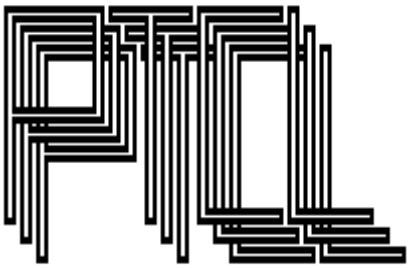


PlexusTCL Crypter

версия 7.00 от 28 января 2026 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, ВАШ ДЛИННЫЙ ЯЗЫК ИЛИ 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 шифрования файлов, начиная с версии 5.10 размером до 4 ЭиБ (Эксбибайт). 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. GUI разработан только под ОС Windows и работает начиная с Windows XP SP3. Если вас интересует смена интерфейса или алгоритмов, вы всегда можете пересобрать проект из исходных текстов. GUI существует в двух экземплярах, в классическом стиле под Windows XP и "улучшенном" под Windows 8. Обе программы собраны средствами C++ Builder 6.0 от 2002 года и Embarcadero RAD Studio 10 соответственно.

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 (для простого пользователя не так важно), 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. По этому, в случае если важна скорость обработки данных, 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 уже не получится. Команда под руководством Брюса Шнайера разработала шифр Threefish для использования в хеш-функции Skein как легкий в реализации, настраиваемый блочный шифр, построенный на базовых 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.

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 учетом времени записи на диск. С обновлением реализации AES использовать ARC4 стало просто незачем.

Важно будет напомнить что процессоры, накопители, операционные системы, компиляторы, оптимизаторы компиляторов, как и совесть, умственные способности и психические расстройства авторов процессоров, накопителей, операционных систем, компиляторов и оптимизаторов, как разнообразны и очень интересны даже для неспециалиста, так и могут напугать до седых подмышек. С первых моделей Intel 80386 и заканчивая AMD Ryzen 9 9950X (на 2024 год) множество инженеров постоянно что-нибудь улучшают, и эти улучшения могут как навредить скорости работы программы так и увеличить её. Например, старый компьютер автора настоящей программы под управлением

Windows XP SP3, с процессором Intel Pentium D 820 обгонял в скорости шифрования компьютер с процессором Intel Core I3-3217U, и это при том, что второй работал под управлением Windows 8.1. Так что скорость обработки данных этой программой, как наконец-то дошло до автора, может зависеть почти от всего, что имеет прямое отношение к вычислительной технике, и именно по этому она не указана в этом README, чтобы не вводить пользователя в заблуждение. На компьютере разработчика скорость шифрования достигает 20 Мб/сек, но он использует старенький HDD со скоростью вращения 7200 об/мин.

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();
C[0] = Ek(IV);
C[i] = P[i] xor Ek(C[i-1]); от i=1 до i=n;

где:
     random - генератор псевдослучайных чисел вашей ОС
     E      - функция шифрования
     k      - ключ шифрования
     IV     - вектор инициализации
     P      - открытый текст
     C      - шифротекст
     xor    - операция побитового исключающего ИЛИ
     n      - длина шифротекста в блоках
     i      - увеличивающийся на единицу счетчик
```

Даже при шифровании файла размером 180 ГиБ, полностью состоящего из нулевых байтов, 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.

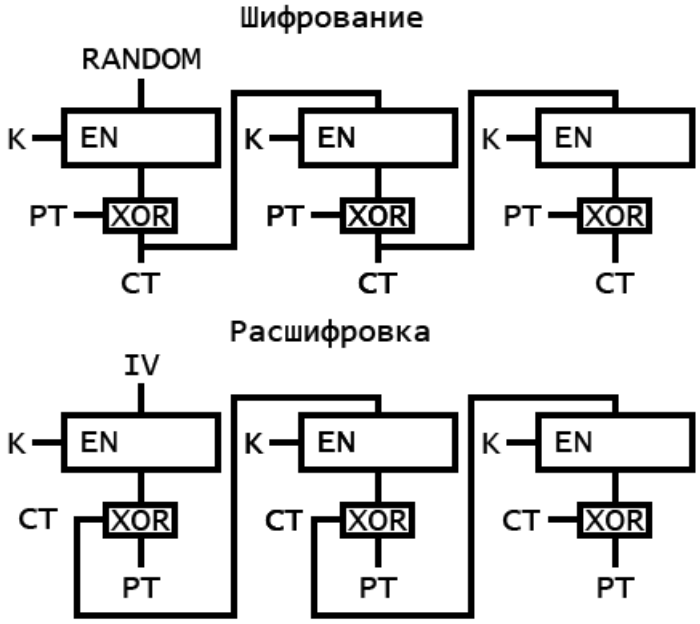
При шифровании двух открытых текстов, шифруемые блоки которых совпадают, уникальный для каждого открытого текста вектор инициализации, используется в режиме CFB для сокрытия этого совпадения, если для

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 криптопровайдер а так-же 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 полученных от одного ключа. 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.

Автор использовал для генерации IV функции стандартного криптопровайдера от Microsoft Windows PROV_RSA_FULL, так как он используется во всех операционных системах семейства Windows начиная с Windows 2000 и заканчивая Windows 11, что обеспечивает программе почти полную совместимость с операционными системами этого семейства, не жертвуя безопасностью.

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 стека, что вносит в IV еще больше неопределенности (какое значение имела переменная кадра стека, к примеру по адресу 0x00408A47, злоумышленник конечно-же установить не сможет). Значение конечно можно подобрать полным перебором всех вариантов, но это не имеет смысла без ключа и на криптостойкость никак не влияет. Автор программы надеется (именно так!) что криптопровайдер операционной системы, под управлением которой работает ваш компьютер, разработан разумными людьми сохранившими хотя-бы остатки совести.

Режим обратной связи по шифротексту (CFB)



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 – случайная строка (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 хеширования, 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 2026 год обычно от 150000 до 2000000 и всегда зависит от производительности функции шифрования или хеширования, 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 её с паролем, многократно увеличить количество вариантов пароля, а так-же избежать атак с использованием радужных таблиц (если пароль хешируется), что многократно усложняет перебор, делая его очень долгим и крайне не выгодным занятием. Если ключ шифрования получается из пароля и соли, а соль для каждого шифрования уникальна и случайна, то пользователь может использовать один и тот-же пароль для шифрования огромного колличества данных, т.к ключ будет каждый раз разный.

