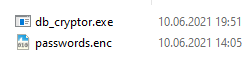
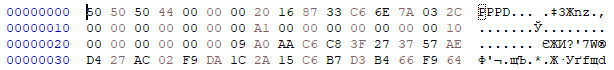
**Описание задачи:** Мой друг попросил меня разработать для него приложение, которое бы шифровала его базу с паролями. При этом он сказал, что оно должно работать только на его компьютере и быть защищено. Ну и ко всему прочему безопасно шифровать. Кажется, что я всё сделал правильно, но может вы проверите мою работу. В базе с паролями есть флаг.

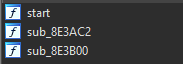
**Решение:** Исходя из описания задания нам потребуется разобраться в том, как расшифровать некоторый файл. У нас есть программа, которая его зашифровала и сам файл.



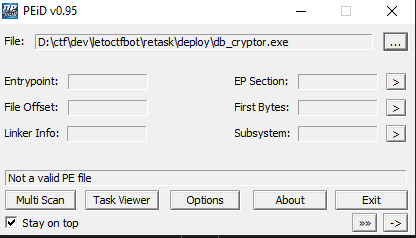
Можно взглянуть на зашифрованный файл и даже увидеть, что он имеет некоторый формат, но прогуглить его не получится.



Начнём анализ исполняемого файла загрузив его в дизассемблер. После анализа файла дизассмеблером можно понять, что он упакован, т.к. у нас есть всего 3 функции.

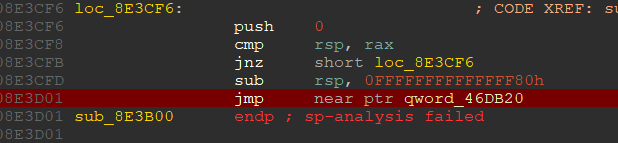


Попробуем определить, чем он упакован.

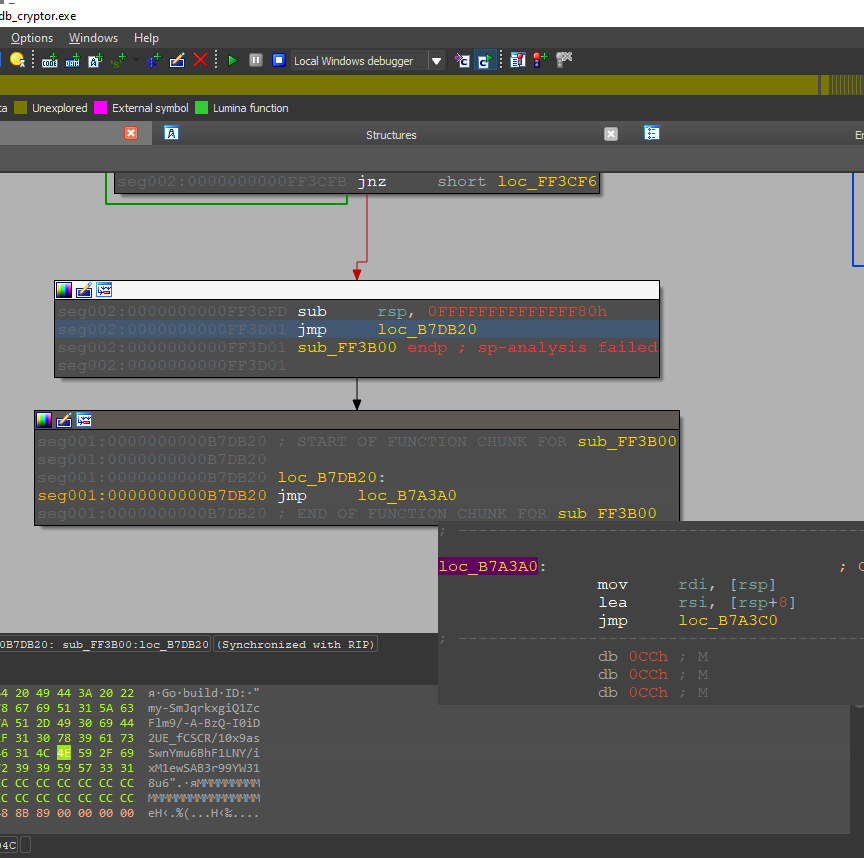


PEiD не выдаёт ничего интересного. Если иметь немного опыта в CTF-задачах или в целом в анализе различных упакованных файлов, то можно заметить, что код распаковки сильно похож на UPX, однако для усложнения распаковки были удалены метаданные. Можно попробовать восстановить их руками с помощью сравнительного анализа с другим файлов, упакованным UPX-ом, но здесь может быть проблема с тем, что разные версии UPX оставляют различные метаданные, поэтому такой путь не 100% стабилен. Наиболее надёжный вариант это сдампить исполняемый файл после распаковки и восстановить таблицу импорта.

