиатуре, злоумышленник потратит количество времени, которое просто не проживет.

Соль, или псевдослучайная константа, нужна для того, чтобы как-либо смешав ее с паролем, многократно увеличить количество вариантов пароля, что многократно усложняет перебор, делая его очень долгим и крайне не выгодным занятием. Разработчик же исключил соль из своей KDF, так как ему не понравилось слово "соль".

В настоящей программе KDF является оригинальной разработкой, так как автор программы есть не кто иной, как "параноик страдающий шпиономанией". Это означает, что ключ шифрования генерируется криптографически стойкой хэш-функцией SHA-2-256, исходный код которой на языке Си, подготовлен NIST (Национальным Институтом Стандартов и Технологий США), длина пароля ограничена только разрядностью регистра процессора пользователя, и на сегодняшний день составляет минимум 4,294,967,295 символов, а **количество итераций хэширования является динамическим**, и генерируется хитрым сплетением битовых операций и вычислением 32-битного циклического избыточного кода (CRC32) различных частей пароля. На рисунке ниже видно, что при увеличении статичного пароля всего на 1 символ, меняется не только количество итераций хэширования, но и время формирования ключа.




Чтобы специалистам было проще понять, как устроена генерация ключа шифрования в настоящей программе, ниже приведен Си код функции KDFCLOMUL:

```
void KDFCLOMUL(SHA256_CTX    * sha256_ctx,
               const uint8_t * password, const size_t password_len,
                     uint8_t * key,      const size_t key_length) {

uint32_t i, j, k, count = 0;
uint8_t  hash[SHA256_BLOCK_SIZE] = {0x00};

  for (i = 0; i < password_len; ++i) {
    count ^= (uint32_t)(CRC32(password, i) + CLOMUL_CONST);
    count -= (password_len + key_length + CLOMUL_CONST + i);
  }

  count  &= CRC32(password, password_len);
  count >>= 18;
  count  |= (uint32_t)1 << 14;
  count  *= CLOMUL_CONST;

  sha256_init(sha256_ctx);

  for (i = k = 0; i < key_length; ++i, ++k) {
    for (j = 0; j < count; ++j) {
      sha256_update(sha256_ctx, password, password_len);
    }

    sha256_final(sha256_ctx, hash);

    if (k >= SHA256_BLOCK_SIZE) {
      k = 0;
    }
    key[i] = hash[k];
  }
}
```

Почему количество итераций хэширования сделано не статическим, как у людей? Чтобы полностью исключить возможность перебора пароля, привязав к количеству итераций хэширования не только пароль с его длиной, но и длину ключа, сделав перебор пароля невозможным в принципе. Но и не только длина пароля влияет на количество итераций, но и каждый байт пароля, что

превращает KDF из обычной функции в "непроходимый таежный лес" хэширования. Если сложность (время) атаки методом перебора увеличивается с количеством итераций хэширования, а это количество зависит не только от длины пароля но и от каждого его байта, то чтобы узнать количество итераций хэширования и попытаться составить схему перебора пароля, нужно знать не только длину пароля но и каждый его символ, что сводит взлом пароля к знанию самого пароля.

Данная KDF получила название KDFCLOMUL, от константы CLOMUL_CONST (Clock multipler), которая все-же позволяет настраивать, насколько большим будет количество итераций хэширования, а значит и время вычисления ключа. Она описана в header файле "clomul.h", который легко найти в каталоге "src", описание же переведено на английский язык средствами Google Translate (оно почти такое же как описание ниже).

Константа CLOMUL_CONST обозначает так называемый "тактовый множитель", использующийся в функции KDFCLOMUL как один из операндов при формировании количества итераций хэширования. Если описывать константу простыми словами, то чем больше ее значение, тем дольше генерируется ключ из пароля, так как для генерации ключа требуется намного больше тактов процессора. Он необходим для генерации ключа шифрования из пароля, так как влияет на количество итераций цикла вычисления хэш-суммы пароля. Сам же цикл использует алгоритм хэширования SHA-2-256. По умолчанию "тактовый множитель" равен 1, но увеличение до значений 12, 16, 38 или 67 заставляет любое оборудование генерировать ключ очень медленно, что делает атаку полным перебором даже на короткий 8 символьный пароль невозможной за приемлемое время. Обращайтесь с этой константой осторожно, так как неправильно подобранное значение может привести к тому, что вы будете ждать окончания генерации ключа из относительно короткого пароля минуты или даже часы.

В случае, если "тактовый множитель" будет слишком большим, то переменная содержащая количество итераций хэширования может переполнится. На практике это означает десятки минут или даже часов ожидания, так как знаковая 32-битная переменная вмещает максимальное число 2,147,483,647. В случае переполнения, значение переменной становится отрицательным, и цикл хэширования становится чудовищно долгим из-за того, что переменная сначала будет расти от отрицательного значения до нуля, и только потом начнет увеличиваться от нуля до заданного предела. Выбирайте значение исходя из предполагаемой общей мощности компьютерного парка вашего противника, так как оборона, адекватная атаке, выстраивается только если известны возможности атакующего. На 2021 год, значение можно сделать от 5, этого хватит еще лет на 5-10. Максимальное же значение 57,344. Никогда не превышайте его!

Значение "тактового множителя" равное 1, обеспечивает генерацию ключа шифрования из пароля "password" на процессоре Intel Core I3-3217U с тактовой частотой 1.8 GHz где-то за 0.18 секунды, что уже превращает полный перебор всех 8 символьных паролей в адский кошмар, так как для перебора всех паролей состоящих только из строчных латинских букв, понадобится перебрать 208,827,064,576 вариантов пароля. Так как данный процессор способен перебрать 5 паролей в секунду, то полный перебор всех паролей на выше указанном процессоре займет 79,462 года. Но учтите, что графические процессоры в разы быстрее обычных, и перебор пароля можно выполнять параллельно на десятках тысяч машин, так что используйте длинные парольные фразы (20 — 30 символов). Но не делайте пароли слишком длинными (40, 50 или 100 символов), так как время генерации ключа прямо пропорционально длине пароля — чем длиннее пароль, тем дольше выполняется генерация ключа. Обратите внимание, что графический процессор стоимостью $1000, на начало 2020 года, может вычислить более 10 миллиардов хэш-сумм SHA-1 за одну секунду, так что не все так радужно, как обычно представляется из цифр.