При разработке первой версии программы, разработчик исключил соль из KDF, так как ему не понравилось слово "соль", по этому в программе не был реализован механизм защиты пароля путем его "соления". Позже разработчик привык к слову "соль" и всё-таки добавил в KDF 128-битную соль, которая создается криптостойким генератором для каждого отдельного файла и хранится первым блоком каждого зашифрованного файла. На этот счет не стоит беспокоится, т.к соль не нужно делать секретной, она нужна только для того чтобы получать разные ключи при использовании одного и того-же пароля а так-же затруднить взлом пароля с использованием предвычесленных хэшей. Если два пользователя используют один и тот-же пароль для защиты своих файлов, и KDF генерирует ключи шифрования только от паролей, то такая KDF сгенерирует одинаковые ключи для двух разных пользователей с одинаковым паролем. Соль нужна чтобы такого не случалось.

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 хеш-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 4294967295 символов, а **количество итераций хеширования является динамическим**, и генерируется на основе первых 32 битов контрольной суммы пароля и соли.

Почему количество итераций хеширования сделано не статическим, как у людей? Чтобы полностью исключить (как верит автор) возможность перебора пароля, привязав к количеству итераций хеширования не только пароль и соль, но и длину ключа, сделав перебор пароля очень затруднительным. 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" хеширования. Если сложность (время) атаки методом перебора увеличивается с количеством итераций хеширования, а это количество зависит не только от длины пароля но и от каждого его байта, то чтобы узнать количество итераций хеширования и попытаться составить схему перебора пароля, нужно знать не только длину пароля но и каждый его символ, что почти сводит взлом пароля к знанию самого пароля. Количество итераций хеширования можно конечно-же подобрать, но такой подбор очень усложняет жизнь злоумышленнику (подбор нужно еще осуществить). Скрипя душой автор программы признает, что большое статическое количество итераций (500000-1500000) почти всегда лучше чем динамическое, но ему как-то не хочется всё переделывать, к тому-же при замене функции формирования ключа, старые версии программы уже третий раз станут

несовместимы с новой и будущими версиями (придется перешифровать все зашифрованные файлы).

Данная KDF получила название KDFCLOMUL, от константы CLOMUL_CONST (Clock multipler), которая все-же позволяет настраивать, насколько большим будет количество итераций хеширования, а значит и время вычисления ключа.

Константа CLOMUL_CONST обозначает так называемый "тактовый множитель", использующийся в функции KDFCLOMUL как один из операндов при формировании количества итераций хеширования а так-же размера используемой при хешировании памяти. Если описывать константу простыми словами, то чем больше её значение, тем дольше генерируется ключ из пароля, так как для генерации ключа требуется намного больше тактов процессора и оперативной памяти. Она необходима для генерации ключа шифрования из пароля, так как влияет на количество итераций цикла вычисления хеш-суммы пароля. Сам же цикл использует алгоритм хеширования 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 десятки минут.

Выбирайте значение "тактового множителя" исходя из предполагаемой общей мощности компьютерного парка вашего противника, так как оборона, адекватная атаке, выстраивается только если известны возможности атакующего. На 2026 год, значение можно сделать от 2, этого хватит еще лет на 20.

Значение "тактового множителя" равное 1, обеспечивает генерацию ключа шифрования из пароля "password" на процессоре Intel Core 2 Quad Q9500 с тактовой частотой 2.83 GHz где-то за 0.12 секунды при затратах памяти в 1.28 Мб и около 600000 итераций хеширования, 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 208827064576 вариантов пароля. Так как данный процессор способен перебрать 9 паролей в секунду, то полный перебор всех паролей на выше указанном процессоре займет ~735 лет. Но учтите, что графические процессоры в десятки (сотни) раз 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 хеш-сумм SHA-1 за одну секунду, так что не все так радужно как обычно представляется из цифр.

Программы имеющие разный "тактовый множитель", **НЕСОВМЕСТИМЫ**!

Сама функция формирования ключа сначала выделает память, размер которой равен сумме размера ключа в байтах, длины пароля в байтах и константы CLOMUL_CONST, помноженной на 32768 * CLOMUL_CONST. Далее вычисляется SHA-2-256 контрольная сумма пароля и соли, и первые 4 байта контрольной суммы используются в качестве количества итераций хеширования пароля на начальном этапе

```
count  = *(uint32_t *)(sha256sum.hash) & 0x000FFFFF;
count |=  (uint32_t)1 << 19;
count *= CLOMUL_CONST;
```

20-й бит устанавливается во избежание слишком маленького колличества итераций хеширования, первые же 12 битов сбрасываются чтобы избежать

наоборот слишком большого колличества итераций. Далее функция приступает к поэтапному выполнению:

1.) Функция "нагревается", т.е хеширует пароль и соль count раз (~600000). Чем длинее пароль и соль, и чем больше размер ключа, тем больше требуется памяти и больше времени занимает хеширование.
2.) Хеширует пароль и соль столько раз, сколько нужно для полного заполения контрольными суммами всей ранее выделенной памяти (мегабайты).
3.) Хеширует заполненную контрольными суммами память и записывает итоговую контрольную сумму в массив, где должен хранится ключ. Если этого не хватает чтобы сгенерировать ключ шифрования нужной длины, хеширует память до тех пор, пока ключ требуемой длины не будет получен.

Помните! Слабое место любой программы, работающей с использование паролей — легкомысленность её 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, глупое, возможно даже мерзкое; т.е что-то такое, что в трезвом уме вы бы никогда не написали на бумаге и не дали кому нибудь прочитать. Пароли вроде "*Епихантожуевский штрульдик чмякнул чипужку*" или "*Хитрый+Бублик+Сп&здил+Рублик*" вполне неплохи для использования русскоязычными пользователями, но пароль "*123456*" может выбрать только тот, кто вообще ничего не знает о парольной защите информации, так как это один из самых популярных паролей в мире вот уже около 30 лет. Храните пароль только в хорошем менеджере паролей со свободным исходным кодом, использование которого одобрено широкими массами и международным экспертным сообществом, если вы не доверяете своей памяти.

Разработчик так-же рекомендует пользователю программы ознакомиться с инструкцией по созданию надежных паролей от "Фонда Электронных Рубежей" (англ. Electronic Frontier Foundation, EFF).