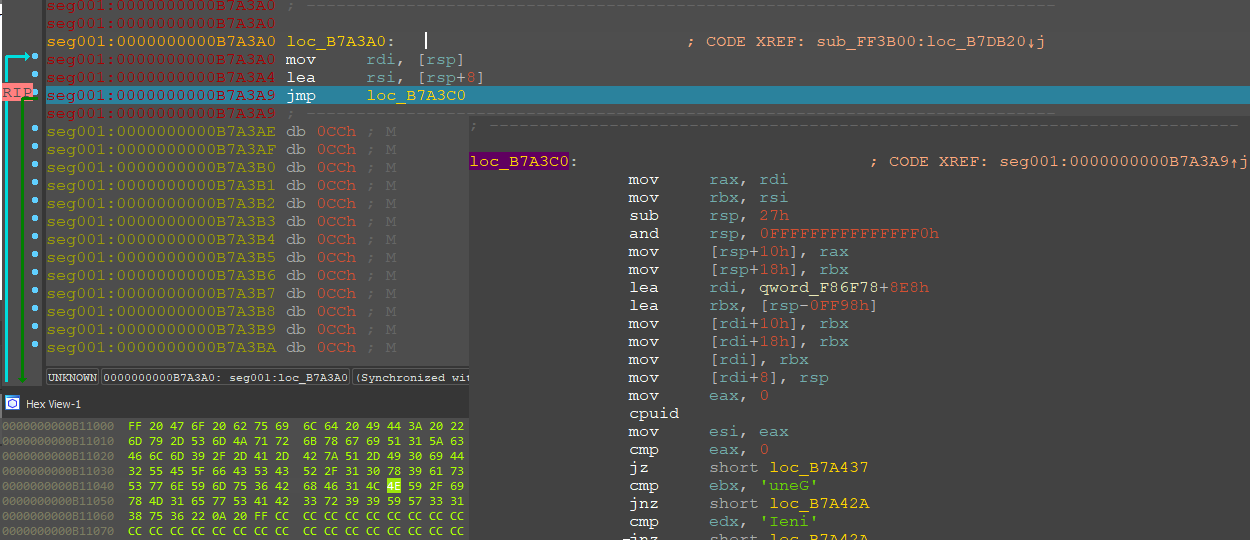
Немного проанализировав код распаковщика, можно найти место, где будет совершён прыжок по некоторому адресу, хранящемуся в памяти.



Поставим брейк-поинт туда и посмотрим, что у нас будет в памяти в этот момент.

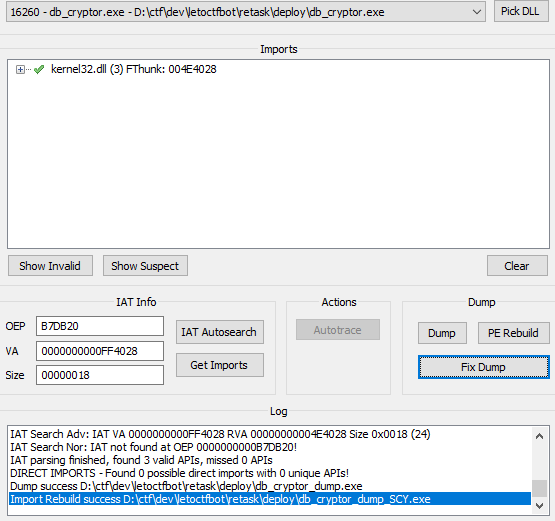


Можно заметить сразу три прыжка. Посмотрим куда будет вести финальный.



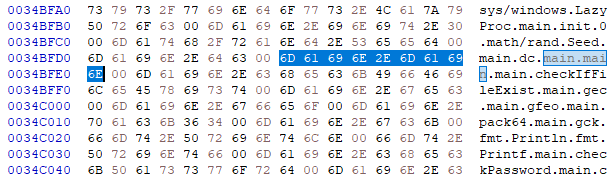
Можно заметить, что финальный прыжок перенесёт нас в некоторую функцию, в которой также встречается инструкция cupid и проверка на то, что мы запущены на процессоре Intel. Это довольно сигнатурный код для исполняемых файлов скомпилированных из Golang. Значит это можно считать нашей точкой входа, то есть код уже распакован. Теперь нам надо всё правильно сдампить. Для дампа будем использовать x64dbg и встроенное в него расширение Scylla.

Выбираем нужный нам процесс и меняем у него значение OEP на найденное нами, после нажимаем «IAT Autosearch», далее «Get Imports». Потом делаем дамп и фиксим его.



В итоге у нас получается распакованный исполняемый файл, который можно спокойно загружать в IDA PRO и анализировать. Мне не удалось нормально восстановить таблицу импорта и довольно необычно выглядит, что Scylla не смогла её найти, видимо компилятор Go как-то необычно формирует её.

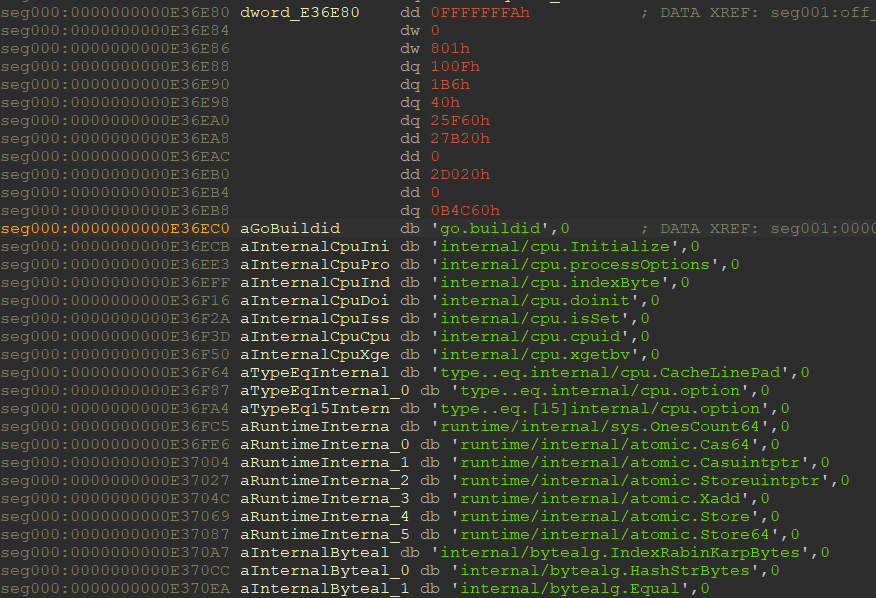
После загрузки восстановленного файла встаёт новая проблема. Как уже было раньше сказано, исходным языком для данного исполняемого файла являлся Golang, при из файла была удалена символьная информация, поэтому у нас не отображаются имена многочисленных функций, которые добавляет компилятор Go, а также имена функций, реализованных пользователем. Несмотря на это, файл получаемый после компиляции устроен так, что в нём всё равно содержаться имена функций, это можно проверить просто поискав строчки вида «main.main».