В случае, если используется базовый "тактовый множитель" равный 1, то количество итераций хэширования пароля составляет около 20,000 на один байт ключа шифрования, и может иметь максимальное значение 32,767, а

время хэширования пароля "password" на вышеуказанном процессоре составляет около 0.18 секунды, что является идеальным компромиссом между безопасностью и быстродействием на сегодняшний день. Если пароль хэшируется около 100,000 раз для генерации всего 5 байт ключа шифрования, то при генерации 256-битного (32-байтного) ключа шифрования, пароль будет хэширован около 640,000 раз. Неплохой результат.

Как показал тест на вышеуказанном процессоре, при "тактовом множителе" равном 64, генерация 256-битного ключа шифрования для алгоритма Rijndael, при использовании пароля "password", длится целых 11 секунд. Чтобы избавить пользователя от ожидания, значение по умолчанию было уменьшено до минимального, ради разгрузки процессора пользователя и комфортной работы с программой, что никак не сказывается на безопасности. Программы имеющие разный "тактовый множитель", **НЕСОВМЕСТИМЫ**!

**Помните! Слабое место любой программы, работающей с использование паролей — легкомысленность ее пользователя!** Пароль должен быть длинным, алогичным и вообще немного идиотским. Нидерландский и американский криптографы Нильс Фергюсон и Брюс Шнайер в своей книге «Практическая криптография» рекомендуют использовать в качестве парольной фразы то, что они назвали "шокирующей ерундой", т.е что нибудь ужасно неприличное, мерзкое; то, что вы в трезвом уме взяли бы написали на бумаге и дали кому нибудь прочитать. Пароль вроде "*Епихантожуевский штрульдик чмякнул чипужку*" вполне неплох для использования русскоязычными пользователями, но пароль "*123456*" может выбрать только тот, кто вообще ничего не знает о парольной защите информации. Храните пароль только в своей голове, или в хорошем менеджере паролей со свободным исходным кодом, использование которого одобрено широкими массами и международным экспертным сообществом.

## Подлинность зашифрованного файла и подтверждение авторства

Для того, чтобы проверить авторство и целостность файлов или текстов, в современном мире используют криптографию на эллиптических кривых, но так как автор программы "параноик, страдающий шпиономанией", он решил не доверять тому, в чем сам мало разбирается и не верить на слово организациям вроде Microsoft или NIST, как это делает множество разработчиков программ по всему миру.

Автор решил пойти самым консервативным путем, который был ему известен из книг по криптографии и защите информации: аутентифицированная парольно-криптографическая защита. Об аутентификации (проверке подлинности) ниже как раз и пойдет речь.

Чтобы обнаружить внесенные в файл изменения, файл обычно хэшируют функцией подсчета контрольной суммы, публикуя файл и его хэш-сумму там, где файл и тем более его хэш-сумму будет сложно изменить. Авторы свободных программ часто используют такой подход: программа публикуется на сайте авторов программы, вместе с текстовым файлом, содержащим хэш-сумму опубликованной программы, доступ же к изменению как программы так и файла с хэш-суммой устанавливается только для администраторов сайта.

Текстовый файл с контрольной суммой, используется пользователем программы для проверки контрольной суммы файла. Пользователь программы, вычисляет ее хэш-сумму, используя тот же алгоритм хэширования, что и разработчики программы, а после сверяет получившуюся контрольную сумму с той, что разработчики любезно опубликовали на своем сайте.

Совершенно ясно, что программа, контрольная сумма которой не совпадает с опубликованной ее разработчиками, никак не может быть названа настоящей. При проверке контрольной суммы файла, мы доверяем сайту, на котором разработчики, как мы думаем, опубликовали контрольную сумму своей программы. Но что если, сайт разработчиков программы будет атакован, программа заменена вредоносной а хэш-сумма подменена? Встает вопрос аутентификации автора, т.е проверки того, кто именно сгенерировал

опубликованную хэш-сумму опубликованной программы, автор или злоумышленник?

На помощь приходит алгоритм HMAC (англ. Hash-based message authentification code). Это тот же алгоритм хэширования данных, но с использованием секрета. Автор позволил себе внести небольшую редакторскую правку в оригинальный алгоритм, назвав его HMAC-SHA-2-256-Uf, чтобы, как он верит, усложнить жизнь потенциальному злоумышленнику еще больше, чем это сделали разработчики оригинального алгоритма.

Если описать алгоритм аутентификации HMAC-SHA-2-256-Uf символами, получится что-то такое:

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

**Где:**

**HMAC – контрольная сумма сообщения и ключа;**
**h – функция вычисления хэш-суммы;**
**key – ключ шифрования, известный отправителю и получателю;**
**alpha – первое секретное число 0x66;**
**beta – второе секретное число 0x55;**
**message – сообщение, подлинность которого нужно доказать;**
**xor – сложение по модулю 2;**
**|| – конкатенация ("склеивание" данных);**

Как же HMAC позволяет не только проверить целостность данных, но и их авторство? Алгоритм действий отправителя:

1.) Сложить по модулю 2 ключ шифрования и второе секретное число.
2.) Совместить с защищаемым сообщением до его шифрования.
3.) Вычислить хэш-сумму.
4.) Сложить по модулю 2 ключ шифрования и первое секретное число.
5.) Совместить с ранее вычисленной контрольной суммой.
6.) Вычислить хэш-сумму.
7.) Зашифровать сообщение и отправить вместе с хэш-суммой.

Злоумышленник, перехвативший такое зашифрованное сообщение, не имея ключа шифрования, не только не может прочитать открытый текст, но и не в состоянии подделать сообщение так, чтобы этого не заметил получатель. Одна из самый распространенных ошибок в криптографии, это думать о зашифрованном сообщении, как о сообщении, которым невозможно манипулировать. Для защиты от манипуляций, как раз и придуман алгоритм HMAC. Не зная ключа шифрования, известного только отправителю и получателю, злоумышленнику невозможно получить правильную хэш-сумму для измененных им данных, а значит, получатель сообщения, имея правильный ключ шифрования, сможет без труда узнать, подделано сообщение или нет. Злоумышленник все еще в состоянии испортить одно или несколько сообщений, прервать передачу сообщений, бесконечно записывать пересылаемые сообщения, перенаправить сообщение "не туда", но на этом его возможности как атакующей стороны заканчиваются.