При шифровании данных блочным шифром при одном векторе инициализации, ключ может быть использован для шифрования только ограниченного количества информации, превышать которое крайне не рекомендуется. Такое количество информации называют "нагрузка на ключ" и чем она больше тем реже следует менять ключ шифрования во избежание атак на криптосистему. В настоящей программе нагрузка на ключ установлена в 2 ГиБ, что очень мало и гораздо меньше чем реальная нагрузка на ключ шифра Blowfish, шифра с самым маленьким блоком из остальных используемых в программе шифров. При превышении установленной нагрузки на ключ, т.е если при шифровании файла количество зашифрованных байт превысило или стало равно 2 ГиБ, программа генерирует новый ключ путем хеширования старого ключа алгоритмом SHA-2-256. Это нисколько не мешает шифрованию/расшифровке, не вносит уязвимость, не требует от пользователя никаких действий и сделано для усиления защиты, так как во всех версиях программы до версии 5.10 программа могла обрабатывать только файлы размером до 2 ГиБ.

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 компаниям вроде Microsoft, Google или организациям вроде 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 хешируют функцией подсчета контрольной суммы, публикуя файл и его хеш-сумму там, где файл и тем более его хеш-сумму будет сложно изменить. Авторы свободных программ часто используют такой подход: программа публикуется на сайте авторов программы, вместе с текстовым файлом, содержащим хеш-сумму опубликованной программы, доступ же к изменению как программы так и файла с хеш-суммой устанавливается только для администраторов сайта.

Текстовый файл с контрольной суммой, используется пользователем программы для проверки контрольной суммы файла. Пользователь программы, вычисляет её хеш-сумму, используя тот же алгоритм хеширования, 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 её 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 хеш-сумма подменена? Встает вопрос аутентификации автора, т.е проверки того, кто именно сгенерировал опубликованную хеш-сумму опубликованной программы, автор или злоумышленник?

На помощь приходит алгоритм HMAC (англ. Hash-based message authentication code). Это тот же алгоритм хеширования данных, но с использованием секрета. Автор использовал свою реализацию алгоритма ключевой аутентификации PRF-HMAC-SHA-256 согласно RFC-4868, но позволил себе внести небольшую редакторскую правку, а имменно, вычислять HMAC для контрольной суммы всего зашифрованного файла.

Если описать символами алгоритм аутентификации PRF-HMAC-SHA-256 с правкой автора программы, получится что-то такое:

HMAC = h((key xor 0x5C) || h((key xor 0x36) || h(c)));

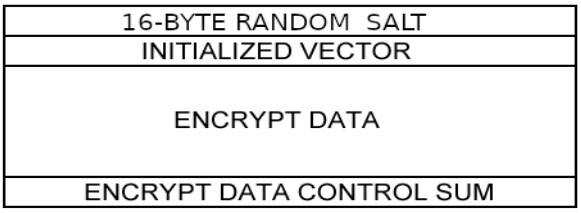
Где:

HMAC – контрольная сумма сообщения и ключа;
h – функция вычисления хеш-суммы;
key – ключ шифрования, известный отправителю и получателю;
c – зашифрованный файл, 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 число 0x36.
2.) Совместить с контрольной суммой зашифрованного сообщения.
3.) Вычислить хеш-сумму.
4.) Сложить по модулю 2 ключ шифрования и число 0x5C.
5.) Совместить с ранее вычисленной контрольной суммой.
6.) Вычислить хеш-сумму.
7.) Отправить сообщение вместе с хеш-суммой.

Для облегчения понимания того, что из себя представляет файл после шифрования, справа приведена схема любого файла, зашифрованного этой программой. Как видно на рисунке, первые 128 битов файла это случайная соль. В зависимости от используемого шифра, следующие за солью 128, 256, 512 или 1024 бита зашифрованного



файла, это вектор инициализации, далее идут зашифрованные данные, а после 256 битов контрольной суммы зашифрованных данных и ключа шифрования.

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 хеш-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 оригинальная функция HMAC на основе алгоритма SHA-2-256 соответствующая RFC-4868, так как автору 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, использовав стойкость хэширования, и как верит автор, даже усложнить жизнь потенциальному взломщику, при этом не внедряя сложную и тяжелую для понимания криптографию с открытым ключом.

Конфигурационный файл

Конфигурационный файл существует что-бы хранить настройки программы, как правило в виде текста. Поддержка конфигурационного файла в программе "PlexusTCL Crypter" реализована начиная с версии 5.12, сам-же конфигурационный файл находится в файле "settings.cry" и должен находится в одной директории с запускаемой программой. Если конфигурационного файла в директории с программой не окажется, или прочитанный из файла параметр окажется некорректен, программа применит настройки по умолчанию. Программа не создает конфигурационный файл, т.е не сохраняет настройки выставленные пользователем или настройки по умолчанию, полностью полагаясь на ответственность пользователя.

Устроен конфигурационный файл следующим образом.

Программа принимает за полезное содержимое конфигурационного файла только знак подчеркивания, большие английские буквы, цифры от 0 до 9, разделитель в виде знака равно, знак # и точку с запятой. Другие символы (все прочие байты) программа игнорирует.

Длина любой строки в файле не должна превышать 127 байт, если длина строки 128 байт или больше, то она будет прочитана из файла частями по 127 байт. Всего из файла программа прочитает не более 10000 строк. Строка начинающаяся со знака # и заканчивающаяся переводом строки это однострочный комментарий, многострочные комментарии как в языке Си, программа не поддерживает. Если в начале или в середине строки встретился знак # то все последующие символы до перевода строки или до знака точки с запятой программа считает комментарием (такой комментарий можно вставлять даже посередине параметра).

В конфигурационном файле можно описать параметры, задаваемые в виде строки вида КЛЮЧ=ЗНАЧЕНИЕ. После ключа должен стоять знак равно, значение должно стоять после знака равно. Если вы хотите описать параметры в одну строку, их нужно разделить знаком точки с запятой.

TOP_COLOR – устанавливает цвет шапки программы. Его, как и другие цвета в программе можно задать только 32-битным числом в виде HEX строки, например 00623E00. Старшие 8 битов числа (в примере 00) ничего не

означают, имеют значение только младшие 24 бита, так-как содержат цвет RGB (Red, Green, Blue). В примере 00623E10 байт 62 означает оттенок красного, байт 3E означает оттенок зеленого и байт 10 оттенок синего.

TOP_TEXT_B_COLOR – устанавливает цвет фона текста в шапке программы.