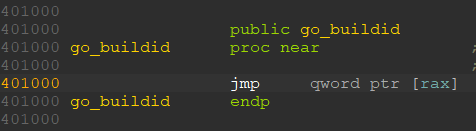
Видим множество имён вида «main.<название-функции>» это код реализованный пользователем. Но у нас появляется новая проблема, нужно как-то применить эти имена к функциям внутри исполняемого файла.

Для решения этой проблемы существует несколько инструментов, наиболее известный это GoUtils от Георгия Зайцева (<https://github.com/sibears/IDAGolangHelper>). Однако для данной версии Go этот набор скриптов не подходит. Поэтому придётся восстанавливать вручную или искать какие-то другие скрипты.

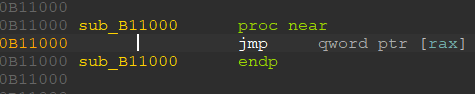
Основной задачей для нас теперь является восстановить имена функций. Для начала нужно найти, где у нас находятся сами имена. Это легко сделать, т.к. мы нашли их в 16-ричном редакторе и теперь можем посмотреть смещение до них внутри файла и найти их в базе IDA.



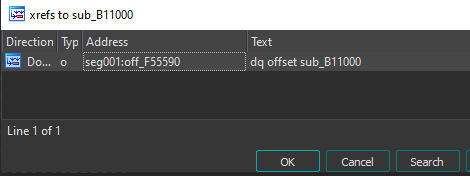
Теперь нам нужно понять, где у нас находятся указатели на функции и как их совместить с данными именами. Если собрать любой исполняемый файл из Go, то можно заметить, что функция “go\_buildid” будет находится прямо в начале секции кода.



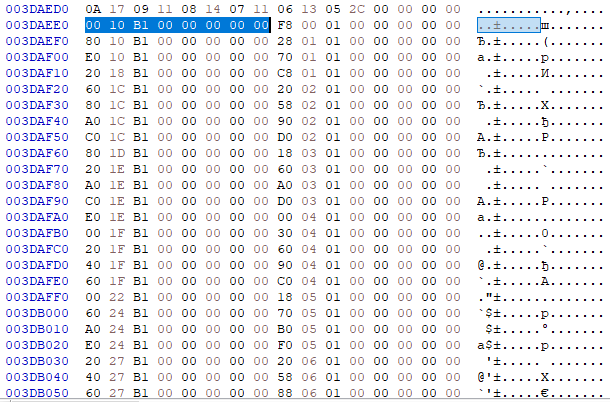
Значит в нашей базе это функция тоже скорее всего будет находиться там.



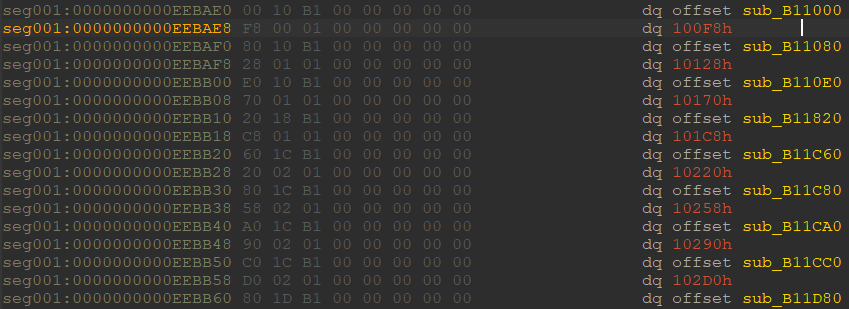
Теперь поищем места в которых есть ссылка на эту функцию и нам интересны не вызовы, а хранение указателя.



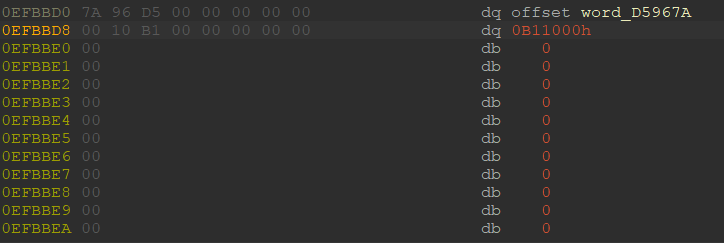
Находим всего лишь одно вхождение, но если посмотрим в ранее скомпилированном нами файле, то их будет гораздо больше. Это может быть связано с тем, что IDA просто не распознала некоторые сегменты и не распарсила их как нужно из-за отсутствия отладочной информации. Можно поискать этот адрес руками в 16-ричном редакторе. Второе вхождение поиска уже выглядит интересно.

\

Если внимательно посмотреть, то можно заметить некоторый шаблон. Каждая запись содержит 16 байт. Первые 8 байт указывают на функции и если посмотреть на них, то можно заметить, что они идут друг за другом. То есть это выглядит как список функций.



