

3.14 Petabytes

Переписки между Доктором F. и Доктором D. порой содержат довольно много вещей, которые не так-то просто поддаются объяснению. Недавно мы столкнулись, например, вот с таким:

[2217-09-11 12:13:45] Fade
Ты уже собрал данные?
[2217-09-11 12:15:02] Demon
Да. Получилось чуть больше 3 РВ...
[2217-09-11 12:15:18] Fade
Да блин, ты снова терагерцовые дампы с тарелок в реалтайме снимал, чтоли?

[2217-09-11 12:15:26] Fade
Пространство дисковое девать некуда?
[2217-09-11 12:17:20] Demon
Да нет, там просто сборка новая подъехала, с исходниками. Они кстати половину почти весят.

[2217-09-11 12:18:46] Fade
Мда... Ну ладно, давай сюда.
[2217-09-11 12:21:21] Demon

[<ссылка скрыта>](#)

[2217-09-11 12:30:56] Fade
Погоди, а что это вообще?
[2217-09-11 12:34:17] Demon

[<изображение>](#)

```
Z:\>copy /b sources.tar+flag.tar+binaries.tar data.tar
sources.tar
flag.tar
binaries.tar
      1 file(s) copied.
```

[2217-09-11 12:35:00] Fade
Окей, ясно, скачиваю.
[2217-09-11 13:50:11] Fade
Погоди, а докачка-то есть хотя-бы?!

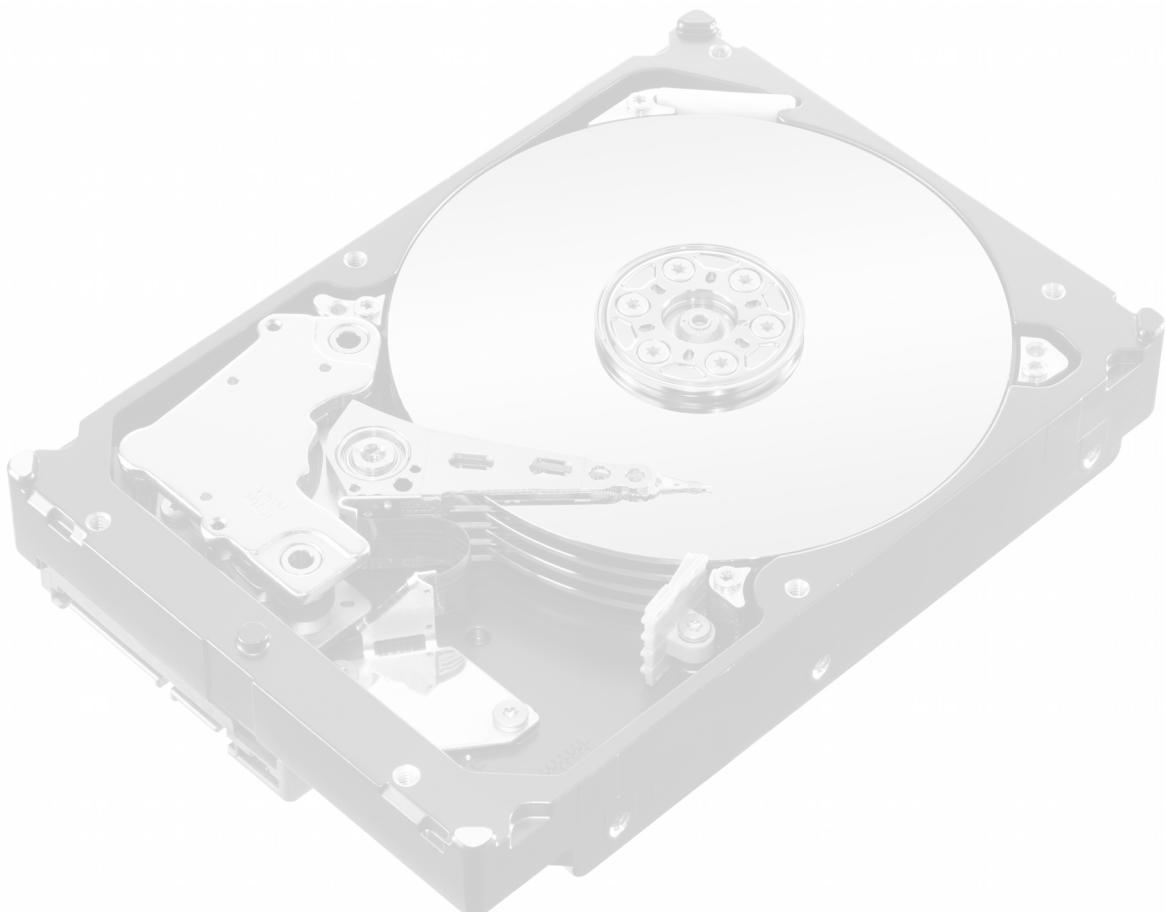
[2217-09-11 13:51:12] Demon
Да, настроил только что



Возможно, что исходные тексты программ объёмом в несколько сотен терабайт — обыденное явление для 2217 года, но нас не это интересует. В силу ограниченности пропускной способности наших сетевых каналов, скачать такой объём данных нам не под силу, хотя

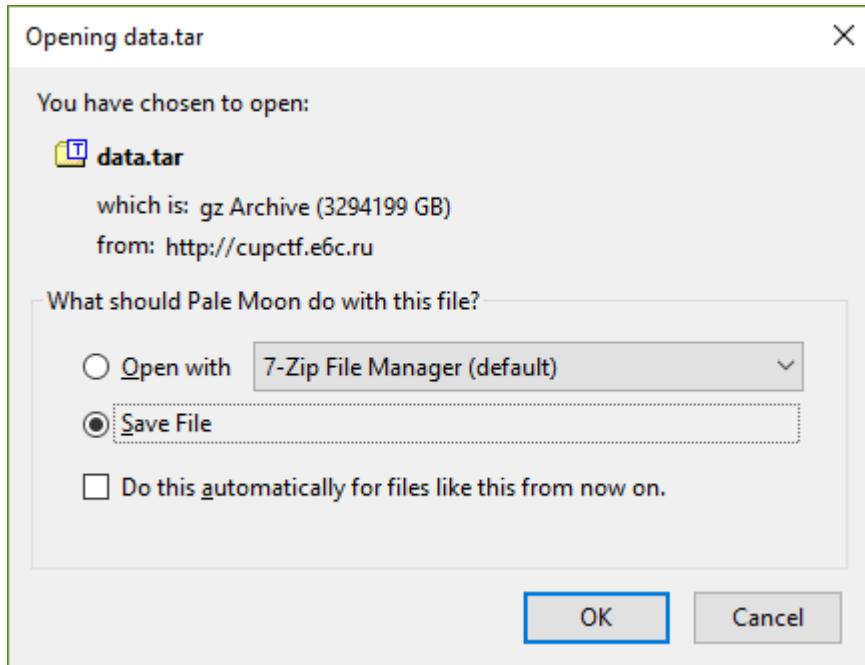
нам даже удалось проложить временной туннель в 2217 год, получив таким образом доступ к скачиванию того самого файла `data.tar`. Вы можете [воспользоваться](#) данным туннелем — он в целом относительно стабилен, однако в силу сильного локального искривления пространства-времени, имеет большие задержки, и невысокую скорость. Иногда соединение рвётся, правда. Но мы на вас рассчитываем...

Ваша цель — заполучить содержимое файла `flag.tar`, который судя по картинке из переписки, был отправлен Доктором Д. среди прочего содержимого `data.tar`. Внутри вы и найдёте флаг...



Флаг: **specific_binary_search_zhd4mhuv**

Итак, по ссылке, которую нам дают, действительно открывается файл **data.tar**:



И его объём действительно около 3.1 петабайта. И нет, скачивать целиком это не надо пытаться. На это уйдёт при выделаемой скорости в 100 кб/с около 1121 года.

Так распоряжает к тому, что-бы немного подумать, найти возможно не совсем стандартный подход.

Во первых, в условии что-то сказано про докачку. А что такое, собственно, докачка в отношении HTTP? Это вкратце заголовок запроса *Range*, и заголовки ответа *Content-Range*, которые (если вкратце), позволяют у сервера запрашивать не весь файл, а произвольными кусочками, в самом простом случае (который используют как минимум Firefox и wget для продолжения оборванной загрузки) позволяя пропустить произвольное количество байтов с начала скачиваемого файла. Пример HTTP запроса и ответа при докачке:

```
GET /bakupka.tgz HTTP/1.0
Range: bytes=122580-
User-Agent: Wget/1.12.1-dev
Accept: */*
Host: example.com
Connection: Keep-Alive
```

```
HTTP/1.1 206 Partial Content
Server: nginx/1.8.1
Date: Mon, 11 Sep 2017 05:50:47 GMT
Content-Type: application/x-gzip
Content-Length: 64689037896
Connection: keep-alive
Last-Modified: Sat, 09 Sep 2017 10:01:52 GMT
ETag: "f0fc6411c-558bec85b2400"
Accept-Ranges: bytes
Content-Range: bytes 122580-64689160475/64689160476
```

В данном примере, файл `bakupka.tgz` запрашивается не весь, а начиная с байта 122580 в нумерации с нуля. Ах да, ну ещё при этом код успешного ответа будет не 200 OK, а 206 Partial Content.

Вся эта наука как-бы намекает нам, что с помощью нехитрого использования этой возможности, у нас есть произвольный доступ к любому месту этого гигантского файла на З РВ, и мы можем легко скачивать любые его кусочки, правда небольшие. Но что это нам даёт?

Небольшой спойлер: пусть файл с флагом вешает около 20 кб. Из чего следует, что вероятность случайно в него попасть, качая случайные кусочки равна приблизительно $20 \times 1024 / (3.1 \times 1024^5) \approx 5.8 \times 10^{-12}$. Честно говоря, я думаю, что вам для успеха не хватит либо терпения, либо везучести (а скорее всего, и того и другого), поэтому, давайте внимательнее посмотрим на условие:

```
Z:\>copy /b sources.tar+flag.tar+binaries.tar data.tar
sources.tar
flag.tar
binaries.tar
      1 file(s) copied.
```

Здесь мы вроде-бы видим намёк, что файл склеен из трех архивов, содержащих однотипные данные. Вернее, flag.tar скорее всего небольшой (иначе, как мы его скачаем-то в итоге?), а вот остальные 2 файла раздуты на все 3 петабайта. И если мы немного покачаем начало файла, то действительно увидим, что скачиваются довольно однотипные данные — исходные тексты программ. А если попробовать скачивать ближе к концу? Да, ожидаемо — нам посыпется мусор в виде каких-то exe и dll, содержащих что-то явно не текстовое. То есть, мы явно можем отличить эти данные друг от друга, даже программно.

Собственно на этом месте, собрав всё воедино, остаётся только сообразить, что тут нужно всего-то использовать [двоичный поиск](#), который в итоге обязан сойтись где-то между этими двумя разнородными архивами, и (сюрприз!) попасть прямо в наш флаг. Неожиданно, правда? Хотя, по моему, всё очень даже ожидаемо...

Ну, тут в общем нечего более добавить. Разве что ещё один спойлер: флаг начинается с байта **1025764463940096**. И да, никакого файла на 3.14 петабайта на самом деле не существует, это всё мистика наличие этого файла нехитрым образом полностью эмулируется скриптом. Хотя если признаться — не таким уж и нехитрым...

Код активации

Много воды утекло после событий на печально известной атомной электростанции, произошедших в далёком 2217 году. Хотя, более точно будет сказать — много снега насыпало... Доктор F. тогда вовремя подсуетился, подыскав себе бункер. Большую часть времени он посвящал расследованию того, что же на самом деле произошло тогда, и как это было связано с таким резким изменением климата.



Из данных, которые тогда Доктором F. совершенно случайно были сняты с некого низкочастотного радиоканала говорили о том, что ПО станции попросту удалённо взломали, и в обход всех возможных систем вызвали критическую перегрузку. Реактор на такое был совсем не рассчитан, и в считанные секунды довольно красиво взлетел на воздух, расплескав по территории города изрядное количество радиоактивных отходов. Казалось бы, вопросов особых не остаётся (за исключением того, кому это было нужно), однако, возникает несостыковка — для наступления ядерной зимы взрыва небольшой электростанции явно недостаточно.



* * *

Объединив все собранные нами данные воедино, напрашивался вывод, что тут было замешано ядерное оружие. Радиопередатчик на 66 кГц установленный прямо под электростанцией появился там не просто так. Если бы Доктор Ф. догадался бы вести свой радиоприём не на фиксированной частоте, а записывать целиком полосу низких частот в высоком разрешении, то он легко бы мог обнаружить, что перед самым взрывом в радиоэфире пронеслось нечто, похожее на многократно повторяющийся цифровой сигнал, несущий в себе какой-то осмысленный двоичный код. Этот код, судя по нашим данным — был ключом к разгадке случившегося.

Нельзя сказать точно, когда именно произошел катаклизм, повлекший за собой такое кардинальное изменение климата, хотя-бы потому, что была сильно повреждена любая существующая информационная (и не только) инфраструктура (что говорить, некоторые города попросту провалились в образовавшиеся разломы в земной коре), и какие-либо записи с того времени вообще разыскать трудно, однако всё указывает на то, что это началось вскоре после этого взрыва, через часы, дни, месяцы...



Теперь отметим вот что. Все ядерные боеголовки, производившиеся с 2160 года, имели стандартизованный блок управления, который как раз-таки умел принимать команды по радио интерфейсу, и формат этих команд был очень похож на тот самый сигнал, отправленный в момент взрыва. Если точнее, то это должен был быть уникальный код боеголовки, который её блок управления и ожидал получить, только для избежания случайного срабатывания код должен был многократно повторяться (что собственно и наблюдалось). Сами коды, кстати, были длиной всего лишь 32 бит.

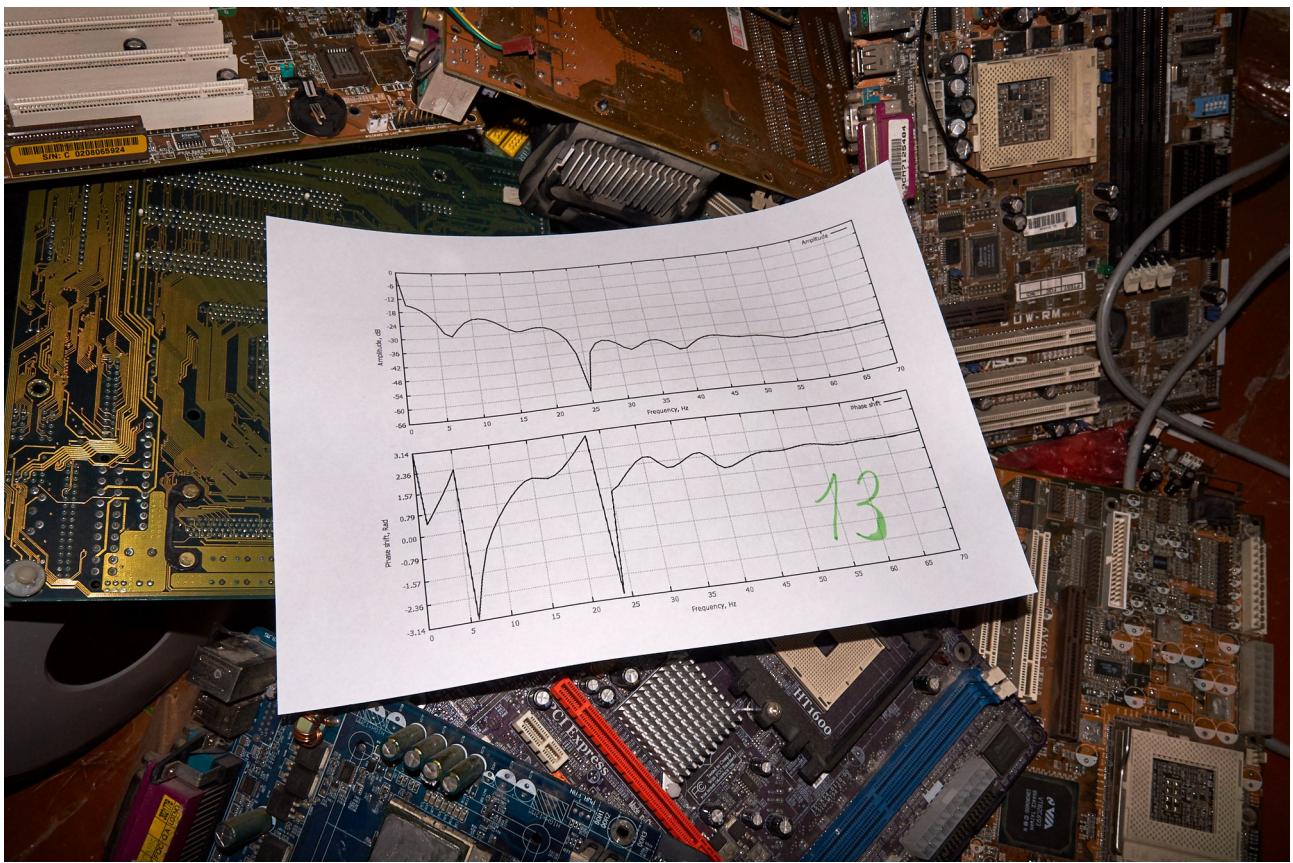
Для каждой боеголовки, код, по идеи, должен был быть уникален. Однако код этот хранился в памяти специальной микросхемы, похожей на флеш-память, и теоретически мог был быть после производства изменён, однако, дела обстояли даже хуже. Похоже, что одна из боеголовок была попросту многократно скопирована молекулярным репликатором — так китайцы в далёком 2200 году додумались удешевлять стоимость производства, благо технологии уже такое позволяли. Соответственно, все боеголовки имели один и тот-же код — их вряд-ли бы стал кто-то потом специально менять.

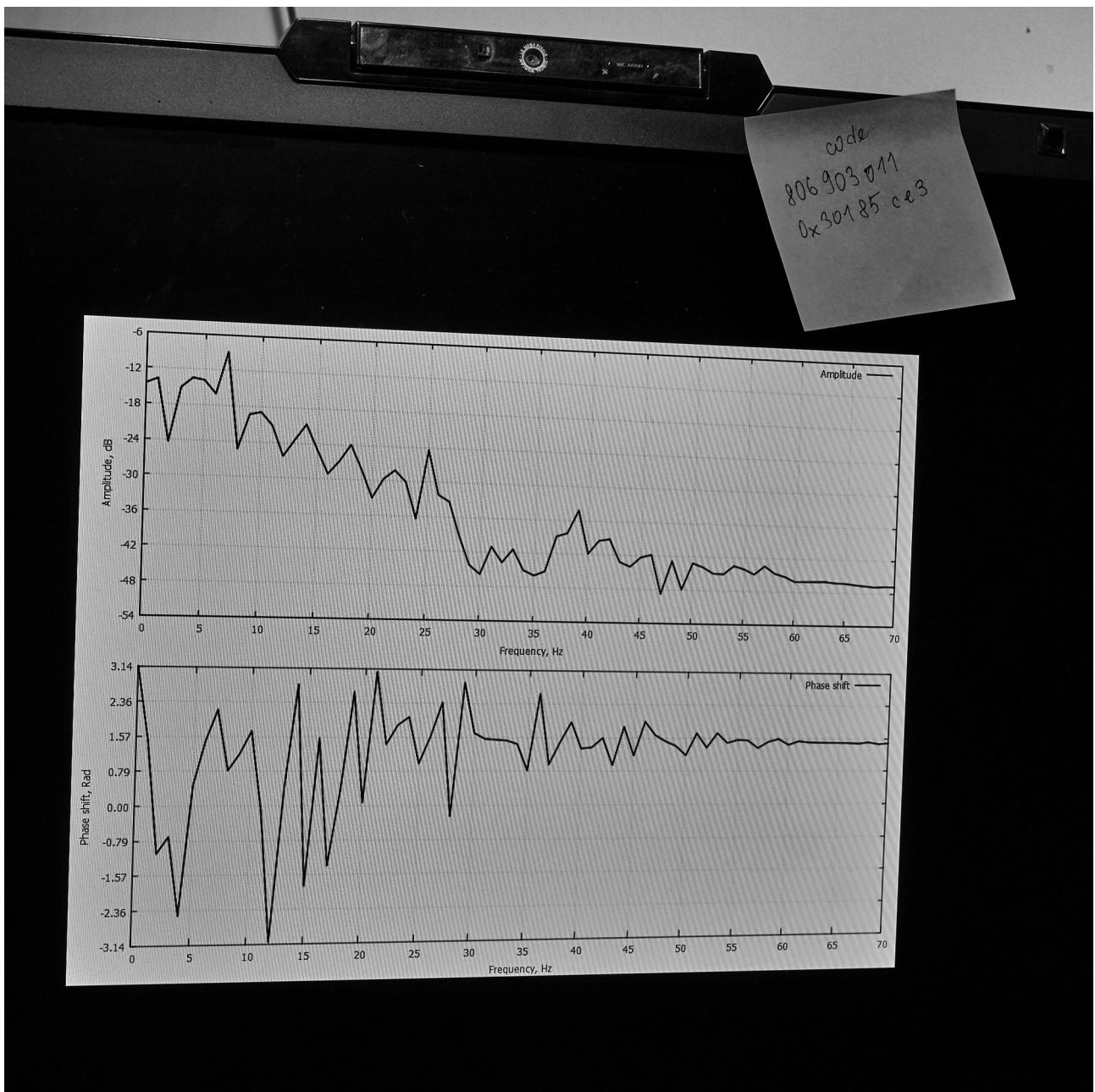
Таким образом, один единственный правильный радиосигнал, разосланный достаточно мощным передатчиком, запросто мог положить начало глобальному ядерному апокалипсису...



* * *

Отладочные комплекты к этим самым боеголовкам были, мягко говоря, странноваты. Например, то-ли в целях безопасности, то-ли просто из-за своей примитивности, они не позволяли нормально перехватывать коды активации. Точнее, позволяли. Но... В общем, в наших архивах было найдено несколько вот таких очень между собой похожих документов:





Доктор F. одно время имел дело с такими блоками управления, и довольно быстро сделал софт, позволяющий по графику в этих документах восстанавливать сам код, и даже генерировать примерную форму его сигнала. Вам мы передадим архив, содержащий полный набор найденных документов, вместе с семплами восстановленных сигналов — все это позволит вам получить базовое представление о том, что они из себя представляют.

Мы хотим поставить, наконец, точку в этой истории. И для этого нам не хватает последнего звена — того самого кода, который был отправлен в момент взрыва электростанции. Хотя по счастливому

стечению обстоятельств, в нашем архиве нашелся соответствующий документ, но только...



Сущности, по вине которых произошла катастрофа хорошо постарались, заметая следы. Но не учли, что это дело попадет в руки к вам. Вы должны повторить «подвиг» Доктора F. и восстановить код активации по этому снимку. В архиве, который мы вам передали вы найдёте эту картинку в оригинальном разрешении.

* * *

По семплам сигнала можно понять, как именно кодируется его численное представление — да и ничего необычного там нет. Попробуйте установить связь снимков с семплами. Это должно помочь понять, как их можно трансформировать обратно в семплы. В любом случае, мы на вас как всегда рассчитываем...

Ваш флаг — код активации в десятичной системе счисления.

Флаг: 2857710472

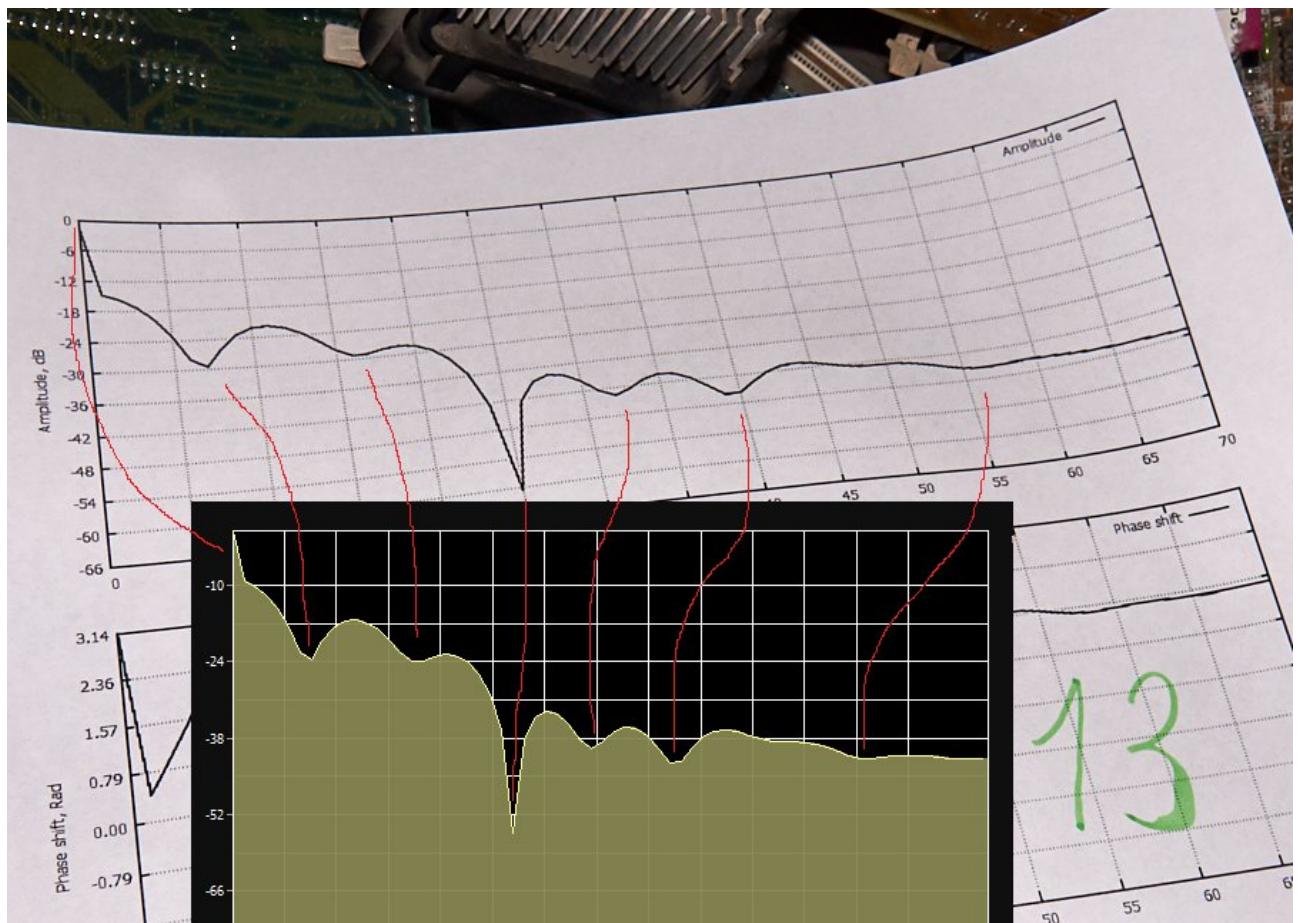
"Нужно больше БПФ в ваших тасках,
милорд"

Dr. Fade

По шкале упоротости, этот таск явно заслуживает 10/10 — добавить тут нечего. Но попробуем разобраться.

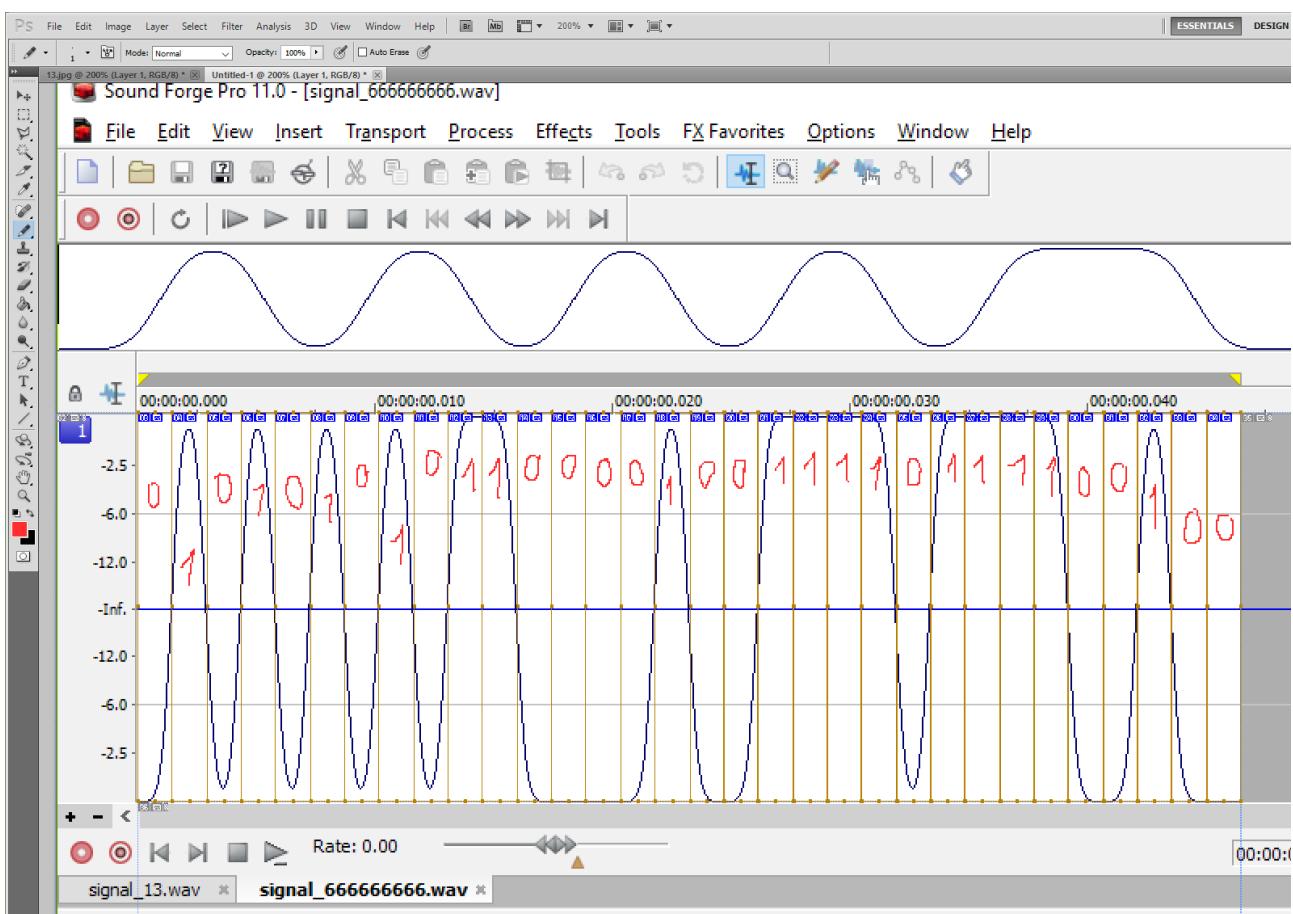
Нам дают серию фоток с чем-то, напоминающим график спектра, и WAV файлы с короткими записями какого-то низкочастотного звука. Для каждого файла есть своя фотография со спектром, и подразумевается, что сигнал должен кодировать какое-то число, скорее всего, 32-битное.

Можно быстро понять, что спектры на фотографиях **почти** полностью сосуществуют **реальным** спектрам аудиозаписей, которые можно увидеть к примеру в Sound Forge (правда, без фазы):



Из некоторых фундаментальных знаний известно, что чтобы восстановить по Фурье-образу исходный сигнал, надо знать его **амплитудную и фазовую** составляющие (либо действительную и мнимую часть комплексных чисел, которые и есть Фурье-образ), из чего можно сделать вывод, что по графику который нам дают **можно** восстановить исходный звуковой файл, с которого график был снят. Точнее, можно восстановить его с приемлемой для нас точностью, о чем мы сейчас и поговорим.

Давайте теперь разберемся, как в этих звуковых файлах кодируются числа. Возьмем пример с числом 666`666`666:



Здесь мы визуально разрезали файл сигнала на 32 части — и сразу все стало ясно. Единичкам соответствуют верхние уровни, нулям — нижние. В итоге получается 32-битная последовательность 0101`0101`0110`0001`0011`1101`1110`0100, и легко убедиться, что это ни что иное, как двоичное представление числа 666`666`666, правда записанная наоборот, начиная с **младшего** бита:

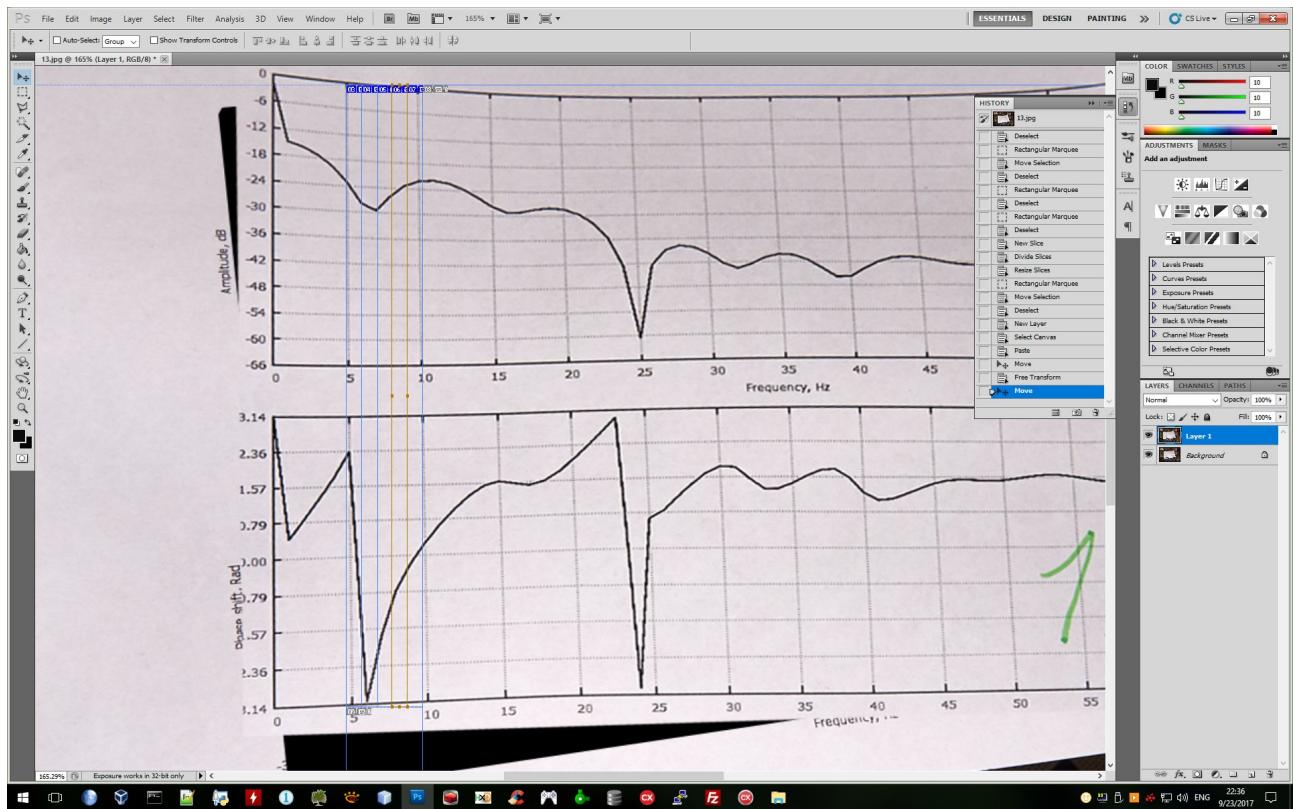
```
>>> int("01010101011000010011110111100100"[::-1], 2)  
66666666
```

То есть, чтобы получить нормальное двоичное представление, строку надо развернуть, что мы тут и делаем.