TOP_TEXT_COLOR – устанавливает цвет текста в шапке программы.

PROG_BAR_COLOR – устанавливает цвет полосы прогресса.

CIPHER – устанавливает используемый по умолчанию шифр. Возможные значения параметра: AES, BLOWFISH, SERPENT, THREEFISH, TWOFISH.

KEY_SIZE – устанавливает длину ключа указанного шифра. Значения параметра для шифров AES, SERPENT и TWOFISH: 128, 192, 256. Для шифра THREEFISH: 256, 512, 1024. Для шифра BLOWFISH этот параметр не используется, его можно оставить любым.

OPERATION – устанавливает производимую программой операцию. Можно задать только значения ENCRYPT или DECRYPT.

PASS_GEN_SIZE – устанавливает длину генерируемого пароля. Можно задать значение от 8 до 256.

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 ГиБ. Утилита, как и её 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 её 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

Crypter for **Con**sole – консольная программа под Linux, аналог графической программы "PlexusTCL Crypter", из которого удален криптостойкий генератор паролей и уничтожитель обрабатываемого файла, так как эти операции могут

быть выполнены штатными средствами большинства дистрибутивов Linux. Все написанное ранее про программу "PlexusTCL Crypter" и её алгоритмы, кроме интеграции в вектор инициализации координат курсора мыши, использование функций криптопровайдера ОС и поддержки конфигурационного файла, 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 2047 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

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

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

при выборе алгоритма Blowfish такими:

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

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

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

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.

Во всех версиях программы до версии 5.10 последний аргумент командной строки всегда указывал на имя файла, содержащего ключ шифрования, и если файл не удалось открыть на чтение или файл не существовал, его имя принималось программой как пароль, преобразуемый в ключ шифрования функцией KDFCLOMUL. Начиная с версии 5.10 программа просит пользователя ввести пароль или имя ключевого файла отдельной строкой приглашая пользователя к вводу диалоговой строкой как в примере ниже:

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

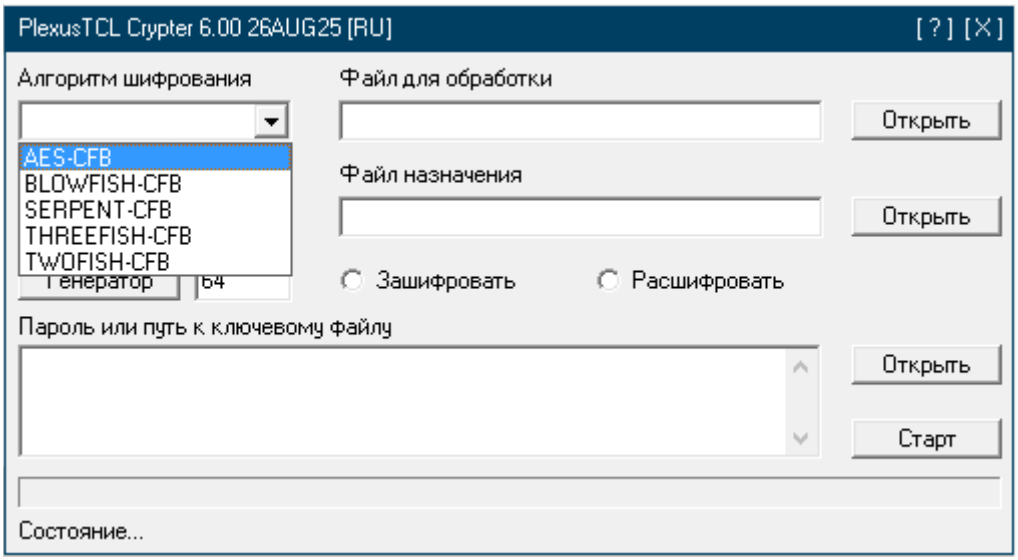
Данные в ключевом файле начиная с версии 7.00 не считаются ключом шифрования как таковым, и теперь подвергаются таким-же преобразованиям как пароль. При использовании ключевого файла, храните его в надежном месте.

Графический интерфейс пользователя

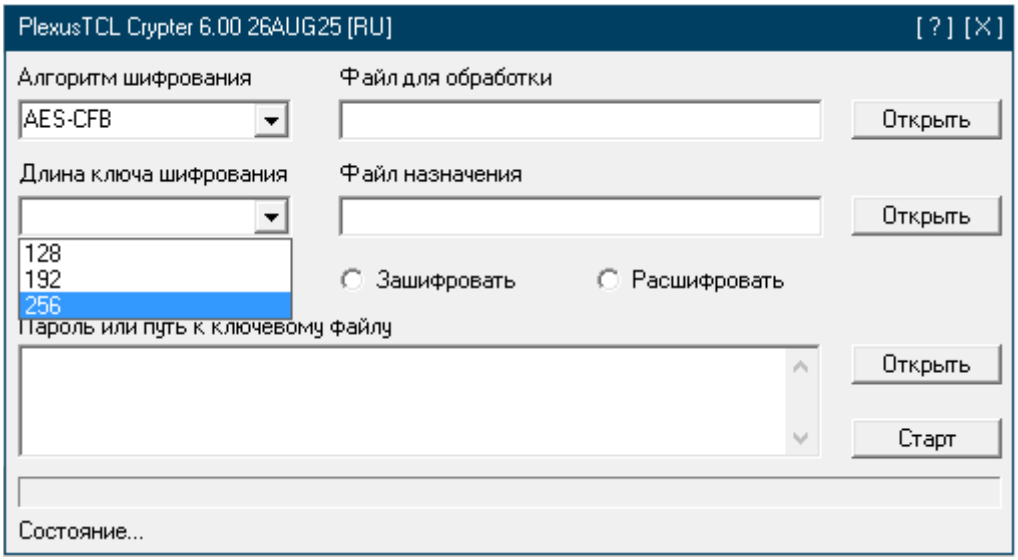
GUI существует чтобы управлять программой с помощью компьютерной мыши, так как не все пользователи успешно работают в командной строке, а многих просто раздражает нужда в постоянной печати команд. К тому же, GUI бывает красив, элегантен, строг, интуитивно понятен и просто приятен в работе с ним. Чтобы использовать программу "PlexusTCL Crypter" для обработки файла, нужно выполнить следующие действия. Программа ничего не сделает, пока пользователь не нажмет "Самую Главную Кнопку", а именно "Старт".