Отлично, но теперь нам надо понять сколько функций у нас есть. Находим конец данной таблицы и видим, что там уже не функция, а ссылка на какое-то поле, исходя из этого можно сделать, что это не таблица функций, а скорее таблица символов, то есть любых объектов используемых внутри программы, которые могут иметь имя.



Теперь с помощью IDA-api приведём данную таблицу к нормальному виду (используем прямо в консоли внутри IDA).

cur = 0xEEBAE0

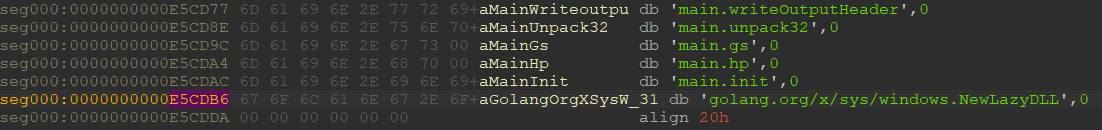
end = 0xEFBBD8

while cur < end:

ida\_bytes.create\_qword(cur, 8)

cur += 8

Отлично, теперь находим окончание секции строк.



И соберём все строки в один список (также запускаем код внутри IDA).

cur = 0xE36EC0

end = 0xE5CDDA

string\_list = []

while cur < end:

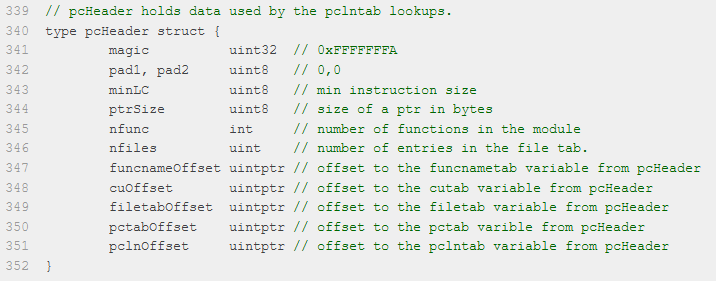
new\_str = ida\_bytes.get\_strlit\_contents(cur, -1, 0)

string\_list.append(new\_str)

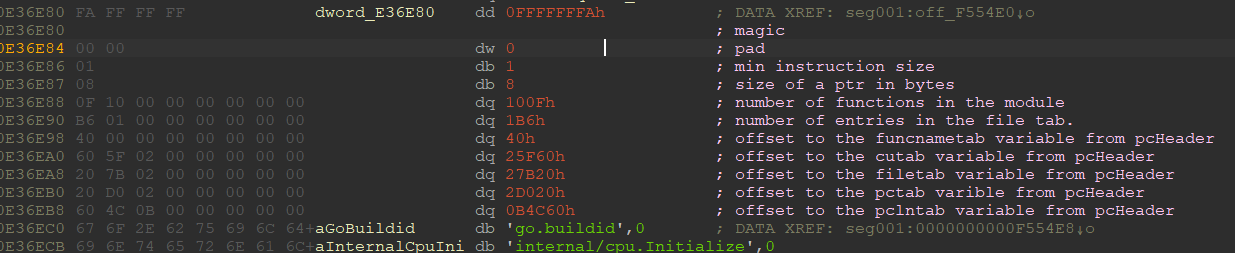
cur += len(new\_str) + 1

Если сравнить количество полученных строк и количество имеющихся функций, то мы получим разные числа, поэтому простое сопоставление по порядку может привести к некорректным результатам. Чтобы корректно восстановить символьную информацию придётся немного разобраться в том, как Golang сохраняет её. Для этого нам понадобятся исходники (<https://golang.org/src/runtime/symtab.go>)

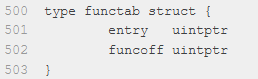
Находим следующую структуру в исходниках.



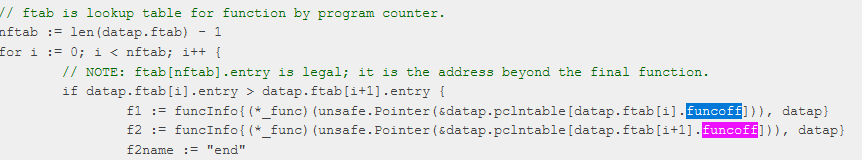
Видим, некоторую магическую константу, которую мы уже видели в месте, где были имена функций. Закомментируем поля, чтобы нам было удобнее анализировать.



Данная структура хранит в себе информацию для того, чтобы искать имена символов. Процесс поиска довольно прост, у нас есть некоторая таблица с функциями, которую мы уже видели. Каждая её запись описывается следующей структурой.



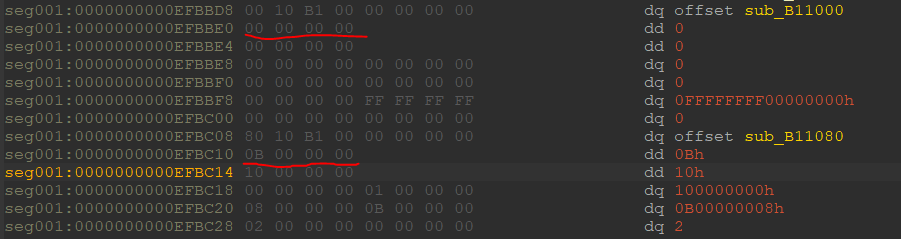
В исходном коде можно найти следующий участок кода.



Как видно, мы используем второе значение в структуре как некоторый индекс внутри таблицы pclntable. То есть чтобы получить расширенную информацию о функции нам надо взять второе поле и прибавить к началу таблицы функций. Например, для первой функции это будет рассчитываться так.



Посмотрим, что лежит по данному адресу.