* * *

Собственно, обладая этой информацией, таск уже можно решить. Ключевое слово **можно**. Для этого всего-лишь нужно с графика, содержащего флаг перерисовать все 70 (или хотя-бы 60) точек, т.е амплитуду и фазу для каждой гармоники, сделать обратное БПФ, и таким же образом как описано выше, посмотреть на полученный сигнал...

На словах оно конечно легко — но ради эксперимента я честно попробовал проделать эту манипуляцию с графиком числа 13, так, как предполагалось, должны делать участники. Вот как **ужасно** это выглядело:



*Пытаемся отчаянно исправить перспективу

The screenshot shows a dual-pane interface. On the left, a Notepad++ window displays a C-style code snippet with various mathematical expressions involving complex exponentials and constants. On the right, a terminal window titled 'Python' shows a corresponding Python session where the same expressions are evaluated, demonstrating the numerical values produced by the software.

```

*new 6 - Notepad++ [Administrator]
File Edit Search View Encoding Language Python
sproxy_qui.h sqlite_tables.sql alles.txt
1 {
2 exp(θ + 3.15i),
3 exp(-3.4539 + 0.45i),
4 exp(-3.68416 + 0.79i),
5 exp(-4.1446 + 1.35i),
6 exp(-4.83546 + 1.85i),
7
8 exp(-5.52624 + 2.36i),
9 exp(-6.4472 + -1.57i),
10 exp(-6.83872 + -2.044i),
11 exp(-5.9176 + -1.044i),
12 exp(-5.2959 + -0.3i),
13
14 exp(-5.0657 + 0.3i),
15 exp(-5.01966 + 0.7i),
16 exp(-5.342 + 1.08i),
17 exp(-5.52624 + 1.41i),
18 exp(-5.98676 + 1.56i),
19
20 exp(-5.98676 + 1.56i),
21 }
22
23
24
25
26 2.3026

Python
>>> 10**(-1.5)
0.03162277660168379
>>> 10**(-1)
0.1
>>> 10**(-1.5)
0.03162277660168379
>>> -15/10*2.3026
-3.4539
>>> -16/10*2.3026
-3.6841600000000003
>>> (0.80+1.57)/2
1.185
>>> -18/10*2.3026
-4.14468
>>> -21/10*2.3026
-4.83546
>>> 1.57*0.3 + 0.7*2.36
2.1229999999999998
>>> 1.57*0.7 + 0.3*2.36
1.807
>>> -24/10*2.3026
-5.52624
>>> -28/10*2.3026
-6.447279999999999
>>> -29.7/10*2.3026
-6.838722
>>> -29.7/10*2.3026
KeyboardInterrupt
>>> 2.36*0.6+1.57*0.4
2.044
>>> -25.7/10*2.3026
-5.917681999999999
>>> -23/10*2.3026
-5.295979999999999
>>> -22/10*2.3026
-5.065720000000001
>>> -21.8/10*2.3026
-5.019668
>>> -23.2/10*2.3026
-5.342032
>>> -24/10*2.3026
-5.52624
>>> -26/10*2.3026
-5.98676
>>> -

```

Примерно на этом этапе меня очень расстроил тот факт, что Wolfram Alpha хоть и имеет команду `Fourier({a + bi, c + di, ...})`, но инпут у него ограничен неприлично маленьким количеством символов, и в него получилось засунуть всего-лишь первые 10 гармоник. Этого откровенно было мало... Пришлось не выpendриваться колхозить:

```

6 CComplex short[128]{}
7 {-0.999965, 0.00840725},
8 {0.0284996, 0.0137669},
9 {0.0176822, 0.0178457},
0 {0.00347329, 0.0154743},
1 {-0.00219003, 0.00763895},
2 {-0.00282631, 0.00280488},
3 {-0.00149368, 0.000531047},
4 {-0.000488666, 0.00095442},
5 {0.00135409, 0.00232812},
6 {0.00479255, 0.00148251},
7 {0.00603191, 0.00186589},
8 {0.00505647, 0.00425901},
9 {0.00225591, 0.0042213},
0 {0.000637517, 0.00393052},
1 {0.0000271382, 0.00251355},
2 {1.35257e-6, 0.00169852},
3 {0.000012121, 0.00112266},
4 {0.0000255966, 0.00123065},
5 {1.05023e-6, 0.00131884},
6 {-0.00025972, 0.00135628},
7 {-0.000548833, 0.00119922},
8 {-0.000710329, 0.000704943},
9 {-0.00045335, 0.000214311},
0 {-0.000248751, 0.0000354586},
1 {-0.0000197955, 2.82177e-6},
2 {7.69069e-7, 6.47777e-7},
3 {0.0000444477, 0.0000448587},
4 {0.0000574771, 0.00014784},
5 {4.88414e-6, 0.000158545},
6 {-9.01358e-6, 0.000113564},

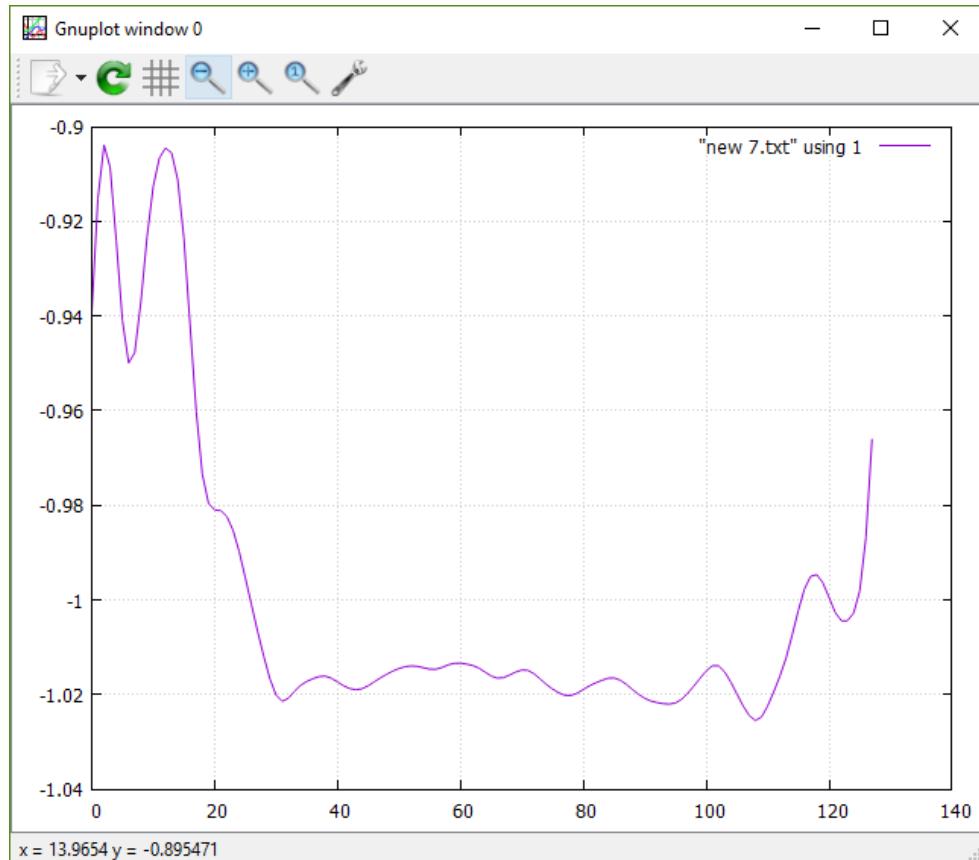
```

```

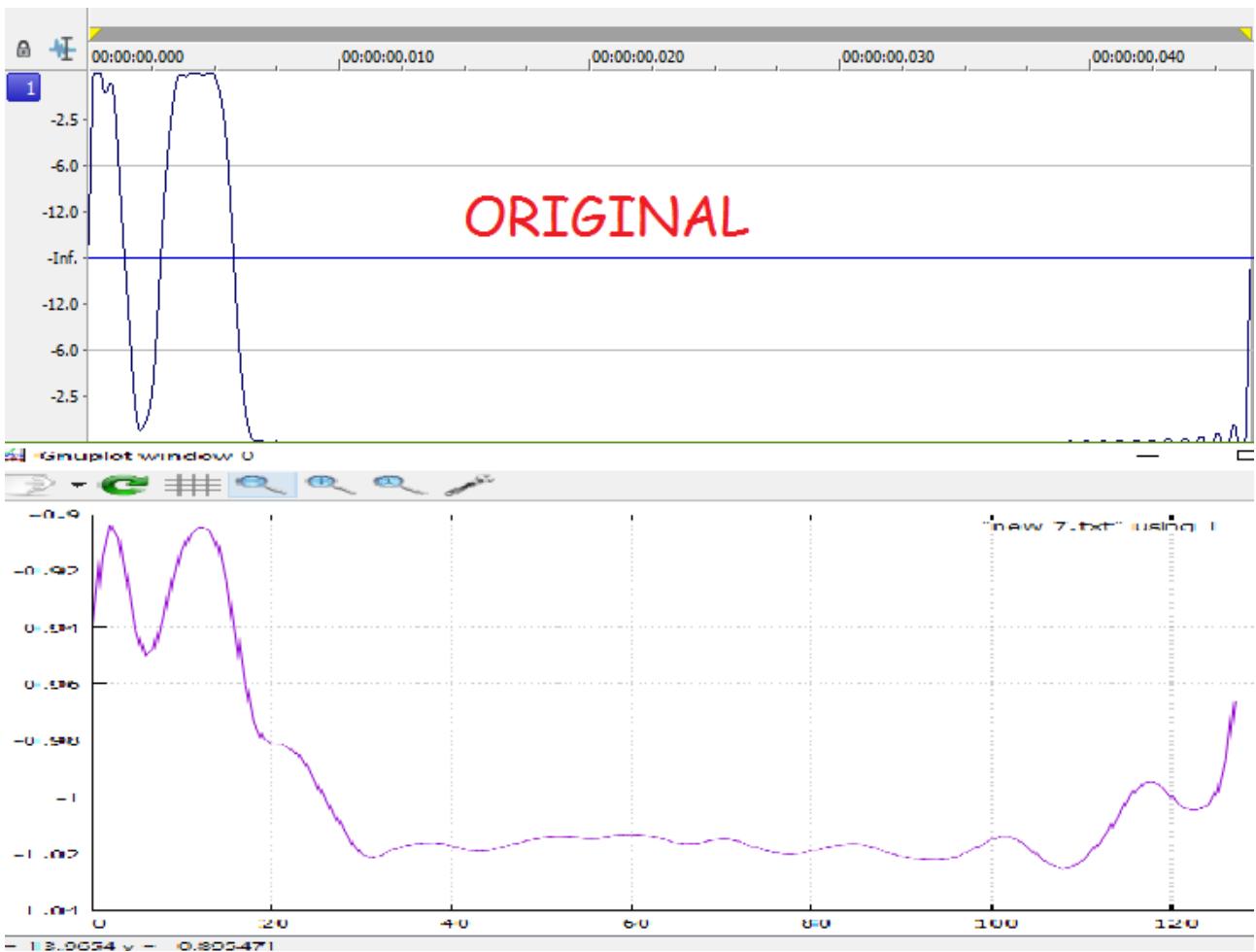
142 int main(int argc, char** argv){
143
144
145 CFFT::FFT(short*, short*, 7, true);
146
147 for(int i = 0; i < 128; ++i){
148     std::cout << res[i].re << ' ' << res[i].im << std::endl;
149 }
150

```

Результатом работы сего колхоза стал файл из 128 отсчетов, который выглядел... Не так плохо, как теоретически мог бы:



В сравнении с исходным файлом сигнала:



А ведь и действительно совсем не так плохо, как могло быть! И это, заметьте, с учетом двух вещей:

1. Моей усидчивости хватило только на «оцифровку» 30 точек из всех 70, то есть, сигнал был восстановлен по 30 гармоникам вместо 70, остальные были нулями
2. Я забыл, что **прежде чем делать обратное БПФ, данные нужно сначала расположить зеркально**. Ненавижу. И тем не менее, форма исходного сигнала всё-равно хорошо прослеживается.

Собственно этого опыта было достаточно, чтобы таск считать решаемым. И понятно, что если приложить чуть больше усидчивости к решению, то извлечь требуемый код из графика с флагом будет вполне реально. Нужно даже заметить что не нужно сильно беспокоиться о точности — в своих экспериментах я намеренно вводил погрешность 10% в аргументах экспоненты — и график всё равно сходился с требуемым. То есть, сигнал, который вы получите

из графика будет толерантен даже к 10%-ным ошибкам в точности при переносе точек с графика в числа.

К слову, для генерации этих сигналов я сделал программу на *C++*, и скрипт для *gnuplot*, генерирующий из вывода программы те самые графики с фото — всё это можно найти в папке *src*.

Unsafe Token

Паранойя зачастую толкает нас на поступки, которые со стороны кажутся абсурдом. Согласитесь ведь, что иметь один пароль на всех сайтах — просто и удобно, двухфакторная авторизация съедает драгоценные минуты времени, а full-disk encryption вообще убивает производительность всей системы. Да ещё и создаёт опасность потери ключа...

Одним из интересных проявлений паранойи у Доктора F. было стремление не хранить секретных ключей на основном рабочем компьютере. И в один прекрасный день, разнообразия ради, Доктор F. решил попробовать хранить один из ключей на своём Android телефоне. Для чего, как обычно, поручил Доктору D. сделать для этого приложение. Ну, а Доктор D., как обычно сделал всё *криво* небезопасно.

Приложение предполагалось использовать для генерации цифровых подписей произвольных коротких текстовых сообщений, на основе встроенного секретного ключа. Нам даже удалось заполучить это приложение — и мы вам его с удовольствием передаём.

Но передаём не просто так.

Дело в том, что нам попалась одна подпись, сгенерированная этим приложением. Файл с её содержимым мы вам так-же передаём. Да, подписи эта штука генерирует не маленькие... Так вот. Может это покажется вам безумным, но нам нужно, чтобы вы **восстановили исходный текст сообщения**, которое было подписано.

Да, вы всё правильно прочитали.

Восстановленный текст в точности будет являться вашим флагом. Желаем удачи...

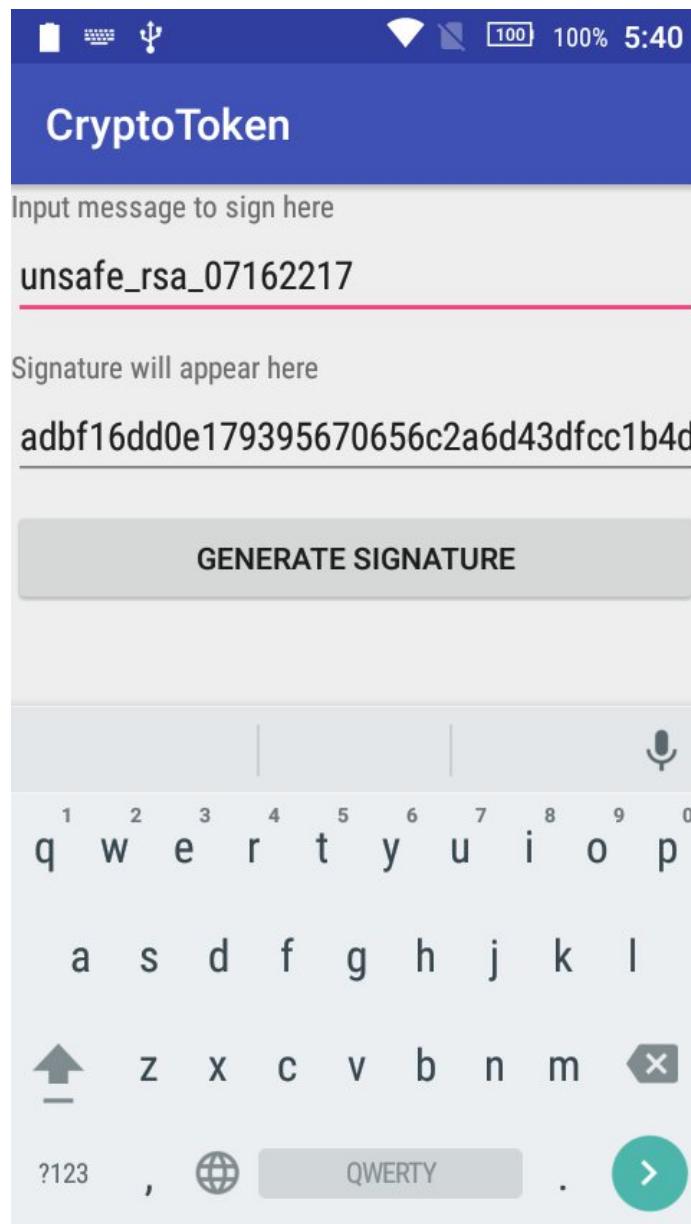
Вот, кстати, копия подписи, на всякий случай:

```
adbff16dd0e179395670656c2a6d43dfcc1b4d967fd007d8f44835d4a7fc7cb  
4e453bd7ca4b71ca433a5e41e63569d56e04e029bbe9a77af997cb50f0d57  
1909eca1755e4531c31adfe0454f2ef5de6a90098c9fda2bff827fdb5587f  
a71f3097391d86a2fbc896ed209f317639ce5eca83245f3dffcc24bd9efe177  
9921f61c0dbb454b2404af19bc136b3911d7e55b3396a4eadfd7e82fd6c25c  
c7ad923d98c4dffce6f205c888a852ab0dc8eda547e118c667d0658a6e98d  
65aa32d747a25311412d2b14c92765d19126a4a3a43fb52bcda2d92fee2ac6  
152f3b2a3eb7dc49da4e0b068943886115e74711cfea896ff45ae79564682e  
c613f2ebcdc988e4
```

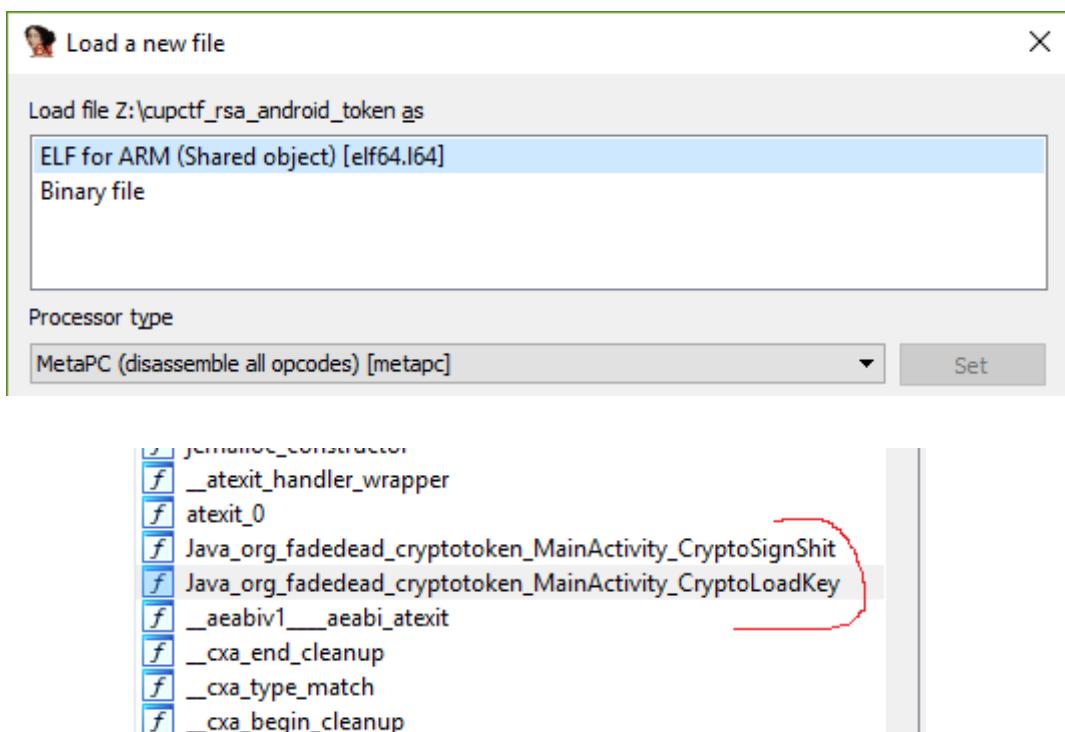


Флаг: **unsafe_rsa_07162217**

Да, действительно, использовать RSA для подписи чего-либо без вспомогательной хеш-функции, несколько, скажем так, криво небезопасно. В контексте того, как это делается приложением из нашего таска — происходит самое простое шифрование сообщения приватным ключом, нам остаётся его только расшифровать, и всё. Но постойте, по порядку.



Нам даётся приложение, содержащее в себе нативную библиотеку для одной единственной архитектуры — arm7. Всё тут потому, что собрать по человечески под все архитектуры, да так, чтобы нормально линковалось с openssl у меня не хватило нервов. А всё потому, что Доктор F. излишне консервативен, но не важно! Важно, что эта библиотека предоставляет в JNI интерфейс, с помощью которого, судя по названиям функций, можно грузить приватный RSA ключ, и что-то им шифровать подписывать.



```

xt:00042C7C
xt:00042C7C ; jboolean __fastcall Java_org_fadedead_cryptotoken_MainActivity_CryptoLoadKey()
xt:00042C7C             EXPORT Java_org_fadedead_cryptotoken_MainActivity_CryptoLoadKey
xt:00042C7C Java_org_fadedead_cryptotoken_MainActivity_CryptoLoadKey
xt:00042C7C
xt:00042C7C prsa          = -0x24
xt:00042C7C var_20         = -0x20
xt:00042C7C
xt:00042C7C env = R0           ; JNIEnv *
xt:00042C7C instance = R1        ; jobject
xt:00042C7C ke_ = R2           ; jstring
• xt:00042C7C LDR      R3, [env]
• xt:00042C80 STMFD   SP!, {env-ke_,R4-R9,LR}
• xt:00042C84 MOU     instance, ke_
• xt:00042C88 LDR      R3, [R3,#0x2A4]
• xt:00042C8C MOU     R7, ke_
• xt:00042C90 MOU     ke_, #0
xt:00042C94 ke_ = R1           ; jstring
• xt:00042C94 MOU     R5, env
• xt:00042C98 BLX      R3
• xt:00042C9C ADD      R4, SP, #0x28+var_20
• xt:00042CA0 MOU     R8, #0
• xt:00042CA4 STR      R8, [R4,-4]!
• xt:00042CA8 MOU     R6, R0
• xt:00042CAC BLX      strlen
• xt:00042CB0 MOU     R1, R0
• xt:00042CB4 MOU     R0, R6
• xt:00042CB8 BL      BIO_new_mem_buf
• xt:00042CBC MOU     R1, R4
• xt:00042CC0 LDR      R4, =(main_rsa_ptr - 0x42CE4)
• xt:00042CC4 MOU     R2, R8
• xt:00042CC8 MOU     R3, R8
• xt:00042CCC MOU     R9, R0
xt:00042CD0 buf = R0           ; BIO *
• xt:00042CD0 BL      PEM_read_bio_RSAPrivateKey
• xt:00042CD4 MOU     R0, buf
• xt:00042CD8 BL      BIO_free_all
• xt:00042CDC LDR      R4, [PC,R4] ; main_rsa
• xt:00042CE0 LDR      R3, [SP,#0x28+prsa]
• xt:00042CF4 MOU     R0, env

```

Думаю, не будет столь сложным догадаться, когда и откуда эти функции вызываются. Вы можете декомпилировать приложение, и убедиться в этом. А я просто покажу кусок исходников:

```

public native boolean CryptoLoadKey(String ke);
public native String CryptoSignShit(String ke);

// Used to load the 'native-lib' library on application startup.
static {
    System.loadLibrary("fadetoken");
}

@Override
public void onClick(View v) {
    if(v.getId() == R.id.buttonSign){
        EditText t = (EditText) findViewById(R.id.editTextMessage);

        String sig = CryptoSignShit(t.getText().toString());

        if(sig.length() > 0){
            EditText ts = (EditText) findViewById(R.id.editTextSignature);
            ts.setText(sig);

            Toast.makeText(getApplicationContext(), "Signature created", Toast.LENGTH_SHORT).show();

            Log.d(LOG_TAG, "Created signature: " + sig);
        }else{
            Toast.makeText(getApplicationContext(), "Input text too long", Toast.LENGTH_SHORT).show();

            Log.w(LOG_TAG, "Unable to create the signature, probably text too long");
        }
    }
}

```

Константа, содержащая приватный RSA ключ находится в классе `MainActivity`, и я думаю не будет так сложно её оттуда вытащить, например, декомпилировав приложение. Мне вот, хватило «декомпиляции» APK с помощью 7-Zip и «дизассемблирования» `classes.dex` с помощью HEX-редактора:

```

..., or qualche tra ..., position ..., offset ..., size ..., size..., type
7665 =..., updated=..., windowActionBarOverlay: ..., windowActionBarModeOve
2D0A rlay: ..., windowNoTitle: ... -----BEGIN PRIVATE KEY-----.
786F MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDiv6qPmNshPuxo
3852 .iEwrBCRsOnO1HDEBdgKo12fWs7mjcgwHiRUz64G0tQu2W1vBSYzf8XTblkh5a8R
4959 K.SpkBUMyftwjrnCQUsVFK9EJgJczTB296Ad2T2+U4BUzeV2kxPzRq1tzEn7u8IY
632F Cs.56rPVKOH/2M65IyCPWF+xBZ/fcTaocdU2czrmyxIIKOEqQzQuzRDTjvoFSLc/
7242 Mg4.XVUt0Lb2HL2ADBSTnBYsYT5AeWXIhKvV1K37GUDpa2OVJk63zUjS/Y6KbVrB
7670 AW1i.y3VIE4iu07Yi1iCqU7EYcTwACFVP1xsQWmt/T70UfNzuhXR1i73iGtlglvp
2F42 1+Hkf.rUk6PNZTAgMBAAECggEBANGqPOH0qUMCh8IH3gVMR6U/1UVcoONxq4sp/B
3843 ZU9G77.NLi1WRJMPDUYO/g3FDwqcCC2Qd9tX95RfKoOf9sf01Qw3FIxYrxPzt58C
5A77 iM7Jg8w.kXfQ+3mhni2R1zyIPIJ0bPZFq+XPGdx6W/96Q/6wA86NC/6DD+jBt4Zw
6D42 TbxpxHhAG.SvX95DLTWQzMGRa97Ay9Iiiy7iWBCVce/97ZwCQI5CW3h86SPB+TmmB
5A51 Ai7gK5cWd.cmfl4PGjcxQb4KFm/FkUdovoBeig6qBpIKpAgHCp21PGBrOxibTwZQ
5A65 K/U9nPq5pM.+6VqIOpKnS2Pm8EhShIFjaz4WhBfZM3sgbFV7LyzGVECgYEA/BXZe

```

Ну, то-есть, если глянуть на скриншот IDA выше, и увидеть там `PEM_read`, то можно сразу понять, что функция принимает приватный ключ в PEM формате. Который начинается на `BEGIN PRIVATE KEY`,

как мы и видим выше. Суть приватного ключа в PEM в том, что он содержит в себе и публичный, а так-же всё необходимое для генерации ключевых пар (поправьте меня, если ошибаюсь).