В настоящей программе используется самостоятельно разработанная автором программы функция HMAC на основе алгоритма SHA-2-256, так как ему лень разбираться в том, как устроенны другие функции ключевой аутентификации, и тем более лень писать на языке Си сложные и длинные функции, за реализацию которых он не получит не то что денег, но даже обратной связи от сообщества. Автор решил использовать что-нибудь простое, быстрое и понятное, одновременно сохранив стойкость оригинальной функции HMAC от 1997 года, и как верит автор, даже усложнить жизнь потенциальному взломщику, при этом не внедряя сложную и тяжелую для понимания криптографию с открытым ключом.

**sha256sum**

Так-же, в пакете "PlexusTCL Crypter", начиная с версии 2.73, присутствует бонус, а именно утилита sha256sum.exe, которая вычисляет SHA-2-256 контрольную сумму как текстов так и файлов размером до 2 Гб включительно. Утилита, как и ее исходный код, распространяется свободно и бесплатно, а в архиве она присутствует на случай, если у пользователя нет программы для вычисления контрольных сумм строк и файлов. Программа принимает на вход три аргумента, а вычисление контрольной суммы строки "PlexusTCL" и ее печать в табличном виде, будет выглядеть так:

**[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**

Утилита sha256sum принимает на вход три аргумента, а именно "-s/t", "-s/f" и простую строку. Строковые эквиваленты первых двух аргументов выглядят как "--string/table" и "--string/file". Первый аргумент позволяет выбрать, в каком виде вы хотите получить контрольную сумму строки или файла, в строковом или табличном. Аргумент "-s/--string" указывает на то, что контрольная сумма будет напечатана в виде строки, а аргумент "-t/--table" на печать контрольной суммы в виде таблицы, как в примере выше. Второй аргумент позволяет явно указать, контрольную сумму чего вычислить, введенной в виде третьего аргумента строки, или файла, именем которого и является третий аргумент. Чтобы вычислить контрольную сумму файла, нужно использовать аргумент "-f/--file", а для вычисления контрольной суммы строки, аргумент "-s/--string". В качестве исходного кода самого алгоритма SHA-2-256 взята реализация от USA NIST (Национальный Институт Стандартов и Технологий США), которая была протестирована с использованием официальных тестовых векторов и полностью безопасна в использовании. Алгоритм был многократно проверен множеством криптоаналитиков со всего мира, и в нем не было найдено ни уязвимостей, ни лазеек. Сам алгоритм SHA-2-256 был разработан и запатентован USA NSA (Агентство Национальной Безопасности США), что ограничивает его использование, но не запрещает использовать его для домашних целей.

## CryCon

**Cry**pter for **Con**sole – консольная программа (не POSIX), аналог графической программы "PlexusTCL Crypter", из которого удален криптостойкий генератор паролей и уничтожитель обрабатываемого файла. Все написанное ранее про программу "PlexusTCL Crypter" и ее алгоритмы, кроме интеграции в вектор инициализации координат курсора мыши, справедливо и для программы CryCon. Как и любая консольная программа, CryCon принимает агрументы, которые интерпретируются программой, и в зависимости от них, программа выполняет какие-либо операции. Длина всех аргументов ограничена 2048 символами.

Первым аргументом программы CryCon всегда выступает строка, указывающая на используемый алгоритм шифрования, если этот аргумент не "-h" или "--help", который указывает на то, что нужно вывести короткую справку. Пользователь сам указывает, какой алгоритм шифрования следует использовать для шифрования или расшифровки файла в виде короткого (буквенного) или длинного (строкового) аргумента. Аргументы, соответствующие алгоритмам шифрования, указаны в таблице ниже, и могут быть переданы программе только в маленьком (строковом) регистре.

| алгоритм шифрования | буквенный аргумент | строковый аргумент |
|---|---|---|
| AES (Rijndael) | -r | --aes |
| Serpent | -s | --serpent |
| Twofish | -w | --twofish |
| Blowfish | -b | --blowfish |
| Threefish | -t | --threefish |

Второй аргумент всегда указывает на то, какую операцию выполнить, шифрование или расшифровку. Этот аргумент может быть коротким (буквенным) или длинным (строковым), и выглядит буквенный аргумент как "-e" и "-d", а строковый как "--encrypt" и "--decrypt".

Третий аргумент интерпретируется в зависимости от того, какой алгоритм был выбран. Если был выбран алгоритм AES, Twofish, Serpent или Threefish, то третий аргумент интерпретируется как указание на то, какой длины ключ следует использовать, 128, 192 или 256-битный. В случае выбора алгоритма Threefish, длины ключа будут 256, 512 или 1024 бита. Этот аргумент может быть коротким (буквенным), а именно "-a", "-b", "-c", или длинным (строковым), а именно "--128", "--192" или "--256". В случае выбора алгоритма Threefish, буквенные аргументы не меняются, а вот строковые изменяются на "--256", "--512" и "--1024". Если был выбран алгоритм Blowfish, то третий аргумент интерпретируется как имя обрабатываемого файла, потому что при использовании этого алгоритма длина ключа не указывается. Все дело в том, что алгоритм Blowfish, это алгоритм с переменной длиной ключа, которая в программе всегда максимальна, и составляет 448 битов. Из этого следует, что аргументы для запуска программы, при выборе алгоритма AES, Twofish, Serpent при 256-битном ключе, будут такими:

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

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

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

при выборе алгоритма Blowfish такими:

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

а при выборе алгоритма Threefish при 512-битном ключе, такими:

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

В случае, если во время работы программы произошла ошибка, например закончилось место на диске, файл для обработки не был открыт (значит что-то мешает), введенный аргумент некорректен (написан неправильно), длина ключа в ключевом файле мала и т.д, то программа уведомит об этом выводом текстового сообщения и прервет все операции.