Мы также находим там некоторую структуру, в первом поле которой лежит указатель на функцию, а второе поле из 4 байт представляет собой смещение внутри таблицы имён функций, это не сложно понять, просто проанализировав две подряд идущих записи. Таким образом мы можем обработать все наши символы и обозначить их именами. Для этого напишем небольшой скрипт с помощью IDA-api. Также в скрипте реализуем автоматического создание функции на месте, которое проименовываем, потому что IDA не распознала все функции корректно. В итоге получаем примерно такой скрипт.

from idaapi import \*

from idc import \*

pclntab = 0xEEBAE0

fname\_tab = 0xE36EC0

def get\_sym\_name(ea):

offset = get\_dword(pclntab + get\_qword(ea + 8) + 8)

return ida\_bytes.get\_strlit\_contents(fname\_tab + offset, -1, 0)

def set\_sym\_name(ea, name):

set\_name(ea, name)

def main():

global pclntab

cur = pclntab

while cur < 0xEFBBD0:

sym\_ptr = get\_qword(cur)

name = get\_sym\_name(cur).decode().replace("\\", "\_").replace("/", "\_")

set\_sym\_name(sym\_ptr, name)

create\_insn(sym\_ptr)

ida\_funcs.add\_func(sym\_ptr)

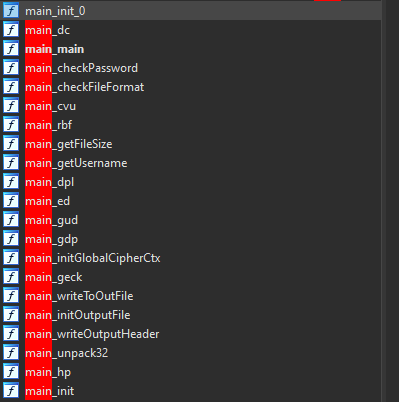
cur += 16

if \_\_name\_\_ == "\_\_main\_\_":

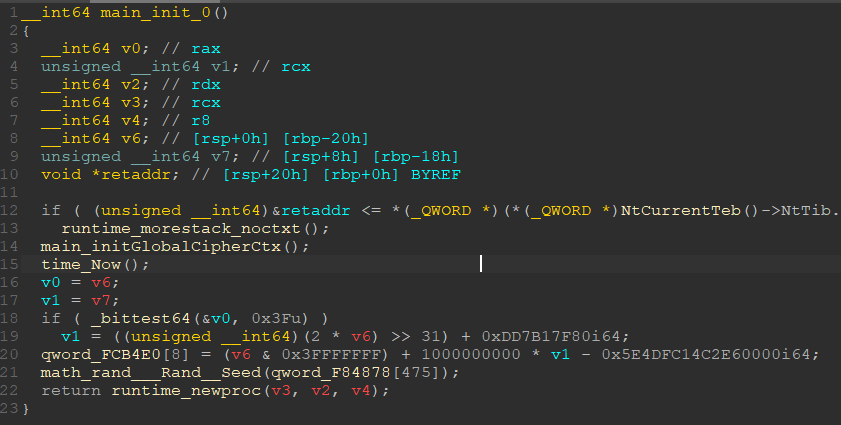
main()

Запускаем его и ждём пока он выполнится, а также пока пройдёт новый автоматический анализ. После первого запуска могут восстановиться не все символы, но это можно исправить, просто пройдясь руками по базе и восстановив некоторые оффсеты в таблице, потом пройти повторно.

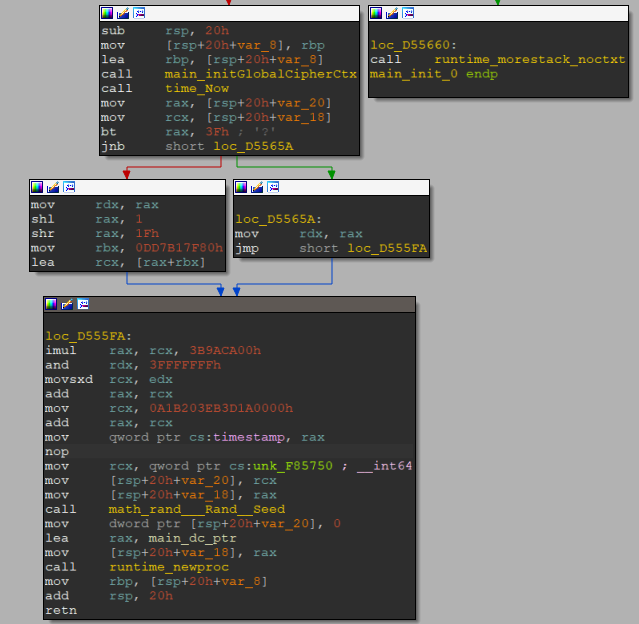
В итоге получим полностью восстановленные символы.



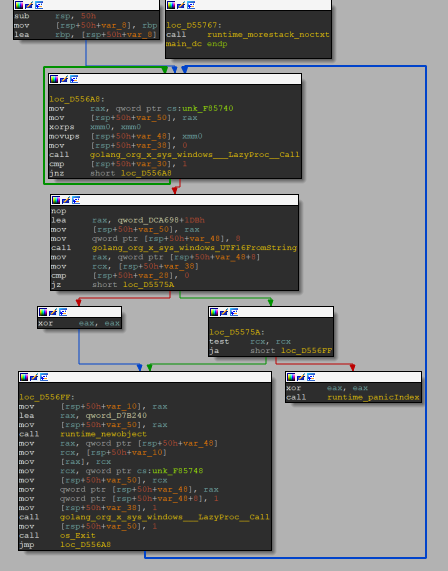
Теперь можно приступать к анализу функционала исполняемого файла. Анализ стоит начать с функции main\_init\_0.