Таким образом, имея на руках подпись, и приватный RSA ключ, нам просто надо выполнить операцию, обратную шифрованию приватным ключом. И это будет расшифровка публичным ключом, то-есть RSA_public_decrypt. Таким образом, решение таска сводится к написанию чего-то подобного:

```
142 FDO::Util::Buffer bubko = FDO::Util::HexToBin("adbf16dd0e179395670656c2a6d43dfcc1b4
143
144
145     std::string keko =
146         std::string("-----BEGIN PRIVATE KEY-----\n") +
147             "MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwgSjAgEAAoIBAQDl6qPmNshPuxo\n" +
148             "iEwrBCRs0N0lHDEBdgK0l2fWls7mjcgwHiRUz64G0tQu2W1vBSYzf8XTblkh5a8RK\n" +
149             "SpkBUMyftwjrnCQUsVFK9EJgJczTB296Ad2Tz+U4BUzeV2kxPzRq1tzEn7u8IYC\n" +
150             "56rPVKOH/2M65IyCPWF+xBZ/fcTaocdIUZczrmyxIIKOEqQzQuzRDTjvoFSLc/Mg4\n" +
151             "XVUt0Lb2HL2ADBSBnBYsYT5AeWXIhKvVlK37GUDpa20VJk63zUjs/Y6KbVrBAWli\n" +
152             "y3VIE4iu07Yi1iCqU7EYcTwACFVP1xsQWmt/T70UfnzuhXR1i73iGt1glvpl+Hkf\n" +
153             "rUk6PNZTAgMBAAECCggEBAnqqPOH0qUMCh8IH3gVMR6U/lUvcoONxq4sp/BZU9G77\n" +
154             "NLiiWRJMPDUYO/g3FDwqcCC2Qd9tX95RfKo0f9sf01Qw3FIxYrxPZt58CiM7Jg8w\n" +
155             "kXFQ+3mhnIZRlzyIPIJ0bPZFq+XPGdx6W/96Q/6wA86NC/6DD+jBt4ZwTbpxHhAG\n" +
156             "SvX95DLTWQzMGRa97Ay9Iiyy7iWBCVce/97ZwCQI5CW3h86SPB+TmmBAi7gK5cld\n" +
157             "cmfL4PGjcxQb4KFm/FkUdovoBeig6qBpIKpAgHCp2lPGBrOxibTwZQK/U9nPq5pM\n" +
158             "+6VqI0pKnSZPm8EhShIFjaz4WhBfZM3sgbFV7LyzGVECgYEABXZek4XK+T7k9jK\n" +
159             "4czzIG/P06yRyf4na4PVrUoMSjrvAJpFLHXiC5EozWdueNmzU+GN/mW+EtTd1TL\n" +
160             "TQUcvvVF9NEerX+jYXeeEcKhWqMJ/w+8YpkeItjwTf/zow4SLKdDTXgD+n5k/bk\n" +
161             "FsdcIxixxF1cpvESunZ1pWKrsZcCgYEAsKuXpKa+s7UIE41jDZN212Q0JnExYvt3\n" +
162             "CPImF3bFZjBsmDeD00kGy3+Emqw09U4tYlVSAiPxN0khTo92WIXFjnPEAGTG2wD\n" +
163             "tfCIL2vvI7xwnftk3i6R8mobuEmaxlt8r6jBsaCKpejo4HJcYGEdVuRX3rGmZT0L\n" +
164             "yoUHwcqzoKUCgYBzGfbnJWjnHfLRGxfn6hD4pzUzz3zwRx2tweFzF24Nxiscd6z\n" +
165             "4k+IQiuE3kl02x8Cd+EtBcwTmZe2XB1h3aGHAw8iLan4RmNvSzLfqSMzo2YZFFCw\n" +
166             "MSSW9e5idxNSVTRD3Wpgcc0Q9yCCMyUKUZT8YyWly4tWnpSSlyw4i8khmQKBgCSq\n" +
167             "8+zgMjX1sffzF8bdqJr779TnSXn9mSlvjG1IHH79Tw2eKuMyEZnc/I9W9RsMoXq\n" +
168             "GDkHCC3owckEx0+pXaAliVnLJmXDeYPLXaRPh7ElHlnLgnBu8gasJuXdsqvLfbtr\n" +
169             "IAztB6cLw6maDG08kzN3ARcRKvIK10QukvulD52RAoGAN20Rz43tz+nU6fi9M0s7\n" +
170             "00eyVKnMBbKS1C5VpTvtqrjhJEVvbCJ+Szz0xkPh0dOTCxEvIbQrcz0vBB50fBo5\n" +
171             "lBV0VG6dhBH4EzeIoVMZ5g5pwR7wKKdJc7ndx2S+J/i0kvHKyEVW1F8TbiNF7uXz\n" +
172             "k1zcvnKs+kDE6gg/mpF/Arc=\n" +
173             "-----END PRIVATE KEY-----\n";
174
175     {
176         RSA *prsa(nullptr);
177         BIO* buf(BIO_new_mem_buf(
178             const_cast<char*>(keko.c_str()),
179             keko.size()
180         ));
181         PEM_read_bio_RSAPrivateKey(buf, &prsa, nullptr, nullptr);
182         BIO_free_all(buf);
183
184         FDO::Util::Buffer bibka(bubko);
185
186         RSA_public_decrypt(256, bubko.data(), bibka.data(), prsa, RSA_NO_PADDING);
187
188         FDO::Log::Warn("TEST", FDO::Util::BinToHex(bibka));
189     }
```

И если посмотреть на то, что это чудо выведет, то мы увидим:

```
6decebeae9e8e7e6e5e4e3e2e1e0dfdedddcdbdad9d8d7d6d5d4d3d2d1d0cfcecdcccbcac9c8c7c6c5c4c3c  
2c1c0bfbebdbbbb9b8b7b6b5b4b3b2b1b0afaeadacabaaa9a8a7a6a5a4a3a2a1a09f9e9d9c9b9a999897  
969594939291908f8e8d8c8b8a898887868584838281807f7e7d7c7b7a797877767574737271706f6e6d6c6  
b6a696867666564636261605f5e5d5c5b5a595857565554535251504f4e4d4c4b4a49484746454443424140  
3f3e3d3c3b3a393837363534333231302f2e2d2c2b2a292827262524232221201f1e1d1c1b1a19181716151  
4131211100f0e0d0c0b0a090807060504030201756e736166655f7273615f3037313632323137
```

Что в переводе на человеческий будет означать:

```
тмлкijизжедгвбаяюэыгышшцхфутсрпонмлкijизжедгвбаиссj»е¶ё·¶игил°ї®  
.¬«©ës;Г¤Jўў үhкњъљ.---·""/ 'ї[нкъ  
млкjihgfedcba`_^]\[ZYXWVUTSRQPOMLKJIHGFE DCBA@?>=<;:9876543210/.  
-,+*) ('&%$#! .....unsafe_rsa_07162217
```

Думаю, не надо объяснять, где тут флаг. Если не очевидно, можно поковырять ассемблерный код функции

[Java_org_fadedead_cryptotoken_MainActivity_CryptoSignShit](#), и обнаружить, что данное она просто засовывает в конец массива, а остальное забивает однообразным паттерном из убывающих чиселок.

```

.text:00042B04      MOU      R5, env
.text:00042B08      LDR      R10, =(main_rsa_ptr - 0x42B20)
.text:00042B0C      LDR      R3, [R3,#0x2A4]
.text:00042B10      BLX
.text:00042B14      LDR      R7, =(_ZNSs4_Rep20_S_empty_rep_storageE_ptr - 0x42B3C)
.text:00042B18      LDR      R10, [PC,R10] ; main_rsa
.text:00042B1C      MOU      R6, R0
.text:00042B20      LDR      R8, [R10]
.text:00042B24      BL      RSA_size
.text:00042B28      MOU      R4, R0
.text:00042B2C rsa_size = R0      ; const int
.text:00042B30 rsa_size = R4      ; const int
.text:00042B30      BLX
.text:00042B34      LDR      R7, [PC,R7] ; _ZNSs4_Rep20_S_empty_rep_storageE_ptr ; st
.text:00042B38      ADD      R8, R7, #0xC
.text:00042B3C      STR      R8, [R11,#reslut]
.text:00042B40      CMP      R8, rsa_size
.text:00042B44      MOU      R9, R0
.text:00042B48 slen = R0      ; int
.text:00042B48      BGE
.loc_42C18
.text:00042B4C      ADD      R3, rsa_size, #7
.text:00042B50      STR      SP, [R11,#var_3C]
.text:00042B54      BIC      R3, R3, #7
.text:00042B58      MOU      R1, #0
.text:00042B5C      SUB      SP, SP, R3
.text:00042B60      MOU      R2, rsa_size
.text:00042B64      ADD      R7, SP, #0x4C+var_44
.text:00042B68      RSB      R8, R9, rsa_size
.text:00042B6C      SUB      SP, SP, R3
.text:00042B70      MOU      slen, R7
.text:00042B74 slen = R9      ; int
.text:00042B74      ADD      R3, SP, #0x4C+var_44
.text:00042B78      STR      R3, [R11,#var_38]
.text:00042B7C      BL      memset
.text:00042B80      ADD      R8, R7, R8
.text:00042B84      MOU      R1, R6
.text:00042B88      MOU      R2, slen
.text:00042B8C      BL      memcpy
.text:00042B90      MOU      R3, #0
.text:00042B94

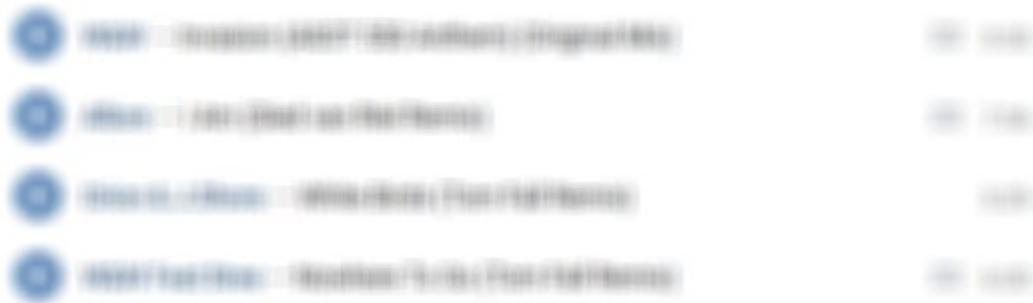
```

К слову, сначала я хотел сделать, чтобы от данных бралось md5, что привело бы к тому, что для успешного решения потребовалось бы ещё расшифровать сбрутить загуглить md5 хеш. Но передумал — и так не самый простой таск, если ты не его автор.

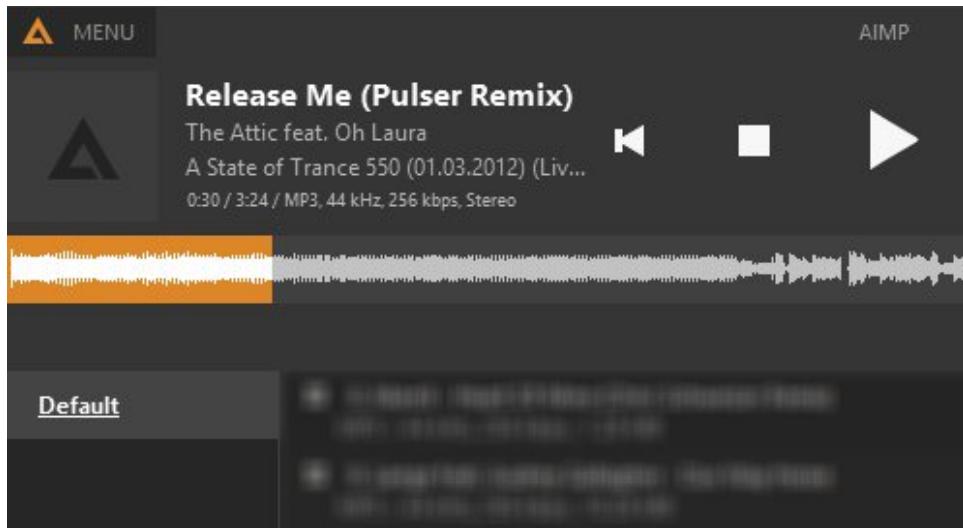
P.S. Оказывается, *Android Studio* что-то нехорошее делает с библиотеками, в результате чего они теряют в весе, и количестве отладочной информации. Поэтому у участника не получится воспроизвести достоверно скриншоты выше. Однако, всё не сильно плохо, и при желании, восстановить алгоритм вполне можно, сохраняются даже имена функций *openssl* и *stl*, а что еще надо-то для счастья?

ASoT 550

Про Доктора F. можно с уверенностью сказать одну вещь — он всегда был просто без ума от A State of Trance! Настолько, что в своём далёком 2217 году несколько раз целиком переслушал ASoT 550. Кажется, он назывался **Invasion**.



Доктор F. слушал ASoT практически постоянно. Он утверждал, что это помогает ему сосредоточиться, сконцентрироваться на работе, как будто на время эта музыка обрывала контакт с внешним миром... Возникает, правда, вопрос, как могут не надоест 2000+ часов реально **однообразной** музыки? Именно поэтому, коллеги его не понимали, частенько приставая с вопросами наподобие «как ты это слушаешь?!», однако Доктор F. на это обычно никак не реагировал. Стоит заметить, что некоторые треки ему нравились больше остальных, он их даже добавлял в закладки. А один из таких треков даже попал в наши руки.



На первый взгляд, ничего необычного. Трек с названием **Release Me (Pulser Remix)** действительно встречается в ASoT 550 дважды, в сетах **Solarstone** и **tyDi**, что, в общем-то, легко проверяется. Подвоха, казалось бы, никакого нет...

Однако, странности появились, когда в наших архивах всплыл некий документ, содержащий фрагмент переписки Доктора F. со своим ассистентом, в котором F. рассказывал про некий файл, который он «хитро замаскировал» под трек из ASoT, пытаясь незаметно передать таким образом нужным лицам какой-то не то сигнал, не то послание. Далее в переписке было скрупулёзное описание того, как именно он это сделал, однако оно сохранилось не полностью, а из того что было, мы попросту ничего не поняли. Там говорилось что-то про частоту Найквиста, пороги слышимости, теорему Пасервала, и было много математических формул, вроде такой:

$$x(t) = \sum_{k=-\infty}^{\infty} x(k\Delta) \prod_{n=1}^M \text{sinc} \left[\frac{\pi}{a^{n-1}\Delta} (t - k\Delta) \right]$$

От Доктора F. ничего другого, впрочем, ждать не приходилось, однако интригу подогревал простой факт, что треков, которые нравились Доктору F. было много, а в наши руки попался почему-то только один. Всего один, из нескольких сотен, а может быть тысяч!

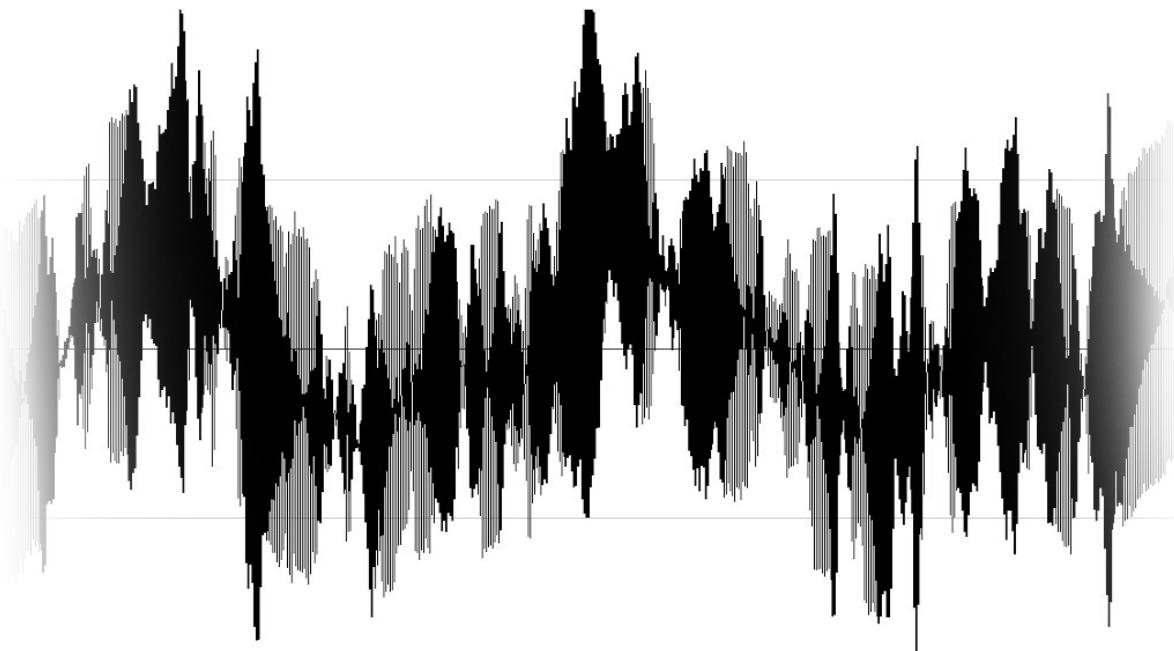
Этот факт, впрочем, не единственное, что натолкнуло нас на мысль, что нам достался тот самый файл, содержащий «послание» ~~внеземной цивилизации~~, достаточно просто, гм... внимательно взглянуть на файл. Например, можно заметить, что он как-то странно обрезан, и его продолжительность с продолжительностью оригинального трека не совпадает...

Это конечно всё домыслы. Но, мы решили дать вам возможность их в полной мере проверить. На основание кое-каких размышлений, мы можем предположить, что упомянутое «послание» в этом файле должно иметь звуковой формат, а если конкретнее, файл помимо трека из ASoT должен содержать в себе **какую-то другую** аудиозапись. Подумайте, как такое возможно, и попробуйте её найти.

* * *

Немного о формате флага. Упомянутая скрытая аудиозапись посвящена некому персонажу. Для того, чтобы получить флаг, необходимо взять имя этого персонажа в кодировке utf-8 в lowercase, и посчитать от него md5-хеш. Проще всего это будет сделать, например, сервисом md5.cz. Хеш и будет вашим флагом. Удачи!

Dr. Fade



Флаг: **c1a50a53284b606f9a3a7ba4a119fc44**

Сразу отмечу, что идея таска была слизана с соответствующего таска NSK CTF (моего-же производства) чуть более, чем полностью, чуть чуть детали только были изменены. Правда, тот таск всё-равно никто не решил... Идея была просто очень уж годная, на мой взгляд.

Итак, поехали. Что вообще дано? Некий файлег, содержащий ровно то, что ожидается — весьма странно обрезанный кусок одного трека с ASoT, хотя, простому слушателю факт того, что трек порезан очевидно не будет. Давайте посмотрим на свойства файла:

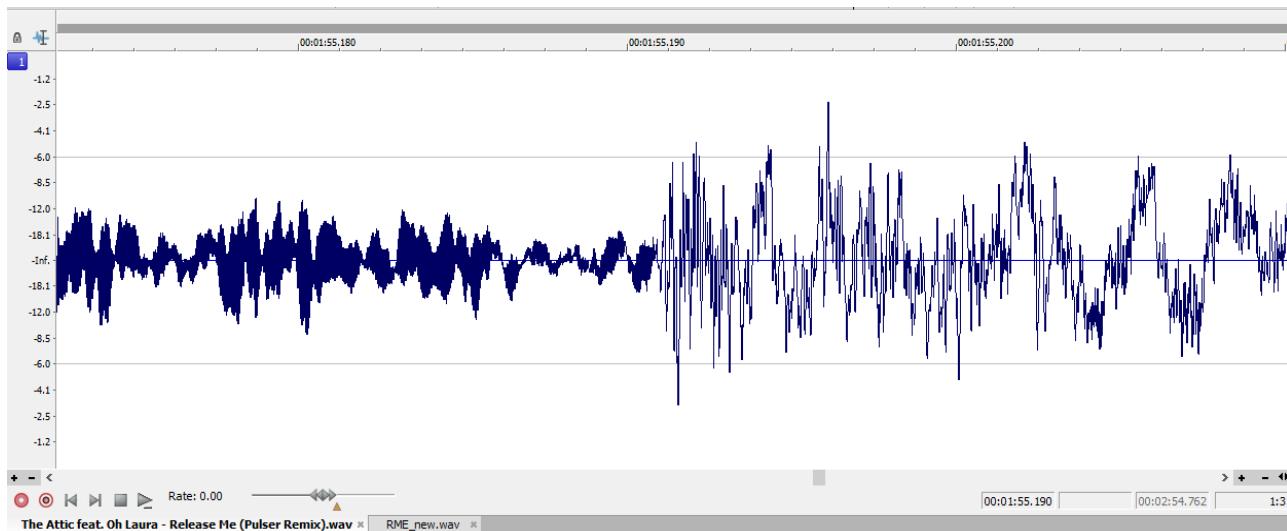
File Properties		
	Attribute	Value
1	File name	The Attic feat. Oh Laura - Release Me (Pulser Remix).wav
2	Location	D:\all\works\cupctf\tasks\asot_550\user\
3	File size	32.77 MB (33,554,476 bytes)
4	File attributes	- --a- -----
5	Last saved	2017-09-28 06:20:58
6	File type	Wave (Microsoft)
7	Audio format	Uncompressed
8	Audio sample rate	96,000 ←
9	Audio bit rate	1,536 Kbps
10	Audio bit depth	16 bit
11	Audio channels	1 (Mono) ←
12	Audio length	00:02:54.762 (16,777,216 samples) ←
13	Video format	No Video
14	Video attributes	No Video
15	Video length	No Video
16	Video field order	No Video
17	Video pixel aspect ratio	No Video

К странностям добавляется 3 факта:

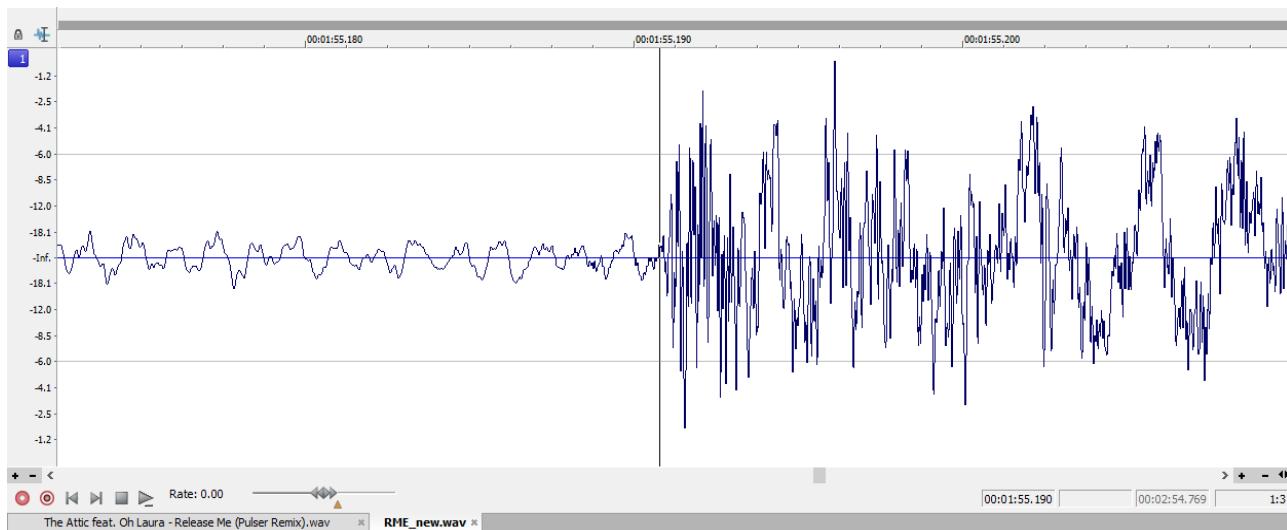
1. Файл имеет избыточную частоту дискретизации 96 кГц;
2. В файле всего 1 канал, что кстати, не лучшим образом сказывается на качестве;
3. В файле **ровно 2^{24} сэмплов!!!** Это слишком точное попадание, чтобы быть просто совпадением...

```
>>> 1 << 24  
16777216
```

Правда пока особо это ничего не даёт. Давайте попробуем внимательно посмотреть на содержимое:



Если бегло посмотреть на форму семплов, то становится заметно, что они выглядят *странны*. Иное слово тут трудно подобрать... Как будто при увеличении появляется какая-то противная паутина, шум, которых по ощущениям быть не должно. На скриншоте выше это проявляется в виде «ожирения» графика в левой части. Чтобы эту странность объяснить, давайте возьмем трек *Release Me* из другого источника, заресемплим его до 96 кГц, и посмотрим на тот-же самый участок. Чтобы было нагляднее, мы ещё и точно так-же его обрежем:

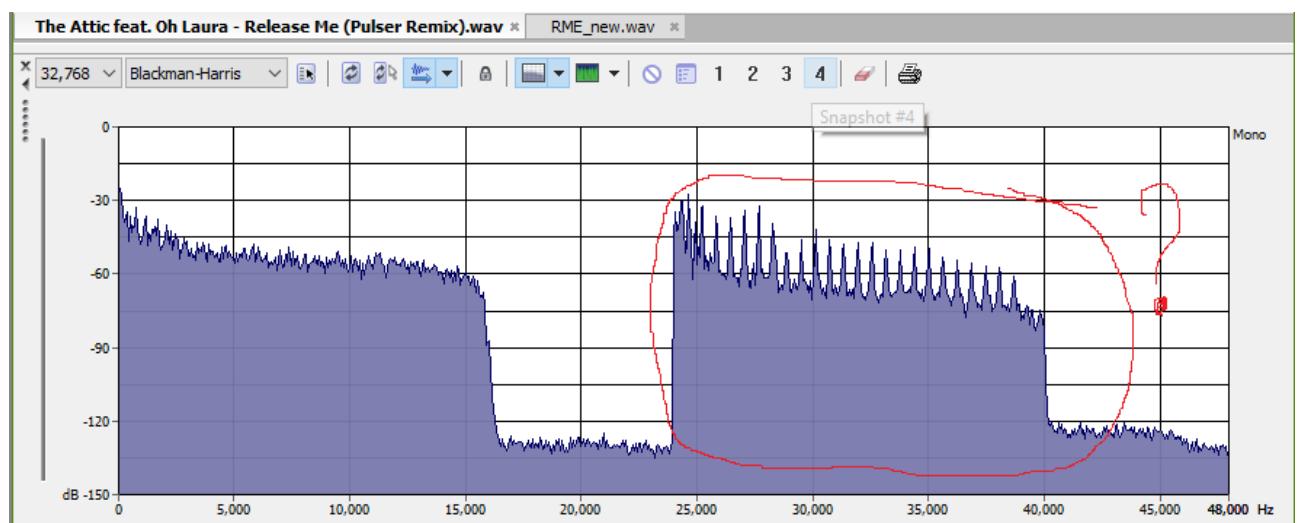


Обратите внимание, фрагмент совпадает до микросекунд. Не правда ли разница налицо? Хотя, это ещё не так наглядно.... Давайте немножко увеличим:



Ну, вот тут более выражено то, что надо увидеть. В нашем треке присутствуют высокочастотные колебания, которые в треке со стороны отсутствуют начисто. Он вообще гладенький, как... как и положено.

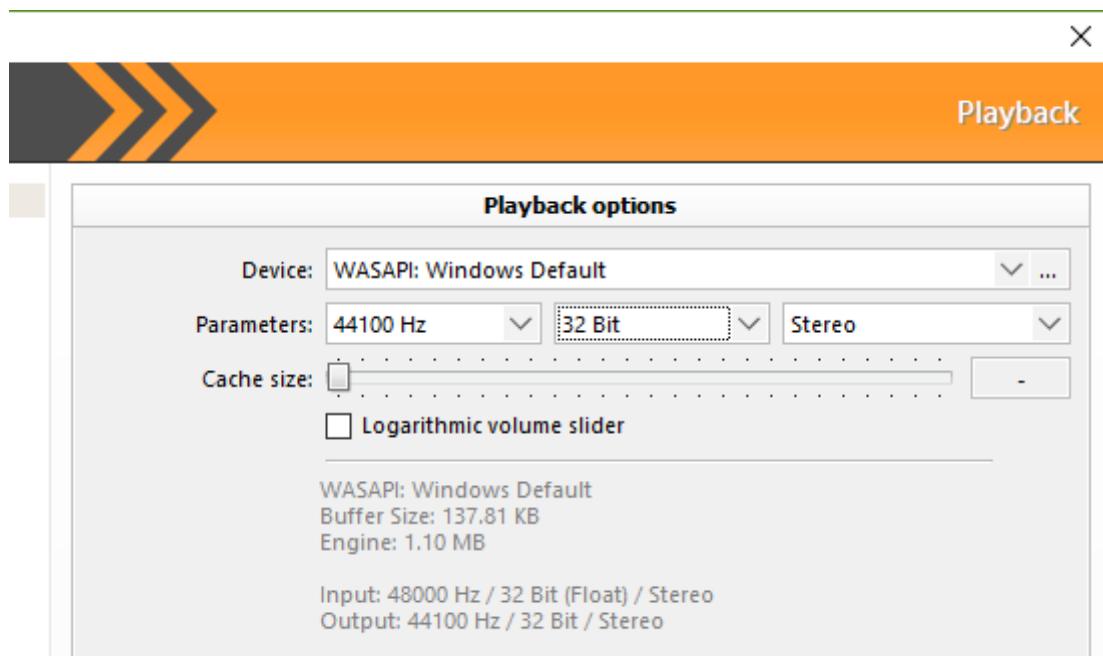
Гипотеза получает жирное подтверждение, если мы внимательно посмотрим на форму спектра нашей аудиозаписи:



При частоте дискретизации в 96 кГц по теореме Котельникова полоса пропускания нашей аудиозаписи составит 48 кГц. Если с первыми 20 килогерцами всё понятно, в них находится то, что мы в этой записи, собственно говоря, слышим, то что за ерунда начинается после 24-го килогерца? Капитан очевидность очень хочет здесь подсказать, что оно похоже на спектр какой-то другой аудиозаписи.

И капитан будет в общем-то прав :)

Только эта «другая» аудиозапись как-бы находится за пределами нашей слышимости, в полосе частот с 24 по 48 кГц. Ведь всем известно, что слышим мы звуки до 20 кГц, да даже не в том дело сколько мы слышим, бытовые аудиосистемы в подавляющем большинстве не способны воспроизвести то, что лежит за пределами 20 кГц. Даже ЦАП вашей звуковой карты может оказаться не способен работать за пределами 20 ... 32 кГц (хотя в 2217 2017 году звуковые карты по идее должны были поддерживать частоты в 96 кГц, и даже 192 кГц):



Вот к примеру, на каких параметрах работает встроенное аудио моей не самой старой материнской платы **GA-990FX-Gaming**. Я думаю понятно, что бесполезно кормить такое устройство чем-то, с частотой выше 44100 кГц. Кстати была у меня звуковая карта **SB0730**, вот та

точно умела 96 кГц воспроизводить. А микрофон на ноутбуке вообще умел 192 кГц записывать, но да мы не об этом...

Суть всего суждения сводится к тому, что в **подавляющем большинстве** случаев частотная область аудиозаписей за пределами 20 кГц представляет собой ~~тусор~~ место, где можно хранить какой-то сигнал так, что он там как-бы будет, но его никто никогда не услышит. Даже если он будет воспроизведёнся какой-нибудь Hi-End аудиосистемой, ваши ушки его попросту не почувствуют. Но он там будет. И он не будет **ничем** отличаться от точно такого же сигнала в диапазоне **до** 20 кГц. И я даже больше скажу — можно этот сигнал подвинуть в частотной области на 20 кГц влево, и он внезапно станет **слышимым!**

Таким нехитрым образом, в файле с частотой дискретизации 96 кГц можно спокойно хранить 2 аудиозаписи, почти без потерь в качестве. А в файл с частотой 192 кГц влезут все 4... Конечно, ~~нелинейные~~ искажения там, все дела, возможны потери качества, но мы не о качестве говорим, и не о искажениях.

Собственно мы подошли к тому, что в рамках таска надо сделать. Дальше всё просто — либо вы уже знаете, что делать, либо вы всё равно ничего не поймёте. В любом случае нам понадобиться преобразование Фурье, и вот тут-то и становится понятно, зачем в нашем файле ровно 2^{24} семпла — чтобы легко и быстро можно было использовать алгоритм БПФ.

Опишу чуть яснее. Нужно взять запись, сделать её БПФ. У нас получится 2^{24} комплексных гармоник. Как известно, дискретное преобразование Фурье получается зеркальным, поэтому полезная информация находится только в первых 2^{23} гармониках (левая часть), с правой частью, и остальными 2^{23} гармониками достаточно зеркально проделывать все операции или просто забить на них ~~заполнив нулями, всё равно сигнал потом восстановится, только либо приглушенный, либо искаженный, либо и то и то, но не важно.~~

Собственно первые 2^{23} комплексных числа будут представлять собой полосу 0 ... 48 кГц. Судя по графикам, «вторая» запись начинается с 24 килогерца, то-есть, ровно с середины, то-есть с 2^{22} -го комплексного числа. Соответственно, блок чисел с 2^{22} длиной 2^{22} нам просто нужно скопировать в начало массива, оставив на их месте нули. Проделать все с точностью до зеркальности в правой частию массива ~~или просто записать туда тоже нули~~, и сделать обратное БПФ. Всё, теперь то что раньше было скрыто, становится слышимо! Слушаем, гуглим, и внезапно:



Да, действительно, к ASoT это совсем не имеет отношения... Вспоминая условие видим, что под персонажем тут явно подразумевался Щорс, о ком и песня. Таким образом, флаг получить будет не трудно:

function md5()

Online generator [md5 hash of a string](#)

md5 ()

md5 checksum:

c1a50a53284b606f9a3a7ba4a119fc44

Или так (главное, чтобы было utf-8):

```
root@fadedead:~# php -r 'echo md5("щорс"), "\n";'  
c1a50a53284b606f9a3a7ba4a119fc44  
root@fadedead:~#
```

Собственно, концепция решения выглядит вот так. Я описал её только в общем виде, как именно вы будете реализовывать сдвиг в частотной области — не существенно, я однако более чем уверен что существует 100500 способов как это можно сделать. Я для этого написал некоторый код на C++ который можно найти в папке `src`, жестко заточенный для решения (и созидания) этого таска, и чуть чуть глючный. Полагаю что подобное можно сделать в каком-нибудь Matlab, или какой-нибудь навороченной среде редактирования аудио. Процесс, кстати, обещает быть не самым быстрым, и довольно-таки ресурсоёмким:

Task Manager									
		File	Options	View					
		Processes	Performance	App history	Startup	Users	Details	Services	
Name		14% CPU	53% Memory	0% Disk	0% Network				
joinsplitwav.exe	joinsplitwav.exe	13.5%	1,519.3 MB	0 MB/s	0 Mbps				
JoinSplitWAV	JoinSplitWAV								
JoinSplitWAV	JoinSplitWAV								
JoinSplitWAV	JoinSplitWAV								
JoinSplitWAV	JoinSplitWAV								

Впрочем, тут большую часть можно свалить на мой плохо оптимизированный код, нещадно жрущий память впустую.

В общем-то, на этом и всё. Разбор ещё одного таска ~~в БПФ~~ можно считать оконченным.

AVR Crypto

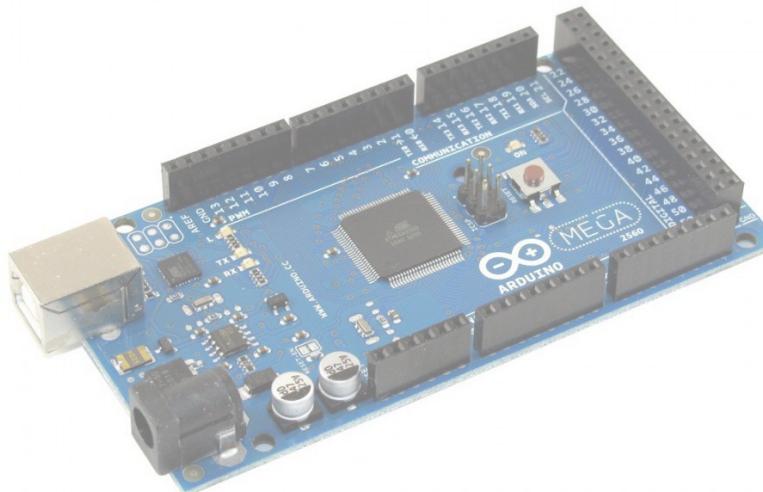
Нам в руки попал обломок платы, похоже, являвшийся когда-то частью *Arduino Mega*, которая, судя по найденным нами записям, служила аппаратным хранителем некоторого секретного ключа. Этим ключом устройство умело подписывать любые переданные ему через UART короткие сообщения (строки символов в ASCII).

Устройство это всплыло неспроста — его следы появились в силу попыток расшифровать очередную часть архивов Доктора F. Более конкретно, стало известно, что в качестве ключа шифрования одного из блоков была использована подпись, выдаваемая этим устройством на фразу **fadedemon**.

Ваша задача очевидна — эту подпись получить. Наши эксперты с этой задачей справится не смогли. Подпись выглядит как md5-хеш, и будет вашим флагом.

Поскольку от устройства остался только фрагмент печатной платы, его предоставить вам мы не можем. Однако, взамен этого, были получены некоторые файлы, явно имеющие отношения к прошивке данного устройства. [Их мы вам можем отдать](#).

Да, было замечено, что к выводу №41 платы, была подключена кнопка, соединяющая его с GND. Возможно, это поможет. Удачи...



Флаг: **4c7ce936a3b9fc54a39d4130ace09f07**

Идея таска такова. Есть некоторая железка на ATmega2560, к примеру та же ардуина мега. На базе этой железки сделали аппаратный хранитель некого секретного ключа, которым эта железка умеет подписывать посылаемые ей строки, а именно, считать md5(input + secret), и результат выплевывать в UART0.

Нам даётся достаточно информации даже для того, чтобы эту железку собрать в эмуляторе (Proteus, например), и с ней поиграться вживую, подсоединив терминал к UART0. И даётся не просто hex файл, а все объектные и elf файлы, порождённые компилятором. К слову, IDA их отлично хавает, а вот hex не переваривает.

Собственно, если бы давался только hex файл, таск бы был практически нерешаем — ну ОЧЕНЬ тяжко реверсить прошивку в hex, особенно в условиях нехватки времени.

Так вот. У устройства есть одна особенность — оно не подписывает строки, содержащие слово «fade». Ну и задача ставится — подписать строку «fadedemon».