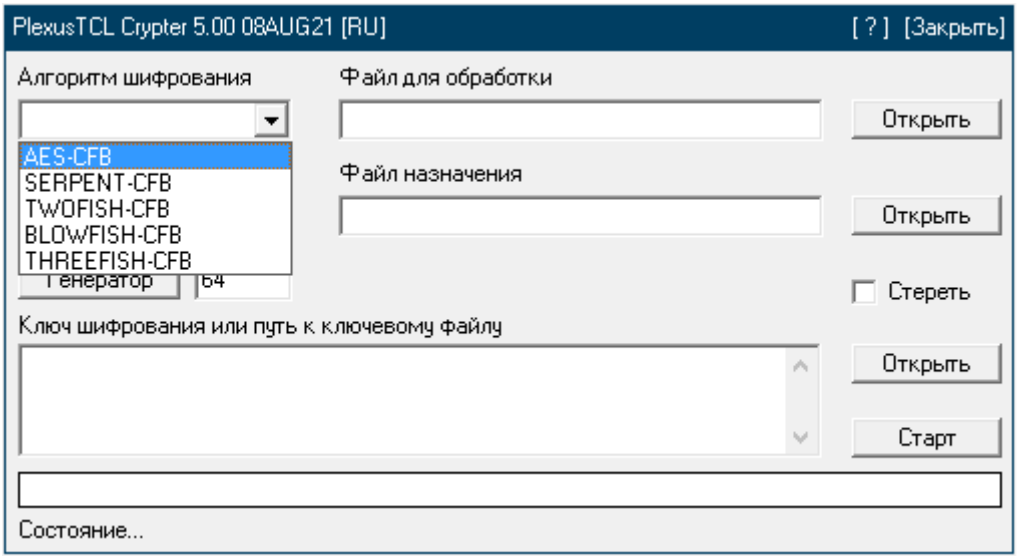
Последний аргумент всегда указывает на имя файла, содержащего ключ шифрования, и если файл не удалось открыть на чтение, его имя интерпретируется программой как строка-пароль, преобразуемый в ключ шифрования функцией KDFCLOMUL. Данные в ключевом файле считаются ключом шифрования как таковым, и не подвергаются никаким преобразованиям, так что при использовании ключевого файла, храните его в надежном месте.

## Графический интерфейс пользователя

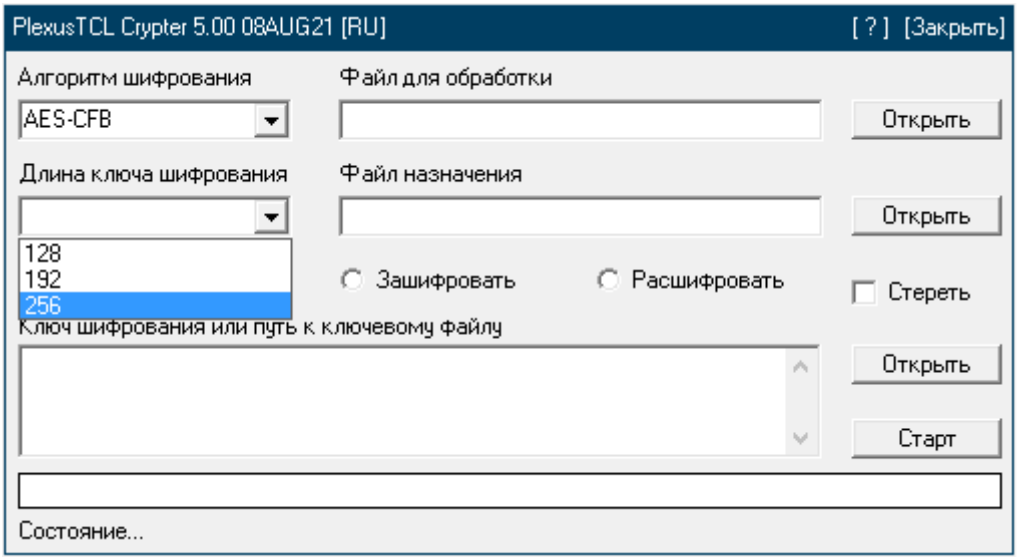
GUI существует, чтобы управлять логикой работы программы с помощью компьютерной мыши, так как не все пользователи успешно работают в командной строке, а многих просто раздражает нужда в постоянной печати команд. К тому же, GUI бывает красив, элегантен, строг, интуитивно понятен и просто приятен в работе с ним. Чтобы использовать программу "PlexusTCL Crypter" для обработки файла, нужно выполнить следующие действия.

1.) Запустите программу PlexusTCL Crypter 5.00.exe

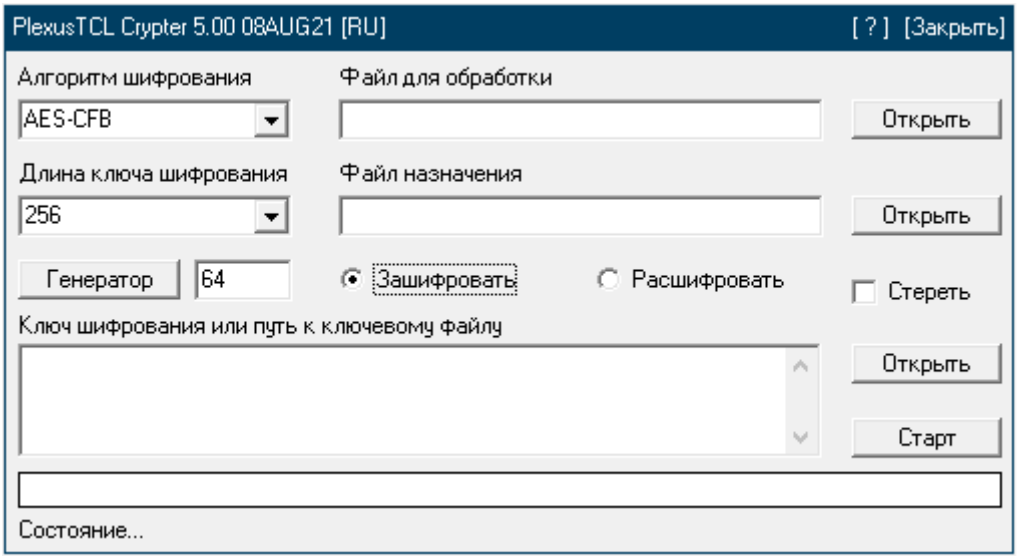
2.) Выберите нужный вам алгоритм шифрования.