В декомпилированном виде выглядит довольно неприятно, поэтому удобнее будет посмотреть ассмеблер. По коду довольно, очевидно, прослеживается вызов функции получения времени, а также запуск горутины. В качестве точки входа для запускаемой горутины используется функция main\_dc. Также можно заметить, что время используется как инициализирующее значение для рандома.



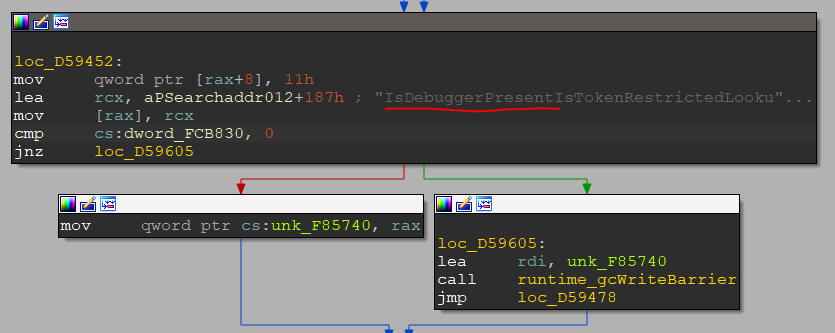
Также интересен вызов функции “main\_initGlobalCipherCtx”, название функции говорит само за себя. Для начала посмотрим на функцию main\_dc. Данная функция содержит бесконечный цикл в котором вызывает некоторую процедуру.



Можно заметить, что происходит вызов метода «golang\_org\_x\_sys\_windows\_LazyProc\_Call» это вариант вызова WinAPI через Golang. Если посмотреть примеры исходного кода, то станет понятно, что чаще всего вызов происходит приблизительно таким образом:

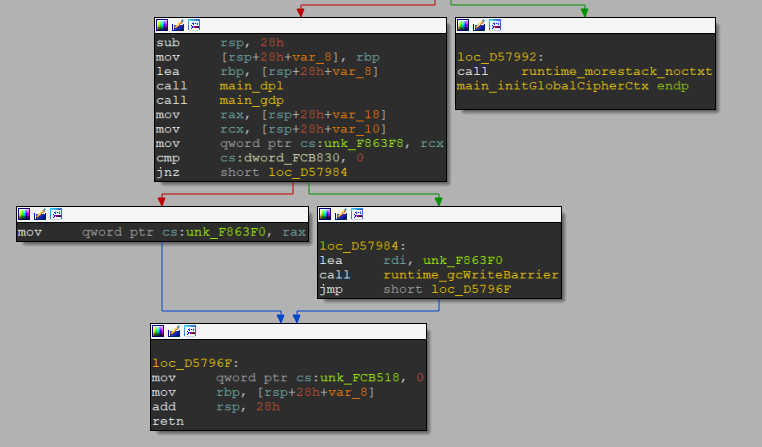
1. Открывается некоторая dll-ка
2. В ней находится нужная функция
3. Адрес нужной функции помещается в некоторую локальную или глобальную переменную.
4. Использую эту переменную происходит вызов

Значит нам нужно найти инициализацию используемых глобальных переменных. По перекрёстным ссылкам это сделать не сложно.



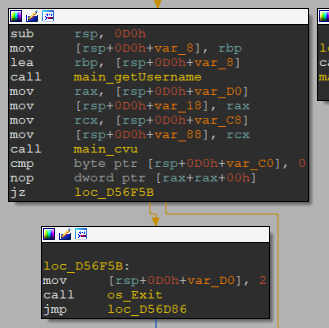
Находим, что одна из функций – это функция проверки отладки. В итоге можно понять, что main\_dc – это Debug Check. То есть отдельный поток для проверки отладки. Можем запатчить данную функцию, чтобы она не обнаруживала отладчик.

Далее проанализируем функцию «main\_initGlobalCipherCtx».

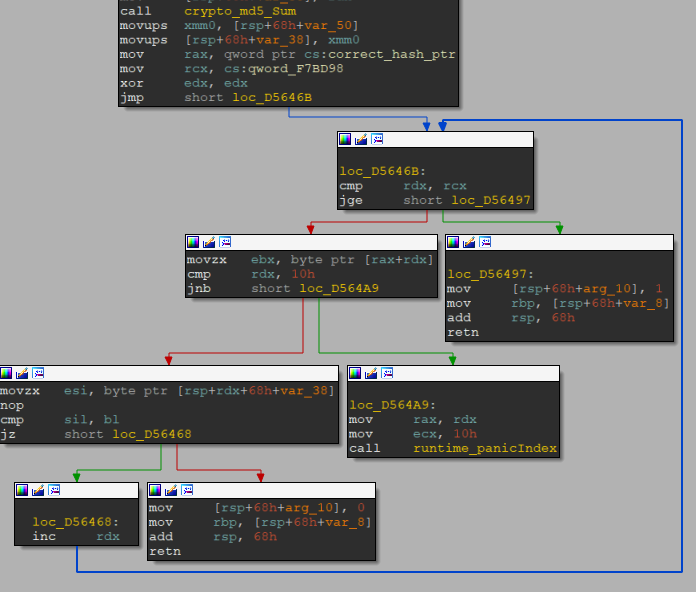


Здесь происходит вызов двух функций “dpl” и “gdp”. Результат gdp сохраняется в некоторой глобальной переменной. Разберём их по порядку.