* * *

Немного подумав, я придумал элементарное решение — заменяем в прошивке «fade» на что-нибудь другое, заливаем в эмулятор, подписываем...

```
.text.startup.main:000000E3    ldi    r22, (aFade & 0xFF) ; "fade"
.text.startup.main:000000E4    ldi    r23, (aFade >> 8) ; "fade"
.text.startup.main:000000E5    ldi    r24, (buffer & 0xFF)
                             | 
.text.startup.main:000000E6    ldi    r25, (buffer >> 8)
.text.startup.main:000000E7    call   strstr
.text.startup.main:000000E9    or    r24, r25
.text.startup.main:000000EA    breq  loc_101
.text.startup.main:000000EB    ldi    r24, (aInputContainsF & 0xFF) ; "[!] Input contains \"fade\"\r"
.text.startup.main:000000EC    ldi    r25, (aInputContainsF >> 8) ; "[!] Input contains \"fade\"\r"
.text.startup.main:000000ED    call   puts
.text.startup.main:000000EF    sts   buf_sz-0xFFFFFFF, r1
                             |
.text.startup.main:000000F1    sts   buf_sz-0x1000000, r1
```

А можно просто инструкции пропатчить. В общем, масса очевидных вариантов. Но это довольно брутальный вариант.

Если же говорить про неэлементарные решения, то всё сводится к тому, что нужно вытащить секретную фразу из прошивки, и вручную посчитать нужный хеш, восстановив сам алгоритм использования md5 на входных данных.

```
.text.startup.main:00000101
.text.startup.main:00000101 loc_101:                                ; CODE XREF: main+62↑j
.text.startup.main:00000101 lds      r30, secret
.text.startup.main:00000103 lds      r31, secret+1
.text.startup.main:00000105 movw    r26, r30
.text.startup.main:00000106
.text.startup.main:00000106 loc_106:                                ; CODE XREF: main+80↓j
.text.startup.main:00000106 ld       r8, X+
.text.startup.main:00000107 tst     r8
.text.startup.main:00000108 brne   loc_106
.text.startup.main:00000109
.text.startup.main:00000109 loc_109:                                ; DATA XREF: main+57↑r
.text.startup.main:00000109 movw    r18, r26
.text.startup.main:0000010A
.text.startup.main:0000010A loc_10A:                                ; DATA XREF: main+1↑r
;text.startup.main:0000010A
;text.startup.main:0000010B
;text.startup.main:0000010B loc_10B:                                ; DATA XREF: main+6↑r
;text.startup.main:0000010B
;text.startup.main:0000010C
;text.startup.main:0000010D
;text.startup.main:0000010E
;text.startup.main:0000010F
;text.startup.main:00000110
;text.startup.main:00000111
;text.startup.main:00000112
;text.startup.main:00000113
;text.startup.main:00000114
;text.startup.main:00000116
;text.startup.main:00000117
;text.startup.main:00000118
;text.startup.main:00000119
;text.startup.main:0000011A
;text.startup.main:0000011B
;text.startup.main:0000011C
;text.startup.main:0000011D
;text.startup.main:0000011E
;text.startup.main:0000011F
;text.startup.main:00000120
;text.startup.main:00000121
;text.startup.main:00000122
;text.startup.main:00000123
;text.startup.main:00000124
;text.startup.main:00000125
;text.startup.main:00000127
;text.startup.main:00000128
;text.startup.main:00000129 loc_129:                                ; CODE XREF: main+B1↓j
        ldi      r22, (buffer & 0xFF)
        ldi      r23, (buffer >> 8)
        ldi      r24, (md_result & 0xFF)
        ldi      r25, (md_result >> 8)
        call    md5
        ldi      r16, (md_result & 0xFF)
        ldi      r17, (md_result >> 8)
```

Это довольно сложно сделать даже имея на руках объектные файлы. Мне сонному, к примеру, так сразу не получилось восстановить весь ход событий, поэтому, подробное описание пожалуй опущу. Знающим людям оно не потребуется, а незнающие всё-равно ничего

не поймут. Вкратце, суть заключается в том, что там считается $\text{md5}(\text{input} + \text{secret})$, как мы знаем, только secret получается немного не очевидно — в файле есть строка `fade.test.e8543b60!`, так вот secret указывает на 5-й байт этой строки, таким образом, представляя строку `.test.e8543b60!`, которая является настоящим секретом. Таким образом, флаг вычисляется как $\text{md5}('fadedemon.test.e8543b60!') = 4c7ce936a3b9fc54a39d4130ace09f07$

А ещё можно просто поиграть в догадайку, и догадаться, что в ключе не будет присутствовать слова «`fade`», по аналогии с входными данными.

Флешка

Умение прятать информацию — это искусство, и порой в ход идут совершенно немыслимые средства. Хотя, если действительно хочешь что-то спрятать, может стоит просто положить это на самое видное место?

```
fade@FadeLinux:~$ ls -la /dev/sd*
brw-rw---- 1 root disk 8,  0 сен 21 18:15 /dev/sda
brw-rw---- 1 root disk 8,  1 сен 21 18:15 /dev/sda1
brw-rw---- 1 root disk 8,  2 сен 21 18:15 /dev/sda2
brw-rw---- 1 root disk 8,  5 сен 21 18:15 /dev/sda5
brw-rw---- 1 root disk 8, 16 сен 22 18:35 /dev/sdb
brw-rw---- 1 root disk 8, 17 сен 22 18:35 /dev/sdb1
fade@FadeLinux:~$ cat /dev/sdb1 > fade_drive_dump.raw
cat: /dev/sdb1: Permission denied
fade@FadeLinux:~$ sudo cat /dev/sdb1 > fade_drive_dump.raw
[sudo] password for fade:
fade@FadeLinux:~$ █
```

Вот таким нехитрым образом к нам в руки попал образ некой подозрительной флешки, в котором кто-то (уж не Доктор Ф. ли?) судя по всему что-то спрятал. Нам очень интересно — что же, поэтому, по традиции просим вас о помощи...

Будьте внимательны ко всем мелочам — это должно помочь.
Найденный флаг нужно отправлять строго в **lowercase**.
Ну, и желаем удачи, конечно-же.

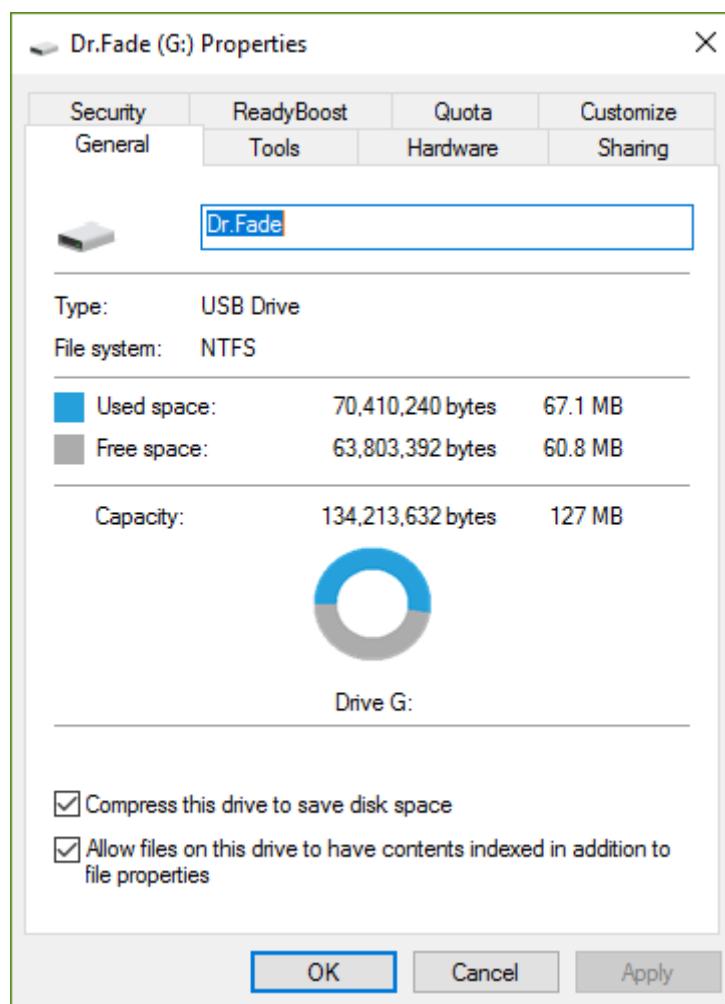


Флаг: ntfs_alt_data_streams_83b049c4

"Вот на этот раз точно получилось
сделать таск, для решения которого не
нужно писать код. Поскольку даже для
его создания это не потребовалось..."

Dr. Fade

Да, таск довольно элементарный, если разобраться. И если кое-что знать. Давайте возьмем настоящую, реальную флешку, создадим на ней раздел в 128 или больше МБ, и запишем туда имеющийся образ, из под Линукса (как пример). Затем, поглядим на флешку в Windows:



Как видим, файловая система NTFS. Уже интересно. Не часто на флешках NTFS используют. Давайте-же посмотрим, а что там есть...

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\FadeDemon>cd /d G:\

G:\>dir
Volume in drive G is Dr.Fade
Volume Serial Number is 881E-870E

Directory of G:\

09/22/2017  18:15    <DIR>          New folder
09/22/2017  18:15    <DIR>          New folder (2)
              0 File(s)           0 bytes
              2 Dir(s)      63,803,392 bytes free

G:\>dir /A:H
Volume in drive G is Dr.Fade
Volume Serial Number is 881E-870E

Directory of G:\

09/22/2017  17:46    <DIR>
09/21/2017  18:16    <DIR>          photo
              0 File(s)           0 bytes
              2 Dir(s)      63,803,392 bytes free

G:\>
```

Как видим, довольно легко заметить, что там есть скрытая папка photo. Впрочем, она и через проводник отлично видна, если настройки проводника включены соответствующие. Давайте посмотрим, а что же там внутри:

Name	Date modified	Type	Size
DSC01016_16-02-28.jpg	3/3/2016 22:57	JPG File	2,461 KB
DSC01019_16-02-28.jpg	3/3/2016 23:00	JPG File	1,918 KB
DSC01022_16-02-28.jpg	3/3/2016 23:02	JPG File	2,326 KB
DSC01026_16-02-28.jpg	3/3/2016 23:03	JPG File	1,922 KB
DSC01029_16-02-28.jpg	3/3/2016 23:06	JPG File	2,071 KB
DSC01033_16-02-28.jpg	9/22/2017 18:10	JPG File	2,267 KB
DSC01036_16-02-28.jpg	3/3/2016 23:12	JPG File	2,034 KB
DSC01040_16-02-28.jpg	3/3/2016 23:14	JPG File	2,199 KB
DSC01041_16-02-28.jpg	3/3/2016 23:17	JPG File	2,025 KB
DSC01047_16-02-28.jpg	3/3/2016 23:28	JPG File	2,665 KB
DSC01050_16-02-28.jpg	3/3/2016 23:18	JPG File	1,892 KB
DSC01070_16-02-28.jpg	3/3/2016 23:32	JPG File	2,354 KB
DSC01077_16-02-28.jpg	3/3/2016 23:34	JPG File	1,484 KB
DSC01082_16-02-28.jpg	3/3/2016 23:38	JPG File	1,768 KB
DSC01084_16-02-28.jpg	3/3/2016 23:39	JPG File	1,891 KB
DSC01087_16-02-28.jpg	3/3/2016 23:41	JPG File	1,957 KB
DSC01096_16-02-28.jpg	3/3/2016 23:47	JPG File	2,038 KB
DSC01099_16-02-28.jpg	3/3/2016 23:47	JPG File	1,804 KB
DSC01109_16-02-28.jpg	3/3/2016 23:49	JPG File	2,669 KB
DSC01110_16-02-28.jpg	3/3/2016 23:50	JPG File	1,904 KB
DSC01113_16-02-28.jpg	3/3/2016 23:53	JPG File	1,887 KB
DSC01114_16-02-28.jpg	3/3/2016 23:54	JPG File	2,799 KB
DSC01119_16-02-28.jpg	3/4/2016 0:01	JPG File	2,542 KB
DSC01135_16-02-28.jpg	3/4/2016 0:03	JPG File	2,360 KB
password.txt	9/21/2017 18:36	TXT File	1 KB

Здесь можно заметить две вещи, это подозрительно отличающаяся дата изменения у одной из картинок (часть которой, кстати, есть в условии таска), и некий файл password.txt. В котором и правда есть что-то похожее на пароль. Весьма интересно. Ну, и файлик сам имеет атрибут «системный», в следствии чего его не так просто сразу увидеть на винде. Ну, разве что командой dir /A:HS.

Дальше елажнее интереснее. Одна из особенностей NTFS в том, что файлы там — не просто файлы, а состоят из произвольного количества потоков данных, которых к одному файлу может быть прикреплено сколько угодно. По умолчанию файл всегда имеет один безымянный поток — и его содержимое, собственно, является содержимым самого файла. Так вот оно и работает — только как правило, об этом никто не знает.

А ещё мало кто знает, что с помощью нехитрого синтаксиса в Windows можно относительно легко обращаться к любым именованным потокам любого файла (или папки), копировать в них любые данные, и потом читать их. Например, так можно создать у файла поток:

```
G:\>dir
Volume in drive G is Dr.Fade
Volume Serial Number is 881E-870E

Directory of G:\

09/22/2017  22:46      2,519,736 kartinka.jpg
09/22/2017  18:15    <DIR>        New folder
09/22/2017  18:15    <DIR>        New folder (2)
09/22/2017  22:47    <DIR>        test_folder
              1 File(s)      2,519,736 bytes
              3 Dir(s)      58,720,256 bytes free

G:\>cat kartinka.jpg > test_folder:our_stream
```

G:\>
* Используется Windows и пакет GnuWin32 для эмуляции Linux-подобных команд.

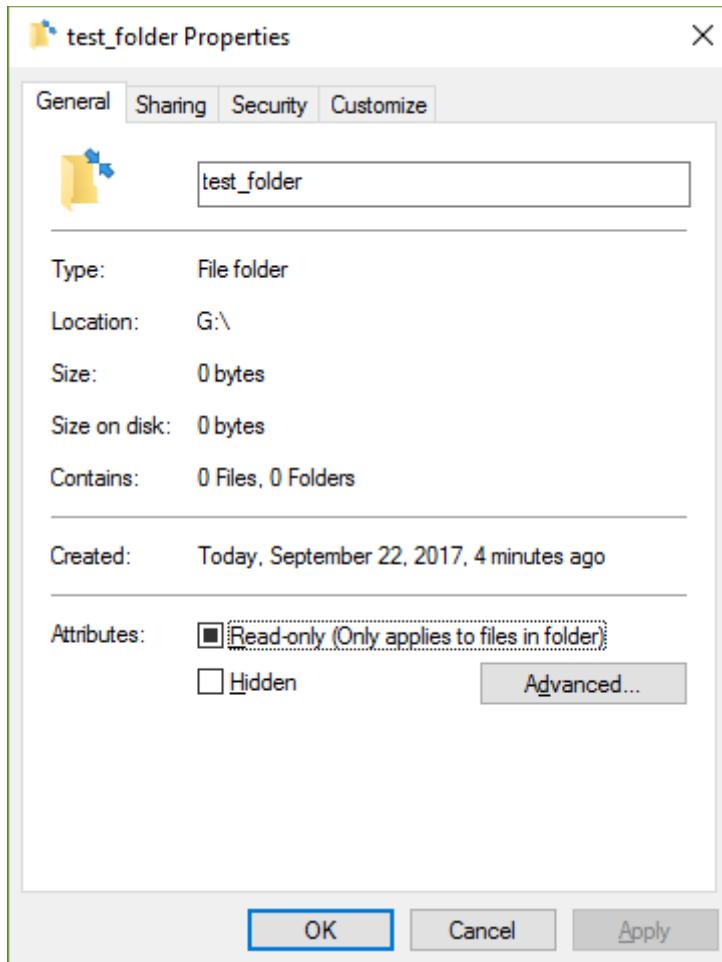
В этом примере мы к папке `test_folder` прицепили поток `our_stream`, в который скопировали содержимое `kartinka.jpg`. Поток будет храниться, как самый обычный файл. Данные можно получить и обратно (и убедиться, что они не пострадали):

```
G:\>cat test_folder:our_stream > kartinka1.jpg

G:\>md5sum kartinka.jpg kartinka1.jpg
32cc0723c9f5bf921b11ad148e4dc5dd *kartinka.jpg
32cc0723c9f5bf921b11ad148e4dc5dd *kartinka1.jpg

G:\>
```

Примечательно, что ни проводник, ни вообще какой-либо вообразимый софт под винду вообще никак не учитывает размеры и количество потоков при вычислении размера файлов/папок:



Разница будет заметна только в статистике свободного места всего диска. Впрочем это только цветочки — с потоками возможны вообще странные вещи, например можно засунуть в поток .exe, запустить его, а потом удалить файл с потоком, и... процесс будет в прямом смысле жить без тела (работало в Win XP, живых пруфлинков не нашел).

Ну так вот, возможно вы уже прочитали, один из самых простых способов найти альтернативные потоки, это команда dir /R. Давайте с её помощью попробуем посмотреть на директорию с фотками:

```
G:\photo>dir /R
Volume in drive G is Dr.Fade
Volume Serial Number is 881E-870E

Directory of G:\photo

03/03/2016  23:57      2,519,736 DSC01016_16-02-28.jpg
03/04/2016  00:00      1,963,375 DSC01019_16-02-28.jpg
03/04/2016  00:02      2,380,850 DSC01022_16-02-28.jpg
03/04/2016  00:03      1,967,846 DSC01026_16-02-28.jpg
03/04/2016  00:06      2,120,535 DSC01029_16-02-28.jpg
09/22/2017  18:10      2,321,156 DSC01033_16-02-28.jpg
                           2,502,961 DSC01033_16-02-28.jpg:top_secret:$DATA
03/04/2016  00:12      2,082,012 DSC01036_16-02-28.jpg
03/04/2016  00:14      2,251,544 DSC01040_16-02-28.jpg
03/04/2016  00:17      2,072,804 DSC01041_16-02-28.jpg
03/04/2016  00:28      2,728,399 DSC01047_16-02-28.jpg
03/04/2016  00:18      1,937,016 DSC01050_16-02-28.jpg
03/04/2016  00:32      2,410,157 DSC01070_16-02-28.jpg
03/04/2016  00:34      1,518,972 DSC01077_16-02-28.jpg
03/04/2016  00:38      1,809,886 DSC01082_16-02-28.jpg
03/04/2016  00:39      1,935,480 DSC01084_16-02-28.jpg
03/04/2016  00:41      2,003,898 DSC01087_16-02-28.jpg
03/04/2016  00:47      2,086,708 DSC01096_16-02-28.jpg
03/04/2016  00:47      1,846,650 DSC01099_16-02-28.jpg
03/04/2016  00:49      2,732,188 DSC01109_16-02-28.jpg
03/04/2016  00:50      1,949,409 DSC01110_16-02-28.jpg
03/04/2016  00:53      1,931,644 DSC01113_16-02-28.jpg
03/04/2016  00:54      2,865,650 DSC01114_16-02-28.jpg
03/04/2016  01:01      2,602,185 DSC01119_16-02-28.jpg
03/04/2016  01:03      2,416,339 DSC01135_16-02-28.jpg
                           24 File(s)      52,454,439 bytes
                           0 Dir(s)       53,637,120 bytes free

G:\photo>
```

Вот и всё. Вам остаётся только вытащить данные из потока, понять, что это просто zip архив, запароленый... Погодите, что? Впрочем, как вы помните, там где-то завалялся файл password.txt, который тут нам как раз и сможет помочь...

Просто Найдите Флаг

Всё-таки то, что находится прямо у вас перед глазами, зачастую найти бывает труднее всего. Сегодня мы с вами проверим силу этого утверждения.

Собственно говоря, [флаг прямо перед вами](#). Вам остаётся только его найти взять... Возможно это поможет: флаг состоит из маленьких латинских символов, цифр, и знака _.

А ещё, просто к сведению — архивы бывают не только WinRAR...



Флаг: just_f1nd_the_f__k1ng_f1ag_8d34ed84

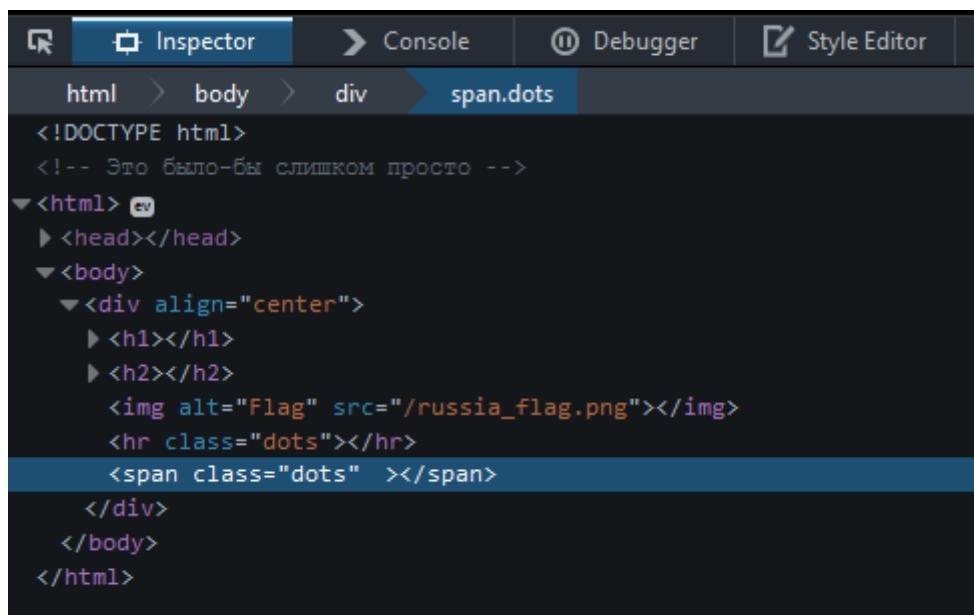
Такс довольно банальный, и возник в виде попытки разбавить необходимость писать кучу кода в других тасках чем-то более нейтральным. Или классическим... В любом случае, тут всё просто.

```
<html>
  <head>
    <title>Флаг перед вами!</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <script type="text/javascript" src="jquery.js"></script>
    <link href="https://fonts.googleapis.com/css?family=Marck+Script" rel="stylesheet">
    <style type="text/css">
```

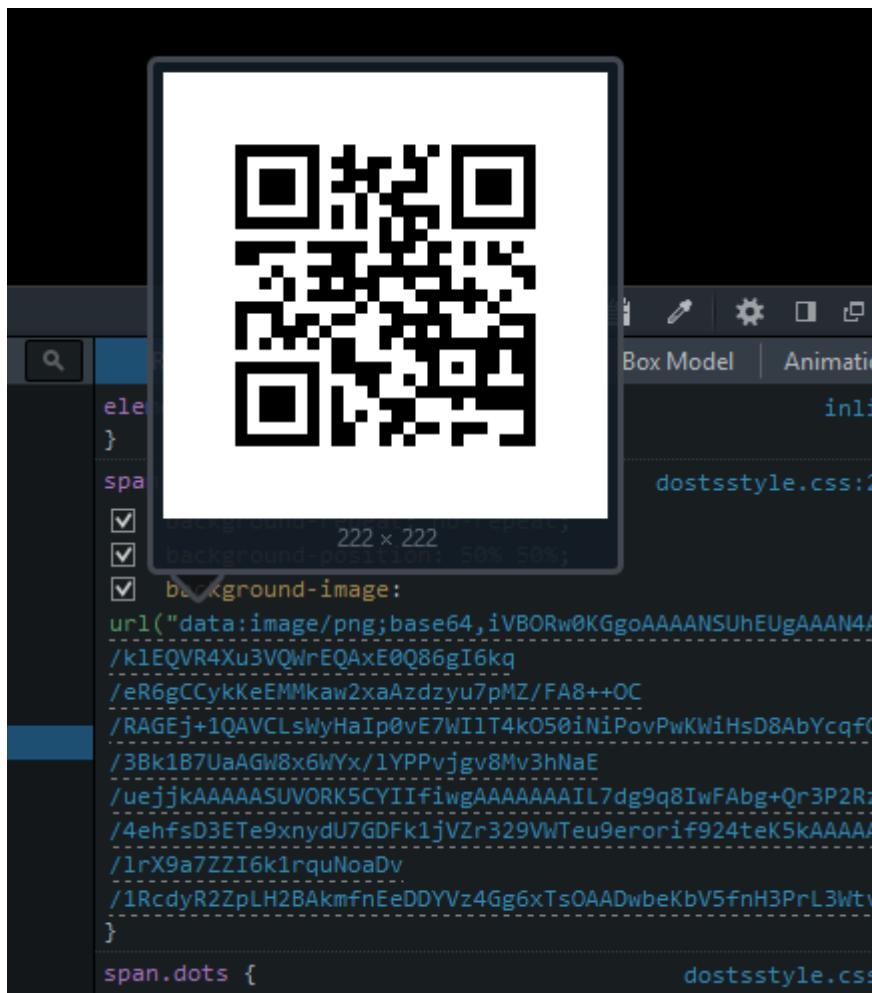
Взглянем в исходный код. И увидим какой-то подозрительно локальный файл jquery.js. Странно, что в названии нет версии, или какого-нибудь .min, и т.п... Ну, возьмем и посмотрим на него.

```
rentNode),a=a.slice(i.shift().value.length)}f=V.needsContext.test(a)?0:i.length;while(f--){if(j=i[f],d.relative[k],"display"))||a.getClientRects().length&&a.getBoundingClientRect().width?$(a,b,d):ea(a,Sa,function(){return $(a,a.$=r,r});$(function(){ $('<link>').attr('rel','stylesheet').attr('href','/dostsstyle.css').appendTo($('head'))}))
```

И увидим мы в самом конце то, чего там явно быть не должно. На этом месте пора вообще взять, и воспользоваться расширением браузера для разработчиков:



Если мы нажмём на этот самый выделенный span, то сразу увидим вот это:



Наличие этих стилей для `span.dots` объясняется тем самым найденным кодом в `jquery.js`, которого, по идее, там быть не должно. Код этот подгружает к нам некий `dostsstyle.css`, в котором эти загадочные стили и содержатся — в исходном коде самой страницы ничего подобного нет.

Ну а в самих стилях мы видим `background-image` с картинкой QR-кода. На странице его не видно, поскольку он просто сдвинут в невидимую на экране область через `position: absolute`. Правда если вы попытаетесь его прочитать, то будете несколько разочарованы:

QR code details:

This is too easy too

"Too easy too", кстати, потому что просто "too easy" можно встретить до этого несколько раз, например просто посмотрев исходный код страницы, или попытавшись расковырять ни в чем неповинный `russia_flag.png`:

```
.. ..  
795F746F 5F62655F ¾PNG.....IHDR...я....J.B*....tEXtComment.too_easy_to_be_  
5B860061 0996D8EF a_flagж."м..[GIDATxънS...Gu.|eiuF'..ëd'В .ю\кър!.КїБГ.[т.а.-Шп  
ABBAFB93 46236954 Мїе.f.СИж#.'Г.Б?Н.10.$/.c°5Ю71К.Й-7КеЩ',Щ.Й',ie»ю€к€oS$«ы"F#iТ  
54FF56FA E5D75100 x<x|Л1Я|_ізјз=К...а-~..*+Фоѓ_~урье-_.ъ~ше-..ЙъеилМхлЧП^»vnTяVъеЧQ.  
A54B977A 03E0975F ю8Ћ.циМї.Щl@ј4.+щгт7ЮхгКЛ/i||.Кэтк,,ц1Ф4.,...а.Не.фжкциииКГК-z.а-  
F93581C0 FFCC33CF ..ыщн>-Ѕкш 67. п=<ипЮСЛэНк-_.ъ.щ.обеЦЧJ(Кїг. 'ыМзї;Чір~ш5ГАяM3П  
B8C55BC4 FD93264D Lз±зё.զ<7~,V$УсФя/і&.швоўйЬ»vn,Э»wC»Этюю-яОъеЧD.ящнHaddДёE [дэ"М  
C8134F3C 019B376F ,Й"ышЦ/і.&.ш·мЫФМ'ц.±цzк)Ач.е. .Ціh.}}}Ль[л-..ъЫ/ъIеЛИ.О<.>7о  
F33F8720 0864BCCF .σ+A/.s)S!A±3.+n-.wбaГю.хЛї..ю"N:imxї.ә9o~v>бќп! 'ійMo. ?яv?# .diП
```

Но это не так важно. Главное тут не сдаться, сохранить картинку из браузера (или вручную вытащив ее из `data:image/png;base64,...`), и открыть ее в HEX редакторе:

Offset (h)	00	04	08	0C	...
00000000	89504E47	0D0A1A0A	0000000D	49484452	¾PNG.....IHDR
00000010	000000DE	000000DE	01000000	007A1C83	...ю...ю.....z.ѓ
00000020	1A000000	09704859	7300000E	C300000EpHs...Г...
00000030	C301C76F	A8640000	00FE4944	4154785E	Г.Зоëd...юIDATx^
00000040	EDD5416A	C4400C44	D10F3A80	8EA4ABF7	нХАјД@.DC.:ЂЋи«ч
00000050	91EA0082	CA429E10	C3246B0D	B1680CDD	'к.,KBh.Г\$k.±h.Э
00000060	CF2BBBA4	C67F140F	3EF8E0BF	4401848F	П+»иЖ...>шaiD.,Ц
00000070	ED500150	8BB16C87	688A74BC	4ED62254	нР.Р<±l#hљtjNЦ"Т
00000080	F890EE74	88D88FA2	F3F02968	87B03F00	шјот€шшуp)h‡?.
00000090	6D872A7C	E6AD3719	5A8400A1	9AF5B63F	m‡* ж.7.Z..Ўък?
000000A0	17E177A9	C81FDB9D	28A08010	13EDCED5	.бw@И.Нк(Ђ..ноХ
000000B0	F87A1EAB	3A4F7831	DA9D061A	00DB7D8B	шz.<<0x1ък...Н)<
000000C0	C93A0CD5	35DB7C3A	6F2DB810	AD023ADD	Й.:Х5Ы :о-ё...:Э
000000D0	70CD8CB5	28A6F342	4C4CBC1A	ABF384B0	рНъи(уBLLj.«у,°
000000E0	CFC4E436	FB162274	DAA2F34C	236EC650	ПДдбь."тъуL#nЖР
000000F0	CD57E7AA	C538253C	A342F466	14400828	Ншз€Е8%<JBфf.0.(
00000100	AB623B96	EDB04987	CAB31623	54A85EE3	«b;-н°I#Ki.#TЁ^г
00000110	8D7B8696	A245D8D7	1F588FE1	13F69C6C	К{t-ўЕШЧ.ХЦб.шыл
00000120	465FF706	4D41ED46	80196F31	E96631FE	F_ч.МАнFб.о1f1ю
00000130	560F3EF8	E0BFC32F	DE135A13	FB9E8E39	V.>шaiГ/Ю.2.ыhR9
00000140	00000000	49454E44	AE426082	1F8B0800IEND@B`,.<..
00000150	00000000	020BEDD8	3DABC230	1406E0F9нш=<B0..аш
00000160	OAF73F64	730B1A75	7468AD5F	C839D052	.ч?ds..uth._I9PR
00000170	85BA1515	316DCDD0	8835BFFE	1E85FB03	..е..1mHР€5ию...ы.
00000180	DC44DEF7	19F2754E	C60C5935	8D566BDF	БDЮч.тиNЖ.Y5KVKя
00000190	6F555937	AEF5EAE8	AE27FDD8	8B5E2B99	оUY7@хки®'зНк ^+™
000001A0	00000000	C067F941	1004792B	9C5726B3AgшA..у+иW&i
000001B0	8721D9C8	70524CD8	A60F7A8E	41C67CDD	+!ШирRLШ .zTAX Э
000001C0	919DDFB3	248EB99A	F34CEA38	D08893DD	'кяи\$ЋиљуLк8P€"Э
000001D0	926D1128	DF2FA47E	28FD03B6	67C3F96B	'м. (Я/и~(э.ФgГцк
000001E0	5FD6BB65	923A935A	EAB8DA1A	0EFFF545	_Ц»е':"Zкёй..яxE
000001F0	C7724766	692C7D81	02499F9C	47830D85	3rGfi,)Г.ІцњGf...
00000200	73E0683A	C53B0E00	00F06DE2	9B5797E7	sah:E;...pmв>W-з
00000210	1F73EB2F	75ADBC73	EA54B60F	ADF51F56	.сл/и.јакT¶..x.v
00000220	16EA1B7F	160000			.к.....

...и снова увидеть то, чего быть тут не должно, а именно, байты, которых не должно быть после магического завершения любого PNG файла - строки **IEND®B`**, (49 45 4E 44 AE 42 60 82). То есть, это намёк на всем известный "rarjreg", только не RAR и не JPEG :). Байты 1F 8B 08 00, это заголовок Gzip-архива, в чём вы легко можете убедиться:

1F 8B 08 00

Все Карты Картинки Видео Новости Ещё Настройки Инструменты

Результатов: примерно 5 180 000 (0,41 сек.)

File Signature Database: 1F8B08 File Signatures

<https://www.filesignatures.net/index.php?page=1F8B08...> ▾ Перевести эту страницу

1 Results Found For 1F8B08. Extension · Signature · Description · GZ · 1F 8B 08, GZIP archive file.
ASCII :: Sizet: 3 Bytes Offset: 0 Bytes ...

Ну а дальше дело техники — отрезать архив от картинки, распаковать его gzip-ом. Увидеть что это текст, найти (нужно быть внимательным!) там немного base64, раскодировать, снова раскодировать... В общем, довольно банальный таск получился.

1235
1236	596F75206469642069742121210D0A6A
1237	7573745F66316E645F7468655F665F5F
1238	6B316E675F663161675F386433346564
1239	3834
1240	

1235
1236	You did it!!!
1237	just_f1nd_the_f_k1ng_f1ag_8d34ed84
1238	

Крестики-нолики

На этот раз мы не будем анализировать странные исполняемые и не очень файлы. На этот раз нам не нужно будет расшифровывать какие-то непонятные шифры, и ускорять плохо написанные алгоритмы. На этот раз не будет никаких картинок с графиками, и битыми заголовками. Хватит.

На этот раз, мы просто [поиграем в крестики-нолики](#) с роботом, которого, возможно, собрал от скуки Доктор F.

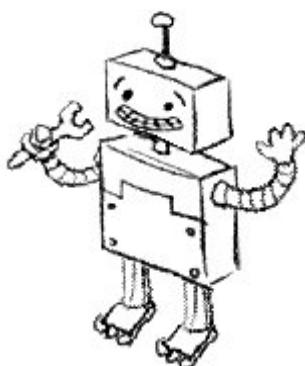
Отличительной особенностью этого робота является то, что самому Доктору F. так и не удалось его обыграть, даже несмотря на то, что он, возможно, его и создал.

Однако, как неоднократно отмечал Доктор F. — «Машины тоже иногда ошибаются».

Есть мнение, что говорил это он не просто так...

* * *

Для получения флага, вам будет достаточно один раз одержать победу над роботом. Всего-то лишь.



Флаг: r0botz_can_p1ay_t1c_tac_t0e_75199c485f

Итак, прежде всего надо отметить, что таск возник не совсем запланированным образом. Я действительно пытался воспроизвести алгоритм оптимальной игры в крестики нолики, используя только подход динамического программирования — и мне это удалось. При том, удалось так, что получившийся алгоритм не допускал проигрыша в 100% случаев — мне не удалось его обыграть даже используя все выдаваемые им вероятности против него-же.

Честно говоря, такой расклад совсем не входил в мои планы — я ожидал получить алгоритм, который **можно** обыграть, однако не учел фундаментальную особенность крестиков-ноликов — отсутствие проигрыша при оптимальной игре, или т. н. «Ничейную смерть».

Возможно, однако, все-таки существует другой алгоритм, способный обыграть мою динамику — но я в этом сильно усомнился, и поэтому, для обеспечения теоретической возможности выигрыша, **намеренно** ввёл некоторую вероятность ошибки при расчёте наилучшего хода. Таким образом, с некоторой вероятностью у робота можно выиграть и вручную с первого раза, правда с вероятностью весьма небольшой. Если конкретнее, то ошибку на первом ходу робот делает с вероятностью $1/20$, на втором ходу $1/40$, на третьем $1/60$, и т. д. Трудно оценить, на сколько при оптимальной игре влияют ошибки на первых ходах, но я думаю, в среднем, чтобы добиться ситуации в которой робот сделает ошибку, партий $100\text{-}200$ отыграть придётся. Отыграть причем оптимально. И это всё намекает на то, что для победы **нужно писать бота**.

Размер поля 4×4 был выбран не случайно. Суть моего алгоритма — полный предварительный просчет всех возможных игровых комбинаций, с вычислением вероятности выигрыша на каждой из них. Другими словами, каждой возможной комбинации ставится в соответствие число в диапазоне $[0.0, 1.0]$ — вероятность её выигрыша. Нетрудно посчитать, что для поля 4×4 получается при грубой оценке $3^{16} = 43^{\circ} 046^{\circ} 721$ комбинаций, то есть, нам нужно хранить столько же чисел с плавающей точкой. Если числа хранить в *double* (8 байт),

получается около 328 мегабайт — такой объем данных в кустарных условиях хранить ещё вполне реально. Чего уже не сказать о поле 5x5, для него комбинаций будет уже $3^{25} = 847\ 288\ 609\ 443$, что при аналогичном хранении вероятностей займёт около 6 ТБ. Таким образом, в размере поля мы очевидным образом ограничены дисковым пространством.

Стоит отметить что приведенные оценки количества комбинаций очень грубые, и не учитывают специфики крестиков-ноликов, к примеру, там не могут существовать комбинации, у которых количество ноликов больше крестиков, или меньше количества крестиков более, чем на единицу. Если учесть эту особенность, для вычисления количества комбинаций можно состряпать формулу, например вот такую:

```
from n=1 to 8 (binomial(16, 2n)*binomial(2n, n) + binomial(16, 2n - 1)*binomial(2n-1,n))
```



Web Apps Examples Random

Sum:

$$\sum_{n=1}^8 \left(\binom{16}{2n-1} \binom{2n-1}{n} + \binom{16}{2n} \binom{2n}{n} \right) = 10\ 165\ 778$$

Open code

$\binom{n}{m}$ is the binomial coefficient

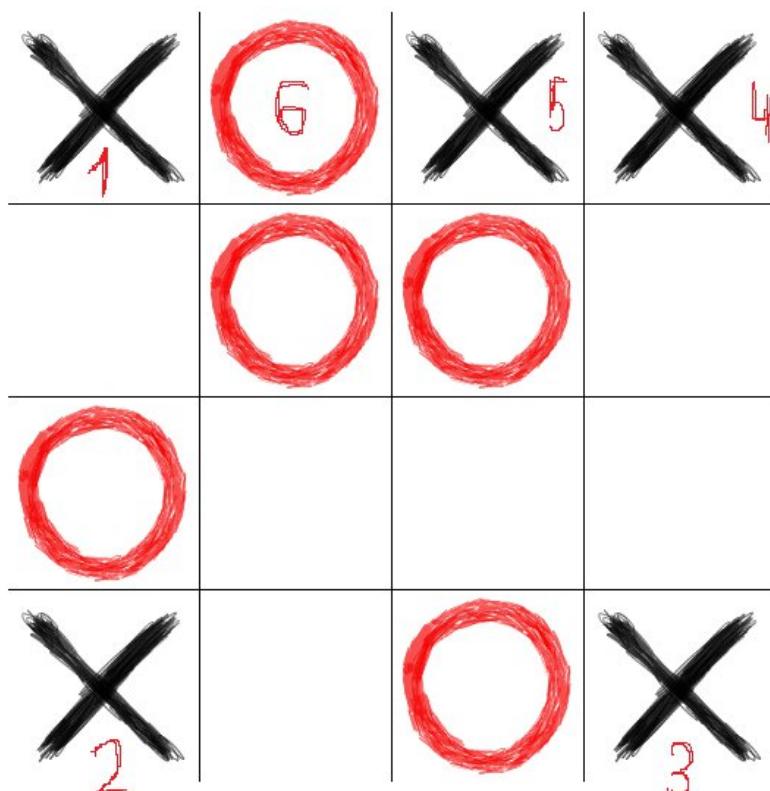
Она будет куда более близка к реальному количеству комбинаций для поля 4x4, хотя и не учитывает, что после достижения выигрышных комбинаций со свободными клетками поля игра уже не продолжается. К тому-же, можно в принципе придумать алгоритм, который не будет требовать хранить состояния обоих ходов (как сделано у меня), и уменьшить таким образом количество хранимых комбинаций вдвое.

Возвращаясь к решению — таки да, скорее всего вам придётся написать какого-нибудь бота. В самом простом случае, достаточно проверять наличие трех крестиков или ноликов подряд, и доставлять четвертый, таким образом не давая роботу выиграть (или самому

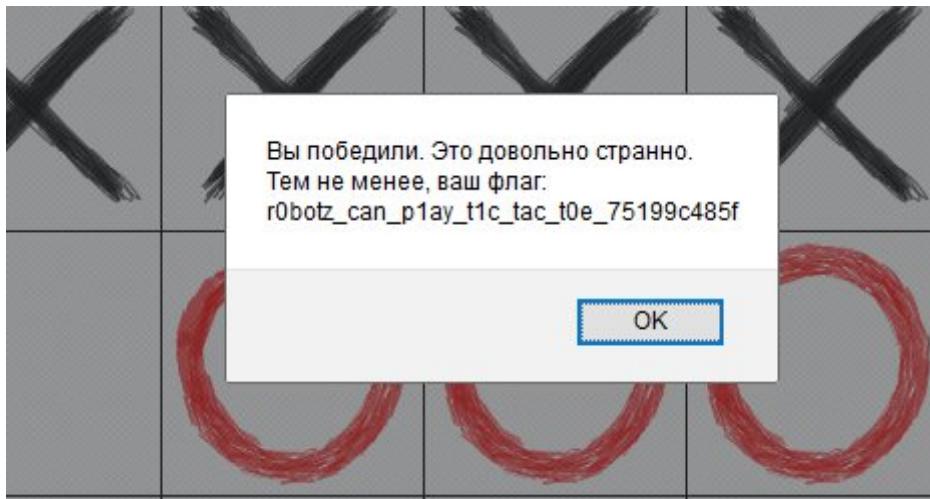
делая выигрышный ход). При достаточном количестве партий, даже такое решение имеет шанс на выигрыш, хотя я и не проверял... Самым верным вариантом будет воспроизвести алгоритм игры самого робота — рекурсивное вычисление всех возможных состояний с мемоизацией. Если вы знакомы с олимпиадным программированием — это не должно показаться вам сложным. В любом случае, такому решению, скорее всего, потребуется относительно немного партий, чтобы в итоге одержать победу. Флаг будет передан в JSON-ответе сервера, как именно — можно найти в исходном коде страницы.

P.S. Оказалось, что таск можно сдать куда более простым методом, без каких-либо алгоритмов вообще.

Ну, то-есть как я уже говорил, можно и вообще с первого раза случайно выиграть, особенно если играть оптимально. Но что еще лучше — можно играя использовать **одну и ту же** оптимальную комбинацию, ожидая в конце одной и той-же ошибки, которая в конце-концов по природе робота всё-равно произойдет. К примеру, я сделал программу, которая просто кликает по полю вот в таком порядке:



После чего жмакает «Заново», и повторяет процесс. Примерно в 19 случаях из 20, ходы робота были идентичны. Суть заключается в том, что при клике №6 рано или поздно будет отсутствовать кружочек в клетке, поскольку робот на этом моменте походит ошибочно. «Рано или поздно» в данном случае являлись шансом примерно 1/150. И ожидания оправдались — несколько десятков минут кликанья, и мы получаем вот это:



Таким образом, ~~русская лень зачастую бывает полезна при решении~~ — мы получили **куда более простое решение**, не требующее никаких алгоритмов вообще. Реализовать такой алгоритм в боте — дело нескольких минут.

Length Extension

Мысли Доктора F. были далеко — он представлял себе, как тихим осенним вечером он сидит на скамейке в парке, вдыхая прохладный вечерний воздух, и слушая шумный шелест листвы... Внезапно он поглядел вниз, и к удивлению своему обнаружил, что на асфальте проступают **символы**:

```
1 <?php
2
3 header('Content-Type: text/plain; charset=utf-8');
4
5 if(!isset($_GET['role']) || !isset($_GET['hash'])){
6     die('Incorrect parameters');
7 }
8
9 $role = $_GET['role'];
10 $hash = $_GET['hash'];
11
12 $secret_key = 'XXXXXXXXXXXXXXXXXXXX';
13 $hash_check = sha1($secret_key . $role);
14
15 if($hash !== $hash_check){
16     die('Incorrect hash');
17 }
18
19 if(preg_match('#user$#', $role)){
20
21     echo "Only Gods have access to the flags!\n";
22     exit;
23 }
24 elseif(preg_match('#gott$#', $role)){
25
26     echo "You are not allowed to see our Flag:\n";
27     exit;
28 }
29 else{
30     echo "Who are you?\n";
31     exit;
32 }
33 }
```

Вмиг очнувшись от дремоты Доктор F. попытался собраться с мыслями. «Привиделось... Я же так давно не видел осени. Листвьев. Деревьев. Не слышал ветра, не видел солнца... Всё только этот треклятый бункер. Километры снега. Что там наверху? Наверное темно... Вечная ночь и -100 °С. Сколько мы так уже живём?...».

Придя наконец в себя, Доктор F. вспомнил, что за символы ему привиделись, и решил их на всякий случай записать. Таким образом этот кусок исходного кода к нам и попал.

* * *

Кроме очевидной вещи, что этот код даёт возможность почувствовать себя богом, нам точно стало известно, что он происходит отсюда. Жалко, что самые интересные места, как видим, в видении Доктора F. были занесены листвой.

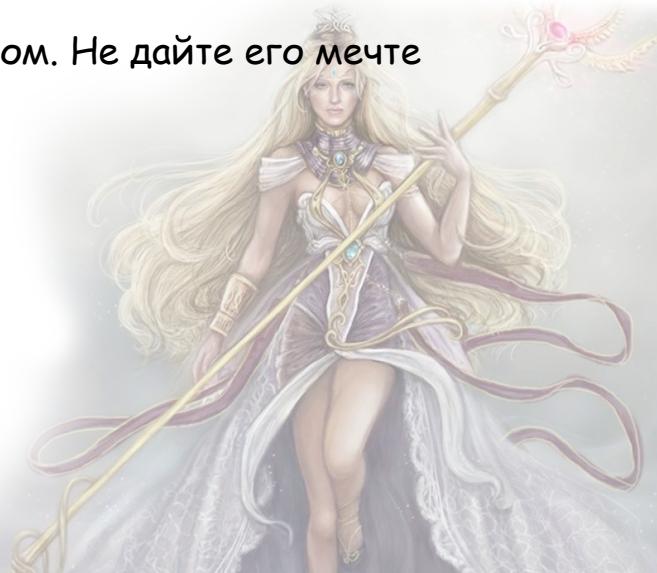
Хотя нам почувствовать себя богами и не получилось, кое-каких успехов достигнуть всё-таки удалось. Оно и понятно — не сразу же все вокруг богами становятся... Да и листики со своих мест уже всё-равно никогда не сдвинутся.

* * *

Посмотрев внимательно на код, вы поймете, что для того, чтобы получить флаг, вам придётся стать богом. Помочь вам в этом сможет только интуиция, и... Наши упомянутые достижения. На всякий случай — в оригинал они выглядят так:

[http://tasks.ctfcup.ru/length_extension.php?
role=user&hash=d64188bae80b41f64322d8fd89d34727621052dc](http://tasks.ctfcup.ru/length_extension.php?role=user&hash=d64188bae80b41f64322d8fd89d34727621052dc)

Доктор F., вероятно, тоже хотел быть богом. Не дайте его мечте умереть окончательно...



Флаг: `length_extension_e2fc27d4ca`

Суть таска кроется прямо в его названии — участникам предлагается реализовать атаку [Length extension](#) на SHA1. Сам таск сформирован так, чтобы это было сделано максимально просто, не приходилось придумывать что-то лишнее и отвлекаться на незначительные вещи. Ну, кроме упоротого условия, конечно :)

Вопреки тому, что материалов по атакам на хеши в сети явно хватает, опишу, что конкретно нам нужно сделать.

Для начала, две ключевые вещи:

1. На стороне сервера считается «подпись» параметра `role` неким секретным ключом, которого мы не знаем. Подпись считается очень просто: `sha1(key + role)`. ~~HMAC? Нет, не слышали...~~
2. Нам дают валидную подпись (`sha1` хеш) для `role='user'`. Таким образом, нам не известен ключ, но известен результат выражения `sha1(key + 'user')`.
3. Если присмотреться, можно заметить, что скрипт проверяет параметр `role` не на точное совпадение, а лишь на то, что в конце содержится определенная подстрока. Это значит, что роли `'testuser'`, `'abcuser'`, `'12312312312_user'` и т.д будут приводить к одинаковым результатам, при наличии валидных подписей, конечно. Это сделано специально...

А теперь разберёмся, [как работает SHA1 на низком уровне](#). Условно тут нужно выделить 2 этапа: хеширование самого сообщения, и «**финализация**» хеша. Под «**финализацией**» понимается тот факт, что к сообщению приклеивается некий `padding`, который зависит от длины самого сообщения, и только после этого сообщение пропускается через хеширующую функцию. Можно ввести некоторую абстракцию — понятие «**нефинализированного**» хеша. К примеру обозначим за `sha1_nf(x)` функцию, выдающую нефинализированный хеш (вообще говоря такой хеш будет существовать не всегда, только когда длина сообщения будет кратна 512 битам, иначе хеширование без паддинга просто не будет производится, но нам нужна только

абстракция). За $P(x)$ обозначим функцию паддинга. За $sha1(x)$ – обычную функцию $sha1$, выдающую привычные хеши. Тогда будет истинным вот такое нехитрое утверждение (знак + означает конкатенацию):

$$sha1(msg) = sha1_nf(msg + P(msg))$$

Что нам это даёт? Напомним, что у нас уже есть значение $sha1(msg)$, а как следствие, есть и нефинализированный хеш вида $sha1_nf(msg + P(msg))$, и при этом неизвестен сам msg . А из чего состоит msg ? Давайте запишем это выражение подробнее:

$$sha1(key + role) = sha1_nf(key + role + P(key + role))$$

Да, точно, $role$ в нашем случае имеет вполне конкретное значение:

$$sha1(key + 'user') = sha1_nf(key + 'user' + P(key + 'user'))$$

Когда мы имеем на руках некий нефинализированный хеш, нам ничто не мешает взять, и продолжить им что-то хешировать, таким образом, получив хеш от (в нашем случае) $key + 'user' + P(key + 'user') + something$. Трюк как видите в том, что новый хеш мы получили, не зная самого key . Значение $P(key + 'user')$ можно также вычислить не зная key – оно зависит только от длины сообщения — таким образом его легко будет подобрать.

Как мы помним, чтобы скрипт нас счёл богом, нам нужно иметь роль, оканчивающуюся на ' $gott$ ' и валидную подпись для неё. Зная всё вышеизложенное, мы теперь можем такую пару легко изготовить. Вот как она будет выглядеть (осторожно, формулы):

$$\text{new_role} = 'user' + \underline{P(key + 'user')} + 'gott'$$

$$\begin{aligned}\text{new_hash} &= sha1(key + 'user' + \underline{P(key + 'user')} + 'gott') = \\ &= sha1_nf(key + 'user' + \underline{P(key + 'user')} + \underline{'gott'} + \\ &\quad + \underline{P(key + 'user')} + \underline{P(key + 'user')} + 'gott'))\end{aligned}$$

Возможно, я описал процесс не очень разборчиво — но после некоторой практики, понять происходящее вполне реально. Достаточно понять, почему в `sha1()` мы подставляем именно такую строку, и откуда она берётся. И теперь от слов мы можем перейти к конкретным действиям.

Для начала, найдём [где-нибудь](#) файл `sha1.c` (есть в папке `src`), и добавим в него вот такой лёгкий костылик:

```
195
196 void SHA1Resume(
197     SHA1_CTX * context,
198     unsigned char digest[20],
199     uint32_t count_0,
200     uint32_t count_1
201 ){
202
203     context->state[0] = 0;
204     context->state[1] = 0;
205     context->state[2] = 0;
206     context->state[3] = 0;
207     context->state[4] = 0;
208
209     context->count[0] = count_0;
210     context->count[1] = count_1;
211
212     for (int i = 0; i < 20; i++) {
213         context->state[i >> 2] |= digest[i] << ((3 - (i & 3)) << 3);
214     }
215 }
216
```

Этот костылик позволит нам легко представлять «финализированные» хеши в виде «нефинализированных», точнее конвертировать их во внутреннее состояние контекста `SHA1_CTX`. А далее этот контекст можно использовать для продолжения хеширования чего-либо, реализуя таким образом нами описанное. Важно, что для восстановления состояния контекста из «финализированного» хеша необходимо знать длину всего сообщения в битах. Впрочем, в случае «финализированного» хеша, представленного как «нефинализированный» она будет всегда кратна 512, чему способствует добавление `padding`.

Использование полученной функции будет выглядеть, собственно, как-то так:

```
1 #include <bits/stdc++.h>
2
3 extern "C" {
4 #include "sha1.h"
5 }
6
7
8 int main(int argc, char** argv){
9
10     SHA1_CTX sutka;
11
12     uint8_t dingo[20] = {
13         0xd6,0x41,0x88,0xba,
14         0xe8,0x0b,0x41,0xf6,
15         0x43,0x22,0xd8,0xfd,
16         0x89,0xd3,0x47,0x27,
17         0x62,0x10,0x52,0xdc
18     };
19
20     SHA1Resume(&sutka, dingo, 64*8, 0);
21
22     /*for(int i = 0; i < 20; ++i){
23         printf("%02x", (int)dingo[i]);
24     }*/
25
26     uint8_t gott[] = "gott";
27
28     SHA1Update(&sutka, gott, 4);
29     SHA1Final(dingo, &sutka);
30
31     for(int i = 0; i < 20; ++i){
32         printf("%02x", (int)dingo[i]);
33     }
34
35     printf("\n");
36 }
```

Как видим, всё довольно просто и понятно. Восстанавливаем состояние `SHA1_CTX` из имеющегося хеша `d64188bae80b41f64322d8fd89d34727621052dc`, хешируем строку `'gott'`, финализируем. Таким образом получаем желанный `sha1(key + 'user' + P(key + 'user') + 'gott')`, то бишь значение `new_hash`.

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\FadeDemon>cd /d D:\all\works\cupctf\tasks\length_extension\src\cupctf_length_extension\Debug
D:\all\works\cupctf\tasks\length_extension\src\cupctf_length_extension\Debug>cupctf_length_extension.exe b47357224b81a091a81770611663755e687f2540
D:\all\works\cupctf\tasks\length_extension\src\cupctf_length_extension\Debug>
```

Отлично, новую подпись мы получили — осталось получить байтовое представление новой роли. Стоит заметить, что выглядеть оно будет неприятно — в паддинге содержатся null-байты...

```
new_role = 'user' + P(key + 'user') + 'gott'
```

Взглянем ещё раз, как будет выглядеть новая роль. Со строками всё понятно. Остаётся получить только значение $P(key + 'user')$. Чтобы это сделать, придётся понять, как строится padding в SHA1 — можно прочитать [по ссылке, которая уже была](#). Хотя, никакого особого таинства там нет:

Предварительная обработка:

Присоединяем бит '1' к сообщению

Присоединяем к битов '0', где k наименьшее число ≥ 0 такое, что длина получившегося сообщения

(в битах) [сравнима по модулю](#) 512 с 448 ($length \bmod 512 == 448$)

Добавляем длину исходного сообщения (до предварительной обработки) как целое 64-битное

[Big-endian](#) число, в [битах](#).

Здесь мы просто берем, и делаем что нас просят. Единственное — нам нужна длина выражения `key + 'user'` в битах — но её будет нетрудно подобрать, ~~если догадаться примерить сколько пикселей занимает строка со скрытым секретным кодом~~. Правильная длина будет равна $(32 + 4) * 8 = 288 = 0x120$. Таким образом, наш паддинг будет являться вот таким неприглядным массивом:

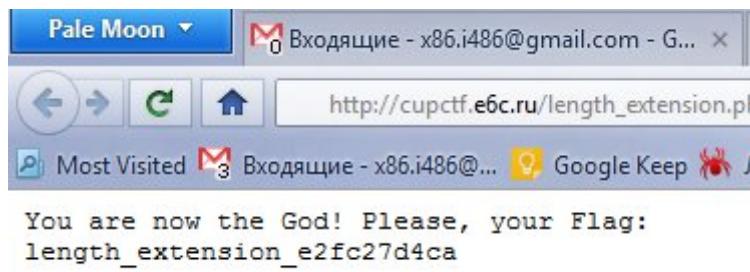
```
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00 00 00 00 01 20
```

Просто берём, и всё вместе подставляем в ссылку. Получаем:

```
http://tasks.ctfcup.ru/length_extension.php?role=user  
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%  
0%00%00%00%00%00%00%01%20gott&hash=b47357224b81a091a8177061166375  
5e687f2540
```

Поздравляю! Вы теперь бог. Правда, не все браузеры и программы

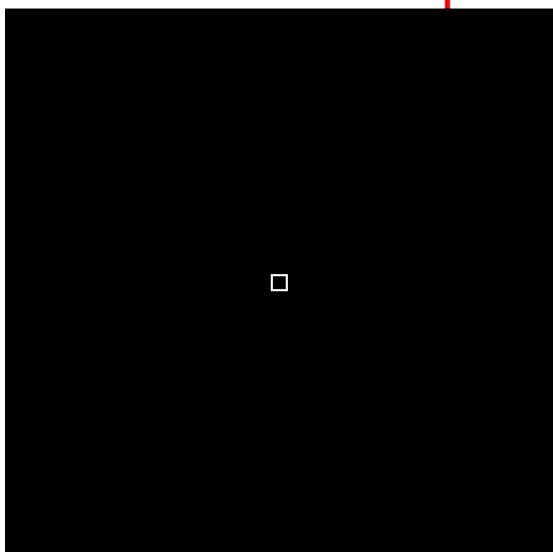
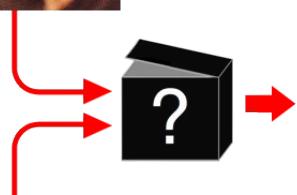
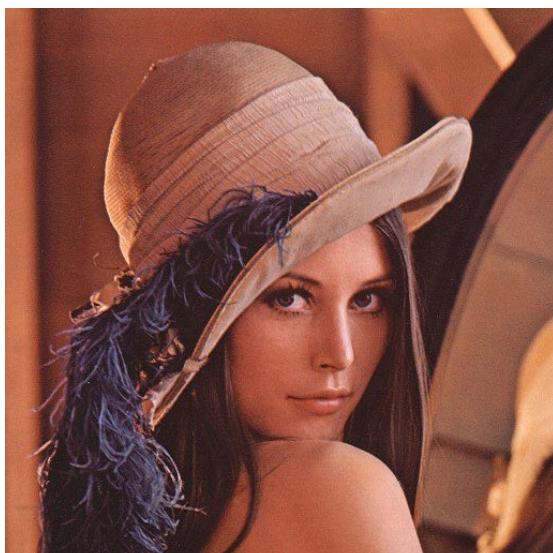
сразу переварят такую вот ссылку, но вот мой Pale Moon вполне справился:



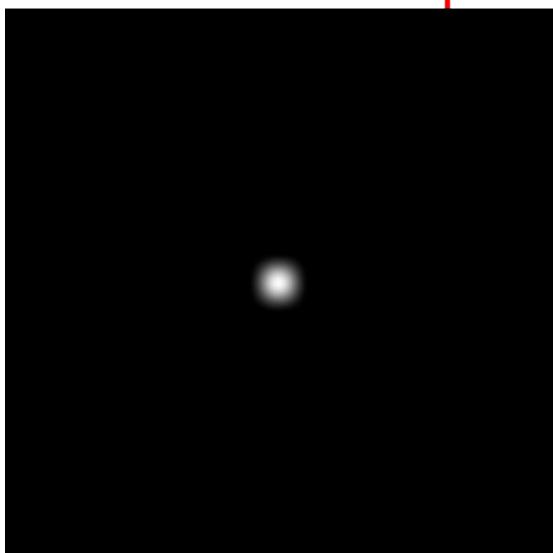
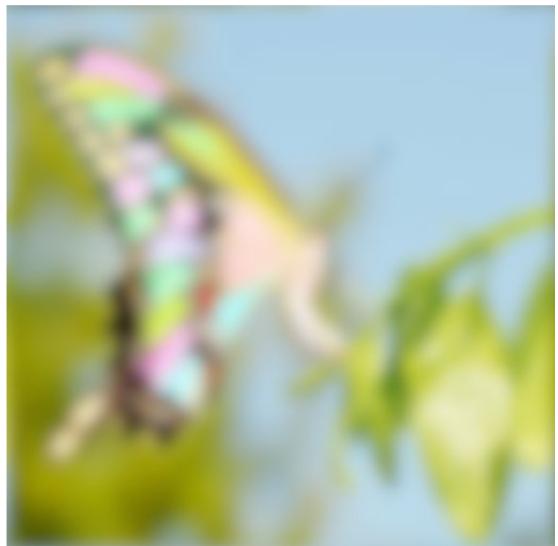
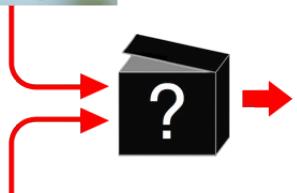
А какая в басне мораль? А мораль басни – не используйте SHA1 используйте HMAC. Ну и вообще, просто пишите адекватный код, и всё будет хорошо. Всем добра.

Lenna Comes Back

Алгоритмы цифровой обработки изображений всегда казались Доктору F. чем-то прекрасным, чарующим, с чем он готов был сутками напролёт ЭКСПЕРИМЕНТИРОВАТЬ. Одно время, если судить по найденным записям, его интересовали алгоритмы сжатия изображений с потерями — неоднократно он пытался изобрести свой. Не особенно успешно, правда. Неудачи, однако, его не сильно расстраивали, и в итоге он решил круто сменить направление исследований результаты которых вам сейчас придётся разгребать.

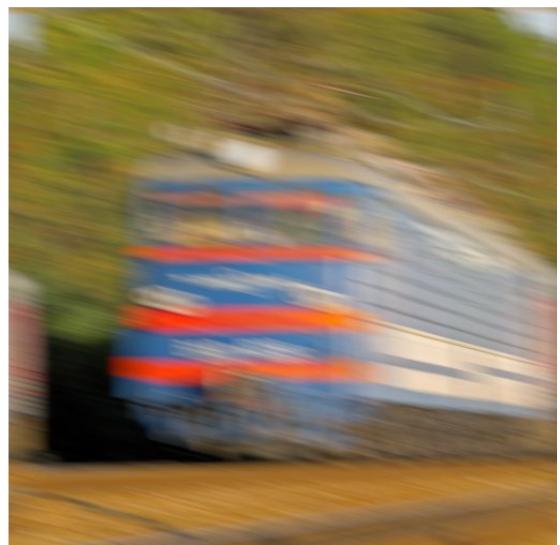
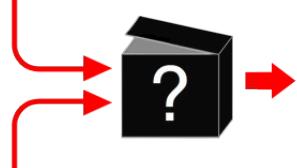
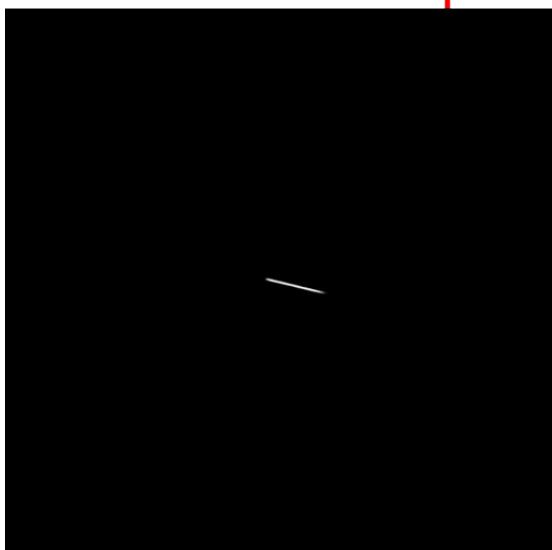


Найденная нами программа принимала на вход два изображения. Результатом работы тоже было изображение. Точнее, одно изображение, некоторым образом искаженное с помощью второго. Что-то это напоминает, правда? Особенно, вот такой пример:



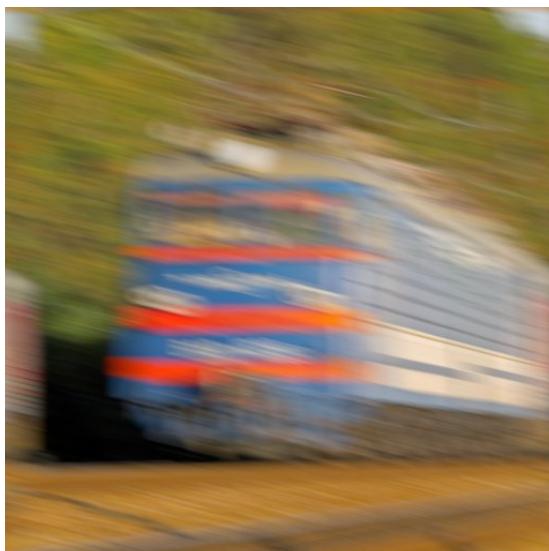
Искажающие изображения могли быть абсолютно произвольными — никаких правил на них не накладывалось. На некоторых, однако, программа начинала глючить, но это скорее были исключения.

Можно было даже воспроизвести что-то похожее на Motion Blur:



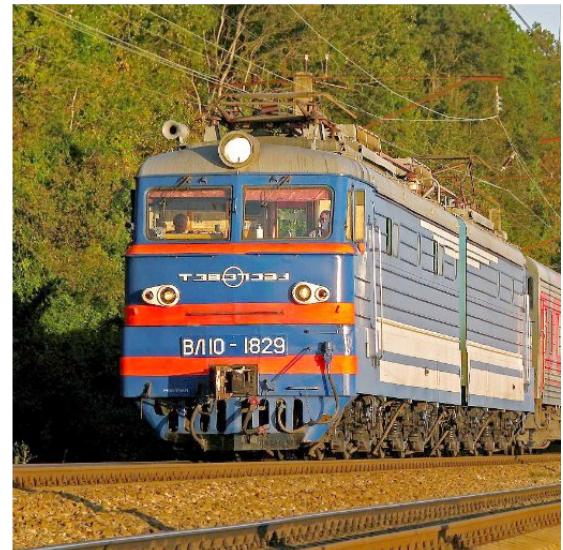
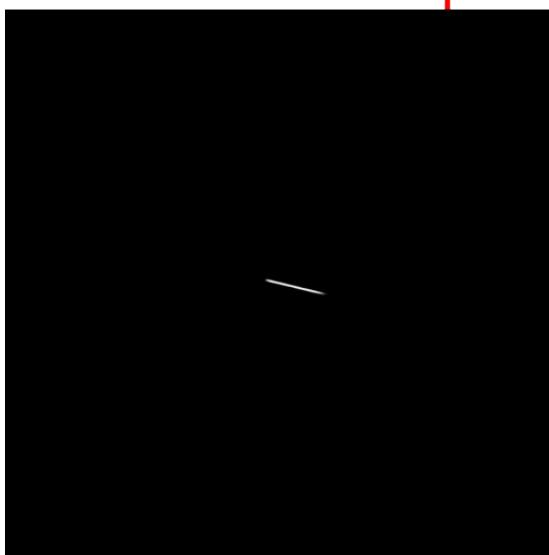
Одной, возможно, не сразу очевидной особенностью программы было то, что она могла проделывать показанное преобразование в **обратную сторону**. То есть, если на вход ей было подать любой результат преобразования, и функцию, в нём использованную, указав флагок «обратное преобразование», то она каким-то чудесным

образом восстанавливалась исходное изображение почти пиксель-в-пиксель. Наглядно говоря:



-reverse

?