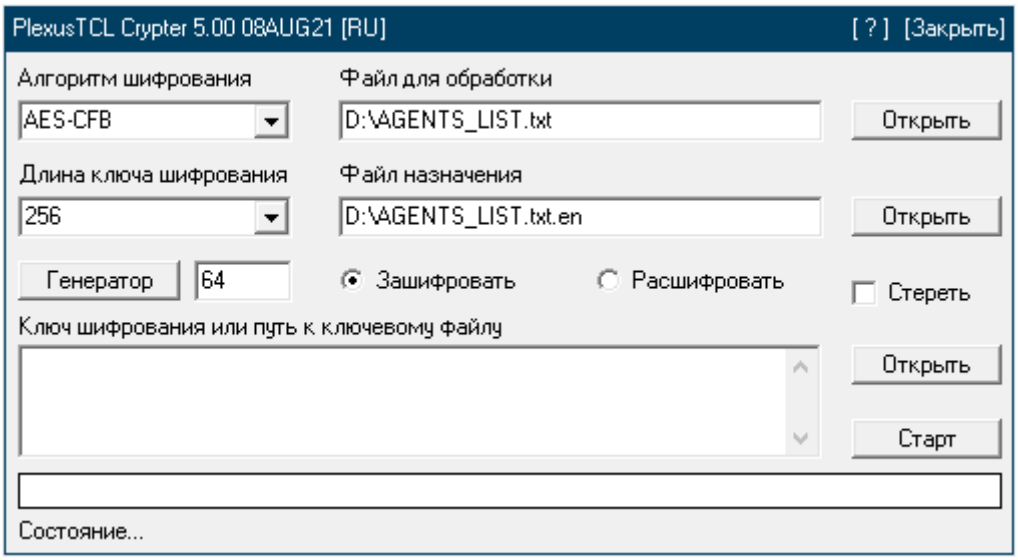

3.) Если появилось поле для выбора длины ключа шифрования, выберете нужную вам длину ключа. Чем больше выбрана длина ключа шифрования, тем больше его криптостойкость, но генерация ключа шифрования как и само шифрование, будут длится дольше.



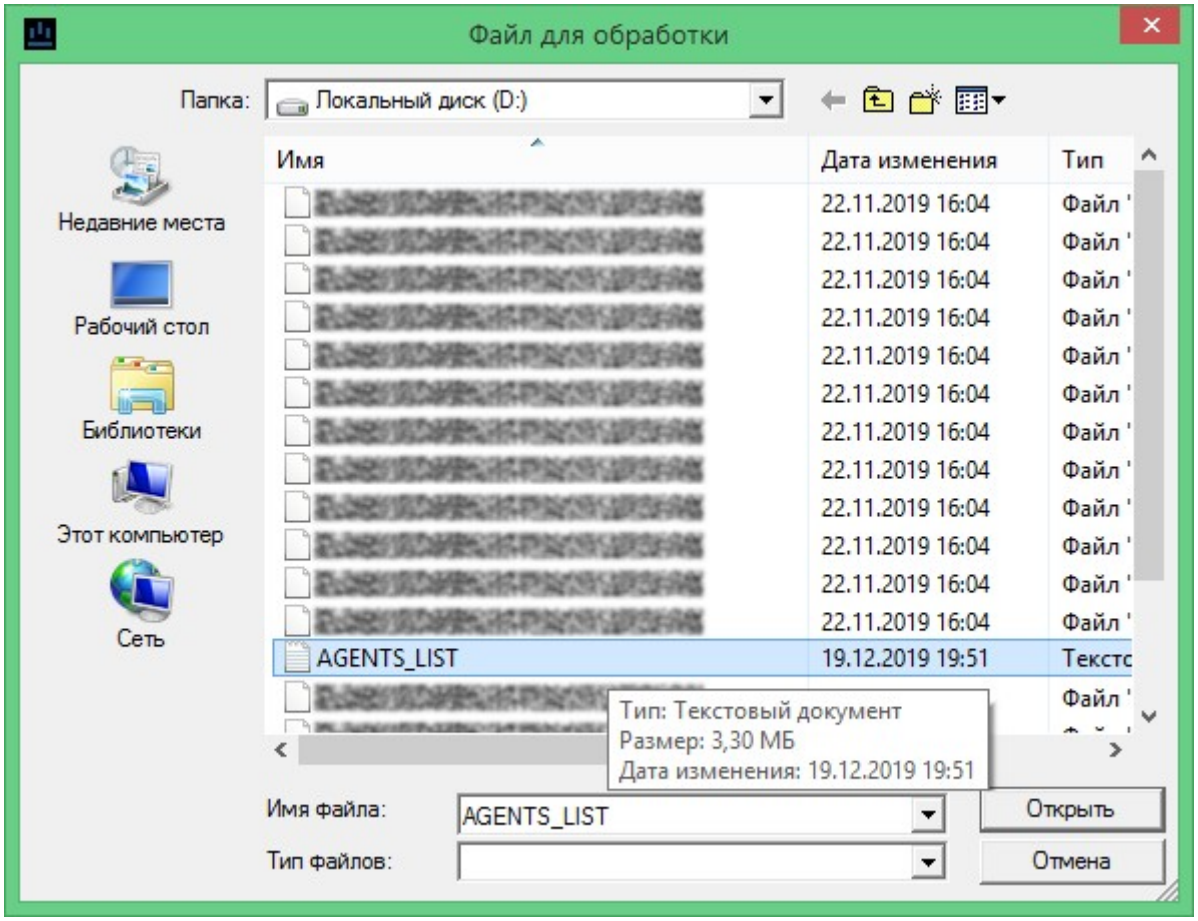

4.) Выберите необходимое действие (зашифровать/расшифровать). Если вы используете блочный шифр Blowfish, то поле для выбора длины ключа шифрования не появится.




5.) Введите в поля озаглавленные как "Файл для обработки" и "Файл назначения" названия файлов, который хотите обработать а так-же в который будут записаны обработанные данные. Строки в полях не должны совпадать!