1.) Запустите программу PlexusTCL Crypter 7.00.exe

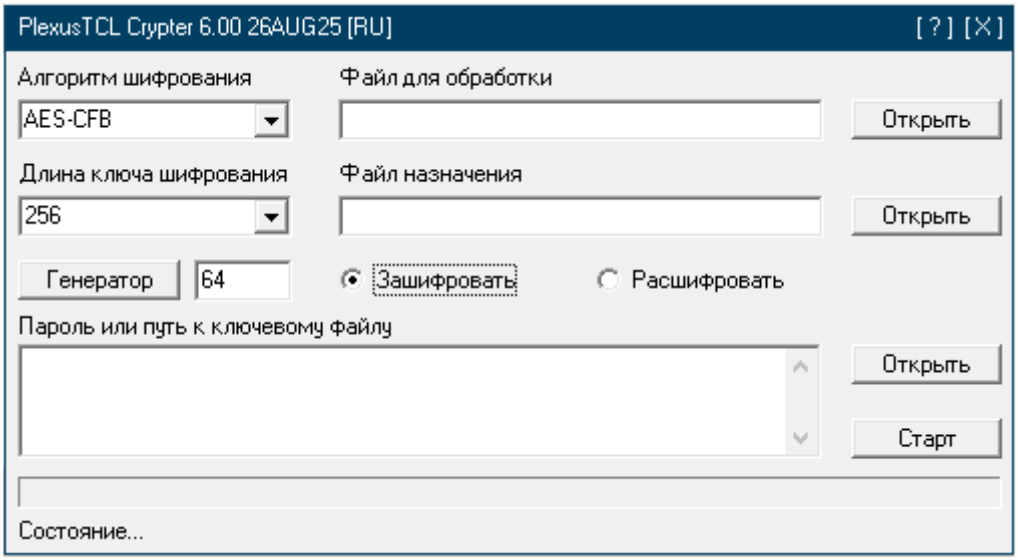
2.) Выберите нужный вам алгоритм шифрования из выпадающего списка, как на рисунке ниже.



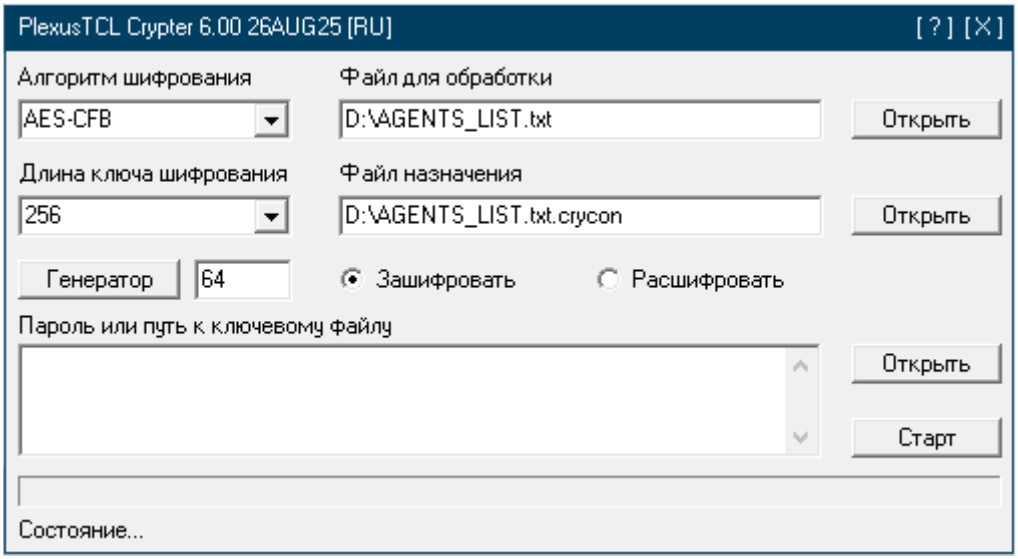
3.) Если появилось поле для выбора длины ключа шифрования, выберете нужную вам длину ключа из выпадающего списка. Чем больше выбранная вами длина ключа шифрования, тем больше криптостойкость, но генерация ключа шифрования как и само шифрование, могут длится дольше.



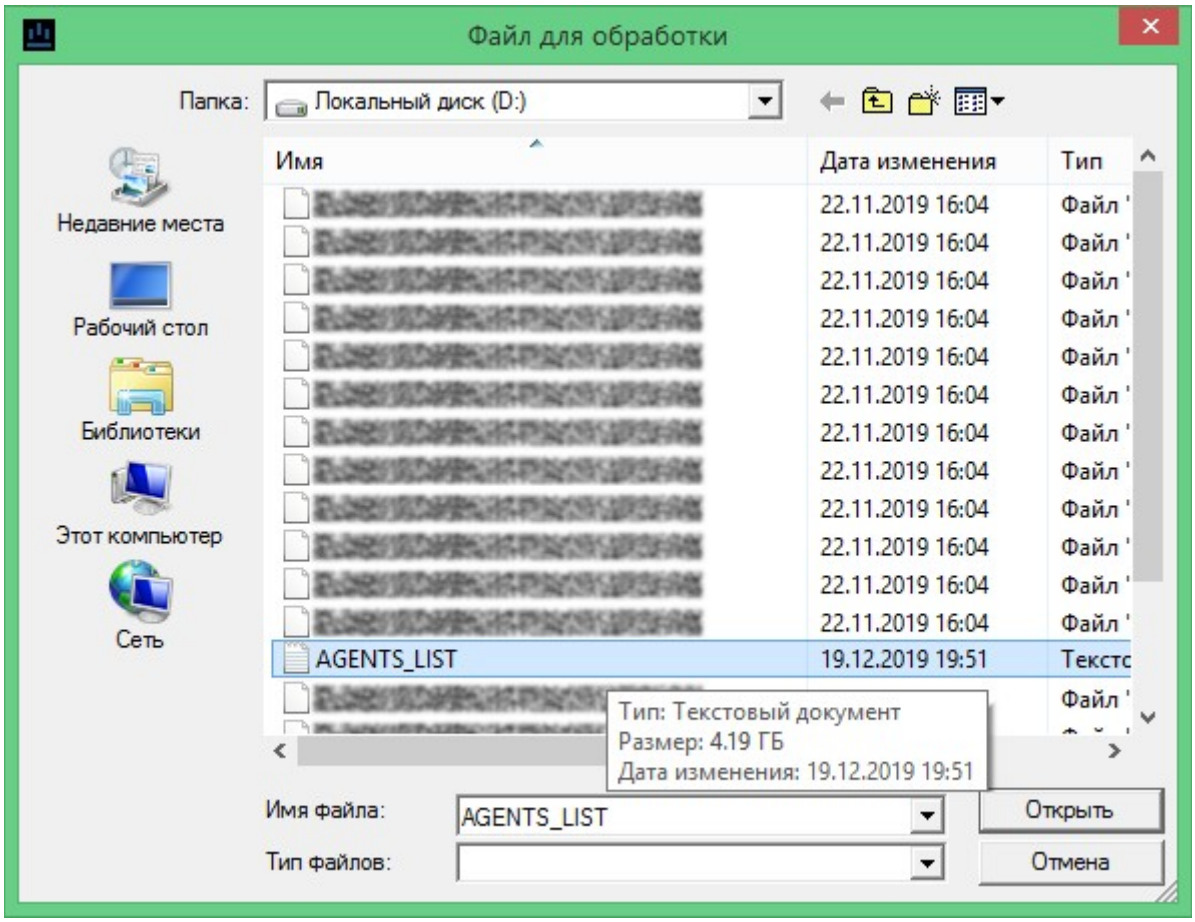
4.) Выберите необходимое действие (зашифровать/расшифровать). Если вы используете блочный шифр Blowfish, то поле для выбора длины ключа шифрования не появится.