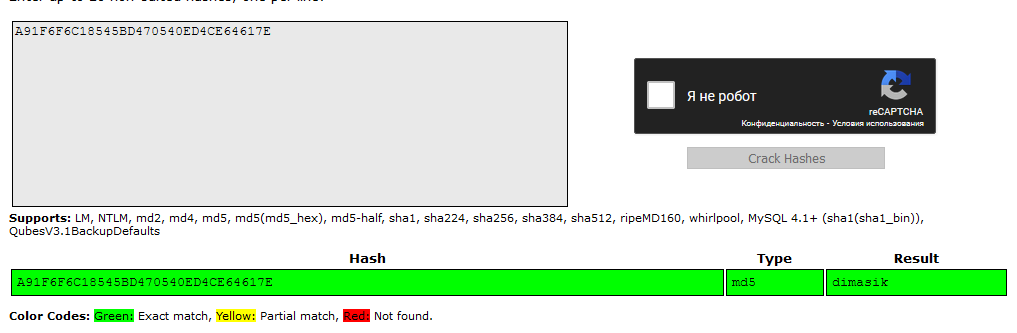
В функции “dpl” в самом начале получается имя пользователя, после чего вызывается функция “cvu”.



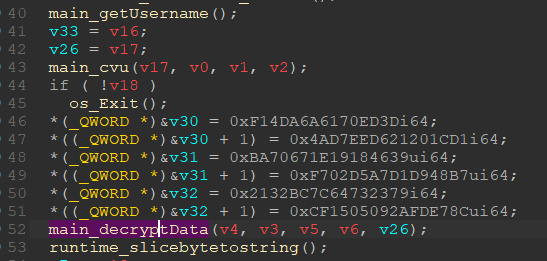
Если проанализировать функцию “cvu” то станет понятно, что она просто рассчитывает MD5-хэш от аргумента и сравнивает с некоторым хэшом в памяти. Не сложно понять, что расчёт хэша идёт от имени пользователя.



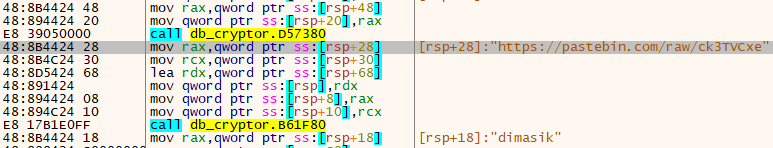
Достанем хэш из памяти и попробуем прогуглить его.



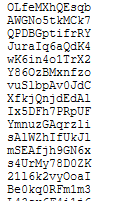
Находим корректное имя пользователя. Теперь мы знаем, что для запуска исполняемого файла нам надо, чтобы имя нашего пользователя было «dimasik». Но, можно и запатчить этот код, однако позже с этим могут возникнуть проблемы. Дело в том, что далее в этой функции расшифровывается некоторый блок данных и ключом для расшифровки служит как раз имя пользователя.



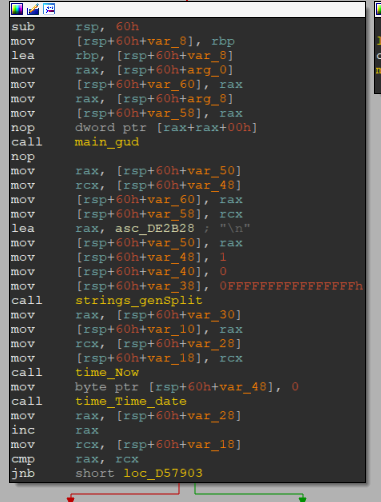
Для того, чтобы быстрее узнать, что именно содержится зашифрованном блоке, мы можем запустить отладку и остановиться именно в этом месте, предварительно обойдя антиотладку (подробно описывать этот процесс я не буду, по сути, мы просто попадаем на OEP и патчим результат функции получения имени пользователя).



Получаем некоторую ссылку на pastebin. Посмотрим, что находится по данной ссылке.



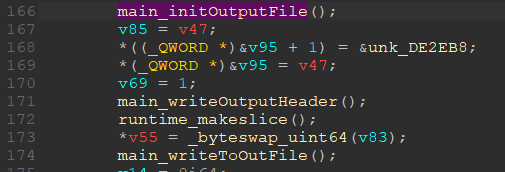
Некоторый список слов одинаковой длины, всего 366 штук. Каждое слово на отдельной строке. Также в отладчике можно заметить, что полученная ссылка передаётся на вход функции «main\_gdp». Проанализируем эту функцию.



Данная функция вызывает «main\_gud» (данная функция получает данные по ссылке, которую мы получили на прошлом шаге), после чего разбивает все полученные данные по символу переноса строки и получает текущий день в году. В итоге мы получаем строку из увиденного нами списка. Индекс строки соответствует текущему дню в календаре (поэтому строк и 366). Полученный пароль записывается в некоторую глобальную переменную.

Отлично, теперь мы знаем некоторый глобальный пароль и даже можем посмотреть, где он используется, но пока что нам это особо не нужно. Теперь нам надо понять, как шифруется файл. С помощью восстановленной символьной информации это сделать не сложно.