* * *

Проблема всей этой ситуации заключается в том, что программа красноречиво описанная выше была безвозвратно утеряна. И всё бы ничего, если бы не одно важное НО.

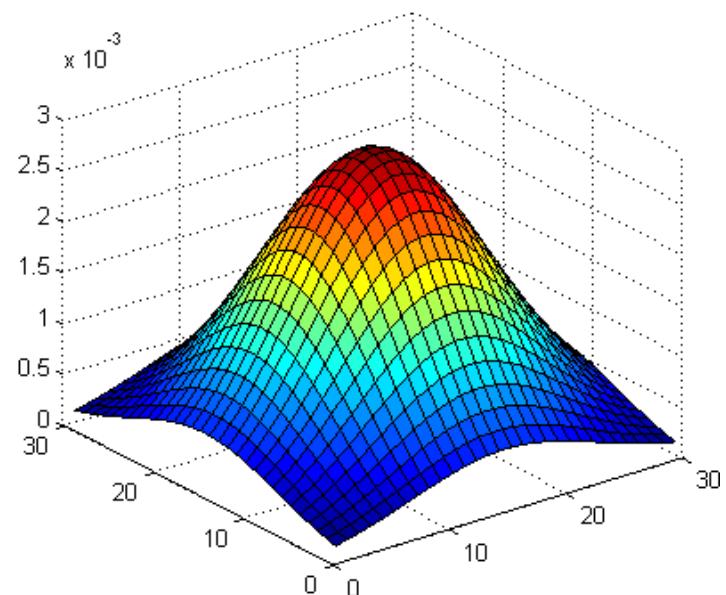
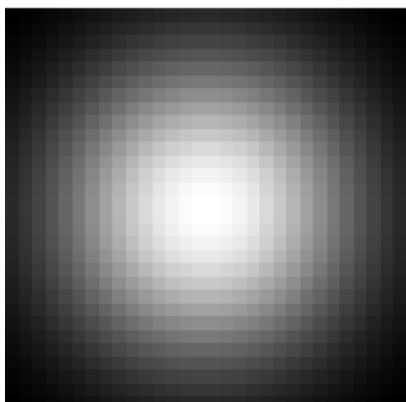
У нас в руках оказалась серия изображений, над которыми, видимо, производились эксперименты с участием злосчастной программы.

Всего изображений было 6, точнее, было 6 разных искажающих изображений, и одно исходное. Среди них не было ничего необычного. Необычным было 7-е изображение, к которому отсутствовал исходник — было только искажающее изображение, и результат искажения.

Искажение было настолько сильным, что ответить на то, что было изображено в оригиналe было абсолютно невозможно. Было похоже, что Доктор F. додумался использовать описанный алгоритм как некоторый прообраз *симметричного шифрования*, в котором искажающее изображение играло роль «ключа».

Все эти изображения [мы отдаём вам](#) — возможно детальный анализ 6-ти первых из них поможет понять, как возможно обратить описанный алгоритм, и узнать, что-же такое было на 7-й картинке. Будьте внимательны с форматом изображений — странный он какой-то, никогда раньше такого не видели.

Вашим флагом будет то, что вы найдёте на исходной 7-й картинке, когда её получите...



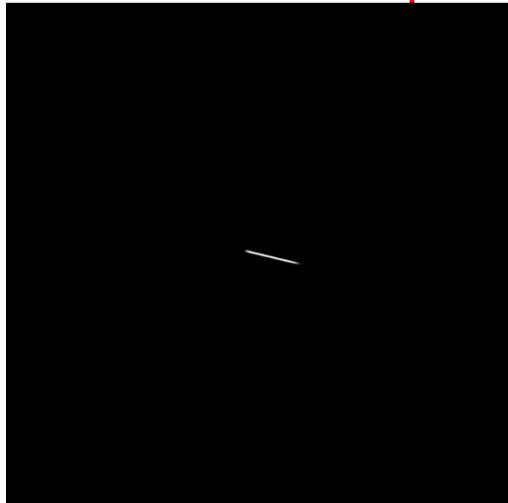
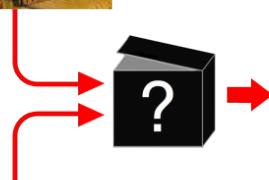
Флаг: lenna_flag_123375

"Мда, теперь надо срочно собраться с мыслями, и самому вспомнить, как вся эта магия, собственно говоря, работает..."

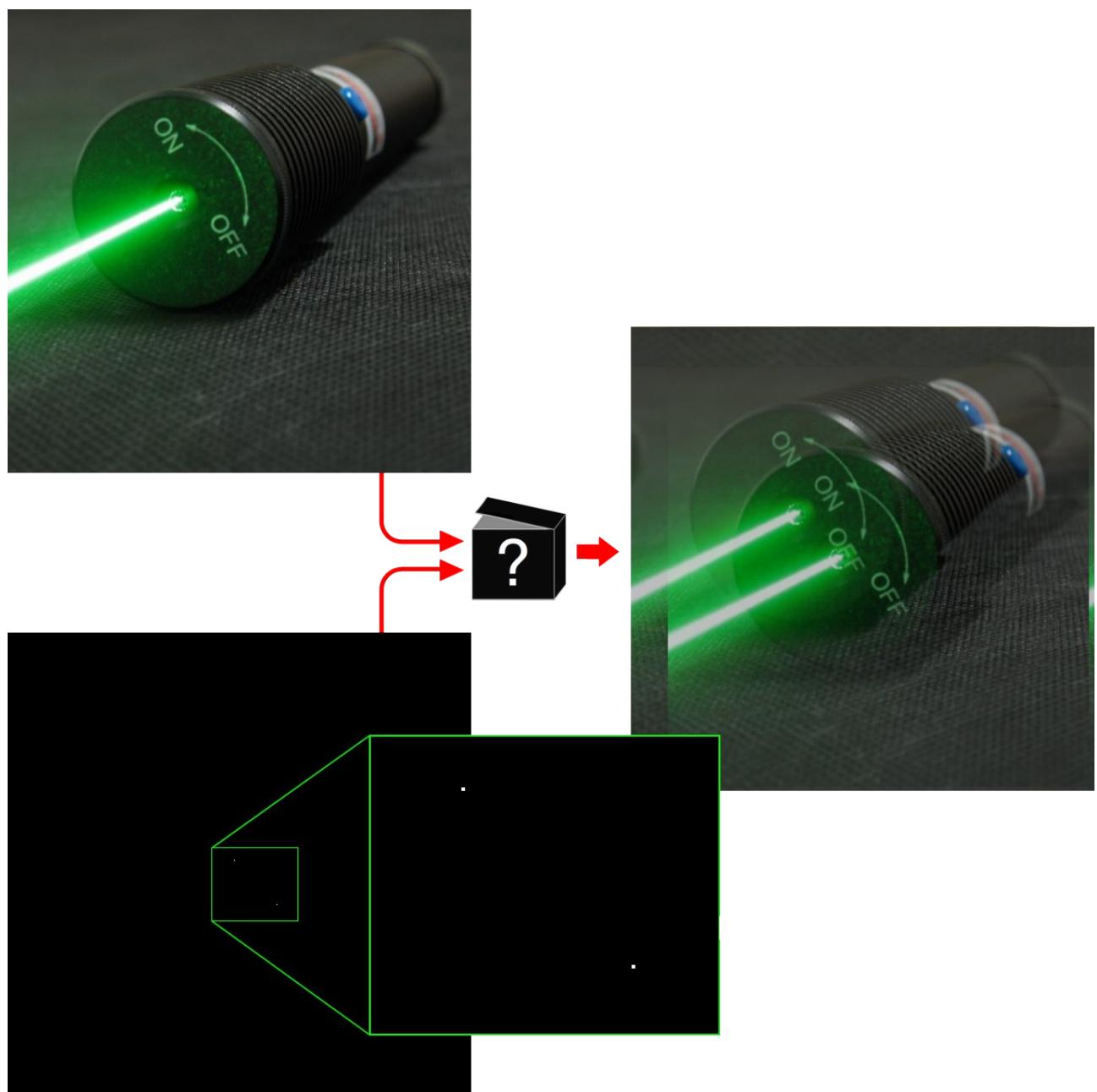
Dr. Fade

Итак, действительно, попробуем сообразить, что тут вообще происходит...

Начнем с того, что по описанному алгоритму нужно было понять, что нам пытаются на пальцах объяснить работу двумерной свёртки, на примере картинок и искажающих функций. Вообще говоря, прочтите [это](#) к примеру, если не в курсе, что это все такое.



Чёрным ящиком на этой диаграмме, собственно, и является свёртка (convolution), в котором участвует исходное изображение (сверху) и искажающая функция (снизу). На пальцах процесс свертки можно представить, как если бы на место каждого пикселя искажающей функции поместили бы исходную картинку (центром), без изменения размеров, интенсивность которой была бы умножена на значение этого пикселя, нормированного к виду $[0.0, 1.0]$. Пример, на котором это хорошо видно:



Обратите внимание на искажающую функцию — это просто два пикселя. Хорошо видно, как они влияют на исходное изображение —

в результате мы имеем как-бы среднее арифметическое двух изображений, немножко сдвинутых относительно центра. К слову, один бы пиксель, помещенный точно в центр никак бы не изменил исходное изображение.

Таким образом, наивное вычисление свертки — не такая уж и большая проблема, хотя и довольно вычислительно трудоёмкая. Можно прикинуть, что для квадратных изображений сложность будет в районе $O(n^4)$, где n — сторона изображения. Но нам, как известно, нужно нечто большее.

Даже понимая что такое свертка, и как она работает, с ходу с трудом можно представить, как её обращать. На ум приходит что-то вроде очень большой системы уравнений, которую, судя по всему, пришлось бы очень долго решать. Я даже не пытался разбираться, как это собственно делается — для этого есть более адекватный способ.

Как вы могли прочитать из статьи выше, существует так называемая «теорема о свёртке». Если эту теорему перевести на человеческий, то она утверждает, что **Фурье образ двумерной свертки двух картинок равен попиксельному произведению Фурье-образов этих картинок.** То есть, алгоритм такой:

1. Берём исходные картинки, делаем над ними двумерное БПФ, получаем двумерные массивы (или матрицы) комплексных чисел, содержащие Фурье-образ. Модули этих чисел — амплитуды, аргументы — фазовая составляющая (но нам это не особо важно).
2. Просто эти матрицы поэлементно перемножаем.
3. Для результата делаем обратное БПФ, из которого оставляем только действительную (*real*) часть комплексных чисел, *imaginary* выбрасываем.
4. Всё, свёртка готова. Действительная часть комплексных чисел будет содержать ни что иное, как свертку двух исходных картинок.

Тут во первых надо отметить две вещи.

1. Во первых, свертка получается циклическая, что можно заметить на картинках — сдвиги изображения там тоже происходят «циклически». Я не особо могу на пальцах объяснить разницу между циклической и линейной сверткой, прочитать можно [здесь](#), например.
2. Во вторых, всё это дело требует большой точности вычислений. Правда постойте, это небольшое забегание вперёд.

Стоит еще заметить, что сильно снижается трудоёмкость вычислений. Если свёртка в лоб требует $O(n^4)$ операций, то свертка через БПФ потребует что-то вроде:

$$O(4n^2 \cdot \log(n) \cdot \log(\log(n)) + n^2)$$

Или если покороче, то

$$O(n^2 \cdot \log(n) \cdot \log(\log(n)))$$

Что, конечно, значительно приятнее, чем $O(n^4)$, согласитесь... И это ещё в том случае, если вычислять двумерное БПФ через одномерное в лоб (по строкам и по столбцам). Если использовать [алгоритм Кули-Тьюки](#), то результаты будут лучше.

Однако, нам всё ещё надо получить обратную свёртку. Зная всё перечисленное, это теперь просто — на этапе поэлементного умножения Фурье-образов, их нужно попросту **не умножать, а делить**, как комплексные числа. В этом случае описанный алгоритм будет вычислять обратную свёртку. И вот тут да! Встаёт боком проблема точности, причем не просто боком — а неприступной стеной.

Если результат свертки сохранить как обычную картиночку в формате 8 бит/канал, или 24 бит/пиксель, то при попытке её обратить мы получим, что-то наподобие этого:



Но, я не совсем был прав, что картинка **абсолютно** необратима становится. Возможно, зависит всё от размеров искажающей функции. Очертания исходной картинки поезда в целом прослеживаются... Если добавить немного блюрца, то:



Поезд проступает чуть явственнее. Какими-то манипуляциями у меня получалось еще качество улучшить — но да речь сейчас не об этом.

Эта та причина, по которой изображения даны в формате TIFF с 32 бит на канал, да еще и float (96 бит/пиксель). Точности флоатов для нашей задачи, как оказалось, достаточно. Да и читать такую картинку легко — совсем не обязательно вообще парсить заголовок TIFF файла, а сами данные представлены в виде троек float, организованных в виде матрицы, если по-русски на C, то:

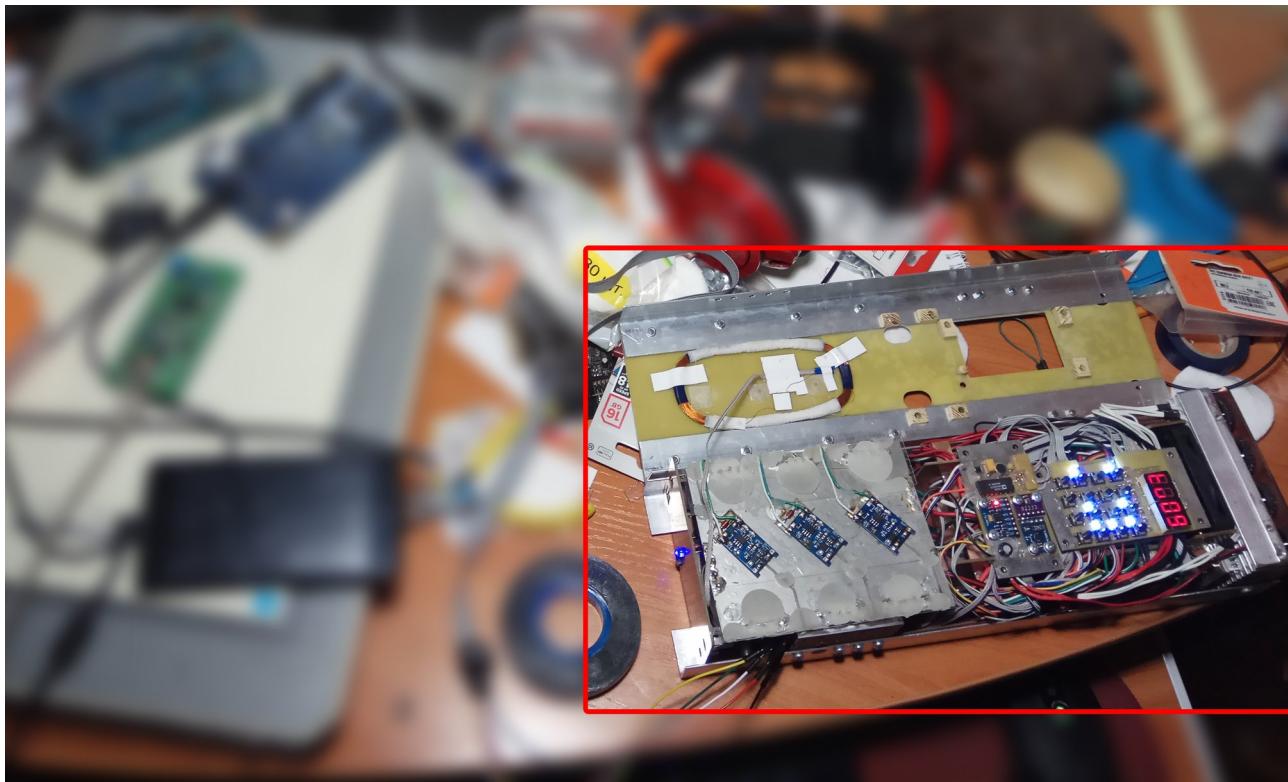
```
float data[WIDTH][HEIGHT][3];
```

Собственно, для всех этих манипуляций со свертками я написал свой код, который можно найти в папке `src`. Он, кстати, курсачом моим был, когда-то (правда сама свёртка в курсач не входила).

Есть мнение, что то-же самое можно проделать в каком-нибудь `matlab`, или чем-то подобном, если судить к примеру по тому что пишут на хабре. Но этот путь решения я честно не прорабатывал — все пишем всегда сами. Код на `C++` для БПФ можно кстати содрать [отсюда](#) (впрочем как и разобраться, как он работает).

Логические уровни

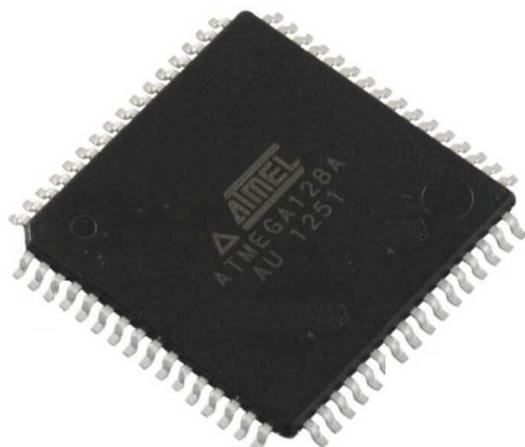
На столе, среди различного мусора, отладочных плат, телефонов и путаницы проводов, будучи, видимо, наполовину разобранным, лежало довольно странное устройство. Его не возможно было спутать с чем-либо, впрочем, как и трудно было понять его основную функцию. Впрочем, мы точно знали, зачем оно нам.



Да, чуть позже стало ясно, что это всего-лишь фонарик, хоть и обладающий весьма недюжинным световым потоком почти в 6000 люмен, и общей ёмкостью аккумуляторов почти 30 ампер/час.

Однако, внешний вид у него не даром изобилует таким количеством проводов — помимо фонарика у устройства были другие, весьма неожиданные функции.

Полученные нами архивные документы из далёкого 2016 года немного рассказали об этом устройстве. В частности, управлялось оно известным в те времена чипом **ATMEGA128A**.



Сведения были довольно скучные, однако мы всё-таки выяснили, что с этим чипом по какой-то общей шине была соединена микросхема **DS1307**.



В целом, конечно, можно сказать, что эта микросхема была предназначена для точного отсчёта временных промежутков, что позволяло её использовать как часы реального времени, но мало кто знает, что помимо этого, на борту этой микросхемы было целых 56 байт постоянной памяти, которая была доступна извне как для чтения, так и для записи. Документы это полностью подтверждают:

Table 2. Таблица памяти микросхемы

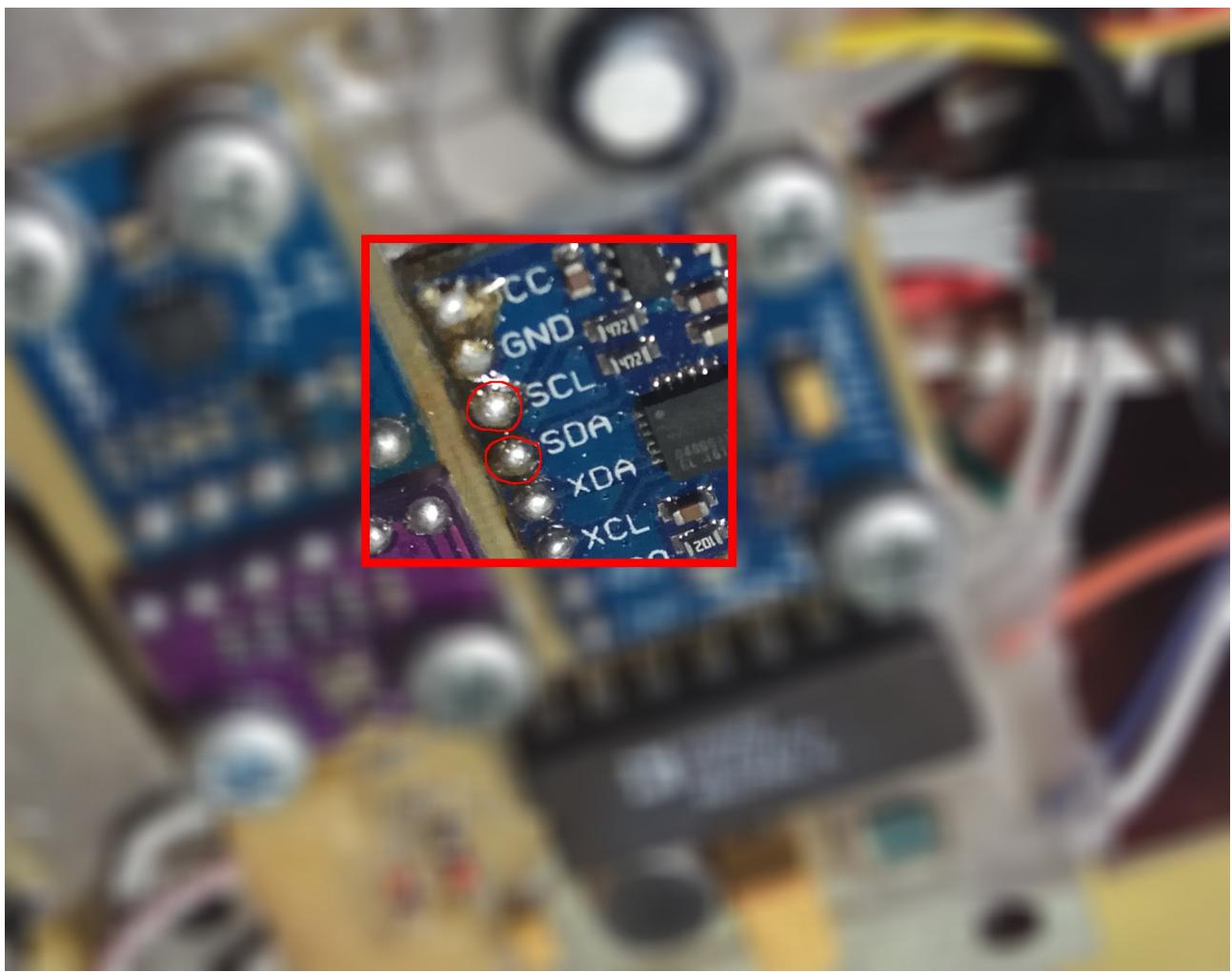
Адреса	Мод 1	Мод 2	Мод 3	Мод 4	Мод 5	Мод 6	Мод 7	Память	Функции
00h								RAM 56 x 8	00h-FFh
01h									
02h									
03h									
04h									
05h									
06h									
07h									
08h-3Fh								RAM 56 x 8	00h-FFh

Всё это мы рассказываем не просто так. Дело в том, что нас очень интересует содержимое этой памяти, конкретно этой микросхемы конкретно в этом чудо-устройстве. И да, не всё так просто, микросхемы-то самой нет, как и каких-либо дампов её памяти... Но, именно поэтому, нам ваша помощь и нужна.

С момента, когда устройство было нами обнаружено, до момента его безвозвратной утери, нам всё-таки кое-что удалось сделать.

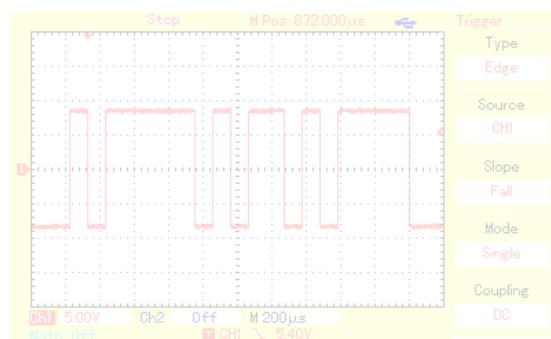
Мы подключили к устройству 2-х канальный осциллограф. Нам точно известно, что интересующая нас микросхема подключена к основному контроллеру через некоторую общую шину, и между ними производится коммуникация, в ходе которой, в частности, и происходит передача содержимого памяти. Нашине, похоже, правда есть ещё несколько устройств.

Осциллограф был подключен к контактам на фото ниже — мы думаем, они имеют отношение к упомянутой шине. Да, общий провод был также соединён с GND.



С помощью осциллографа была создана запись логических уровней на указанных контактах на протяжении всех режимов работы устройства. Точно известно, что содержимое памяти микросхемы должно было считываться за указанный промежуток времени, возможно, не один раз, и следовательно, как-то отобразиться на сделанной записи.

Эта запись — единственное, что мы вам можем предоставить. Ваш флаг вы найдёте в первых байтах содержимого памяти микросхемы, когда его получите. Желаем удачи.

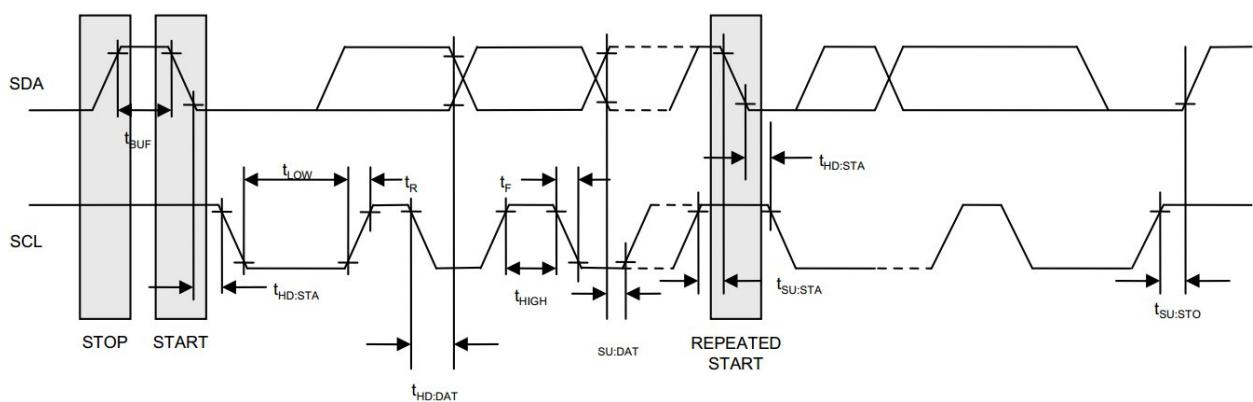


Флаг: i2c_flag_b3059388

О чудо, у меня кажется получилось сделать таск, для решения которого не нужно писать код!!! Хотя, это как посмотреть. В целом, можно и код написать, хоть в данном случае — это излишество. А теперь, в чём суть.

Ну, я думаю можно будет довольно быстро понять, что под шиной недвусмысленно имеется ввиду I²C. И по содержимому файла, который нам дают, не слишком сложно понять, в каком канале там SCL, а в каком SDA, особенно если вы знаете как работает I²C.

Как только вы прочтёте, как работает I²C, нужно приступить к лобовому декодированию записи. Ну серьезно, описаний протокола I²C более чем навалом, да и не такой он сложный, при том, что его описание можно найти прямо в даташите на DS1307:



На картинке ничего не понятно, но там рядом уже более адекватная информация имеется. И да, мне честно лень тут описывать протокол I²C, его правда легко [нагуглить](#) :)

START data transfer: A change in the state of the data line, from HIGH to LOW, while the clock is HIGH, defines a START condition.

Discussion board
Use the discussion board
web-based discussion

STOP data transfer: A change in the state of the data line, from LOW to HIGH, while the clock line is HIGH, defines the STOP condition.

Data valid: The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.

Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between START and STOP conditions is not limited, and is determined by the master device. The information is transferred byte-wise and each receiver acknowledges with a ninth bit. Within the I²C bus specifications a standard mode (100kHz clock rate) and a fast mode (400kHz clock rate) are defined. The DS1307 operates in the standard mode (100kHz) only.

Acknowledge: Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse which is associated with this acknowledge bit.

A device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is stable LOW during the HIGH period of the acknowledge related clock pulse. Of course, setup and hold times must be taken into account. A master must signal an end of data to the slave by not generating an acknowledge bit on the last byte that has been clocked out of the slave. In this case, the slave must leave the data line HIGH to enable the master to generate the STOP condition.

В целом, ваша задача сводится к тому, чтобы находить START-запросы на шине, и смотреть, по какому они адресу идут. А по какому адресу нужно — очевидно написано в уже упомянутом даташите на DS1307:

Figure 4. Data Write—Slave Receiver Mode

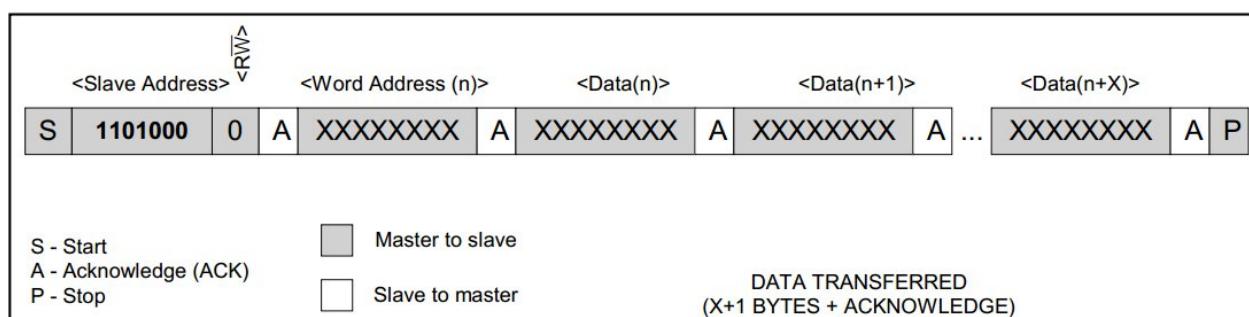
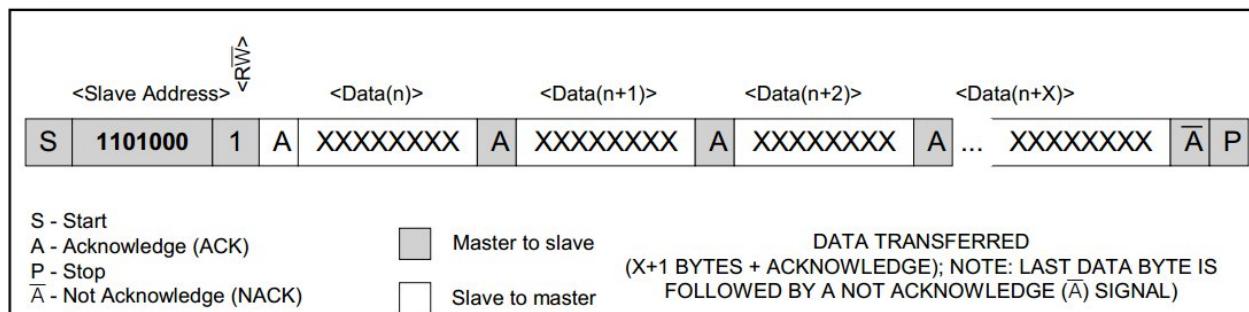


Figure 5. Data Read—Slave Transmitter Mode



Здесь нужно заметить, что вы можете запариться это делать без автоматизации — там дофига однотипного опроса устройств. Но... Если вы додумаетесь просто послушать запись (да, это же всего-лишь wav-файл в конце концов!), то заметите, что там действительно постоянно повторяется один и тот-же паттерн. А вот в двух местах записи, этот паттерн явно чем-то сменяется. Эти места начинаются примерно с секунд 42.700 и 86.400, и это и есть интересующие нас запросы. Вам остаётся их просто декодировать, и дело сделано.

К слову, в качестве «осциллографа» использовалась плата STM32VLDISCOVERY, и соответствующий проект под Coccox IDE есть в папке `src`. Пишет 2 канала АЦП с частотой 44100, и не просто так, а в реалтайме шлёт в UART (это 88200 байт или 705600 бит в секунду, между прочим) — можно использовать почти как стерео микрофон, если каскад входной сделать правильный. Там же можно найти приложение под винду, умеющее дампить вывод с UARTа этой штуки в WAV файл.

Mutated Picture

Из далёкого 2217 года, вам досталась картинка в формате PNG, размером около 4.67 МВ. Картинка, видимо, испорчена — не получается рассмотреть, что на ней изображено.

Нам кажется странным, что формат картинки PNG. Как известно, Доктор F. предпочитал экспериментировать с более простыми форматами, например BMP, аргументируя это тем, что с ними гораздо проще работать программно.

Есть предозрение, что картинка содержит нечто таинственное, наличие чего крайне трудно там предположить — пусть это не будет вас UNEXPECTED. От того, сможете ли вы это осознать, отбросив всё лишнее будет зависеть, сможете ли вы получить флаг. И помните — важен порядок строк, а иногда и столбцов. Не забывайте про это, #000262 как и про то, что **IMPORTANT** значение каждого byte, и пусть числа 262 и 346 помогут вам, хоть они и не про***.