5.) Введите в поля озаглавленные "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, а это значит, что имя файла назначения должно отличатся от имени обрабатываемого файла и не должно совпадать с именем какого-либо файла в каталоге назначения (только если вы не собираетесь перезаписать существующий файл).



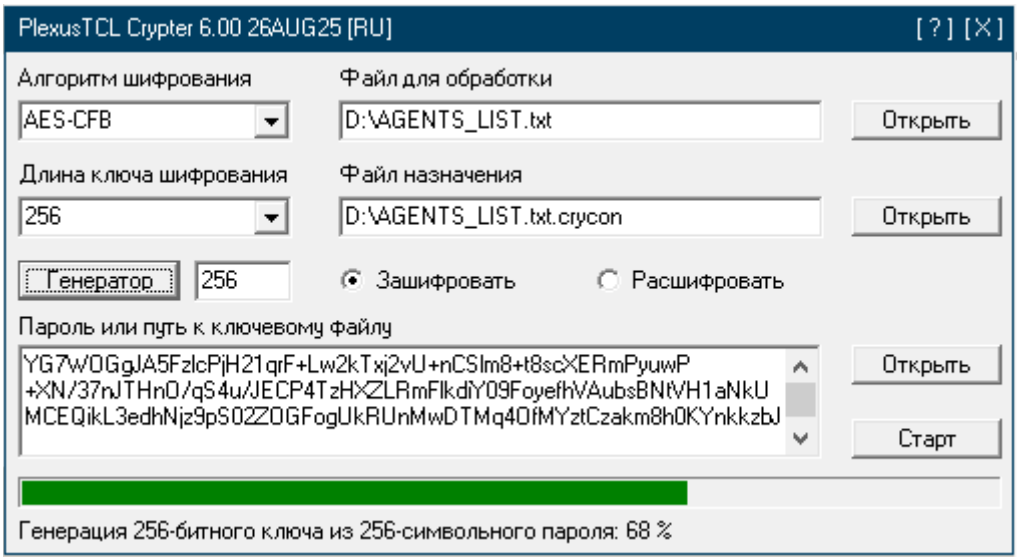
Нужно вводить название файла с его расширением если обрабатываемый файл лежит в одном каталоге с программой, которая будет с ним работать, так как программа не может определять расширения файлов с которыми работает. Если файл лежит в другом каталоге, нужно вводить название файла вместе с полным путем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



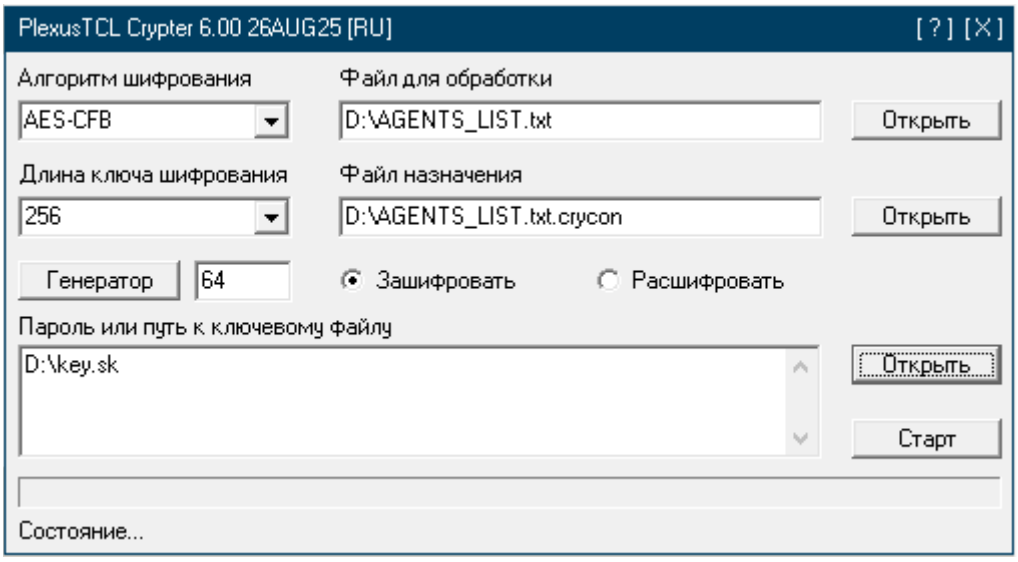
Выбранным для обработки файлом, считается тот файл, название которого появилось в поле "Имя файла" и который одновременно стал выделенным. Чтобы утвердить файл для обработки, нажмите "Открыть", и его имя вместе с полным путем к нему, появится в соответствующем поле. Имя файла, в который будут сохранены 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.

6.) Если вы хотите использовать в качестве ключа шифрования простую строку или ключевую фразу, введите её в поле озаглавленное как "Пароль 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 хеширования 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.