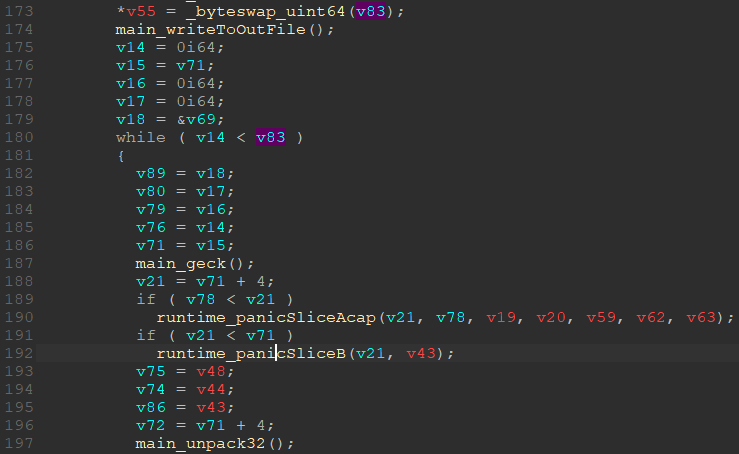
В функции «main\_main» можно найти следующий участок кода.



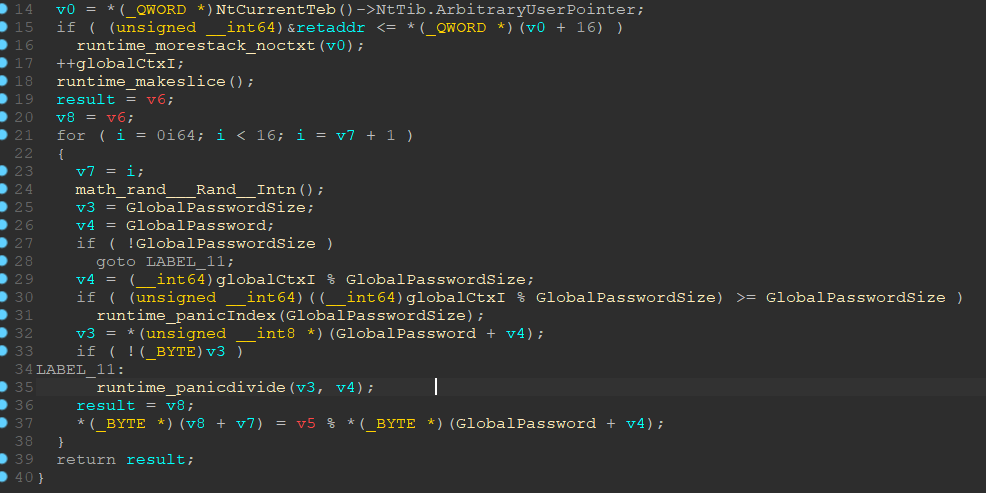
Судя по всему здесь инициализируется выходной файл и в него записывается заголовок. Проанализировав функцию «main\_writeOutputHeader» можно понять, что в файл последовательно записывается следующее:

1. Магическая константа из 4 байт - 50 50 50 44
2. Число 32 в формате int32 BE
3. Timestamp в формате uint64
4. Номер текущего дня в формате uint64

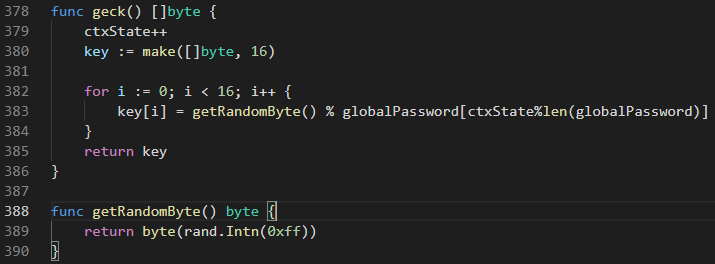
Далее в файл записывается некоторое число, которое обозначает количество записей в базе данных паролей. Это можно выяснить путём анализа последующего цикла, который на каждом из шагов шифрует одну запись и добавляется их в некоторый список.



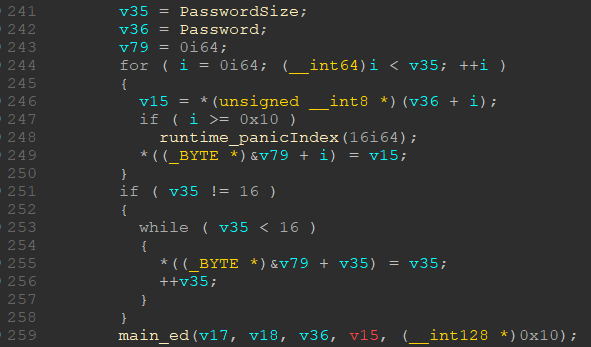
При этом каждая запись шифруется своим ключом. Ключ получается с помощью функции «main\_geck». Данная функция выполняет некоторые преобразования используя глобальный пароль (получаем на основе текущего дня в году) и генератора случайных чисел.



Если протрассировать код или просто внимательно проанализировать в статике, то станет понятно, что каждый пароль будет размером 16 символов. Очередной символ рассчитывается как остаток от деления некоторого случайного байта, получаемого через генератор случайных чисел на символ пароля, который выбирается циклическим образом на основе глобального счётчика. В исходном коде это выглядит так.



Таким образом зная инициализацию рандома и день года, мы можем восстановить пароль для каждой из записей. Однако, это не поможет нам расшифровать все записи, потому что перед записью в файл они шифруются все вместе и ключом является пароль, введённый пользователем.



Можно заметить, что после записи зашифрованных данных записываются ещё некоторые данные. Это результат функции «main\_hp» которая в качестве аргумента принимает пароль пользователя. Данная функция преобразует пароль, но не хэшированием а просто заменой по таблице. Таким образом, что его можно восстановить.

В итоге мы получаем следующий план действий:

1. Считать временную метку и инициализировать рандом
2. Считать день года и получить пароль с pastebin-a
3. Считать преобразованный пароль из конца файла и преобразовать обратно
4. Расшифровать с помощью пароля основной блок
5. Расшифровать каждую запись

Пример решения находится в папке solution/.