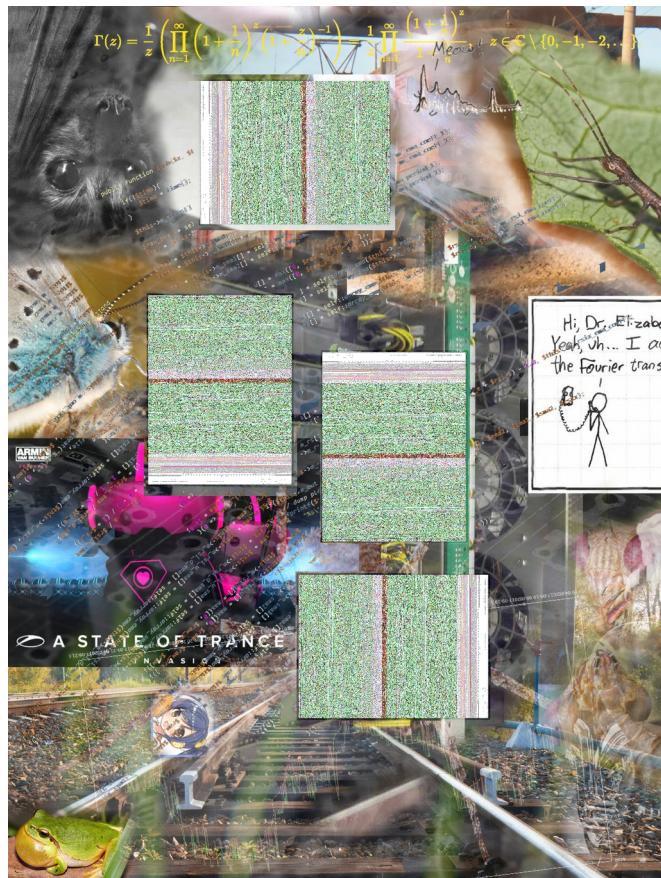


Флаг: mutated_04477a87baa66701

Надо сказать честно — таск извращенский, с элементами игры в догадайку, которые я даже сам ненавижу. По ходу решения, он из нечего похожего на стеганографию весьма плавно превратится в пионерский реверс-инжиниринг. Или не пионерский. Впрочем, давайте попорядку.

Для начала нам дают картинку, которая плохо выглядит. Собственно, первый элемент догадайки — надо заметить, что у нее испорчена ширина. То есть, поток пикселей, содержащийся в ней изначально предназначен для размера 1600 x 1200. Притом, поскольку порча накладывалась в BMP формате, снимать ее надо тоже в этом-же формате, либо насиловать себе мозги, как это можно сделать по другому. Впрочем, про формат BMP был дан намёк в условии.

После того, как с картинки снимем порчу, мы увидим нечто, отдающее лёгкой психodelией...

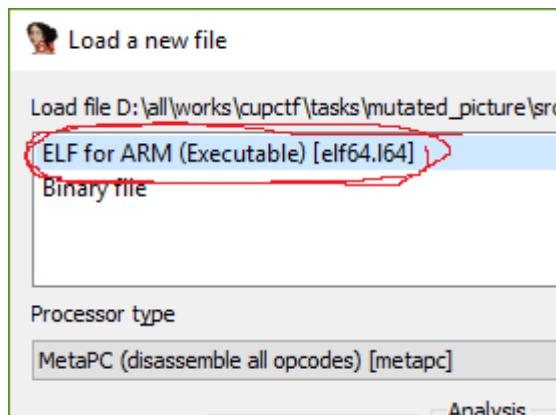


Но не отвлекаемся — похоже, картинка тут снова испорчена, вот этими кусками месива из пикселей. Вы быстро убедитесь, что они идентичны, и являются зеркальными отражениями/поворотами одной и той-же картинки размером 262x346. А ещё, у картинки есть альфа-канал, в котором явно видно какие-то данные на месте этих самых пикселей.

Если вы после всего этого сохраните картинку в формате BMP (возможно, TGA, и любом другом формате без сжатия, с порядком байт совпадающим с BMP), и поковыряете файл HEX-редактором, то найдёте... ВНЕЗАПНО, заголовок исполняемого ELF файла. Правда только заголовок. Или кусок заголовка. И найдёте его именно в том месте, где находится одно из месив пикселей. А если проявите фантазию, то найдете его во всех месивах, они же одинаковы, надо только повернуть правильно...

Дальше, нужно восстановить весь ELF файл. Ну, по моему достаточно очевидно, что надо сделать crop на одном из месив, и поэкспериментировать с отражением/поворотом. Только не забывать, что там есть альфа-канал, и что ни в коем случае никак нельзя менять цвета пикселей, а то всё пропадёт. Я для этого использовал Photoshop CS5, там всё корректно делается, и он умеет понимать альфа-канал в файлах TGA.

Дальше можно просто опять сохранить в BMP, и отрезать заголовок. И вуаля, мы получим исполняемый ELF. Но эльф не простой, а под архитектуру ARM. Неожиданно, правда?



Ладно, предположим. Думаю, запустить его на эмуляторе вы уж как-нибудь сможете. Но только когда запустите, увидите приблизительно это:

```
Remaining: 77123775  
Please, press return...  
  
Remaining: 77123774  
Please, press return...  
  
Remaining: 77123773  
Please, press return...  
  
Remaining: 77123772  
Please, press return...  
  
Remaining: 77123771  
Please, press return...
```

То есть, вас попросят 700 миллионов раз нажать Enter. И в целом, можно так и сделать — программа после этого честно выдаст флаг. Нет, серьезно, можно например создать файл на 800 мб, состоящий из '\n', и скормить его в `stdin` этому шедевру. Но мы, пожалуй, поговорим о менее брутальных способах.

Число 77123777, которое там показывается в самом начале, в hex будет выглядеть как 0x498d0c1. Тут надо сделать лирическое отступление, и вспомнить, что в ARM нельзя так просто взять, и сделать `mov eax, 0x498d0c1`, поскольку это RISC, и все команды там 4-х байтные, а 0x498d0c1, это и так четыре байта. А следствием этого является то, что константа эта начисто отсутствует в файле, и нельзя так просто взять, и заменить ее на ноль, например. Или, всё-таки можно? Давайте спросим IDA Pro:

```
.text:00008F44      MOV      R0, #0xF          -  
.text:00008F48      LDR      R8, =(aRemaining - 0x8F64)  
.text:00008F4C      MOV      R6, #0xD0C1 ↙  
.text:00008F50      LDR      R3, [R7,R3]  
.text:00008F54      MOUT    R6, #0x498 ↘  
.text:00008F58      LDR      R9, =(aPleasePressRet - 0x8F6C)  
.text:00008F5C      ADD      R8, PC, R8 ; "Remaining: "  
.text:00008F60      STR      R3, [R1,#-0x10]!  
.text:00008F64      ADD      R9, PC, R9 ; "Please, press return..."  
.text:00008F68      BLX      sub 323B4
```

Фрагмент до боли похож на то, что нам нужно. Но только вот ни 0xd0c1, ни 0x0490 нету в файлике, ни в Big endian, ни в Little endian. ARM загадочнее чем кажется:)

Но мы останавливаться, конечно-же, не будем. Пойдем, воспользуемся каким-нибудь онлайн ассемблером/дизассемблером ARM, например <http://armconverter.com/>

The screenshot shows a web interface for translating ARM assembly code between GDB/LLDB and HEX formats. At the top, there is a text input containing four assembly instructions:

```
MOV R6, #0xD0C1
MOVT R6, #0x498
MOV R6, #0x1
MOVT R6, #0x0
```

Below the input, a note reads: "For better results, convert only one instruction at a time. If there's an in output, modify it and ."

At the bottom, two output sections are shown:

ARM GDB/LLDB - Copy	ARM HEX - Copy
E30D60C1 E3406498 E3A06001 E3406000	C1600DE3 986440E3 0160A0E3 006040E3

Мы видим тут (результат справа), что формат команды действительно несколько размазан, поэтому мы не можем найти данных в формате с человеческим лицом. Но да мы уже сгенерили нужные команды, и знаем, что и где нужно на них заменить (можно было и в IDA в Hex view посмотреть).

```
shell@A2010-a:/ $ cd /data/local/tmp
shell@A2010-a:/data/local/tmp $ chmod 755 execu
shell@A2010-a:/data/local/tmp $ ./execu
Remaining: 1
Please, press return...

Easy, right?
Just go to jury, and submit the flag:
mutated_04477a87baa66701
shell@A2010-a:/data/local/tmp $
```

Ну а дальше, собственно, запускаем измененный файл на телефоне эмуляторе, и видим Это. Радуемся, бежим, сдаём флаг. Ах да, я кажется забыл упомянуть, что файл скомпилирован под Андроид, с флагами `-static -s -Os`. Ну а иначе он 4 мегабайта вешать начинал :)

Need Some Optimization

Давайте немного поговорим о медленном коде. Очевидно, что код можно сделать медленным как минимум двумя способами:

1. Использовать неподходящий алгоритм с плохой сложностью;
2. Пренебречь оптимизацией под целевую архитектуру.

К примеру, есть подозрения, что нижеследующий код пострадал как раз от первого случая:

```
#include <bits/stdc++.h>

uint64_t iter_count = 2017201820192020ull;

std::vector<uint8_t> flag_to_encode{/* FLAG IS HIDDEN */};

std::vector<uint8_t>
vector_iterate(const std::vector<uint8_t>& v){
    if(8 != v.size()) exit(1);

    uint8_t a = v[0] + v[2] + v[6];
    uint8_t b = v[1] + v[5] + v[7];

    uint8_t three = v[3];
    uint8_t seven = v[7];

    return {
        v[1],
        static_cast<uint8_t>(v[0] * (uint8_t) 3 + seven),
        a, b,
        static_cast<uint8_t>(v[2] * (uint8_t)5),
        static_cast<uint8_t>(v[4] * (uint8_t)5 + three),
        static_cast<uint8_t>(seven * 171 + three),
        v[6]
    };
}

void vector_print(const std::vector<uint8_t>& v){
    for(int i = 0; i < 8; ++i){
        std::cout << static_cast<int>(v[i]) << ' ';
    }
    std::cout << std::endl;
}
```

```
int main(int argc, char** argv){

    // Simple solution
    std::vector<uint8_t> w(flag_to_encode);

    for(uint64_t i(0); i < iter_count; ++i){
        w = vector_iterate(w);
        std::reverse(w.begin(), w.end());
    }

    vector_print(w);
    return EXIT_SUCCESS;
}
```

Проблема первого случая медленного кода заключается в том, что как правило, для того, чтобы его ускорить, код придётся написать новый, который будет принципиально от старого отличаться. Тогда как во втором случае, зачастую достаточно включить -O2 в настройках компилятора, или добавить парочку ассемблерных вставок в код, с командами по типу ADDSUBPD, или PCMPISTRM.

* * *

Каким бы медленным не был код приведённый выше, однако будучи однажды запущенным, в далёком 2217 он всё-таки завершил свою работу, и вывел в терминал вот такой набор чисел:

131 245 109 119 89 169 103 11

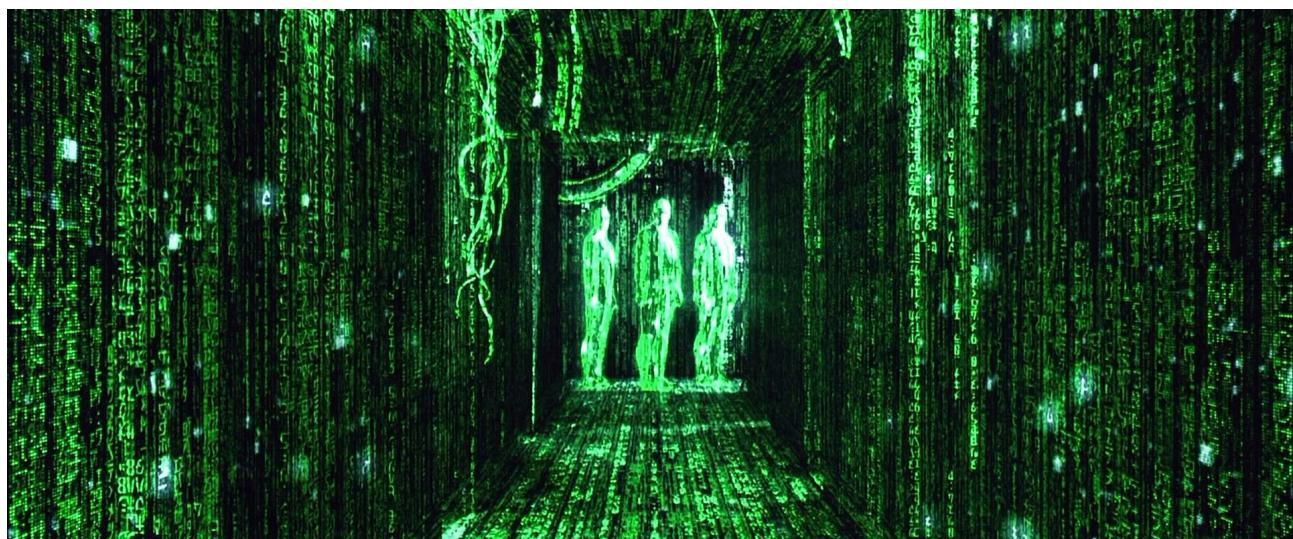
Возможно вы уже заметили, но код в данном документе приведён не полностью. В нём были намеренно скрыты 8 чисел-констант, которые являлись исходными данными, с которыми был запущен код.

flag_to_encode{/* FLAG IS HIDDEN */};



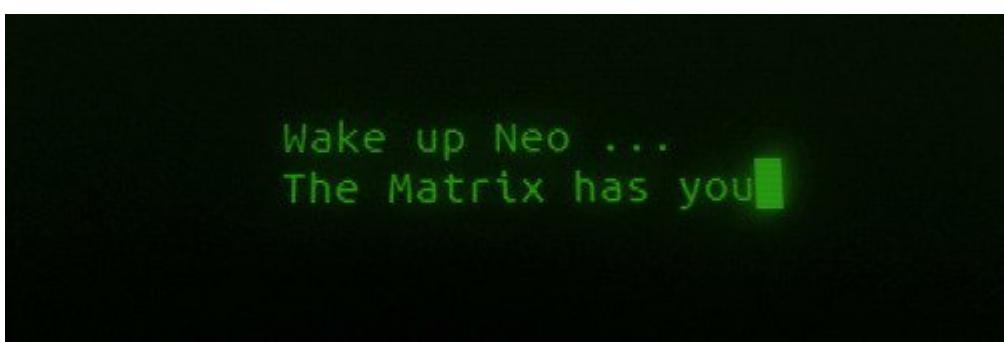
Возможно это не сразу очевидно, но зная, что программа вывела, эти 8 чисел можно **однозначно восстановить**. А раз это возможно, то почему-бы не попробовать?

Вашим флагом будут 8 исходных чисел, отсутствующих в коде, записанных подряд без пробелов в десятичной системе счисления. К примеру для чисел 10 20 30 40 50 60 70 80 флагом будет строка 1020304050607080. И поосторожнее — следите, чтобы матрица случайно вас не поглотила...



Флаг: 1113217117121191211

Итак, нам дана некоторая функция, которая что-то делает с вектором (массивом) чисел (преобразует его). Функция вызывается в цикле непомерное (невычислимое за приемлемое время) количество раз, каждый раз изменяя состояние вектора, полученного на предыдущей итерации. Нам даётся результат работы программы (сколько же авторы ждали времени, чтобы его получить?). А из этого результата... Правильно, надо восстановить исходное состояние вектора. Оно и будет флагом.



Буду краток. Идея таска нацелена на то, чтобы вспомнить, что такое матрицы, вспомнить, что они бывают целочисленные в кольцах по модулю, ну и что их можно возводить в степень. Да, надо заметить, что такими знаниями обычно обладают продвинутые олимпиадные программисты. Ну, или математики, криптографы. То есть, таск по идеи, реально сложный, хотя и заключается в банальном «возьми алгоритм, и примени его». Ну, в коде чуть-чуть надо разобраться еще.

Итак, по порядку. Мы видим, что везде используются тип `uint8_t`, что означает, что все вычисления производятся по модулю 256. Это первое важное наблюдение. Второе важное наблюдение — функцию `vector_iterate` можно заменить умножением вектора на некоторую матрицу. Возможно, это не так просто сразу заметить, однако, в пользу этого говорит то, что новое состояние вектора вычисляется из сложения некоторых элементов старого состояния, опционально умноженных на какую-то константу. То есть, ровно то, что и происходит при матричном умножении. Операция `std::reverse` также представляется матрицей.

Ну и здесь можно окончательно сорвать покровы, вспомнив про самое обычное бинарное возвведение в степень, которое да, отлично работает на матрицах. Мы просто представляем одну итерацию цикла матрицей, возводим ее в нужную степень... И эм, с помощью этой матрицы мы можем получить пока что из флага имеющийся у нас результат. Но нам-то надо кое-что другое, правда?

Собственно, хоть матрица у нас и в кольце по модулю, ее после некоторых телодвижений можно точно так-же инвертировать, как и обычную вещественную матрицу. Только операции деления нужно заменить нахождением обратного элемента по модулю. Здесь есть некоторые подвохи, к примеру, что четные числа по модулю 256 не инвертируются, поскольку не очень с ним взаимно просты. Но именно поэтому, в функции перехода четных чисел и нет, а остальное поправимо :)

В общем мы просто берем, нашу матрицу инвертируем, и умножаем на результат. И да черт возьми, магическим образом получаем исходный флаг.

Решение полностью продемонстрировано в файле `src/cdt/main.cpp`. Не используется никаких-либо внешних библиотек, для инвертирования используется способ с присоединенной матрицей, для нахождение детерминантов используется допиленный метод Гаусса, умеющий бороться с четными числами.

Запуск этого файла с параметром `decrypt` выведет флаг, что, собственно, вы можете и сами увидеть в исходном коде, как и конкретные матрицы, имитирующие наши преобразования. Их там, кстати, две — одна из них имитирует функцию, вторая — операцию `std::reverse`, то есть, переворот вектора. Это кстати просто отраженная единичная матрица. Нетрудно догадаться, что матрица, которую надо возводить в степень получается путем перемножения этих матриц.

На этом думаю, всё. Спасибо за внимание :)



Довольно странный EXE

Собственно говоря, у нас для вас имеется [довольно странный исполняемый файл](#) под Windows. Впрочем, вся странность его заключается в том, что он попросту не запускается.

Хотя вроде-бы должен...

Не факт, что файл как-то связан с Доктором Ф. и что он был получен из далёкого 2217 года, но похоже, что единственный способ для вас получить флаг — это понять, что с этим файлом не так, и какие секреты он в себе скрывает.

Флаг будет узнать просто — это, как всегда, строка, состоящая из маленьких латинских символов, цифр, и символа '_'.

Остаётся только пожелать удачи. ~~И да, будьте внимательны — есть мнение, что разработчик файла был пьян, и кушал грибы.~~



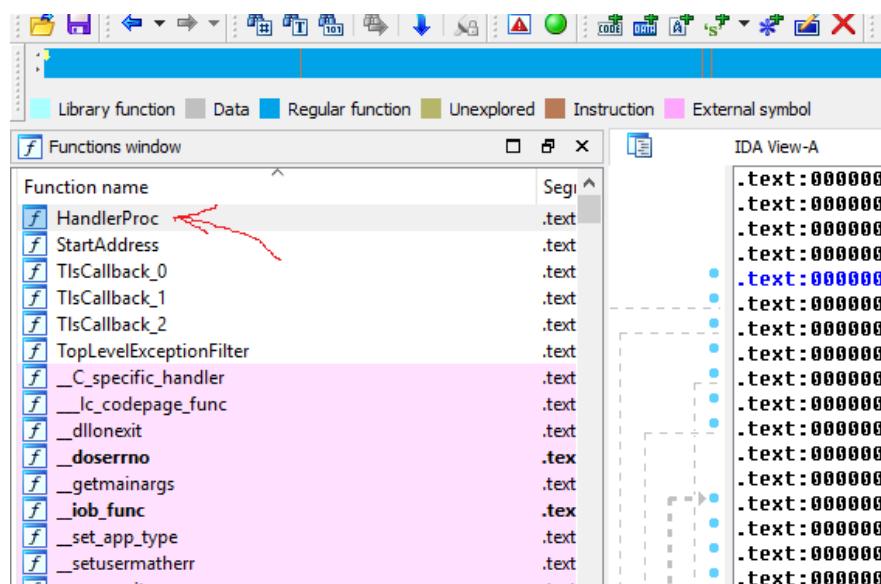
Флаг: just_windows_service_f3d05dfa2

Ну, тут всё довольно просто. Файл представляет собой службу для Windows, и все что нужно сделать — запустить его собственно, как сервис. Ну, почти. Если быть более точным, то как сервис его нужно прописать с параметром /SERVICE. А если запустить его с параметром /REGISTER, то в качестве бонуса он сам создаст нужный сервис (через API [CreateService](#)), правда путь к исполняемому файлу впишет кривой — нужно будет в реестре исправить вручную, не забыв про параметр. Ну, и там ещё подвох один есть, но не важно.

После того, как сервис мы запустим (к примеру как смертные юзеры, через services.msc), он создаст на диске C: файл, в который будет один раз в секунду записывать флаг. И всё, что вам остаётся, по идее — флаг из этого файла вытащить, правда всё-таки, в этом месте не стоит быть сильно наивными...

Давайте попробуем описать не претендующий на однозначность процесс решения в картинках.

Когда мы откроем файл в IDA, то увидим, что там не так много функций, имеющих название. Однако одна из них несколько выделяется — это `HandlerProc`. Название это трудно отнести к какой-то библиотечной функции.



Эта функция не вызывается в коде нигде напрямую. Однако, она всё-таки встречается в коде, вот в таком вот интересном месте:

```
000000000040182A    push    rsi
000000000040182B    push    rbx
000000000040182C    sub     rsp, 0A8h
0000000000401833    lea     rdx, HandlerProc ; lpHandlerProc
000000000040183A    lea     rcx, ServiceName ; "D"
0000000000401841    mov     cs:ServiceStatus.dwServiceType, 10h
0000000000401848    mov     cs:ServiceStatus.dwCurrentState, 2
0000000000401855    mov     cs:ServiceStatus.dwControlsAccepted, 5
000000000040185F    mov     cs:ServiceStatus.dwWin32ExitCode, 0
0000000000401869    mov     cs:ServiceStatus.dwServiceSpecificExitCode, 0
0000000000401873    mov     cs:ServiceStatus.dwCheckPoint, 0
000000000040187D    mov     cs:ServiceStatus.dwWaitHint, 0
0000000000401887    call    cs:RegisterServiceCtrlHandlerW
000000000040188D    lea     rdx, ServiceStatus ; lpServiceStatus
0000000000401894    mov     rcx, rax      ; hServiceStatus
0000000000401897    mov     cs:qword_4D6030, rax
000000000040189E    mov     cs:ServiceStatus.dwCurrentState, 4
00000000004018A8    call    cs:SetServiceStatus
```

Функция [RegisterServiceCtrlHandler](#), как-бы намекает, что этот кусок кода скорее всего является частью сервиса, точнее функции сервиса (*ServiceMain*). Если принять во внимание, что код выше начинается как функция [sub_401820](#), то этот факт подтверждается следующим:

```
0000000000401E34    lea     rdx, aService    ; "/SERVICE"
0000000000401E3B    mov     rcx, rbx      ; Str
0000000000401E3E    xor     ebx, ebx
0000000000401E40    call    strstr
0000000000401E45    test   rax, rax
0000000000401E48    jz    loc_401DC6
0000000000401E4E    lea     rax, ServiceName ; "D"
0000000000401E55    lea     rcx, [rsp+58h+Memory] ; lpServiceStartTable
0000000000401E5A    mov     [rsp+58h+Memory], rax
0000000000401E5F    lea     rax, sub_401820 ←
0000000000401E66    mov     [rsp+58h+var_30], rax
0000000000401E6B    call    cs:StartServiceCtrlDispatcherW
0000000000401E71    mov     eax, ebx
0000000000401E73    add     rsp, 48h
0000000000401E77    pop    rbx
0000000000401E78    pop    rsi
0000000000401E79    retn
```

Функция [StartServiceCtrlDispatcher](#) используется для того, чтобы указать, какой системный сервис должен выполняться в контексте текущего потока, и указать для него «точку входа». Точкой входа является наш [sub_401820](#). Более подробно обо всем этом можно почитать в MSDN, а в грубом приближении работает оно именно так.

Ну, собственно на этом месте, уже станет очевидно, что имеем мы дело с сервисом. Ещё можно глянуть на код выше чуть шире:

```
0000000000401DD9          mov    edx, 1           ; bInitialOwner
0000000000401DDE          call   cs>CreateMutexW
0000000000401DE4          lea    rdx, SubStr      ; "/REGISTER"
0000000000401DEB          mov    rcx, rbx       ; Str
0000000000401DEE          call   strstr
0000000000401DF3          test   rax, rax
0000000000401DF6          jz    short loc_401E34
0000000000401DF8          lea    rdx, aBuypassCheck ; "--buypass-check"
0000000000401DFF          mov    rcx, rbx       ; Str
0000000000401E02          call   strstr
0000000000401E07          test   rax, rax
0000000000401E0A          jz    short loc_401E80
0000000000401E0C          lea    rsi, [rsp+58h+Memory]
0000000000401E11          mov    rcx, rsi
0000000000401E14          call   sub_401BD0
0000000000401E19          mov    rcx, rsi
0000000000401E1C          call   sub_401710
0000000000401E21          mov    rcx, [rsp+58h+Memory] ; Memory
0000000000401E26          add    rsi, 10h
0000000000401E2A          cmp    rcx, rsi
0000000000401E2D          jz    short loc_401E34
0000000000401E2F          call   j_free
0000000000401E34          ; CODE XREF: sub_401D80+76↑j
0000000000401E34          ; sub_401D80+AD↑j
0000000000401E34          lea    rdx, aService     ; "/SERVICE"
0000000000401E3B          mov    rcx, rbx       ; Str
0000000000401E3E          xor    ebx, ebx
0000000000401E40          call   strstr
0000000000401E45          test   rax, rax
0000000000401E48          jz    loc_401DC6
0000000000401E4E          lea    rax, ServiceName ; "D"
0000000000401E55          lea    rcx, [rsp+58h+Memory] ; lpServiceStartTable
0000000000401E5A          mov    [rsp+58h+Memory], rax
0000000000401E5F          lea    rax, sub_401820
0000000000401E66          mov    [rsp+58h+var_30], rax
0000000000401E6B          call   cs>StartServiceCtrlDispatcherW
0000000000401E71          mov    eax, ebx
0000000000401E73          add    rsp, 48h
0000000000401E77          pop    rbx
0000000000401E78          pop    rsi
0000000000401E79          retn
```

И увидеть странные строки по типу /REGISTER, /SERVICE, и --buypass-check. Я не буду расписывать всю логику, просто отмечу, что в исходном коде это выглядело так (а как связано то что здесь с исходным кодом, попробуйте сами проследить):

```

int CALLBACK WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR     lpCmdLine,
    int       nCmdShow
){
    HANDLE hmtx = OpenMutexW(MUTEX_ALL_ACCESS, TRUE, MUTEX_NAME);

    if(NULL != hmtx){
        MessageBoxW(0, L"System is busy", L"Error", MB_ICONERROR);
        return 1;
    }

    CreateMutexW(NULL, TRUE, MUTEX_NAME);

    if(strstr(lpCmdLine, "/REGISTER")){
        if(!strstr(lpCmdLine, "--buypass-check")){
            MessageBoxW(NULL,
                L"Registration disabled in your country due to legal reasons", L"Error",
                MB_ICONWARNING);
            return 2;
        }
        SVCCreateService(SVCGetApplicationExeFilename());
    }

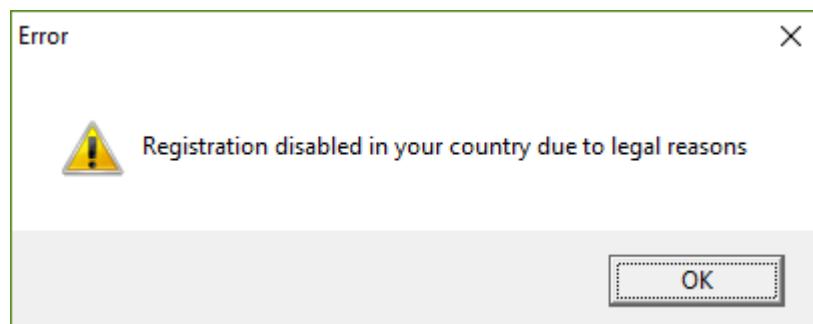
    if(strstr(lpCmdLine, "/SERVICE")){
        SERVICE_TABLE_ENTRYW ServiceTable[1];
        ServiceTable[0].lpServiceName = const_cast<LPWSTR>(SERVICE_NAME);
        ServiceTable[0].lpServiceProc = (LPSERVICE_MAIN_FUNCTIONW)ServiceMain;

        StartServiceCtrlDispatcherW(ServiceTable);
    }

    return 0;
}

```

И действительно, если запустить exe с параметром /REGISTER, оно попытается вас немножко потроллить:



Для того, чтобы оно этого не делало, его нужно запустить с двумя параметрами: /REGISTER и --buypass-check. Тогда приложение честно создаст сервис с именем DrFadeSVC:

```

#define SERVICE_NAME L"DrFadeSVC"

void SVCCreateService(const std::wstring& exe){
    SC_HANDLE schSCManager, schService;

    schSCManager = OpenSCManagerW(
        NULL, // local machine
        NULL, // SERVICES_ACTIVE_DATABASE database is opened by default
        SC_MANAGER_ALL_ACCESS); // full access rights

    // Create/install service...
    // If the function succeeds, the return value is a handle to the service.

    schService = CreateServiceW(
        schSCManager, // SCManager database
        SERVICE_NAME, // name of service
        SERVICE_NAME, // service name to display
        SERVICE_ALL_ACCESS, // desired access
        SERVICE_WIN32_OWN_PROCESS, // service type
        SERVICE_AUTO_START, // start type
        SERVICE_ERROR_NORMAL, // error control type
        exe.c_str(), // service's binary
        NULL, // no load ordering group
        NULL, // no tag identifier
        NULL, // no dependencies, for real telnet there are dependencies lor
        NULL, // LocalSystem account
        NULL
    ); // no password

    CloseServiceHandle(schService);
}

```

Надо ещё обратить внимание, что IDA коряво распознаёт строки в Unicode, хотя все вызываемые функции тут имеют буковку *W* на конце, что означает использование Unicode версии функции. Распознаёт она их настолько коряво, что на месте, где должна хранится строка DrFadeSVC, мы видим это:

```

.rdata:00000000004AB06A ; const WCHAR ServiceName
.rdata:00000000004AB06A ServiceName     db 'D',0           ; DATA XREF: sub_401710+1B↑o
.rdata:00000000004AB06A                               ; sub_401820+1A↑o ...
.rdata:00000000004AB06C                               dd offset loc_460070+2
.rdata:00000000004AB070 aAdesvc:
.rdata:00000000004AB070                               unicode 0, <adeSVC>,0
.rdata:00000000004AB07E ; const WCHAR FileName
.rdata:00000000004AB07E FileName      dw 5Ch           ; DATA XREF: sub_401820+B2↑o
.rdata:00000000004AB080 a_CCon:

```

Опустим язвительные комментарии, и просто откроем HEX View. Там слава богу, всё пока что нормально:

```

000000000004AB05C 75 6C 6C 20 6E 6F 74 20 76 61 6C 69 64 00 44 00 ull-not-valid.D.
000000000004AB06C 72 00 46 00 61 00 64 00 65 00 53 00 56 00 43 00 r.F.a.d.e.S.U.C.
000000000004AB07C 00 00 5C 00 5C 00 2E 00 5C 00 43 00 3A 00 5C 00 ..\.\...\.\C.:.\.
000000000004AB08C 63 00 6F 00 6E 00 00 00 00 00 00 54 68 69 73 c.o.n.....This
000000000004AB09C 5F 69 73 5F 6E 6F 74 5F 61 5F 66 6C 61 67 5F 62 _is_not_a_flag_b
000000000004AB0AC 30 62 64 64 37 33 32 61 31 62 35 00 62 61 73 69 0bdd732a1b5.basi
000000000004AB0BC 63 5F 73 74 72 69 6E 67 3A 3A 61 70 70 65 6E 64 c_string::append
000000000004AB0CC 00 0A 00 00 47 00 6C 00 6F 00 62 00 61 00 6C 00 ....G.l.o.b.a.l.

```

Собственно, мы разобрались с тем, как наш недосервис запустить. Теперь давайте вернёмся в код `ServiceMain`, и разберёмся, что собственно наш сервис делает.

```

000000000004018AE xor    r9d, r9d      ; lpSecurityAttributes
000000000004018B1 xor    r8d, r8d      ; dwShareMode
000000000004018B4 mov    [rsp+0E8h+hTemplateFile], 0 ; hTemplateFile
000000000004018BD mov    [rsp+0E8h+dwFlagsAndAttributes], 4000106h ; dwFlagsAndAttributes
000000000004018C5 mov    [rsp+0E8h+dwCreationDisposition], 2 ; dwCreationDisposition
000000000004018CD mov    edx, 0C0000000h ; dwDesiredAccess
000000000004018D2 lea    rcx, FileName ; lpFileName
000000000004018D9 call   cs>CreateFileW
000000000004018DF cmp    rax, 0xFFFFFFFFFFFFFFFh
000000000004018E3 mov    [rsp+0E8h+hFile], rax
000000000004018E8 jz    loc_401A60
000000000004018EE cmp    cs:ServiceStatus.dwCurrentState, 4
000000000004018F5 jnz   loc_401A60
000000000004018FB lea    rdi, [rsp+0E8h+lpBuffer]
00000000000401903 mov    r14, cs:qword_4AB1F0
0000000000040190A mov    r15, cs:qword_4AB1F8
00000000000401911 lea    rsi, [rsp+0E8h+Memory]
00000000000401916 lea    rbp, [rdi+10h]
0000000000040191A nop    word ptr [rax+rax+00h]
00000000000401920
00000000000401920 loc_401920: ; CODE XREF: sub_401820+23A4j
00000000000401920 lea    rax, [rsi+10h]
00000000000401924 lea    r8, aThis_is_not_a_+1Fh ; ""
00000000000401928 lea    rdx, aThis_is_not_a_ ; "This_is_not_a_flag_b0bdd732a1b5"
00000000000401932 mov    rcx, rsi
00000000000401935 mov    [rsp+0E8h+Memory], rax
0000000000040193A call   sub_401680
0000000000040193F mov    ecx, 20h
00000000000401944 call   sub_4A6440

```

И он действительно создаёт какой-то файл. Пока даже непонятно какой.

```

000000000004019D5 lea    rdx, asc_4AB0CD ; "\n"
000000000004019DC mov    r8d, 1
000000000004019E2 mov    rcx, rdi
000000000004019E5 call   sub_4947C0
000000000004019EA mov    [rsp+0E8h+NumberOFBytesWritten], 0
000000000004019F2 mov    qword ptr [rsp+0E8h+dwCreationDisposition], 0 ; lpOverlapped
000000000004019FB lea    r9, [rsp+0E8h+NumberOFBytesWritten] ; lpNumberOfBytesWritten
00000000000401A00 mov    r8d, [rsp+0E8h+nNumberOfBytesToWrite] ; nNumberOfBytesToWrite
00000000000401A08 mov    rdx, [rsp+0E8h+lpBuffer] ; lpBuffer
00000000000401A10 mov    rcx, [rsp+0E8h+hFile] ; hFile
00000000000401A15 call   cs:WriteFile
00000000000401A18 mov    ecx, 3E8h      ; dwMilliseconds
00000000000401A20 call   cs:Sleep

```

А потом он даже в него что-то записывает. Да ещё и отсыпается 0x3E8 1000 миллисекунд.