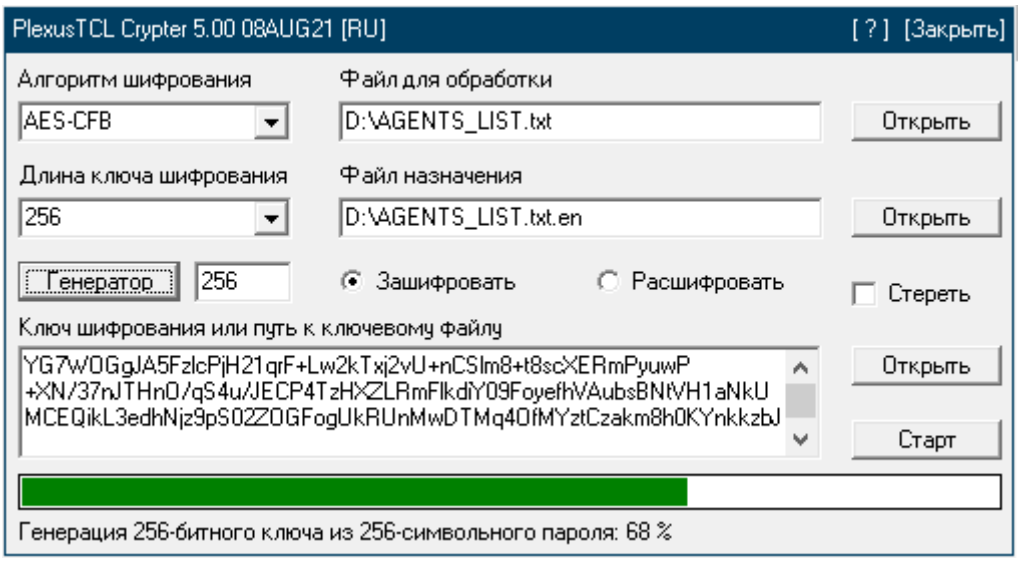



Нужно вводить название файла с его расширением если обрабатываемый файл лежит в одном каталоге с программой, которая будет с ним работать, так как программа не может определять расширения файлов с которыми работает. Если файл лежит в другом каталоге, нужно вводить название файла вместе с полным путем к нему. Если вы не желаете вводить имя файла с полным путем к нему, то вы можете просто вызвать диалоговое окно выбора файла, нажав на кнопку с надписью "Открыть" справа от соответствующего поля.



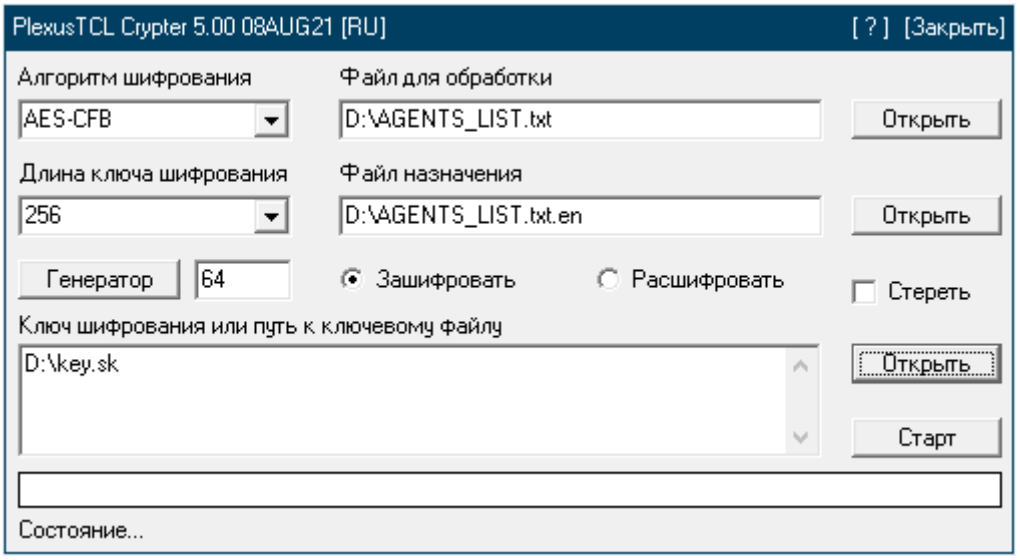

Выбранным для обработки файлом, считается тот файл, название которого появилось в поле "Имя файла" и который одновременно стал выделенным. Чтобы утвердить файл для обработки, нажмите "Открыть", и его имя вместе с полным путем к нему, появится в соответствующем поле. Имя файла, в который будут сохранены обработанные данные, вводится в поле "Файл назначения" или выбирается так-же, как и файл для обработки, после нажатия на кнопку "Открыть" справа от соответствующего поля. Каждая кнопка соответствует полю, находящемуся слева от нее. Начиная с версии 4.91, программа, обнаружив совпадение имен обрабатываемого файла и файла назначения, автоматически добавляет в конец имени файла назначения, строку ".crycon", чтобы избавить пользователя от самостоятельного изменения имени файла назначения или его расширения.

6.) Если вы хотите использовать в качестве ключа шифрования простую строку или ключевую фразу, введите ее в поле озаглавленное как "Ключ шифрования или путь к ключевому файлу". Не стоит беспокоится на счет безопасности использования строки в качестве ключа шифрования, так как строка не используется в качестве ключа шифрования. Строка будет преобразована в ключ шифрования функцией формирования ключа KDFCLOMUL (оригинальная разработка) на основе алгоритма хэширования SHA-2-256. В примере ниже, в качестве пароля, используется строка состоящая из 256 псевдослучайных заглавных, строчных латинских букв, арабских цифр и специальных символов, полученная с помощью встроенного генератора, который может генерировать строки от 8 до 256 символов.