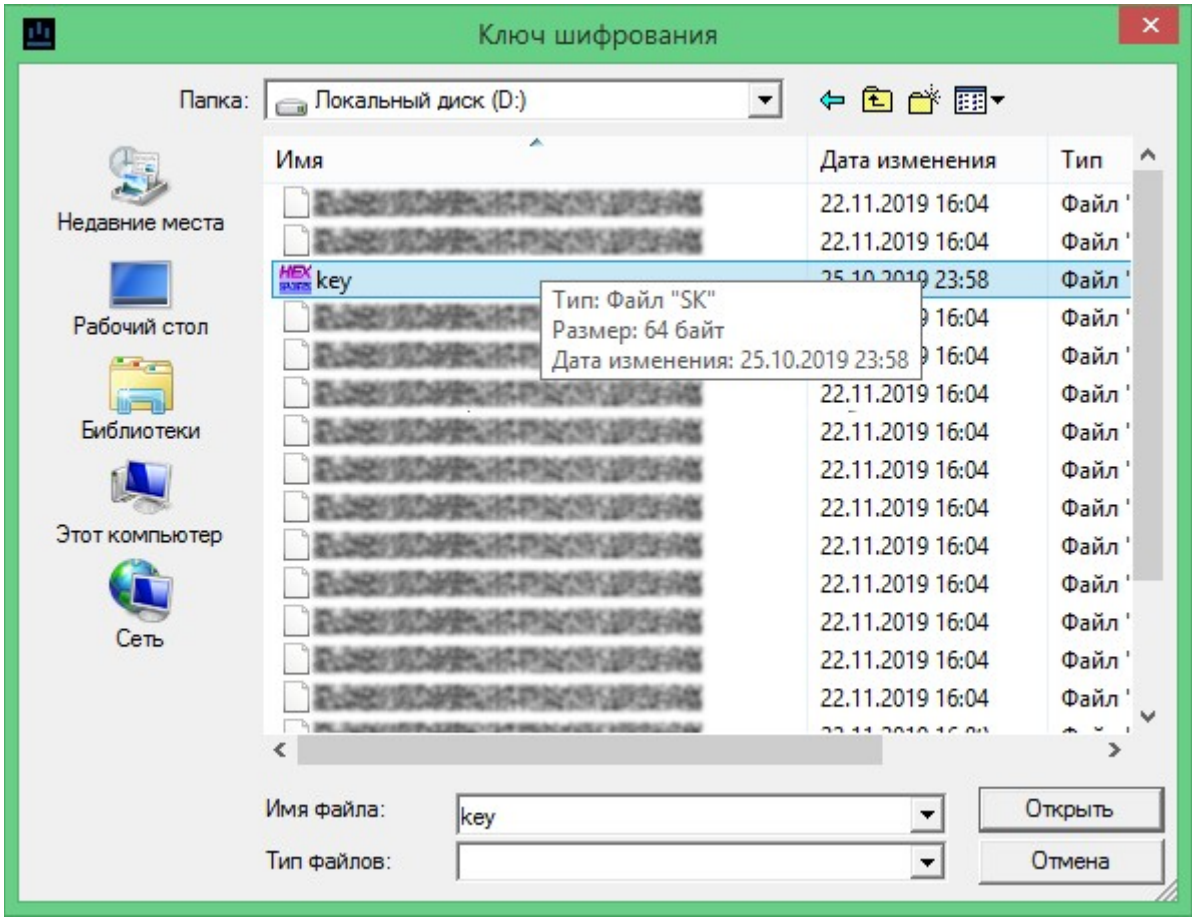


В примере выше, как раз показан прогресс генерации ключа шифрования из 256 символьного пароля. Генерация запускается только после нажатия кнопки "Старт", когда пароль введен, его длина от 8 до 256 символов и программа полностью готова к обработке данных.

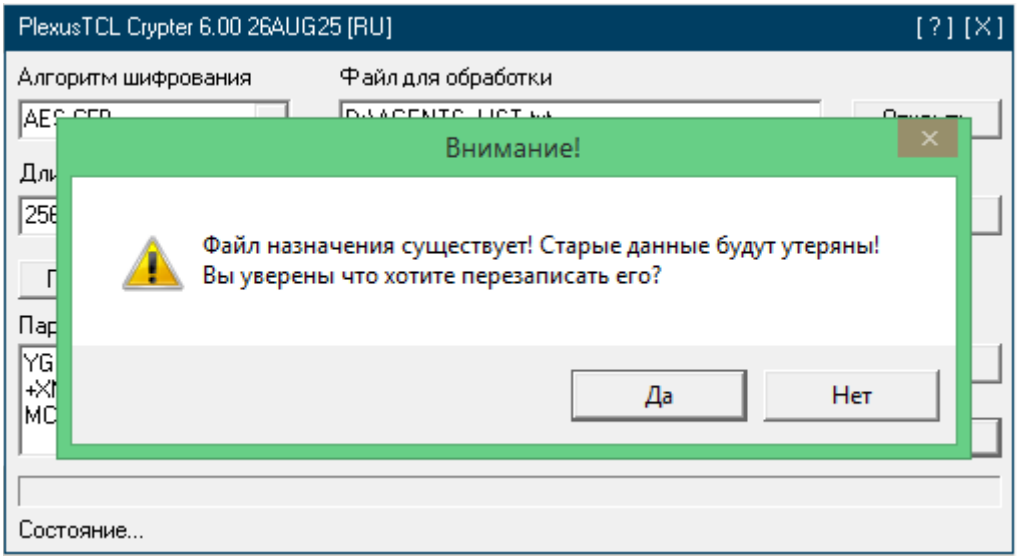


Если вы хотите использовать в качестве ключа шифрования, данные из файла (файл может быть любым, но его размер должен быть больше длины ключа шифрования в байтах или равен ей), введите в поле название файла. Это сделано для того, чтобы пользователь мог использовать в качестве ключа истинно случайные данные, хранящиеся в файле. Если вы решили использовать файл в качестве источника ключа шифрования, спрячьте этот файл подальше и никому о нем не говорите.

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.