Вы можете сами поизучать, что происходит между созданием файла и записью, а я для простоты просто покажу исходник:

```
HANDLE hfile = CreateFileW(
    L"\\\\.\\C:\\con", GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_SYSTEM |
    FILE_ATTRIBUTE_TEMPORARY | FILE_FLAG_DELETE_ON_CLOSE, NULL);

if(INVALID_HANDLE_VALUE != hfile){

    while(true){

        if(serviceStatus.dwCurrentState != SERVICE_RUNNING){
            break;
        }

        std::string not_a_flag("This_is_not_a_flag_b0bdd732a1b5");
        std::vector<int8_t> flag_mask{
            0x3e, 0x1d, 0x1a, 0x07, 0x00, 0x1e, 0x1a, 0x31,
            0x0a, 0x00, 0x03, 0x2c, 0x3e, 0x2c, 0x03, 0x1e,
            0x17, 0x0e, 0x3c, 0x07, 0x6f, 0x04, 0x57, 0x00,
            0x07, 0x06, 0x56, 0x07, 0x55, 0x03, 0x07, 0x00
        };

        std::string the_flag(not_a_flag);

        for(int i = 0; i < 32; ++i){
            the_flag[i] = transform(not_a_flag[i], flag_mask[i]);
        }

        the_flag += "\n";

        DWORD wro(0);

        WriteFile(hfile, the_flag.c_str(), the_flag.size(), &wro, NULL);

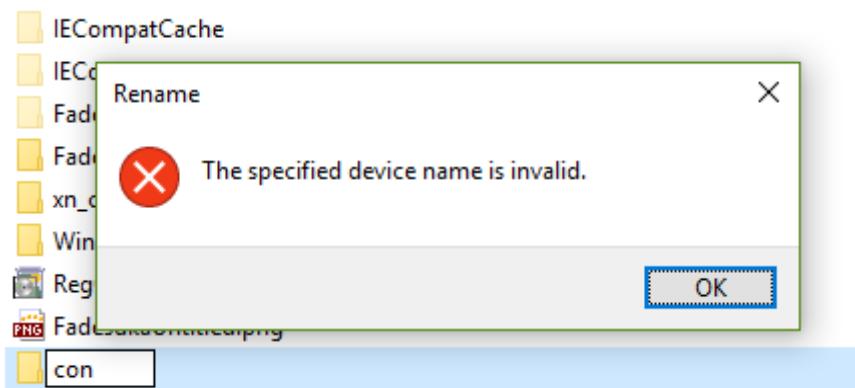
        Sleep(1000);
    }
}
```

И здесь нас ожидает настоящий **Triple Kill!!!**

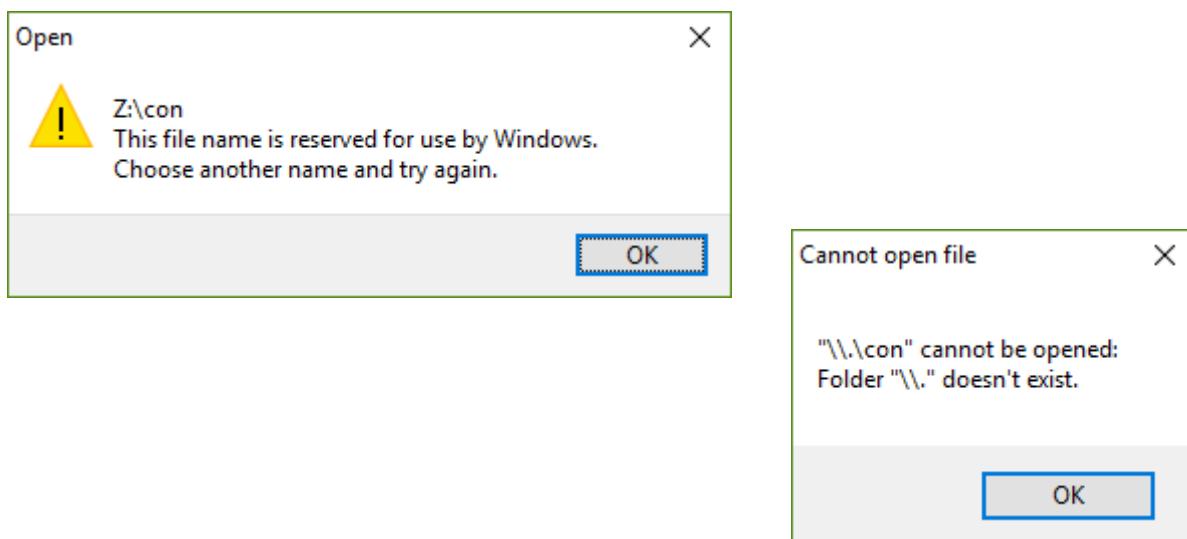


Начать можно с того, что строка `This_is_not_a_flag_b0bdd732a1b5`, действительно флагом не является. То, что на самом деле записывается в файл, получается из неё путем некоторой «трансформации» (спойлер:), и в явном виде в файле не встречается. Что, уже думаете, что её можно сразу легко достать из файла?

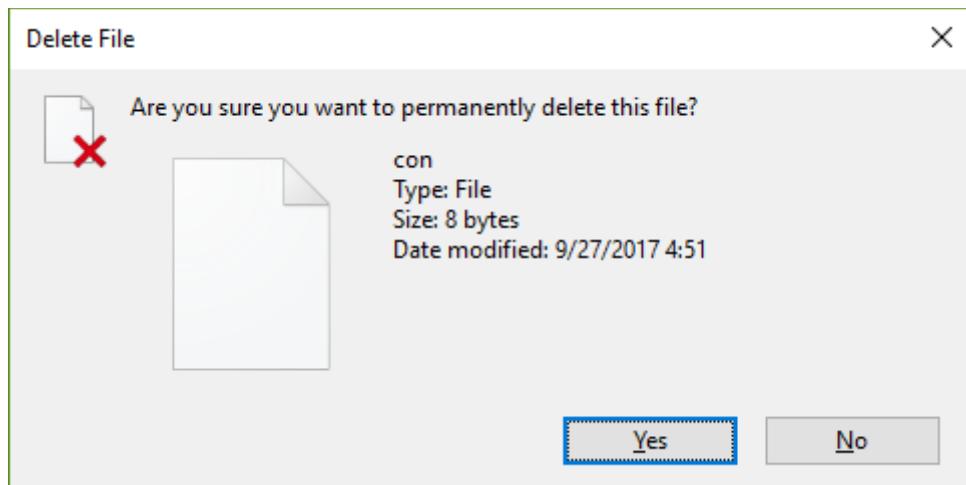
Вот скажите, вы когда-нибудь пытались создать файл или папку с именем `con` в Windows? Или файл `prn`? Если не пытались, то попробуйте. Вы будете приятно удивлены:



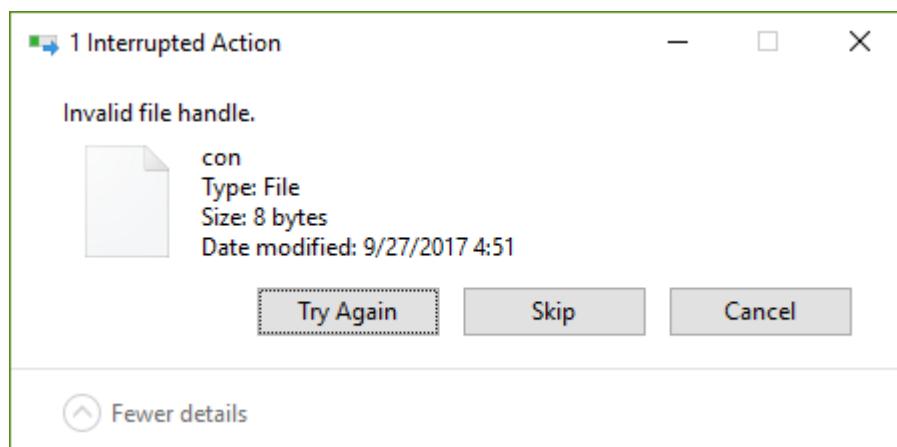
Если попытаться открыть такой файл уже существующий, большинство приложений вас попросту пошлёт:



Более того, вы даже удалить такой файл не сможете:



Жмакаем Yes, и...



Сия особенность Windows может вызвать немало лулзов и [интересных теорий](#), однако проблема здесь в том, что имена **CON**, **PRN**, **AUX** и некоторые другие являются именами зарезервированных устройств, тянувшихся, похоже, еще от DOS. А если копнуть чуть глубже, то оказывается, что для нормальной работы с такими файлами достаточно использовать [UNC](#)-префикс \\.\ и все нормально создаётся и удаляется:

```
C:\Users\FadeDemon>rm Z:\con
rm: cannot remove `Z:\\\\con': Permission denied

C:\Users\FadeDemon>cat \\\\Z:\\con
This is con

C:\Users\FadeDemon>rm \\\\Z:\\con

C:\Users\FadeDemon>cat \\\\Z:\\con
cat: \\\\Z:\\con: No such file or directory
```

А ещё, CON в винде, это низкосортный аналог /dev/stdout и /dev/stdin линукса в одном лице:

```
C:\Users\FadeDemon>cat con
hello con
hello con
some test bytes
some test bytes

C:\Users\FadeDemon>echo some bullshit > con
some bullshit
```

Но это маловажно. Теперь вы знаете, что с такими файлами делать. Однако, можно было просто пропатчить exe, заменив con, скажем, на can, или на ctf, и проблема была бы решена. Правда только на половину.

Давайте чуть внимательнее приглядимся, как вызывается [CreateFile](#).

```
HANDLE hfile = CreateFileW(
    L"\\\\.\\\\C:\\\\con", GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_SYSTEM | FILE_ATTRIBUTE_TEMPORARY | FILE_FLAG_DELETE_ON_CLOSE, NULL);
```

И как она должна вызываться:

C++

```
HANDLE WINAPI CreateFile(
    _In_     LPCTSTR             lpFileName,
    _In_     DWORD               dwDesiredAccess,
    _In_     DWORD               dwShareMode,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_     DWORD               dwCreationDisposition,
    _In_     DWORD               dwFlagsAndAttributes,
    _In_opt_ HANDLE              hTemplateFile
);
```

Здесь кроются сразу два подвоха. Во первых, можно обратить внимание, что dwShareMode = 0, и это означает ровно то, что любой доступ к такому файлу во время работы нашего приложения будет

запрещён. Полностью. И на чтение тоже. Пока служба работает, вы не сможете его открыть. А если вы её остановите, то обнаружите второй подвох: наличие FILE_FLAG_DELETE_ON_CLOSE заставляет файл удаляться сразу после закрытия. Неприятная неожиданность, не так-ли? Файл-то есть, а лучше от этого не становится...

Давайте думать, что тут можно сделать. Точнее, я перечислю, что мы можем тут сделать в порядке от простого к сложному.

1. Во время работы сервиса немного подождать, и отрубить питание компьютера/виртуальной машины. Практика показывает, что в этом случае файл с FILE_FLAG_DELETE_ON_CLOSE не удаляется, и после перезагрузки вы всё-таки сможете его прочитать.
2. Использовать что-то вроде [Unlocker](#). Правда нет гарантии, что он сможет отвязать файл от создавшего его процесса без побочного эффекта FILE_FLAG_DELETE_ON_CLOSE, но будет хотя-бы nice try.
3. Пропатчить exe, убрав неугодные нам флаги и/или share mode. И где этот пункт раньше то был?!
4. Вариант для ценителей: мы можем своровать хэндл файла у сервиса с помощью [DuplicateHandle](#), и спокойно всё оттуда прочитать. Как узнать хэндл открытого файла — немножко [другая история](#), поскольку в винде нету аналога lsof. Если посмотреться [повнимательнее](#), то можно заметить, что MS эти полезные возможности нагло скрывают.

Ну, вот собственно говоря, и финал — все что нужно для решения этого простого таска мы теперь знаем. С чем должен вас поздравить, и откланяться.

Трансцендентность

Доктор F., как известно, в свободное, да и не очень свободное время очень интересовался теорией чисел, да и числами в принципе. Так интересовался, что долгие вечера в бункере зачастую проводил над разнообразными расчетами, и вычислительными экспериментами. Нельзя его винить за такой образ жизни — мало чем ещё можно заняться, находясь на глубине 120 метров под землёй, когда снаружи свирепствует холодная, снежная ядерная зима... Нельзя даже дверь наружную открыть — на ней метров 10 снега навалило.

Некоторые его записи и исследования сохранились, и даже попали в наши руки. Среди них есть интересные, и... Довольно странные работы. Сейчас мы и познакомимся с одной из них.

В тот вечер Доктор F. размышлял над такой концепцией. Все мы знаем такие математические константы, как число Пи, и основание натурального логарифма — число e . Думаю, всем известно, что выглядят они как-то вот так:

$$\pi = 3.1415926535897932384626433832795\dots$$

$$e = 2.7182818284590452353602874713527\dots$$

Конечно, мы знаем, что десятичная запись этих чисел бесконечна. Числа можно получить с помощью разнообразных формул, и они всплывают во многих местах математического анализа. Однако, не это интересовало Доктора F.

* * *

Представим некую абстрактную машину, у которой есть целочисленные регистры: счётчик C , и сумма S , изначально равные нулю. Пусть эти регистры для простоты могут вмещать бесконечно большие числа.

Машина делает следующее:

1. Берет из десятичного разложения числа Пи, и числа e по одной цифре, находящейся по индексу C , и перемножает их.
2. Результат умножения прибавляет к сумме S
3. Увеличивает счётчик C на единицу.
4. Переходит к пункту 1.

Собственно, мы предположим, что таким образом машина работает бесконечно долго (числа-то ведь бесконечные, верно?)

У машины есть дисплей, достаточный, чтобы вместить число с 1000 знаков после запятой. Показания на этом дисплее обновляются после каждой итерации работы машины, описанной выше.

Число, которое отображается на дисплее, вычисляется следующим образом: от отношения S / C вычитается $\frac{1}{4}$, результат делится на 10, после чего из него извлекается квадратный корень.

* * *

Нелепо, правда? Так вот. Доктора F. интересовало это самое число, которое постепенно проявлялось на дисплее. То есть, он считал, что постепенно, по мере работы машины, это число проявлялось всё точнее, и точнее, говоря более математическим языком — вычисления сходились к какой-то одной константе.

Собственно, он хотел выяснить, к какой. Неизвестно, сколько тысяч лет он там вычислял, и сколько мощностей использовал, но в итоге, число это он выяснил. Он вообще любил подобные эксперименты до конца доводить, любой ценой...

Возможно вы уже догадались, но нам нужно будет повторить подвиг Доктора F., и тоже получить это число, которое должно, по идее, отобразиться на дисплее, когда пройдет, эм «бесконечное» количество лет.

Флаг можно будет получить, взяв 20 цифр из десятичного разложения этого числа, начиная с позиции 1000. Чтобы было проще — умножьте число на 10^{1000} , и первые 20 цифр после десятичной точки и будут вашим флагом. Удачи.

5 : 7 : 5 : 5 :

Флаг: 08969463386289156288

Задача эта, хоть и звучит несколько странно, но сама по себе не отличается большой сложностью, всего-лишь нехитро комбинируя теорию чисел, и теорию вероятности. Давайте разберёмся.

Во первых, при чём тут вообще числа π и e ? Суть в том, что есть мнение, что они «[нормальны](#)», а если по простому — то все цифры встречаются в них с одинаковой вероятностью, ровно как и любые последовательности цифр. То есть, вместо π и e с таким же успехом могут быть корни из 2 и 5, например, или вообще просто качественный ГПСЧ.

А дальше, думаю, не интересно, все сводится к скучной теории вероятности. Сначала посчитаем математическое ожидание произведения очередных двух цифр. Это несложно сделать скриптом, например. Оно будет равно 20.25, здесь и понятно, зачем $\frac{1}{4}$... Постойте, так ответ — корень из двух?!

И таки да, до этого можно догадаться и без тервера — просто повычислять пару часиков. Да, вычисление среднего значения произведений хоть и МЕДЛЕННО, но сходится к его матожиданию в 20.25. Отняв $\frac{1}{4}$, поделив и на 10 мы будем получать 2 со все большей точностью. Кстати ровно этот процесс схождения как раз и описывается [Законом больших чисел](#).

Собственно, исчерпывающее решение, выдающее флаг приведено в `src/solution.py`

Passwords Generator

Подслушивать диалоги иногда полезно. Особенно, если диалоги из будущего. Или прошлого?

- Итак. Путешествием в прошлое это, конечно, не назвать, но замедлиться мы должны были почти в 1800 раз. Да, что там с паролем на архив?
- Сгенерировал. Буквально минуту назад. Той самой программой, кстати, про которую ты говорил.
- Я вообще-то тебе говорил не использовать ту программу для генерации паролей. Идиот.
- Почему? Она точно безопасна. Я же сам её написал!
- Я же говорил тебе **больше не писать программы!** Нет, ты точно идиот. От тебя и твоих программ одни проблемы.
- Но...
- Лучше скажи, сколько там сейчас времени по твоим контрольным часам.
- Tue, 05 Jan 2021 17:53:19 GMT
- Ага! Эксперимент похоже удался. Как тебе возможность растянуть 1 секунду на 30 минут, а?

Обойдёмся без лишних комментариев — ваш флаг находится в этом архиве...



Флаг: unsafe_random_86643b02

Устраивайтесь поудобнее — в этот раз снова придётся немного пореверсить. Идея таска — фундаментальная небезопасность использования `std::srand(time(NULL))` для генерации паролей. Этую небезопасность нам сейчас предстоит использовать вполне конкретным образом — восстановить пароль, зная временную метку. Я, однако, схитрю, и, наверное, снова буду поглядывать при объяснении в исходный код.

Обратим для начала внимание, что программа явно собрана в **RAD Studio XE**, что в некотором роде роднит её внутренности с Delphi.

```
Embarcadero RAD Studio - Copyright 2015 Embarcadero Technologies
, Inc.....@.Ф.Ф.Ф.Ф.Н!@.....д)@.х.п.к.п....."Но.Но."Jo.0Jn.
€Kn.TLn.иMn..%o...<Jo.$%o.Ф о.....Н...
.....$...
.....яяяя....Ш%Ф.ш"Ф. `Bo.dBo.и&Ф...
.....С....Golden Graphite...яяяя....,*@.,,Ф...
```

IDA не особенно хорошо справляется с анализом программы — как результат улучшить разбираться времени у меня не было. Однако, после некоторого копания, в строках можно найти это:

The screenshot shows two panes of IDA Pro. The left pane displays assembly code with several entries highlighted in blue, such as `ButtonGenerateClick`. The right pane shows a memory dump with the same blue-highlighted entries. Red arrows point from the assembly code to the corresponding memory dump entries, specifically highlighting the string "Golden Graphite".

Assembly Address	Assembly Instruction	Memory Dump Address	Memory Dump Content
006EC5FE	ButtonGenerateClick	006EC5FE	Golden Graphite
00671858	ButtonGroup6	00671858	Golden Graphite
00669304	ButtonItem6	00669304	Golden Graphite

Да, `sub_402E00` ни что иное, как функция `FormCreate`, т.е то, что исполняется при создании формы. Это можно проверить:

```
.text:00402E00          = dword ptr -8
.text:00402E00 var_8      = dword ptr -4
.text:00402E00 var_4      = dword ptr -4
.text:00402E00
.text:00402E00 push    ebp
.text:00402E01 mov     ebp, esp
.text:00402E03 add     esp, 0FFFFFFF8h
.text:00402E06 mov     [ebp+var_8], edx
.text:00402E09 mov     [ebp+var_4], eax
.text:00402E09 push    0
.text:00402E0E call    sub_6E63E0
.text:00402E13 pop     ecx
.text:00402E14 push    eax
.text:00402E15 call    sub_6E1F44
.text:00402E1A pop     ecx
.text:00402E1B pop     ecx
.text:00402E1C pop     ecx
.text:00402E1D pop     ecx
.text:00402E1E ret
.text:00402E1E sub_402E00 endp
text:00402E1E
```

Если попытаться чуть-чуть восстановить алгоритм, очень уж напрашивается конструкция `sub_6E1F44(sub_6E63E0(0))`. А если бегло посмотреть на то, что эти функции делают...

```
.text:006E63E0
.text:006E63EA lea     eax, [ebp+SystemTime]
.text:006E63ED push   eax ; lpSystemTime
.text:006E63EE call   GetLocalTime
.text:006E63F3 push   0FFFFFFFh
.text:006E63F5 movzx edx, [ebp+SystemTime.wSecond]
.text:006E63F9 push   edx
.text:006E63FA movzx ecx, [ebp+SystemTime.wMinute]
.text:006E63FE push   ecx
.text:006E63FF movzx ecx, [ebp+SystemTime.wHour]
.text:006E6403 push   ecx
.text:006E6404 movzx eax, [ebp+SystemTime.wDay]
.text:006E6408 dec    eax
.text:006E6409 push   eax
.text:006E640A movzx edx, [ebp+SystemTime.wMonth]
.text:006E640E dec    edx
.text:006E640F push   edx
.text:006E6410 movzx ecx, [ebp+SystemTime.wYear]
.text:006E6414 add    ecx, 0FFFFF894h
.text:006E641A push   ecx
.text:006E641B call   sub_6E6048
.text:006E6420 add    esp, 1Ch
.text:006E6423 test   ebx, ebx
.text:006E6425 jz    short loc_6E6429
.text:006E6427 mov    [ebx], eax
.text:006E6429 loc_6E6429:           Из года вычитается 1900
.pop   ebx
.text:006E6429 pop    esp, ebp
.text:006E642A mov    esp, ebp
.text:006E642C pop    ebp
.text:006E642D ret
.text:006E642D sub_6E63E0 endp
```

; CODE XREF: sub_6E63E0+45↑j

З три раза встречается константа 0x3c = 60

Это для `sub_6E63E0`. Для `sub_6E1F44`:

```
.text:006E1F44          proc near               ; CODE XREF: sub_402E00+15↑p
.text:006E1F44 sub_6E1F44      arg_0           = dword ptr 8
.text:006E1F44
.text:006E1F44     push    ebp
.text:006E1F45     mov     ebp, esp
.text:006E1F47     call    sub_6E5B94
.text:006E1F4C     mov     edx, [ebp+arg_0]
.text:006E1F4F     mov     [eax+48h], edx
.text:006E1F52     call    sub_6E5B94
.text:006E1F57     xor     ecx, ecx
.text:006E1F59     mov     [eax+4Ch], ecx
.text:006E1F5C     call    sub_6E1F70
.text:006E1F61     pop    ebp
.text:006E1F62     retn
.text:006E1F62 sub_6E1F44      endp
.text:006E1F62
.text:006E1F62 ; -----
.text:006E1F63     align 4
.text:006E1F64
.text:006E1F64 loc_6E1F64:      push    1
.text:006E1F64     call    sub_6E1F44
.text:006E1F66     pop    ecx
.text:006E1F6C     retn
.text:006E1F6C ; -----
.text:006E1F6D     align 10h
.text:006E1F70
.text:006E1F70 ; ===== S U B R O U T I N E =====
.text:006E1F70
.text:006E1F70 sub_6E1F70      proc near               ; CODE XREF: sub_402E20+71↑p
;text:006E1F70          ; sub_6E1F44+18↑p
.text:006E1F70     push    ebx
.text:006E1F71     call    sub_6E5B94
.text:006E1F76     imul   ebx, [eax+48h], 15A4E35h
.text:006E1F7D     inc    ebx
.text:006E1F7E     call    sub_6E5B94
.text:006E1F83     mov    [eax+48h], ebx
.text:006E1F86     call    sub_6E5B94
.text:006E1F8B     mov    eax, [eax+48h]
.text:006E1F8E     shr    eax, 10h
.text:006E1F91     and    eax, 7FFFh
.text:006E1F96     pop    ebx
.text:006E1F97     retn
.text:006E1F97 sub_6E1F70      endp
.text:006E1F97
```

Мало что понятно, но попытки погуглить число `0x15A4E35` = `22695477` приводят вот к такому интересному результату:

[Линейный конгруэнтный метод — Википедия](https://ru.wikipedia.org/wiki/Линейный_конгруэнтный_метод)

https://ru.wikipedia.org/w/index.php?title=Линейный_конгруэнтный_метод&oldid=1013904223

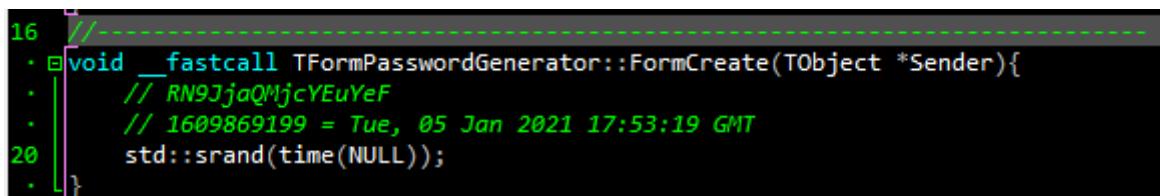
Линейный конгруэнтный метод — один из методов генерации псевдослучайных чисел.

Numerical Recipes, 2³², 1664525, 1013904223. Borland C/C++, 2³², 22695477, 1, bits 30..16 in rand(), 30..0 in lrand(). glibc (used by GCC), 2³¹ ...

[Описание](#) · [Свойства](#) · [Часто используемые](#) ... · [Возможность](#) ...

Вы посещали эту страницу несколько раз (2). Дата последнего посещения: 01.09.17

О да! Что-то проясняется! Вернее, проясняется только то, что эта функция как-то связана с генератором случайных чисел, особенно если вспомнить, что в Windows `std::rand` обычно возвращает 15-битное число, и увидеть при этом `and eax, 0x7FFF`. Резюмируя — в `FormCreate` вызываются две функции, первая что-то мутит со временем, вторая похожа на вызов `std::rand`. Что-ж, от истины мы определённо не далеки:



```
16 //-----
17 // void __fastcall TFormPasswordGenerator::FormCreate(TObject *Sender){
18 //     // RN9JjaQMjcYEuYeF
19 //     // 1609869199 = Tue, 05 Jan 2021 17:53:19 GMT
20 //     std::srand(time(NULL));
21 }
```

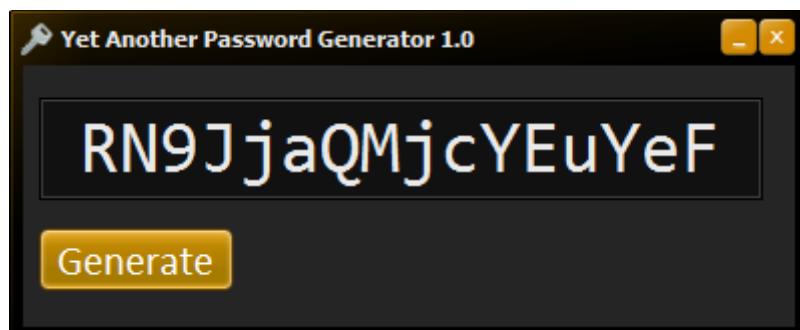
Я предлагаю вам самим исследовать код `ButtonGenerateClick`, чтобы убедиться, что там где-то вызывается `std::rand`, а нас сейчас слабо интересует, как именно он вызывается.

Суть того, почему пароль, генерируемый нашей программой небезопасный, заключается в возможности подобрать начальное значение генератора случайных чисел, и таким образом, сгенерировать тот-же самый пароль. В нашем случае (а наш случай наиболее типичный) подобрать нужно будет момент времени в UNIX формате, когда была запущена программа, чтобы получить доступ к паролям, которые она бы в этот момент сгенерировала. Именно для этого нам в условии дают временную метку.

Таким образом, достаточно в момент вызова `time`, подставить в качестве возвращаемого значения нашу временную метку, которая будет в UNIX формате выглядеть как **1609869199** или **0x5ff4a78f**. Давайте это сделаем:

Thread #7148			
006E63DD 0000	add [eax],al	EAX 5FF4A78F	CF 0
006E63DF 90	nop	EBX 030F0670	PF 1
time(long *):		ECX 59DA40B8	AF 0
006E63E0 55	push ebp	EDX 0000000B	ZF 1
006E63E1 8BEC	mov ebp,esp	ESI 006EC048	SF 0
006E63E3 83C4F0	add esp,-\$10	EDI 00402194	TF 0
006E63E6 53	push ebx	EBP 0019FE5C	IF 1
006E63E7 8B5D08	mov ebx,[ebp+\$08]	ESP 0019FE4C	DF 0
006E63EA 8D45F0	lea eax,[ebp-\$10]	EIP 006E642D	OF 0
006E63ED 50	push eax	EFL 00200246	IO 0
006E63EE E8BD3E0000	call \$006ea2b0	CS 0023	NF 0
006E63F3 6AFF	push \$ff	DS 002B	RF 0
006E63F5 0FB755FC	movzx edx,[ebp-\$04]	SS 002B	VM 0
006E63F9 52	push edx	ES 002B	AC 0
006E63FA 0FB74DFA	movzx ecx,[ebp-\$06]	FS 0053	VF 0
006E63FE 51	push ecx		
006E63FF 0FB74DF8	movzx ecx,[ebp-\$08]	0019FED4 0019FEE0 ..юа	
006E6403 51	push ecx	0019FED0 006489E9 .дэй	
006E6404 0FB745F6	movzx eax,[ebp-\$0a]	0019FECC 0019FF04 ...я.	
006E6408 48	dec eax	0019FEC8 030F0670 ...р	
006E6409 50	push eax	0019FEC4 00499EFF .Инъ	
006E640A 0FB755F2	movzx edx,[ebp-\$0e]	0019FEC0 00000012	
006E640E 4A	dec edx	0019FEBC 005FDFA2 ..яў	
006E640F 52	push edx	0019FEB8 0257DA00 .Wб.	
006E6410 0FB74DF0	movzx ecx,[ebp-\$10]	0019FEB4 0000000C	
006E6414 81C194F8FFFF	add ecx,\$fffff894	0019FEB0 0019FE9C ..юъ	
006E641A 51	push ecx	0019FEAC 006EC2E0 .нВа	
006E641B E828FCFFFF	call \$006e6048	0019FEA8 006E6477 .ndw	
006E6420 83C41C	add esp,\$1c	0019FEA4 0019FED4 ..юФ	
006E6423 85DB	test ebx,ebx	0019FEA0 FF5F23B7 я_#.	
006E6425 7402	jz \$006e6429	0019FE9C 0259E0C0 .YaA	
006E6427 8903	mov [ebx],eax	0019FE98 00402D7C .@-	
006E6429 5B	pop ebx	0019FE94 030F0670 ...р	
006E642A 8BE5	mov esp,ebp	0019FE90 006EA04B .н К	
006E642C 5D	pop ebp	0019FE8C 006EC450 .нДР	
006E642D C3	ret	0019FE88 006EC048 .нAH	
006E642E 90	nop	0019FE84 0063D567 .сXg	
006E642F 90	nop	0019FE80 0019FECC ..юM	
_InitExceptBlockLDTc(void *):		0019FE7C 030F0670 ...р	
006E6430 53	push ebx	0019FE78 030F0670 ...р	
006E6431 8BDD	mov ebx,ebp	0019FE74 006EC048 .нAH	
006E6433 035808	add ebx,[eax+\$08]	0019FE70 00402194 .@!»	
006E6436 894308	mov [ebx+\$08],eax	0019FE6C 0019FE80 ..юћ	
		0019FE68 0063D955 .сЩU	
		0019FE64 0019FE84 ..	

И, та-дам! Первый сгенерированный нашей программой пароль:



Собственно, берём его, открываем архив, находим там флаг... Но я думаю, это вы уже как-нибудь сами.

Writeups

CyberSamurai

Начальную картинку открываем с помощью outguess с паролем Azimov ,далее находим ссылку на страницу в telegra.ph, на которой находим странные хокку, эти хокку являются необычной записью ip-адреса , гуглим haiku ip .находим сервис для декодирования, получаем ip адрес, это тоже страница в telegra.ph конкретики придает надпись под самим хокку, и в последнем сообщении скрыто имя isaac ,от которого и нужно взять md5

Foren

Дан дамп трафика,нужно обратить внимание на rtp , из него можно вытащить аудиозапись звонка по voip , в ней и нужно расслышать флаг