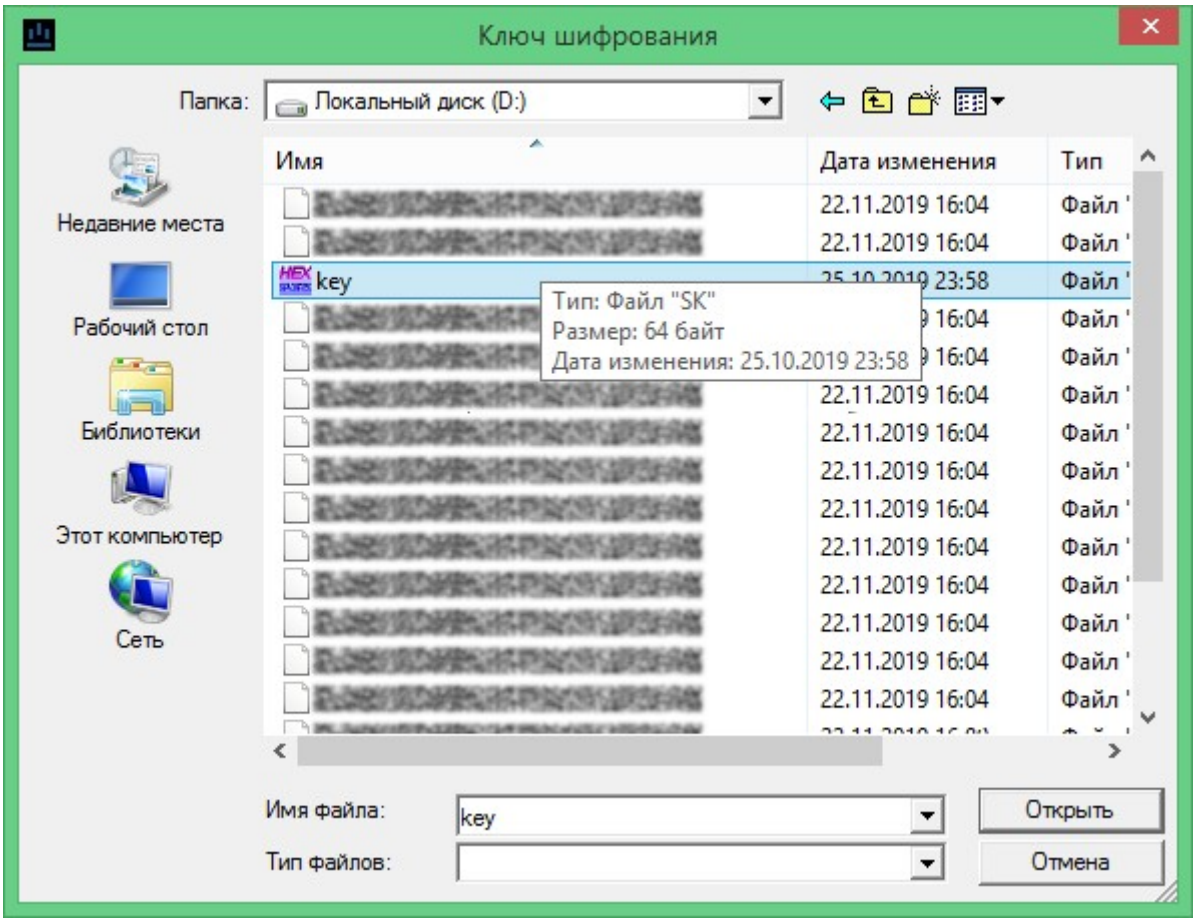



В примере выше, как раз показан прогресс генерации ключа шифрования из 256 символьного пароля. Генерация запускается только после нажатия кнопки "Старт", когда пароль введен, его длина от 8 до 256 символов и программа полностью готова к обработке данных.

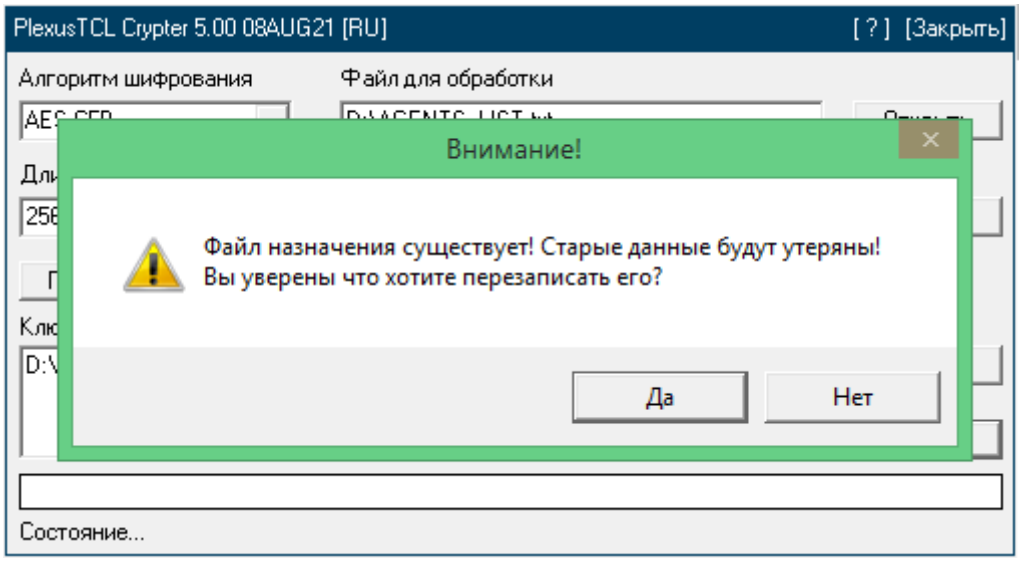
Если вы хотите использовать в качестве ключа шифрования, данные из файла (файл может быть любым, но его размер должен быть больше длины ключа шифрования в байтах или равен ей), введите в поле название файла. Важно отметить, что данные из ключевого файла НЕ подвергаются никаким изменениям — данные из ключевого файла считаются ключом шифрования как таковым. Просто знайте это!




Вы так-же можете вызвать диалоговое окно выбора ключевого файла (как на рисунке ниже), если хотите выбрать один из множества файлов, нажав на "Открыть" рядом с полем для ввода ключа. Файл считается утвержденным так-же, как в примере с выбором обрабатываемого файла.

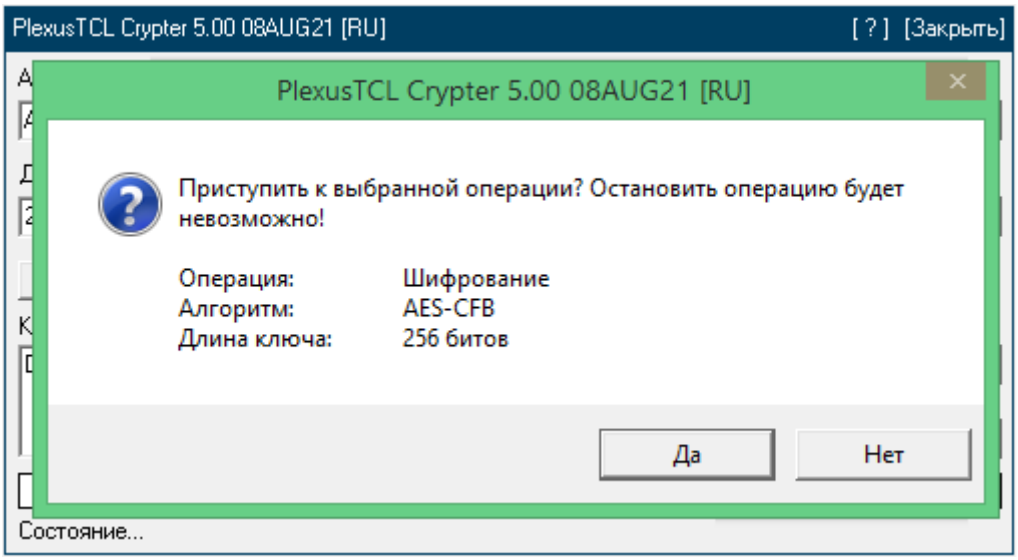



7.) Нажмите "Старт", чтобы начать операцию шифрования/расшифровки. В случае, если вы допустили ошибку, программа уведомит об этом выводом текстового сообщения. Так-же, программа уведомит о том, что в качестве файла назначения был выбран существующий файл, и предложит перезаписать его.