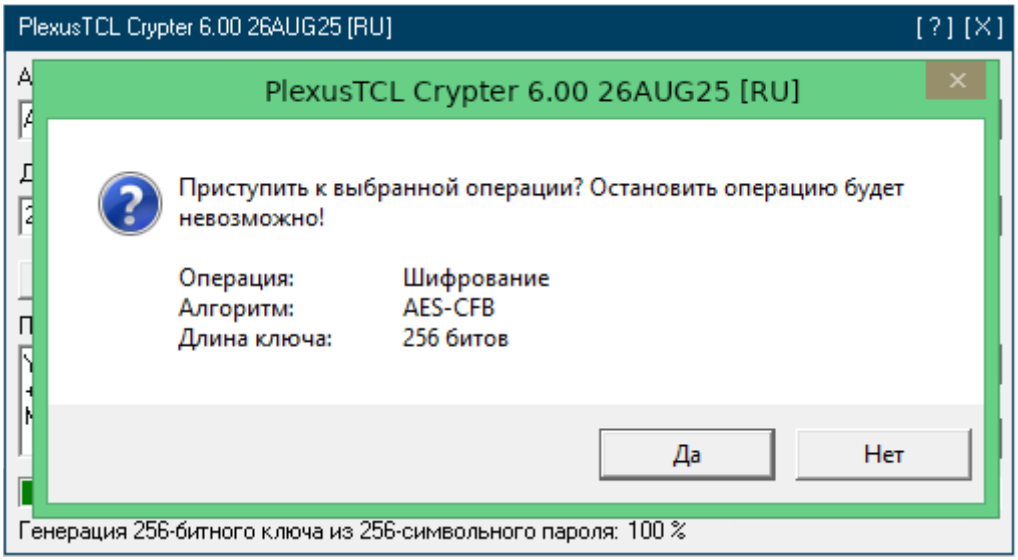


7.) Нажмите "Старт", чтобы начать операцию шифрования/расшифровки. В случае, если вы допустили ошибку, программа уведомит об этом выводом текстового сообщения. Так-же, программа уведомит о том, что в качестве файла назначения был выбран существующий файл, и предложит перезаписать его.

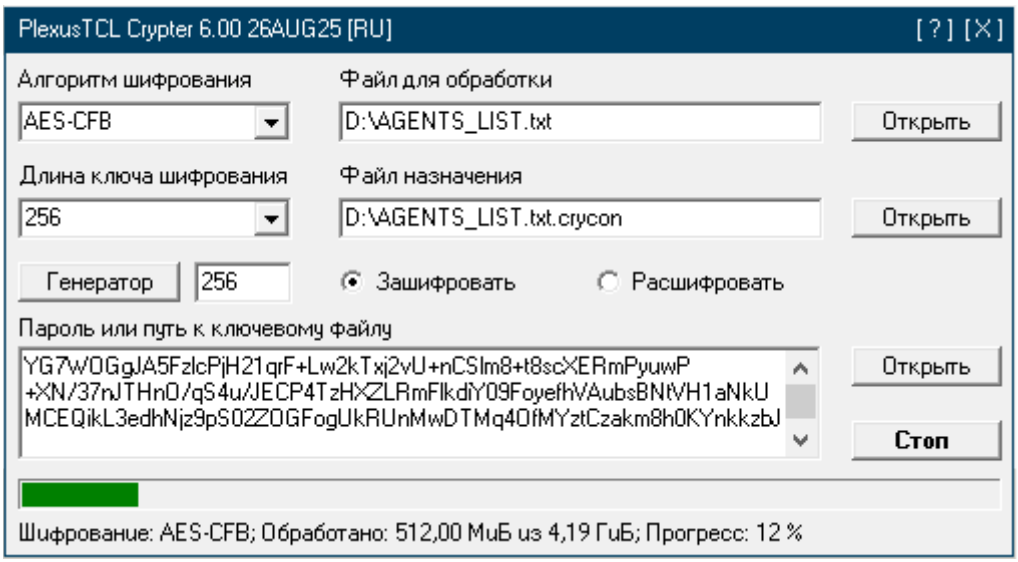


Если вы согласитесь на перезапись существующего файла, программа немедленно уничтожит все данные в файле назначения путем записи обработанных данных из файла для обработки, поверх старых данных. В случае совпадения строк в любых двух и более полях, а именно в полях "Файл для обработки", "Файл назначения" 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.



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 генерации ключа шифрования, программа отключает кнопку "Старт". Во всех программах версии ниже 5.09 кнопка отключалась так-же перед началом операции шифрования. К выпуску программы версии 5.09, автор программы наконец-то соизволил изучить мьютексы, семафоры и наконец добрался до изучения критических секций по данным, так-что с выходом этой версии шифрование стало возможно не только приостановить но и отменить. Начиная с версии 6.00, программа стала многопоточной, что облегчило использование программы (теперь GUI реагирует мгновенно). С начала шифрования файла надпись кнопки "Старт" меняется на "Стоп". Отмена запущенной операции осуществляется нажатием на кнопку "Стоп", но она не удаляет созданный файл назначения с диска.

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

Исходные коды

Программы crycon и sha256sum написаны на языке программирования C (Си) и скомпилированы в исполняемые файлы компилятором GCC версии 12.2.0.

GUI под ОС Windows представлен в двух экземплярах, в стиле Windows XP и Windows 8. GUI написан на языках программирования C, C++ и языке ассемблера под x86, и скомпилирован в исполняемый файл компилятором среды быстрой разработки приложений C++ Builder версии 6.0 и Embarcadero RAD Studio 10 соответственно.

Все исполняемые файлы в пакете "PlexusTCL Crypter" и их исходные коды, распространяются под лицензией GNU GENERAL PUBLIC LICENSE версии 3 от 29 июня 2007 года.

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

Имя файла	**SHA-2-256 контрольная сумма**
PlexusTCL Crypter 7.00 [RU].exe	8bf33feace4c53c9a354b8f8c8d84373 59857d5b399d670fb662d2bcd105df7e
PlexusTCL Crypter 7.00 [RU] [W81].exe	ffa9da88708873de4030d72c684f7100 3a8b8367e113259314a999156f7f7192
PlexusTCL Crypter 7.00 [EN].exe	8aff900a371f096363b1cbb86323369f 906cfec83c4929160e222bcdcb6a6ed6
PlexusTCL Crypter 7.00 [EN] [W81].exe	55bc5b539d33c96b56415f0066bc0d9f f96de8b7b39c899ee66c8292261e120a
crycon	5c76275ef94a6d35f48a4ac82cc4191f 3e76162da2ea849d9dadeccf6f8d80be
sha256sum.exe	01ddbf8b37961cbac3b149536fca0c3e 319f4fcd34a887f0d775bc472c0b1cb5