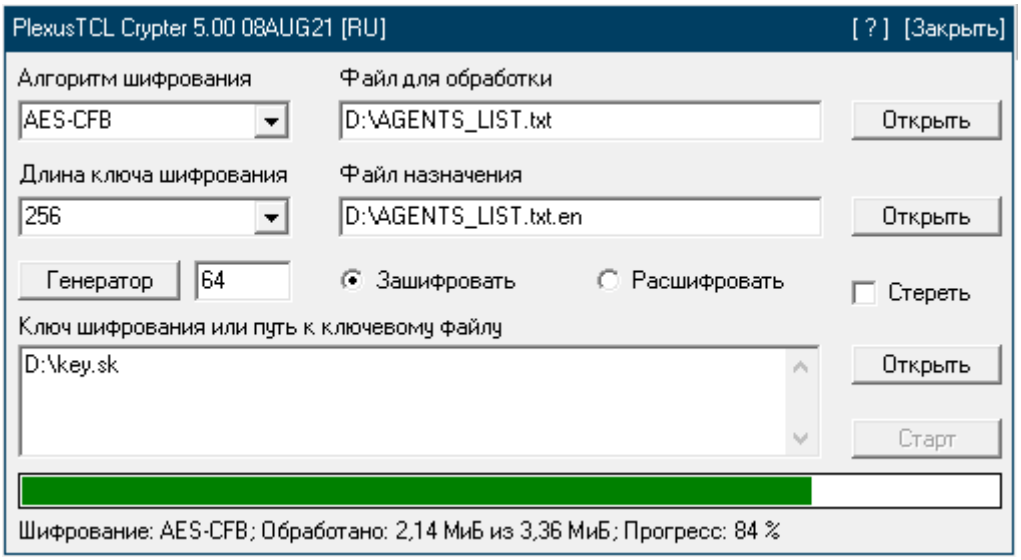



Если вы согласитесь на перезапись существующего файла, программа немедленно уничтожит все данные в файле назначения путем записи обработанных данных из файла для обработки, поверх старых данных. В случае совпадения строк в любых двух и более полях, а именно в полях "Файл для обработки", "Файл назначения" и "Ключ шифрования или путь к ключевому файлу", программа прервет операцию и уведомит о равенстве строк, потому что оно недопустимо.

Если ошибки не были допущены и программа готова к обработке файла, программа уведомит об этом выводом диалогового окна. Обратите внимание на то, что **отмена выбираемой операции невозможна** (придется ждать окончания обработки файла).

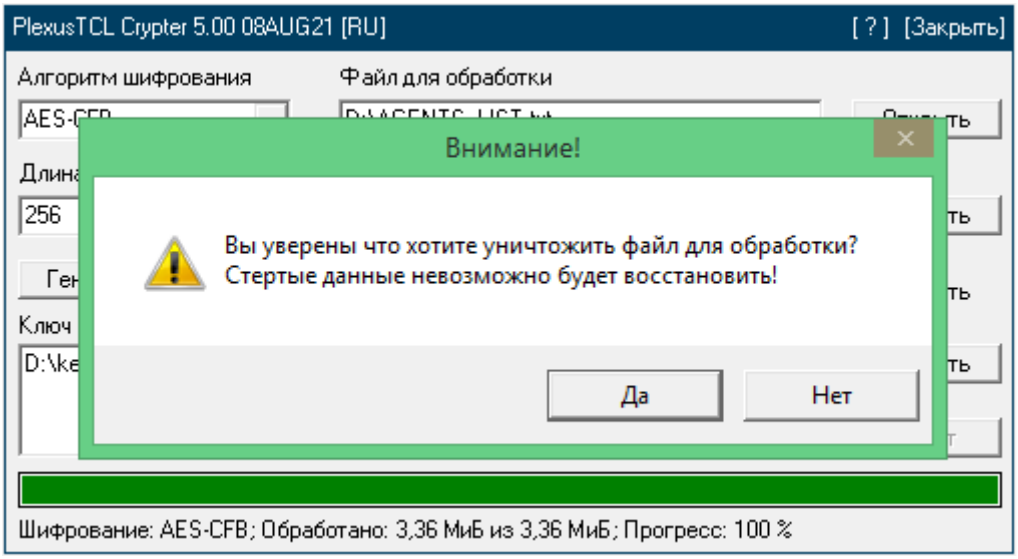



Выполняемая операция, используемый алгоритм, количество обработанных данных и процент прогресса, будут видны в самом нижнем поле, как на рисунке ниже.




Когда операция будет завершена, программа уведомит об этом выводом сообщения. Обратите внимание на то, что на время выполнения операции, программа отключает кнопку "Старт". Это сделано для того, чтобы у пользователя не было возможности запустить обработку одного файла сразу в двух потоках одновременно.

Перед запуском процесса шифрования, вы можете поставить галочку в поле "Стереть", если желаете уничтожить файл для обработки после того, как он будет обработан. Если галочка установлена, то после окончания шифрования, вам будет показано диалоговое окно, в котором можно выбрать, уничтожить файл для обработки, или нет.




Если галочка установлена и вы согласитесь на уничтожение файла, файл для обработки будет перезаписан нулями, его размер будет усечен до нуля и он будет удален стандартной функцией DeleteFile из библиотеки Kernel32.dll.

8.) Чтобы расшифровать файл, заполните поля так-же как и для шифрования, но выберите "Расшифровать" перед нажатием кнопки "Старт".

**Исходные коды**

Программа crycon, как и утилита sha256sum, написаны на языке программирования C (Си) и скомпилированы в исполняемые файлы компилятором TCC (Tiny C Compiler) версии 0.9.27. GUI написан на языках программирования C/C++/Pascal (Си, Си++ и Паскаль) и скомпилирован в исполняемый файл компилятором C++ Builder версии 6.0. Все исполняемые файлы в пакете "PlexusTCL Crypter" и их исходные коды, распространяются свободно и бесплатно.

Ниже указаны SHA-2-256 контрольные суммы всех трех программ входящих в "PlexusTCL Crypter":

| **Название файла** | **SHA-2-256 контрольная сумма** |
|---|---|
| PlexusTCL Crypter 5.00.exe | 675446B9639FD3369DE523D10A6742C2<br>6F60047115F45CA299B8F95EB5158F46 |
| crycon.exe | 98CF5434EFFA37DC4C1CF46410571608<br>86804BBDCFB3B6B0B1D0CC7D6B723F36 |
| sha256sum.exe | 83CD0D3A42E11DE25D1C5EEEDBE00F60<br>4B2F59FAD4EDF3378E6125DB99BE028A |