Задания, решения и критерии отборочного этапа олимпиады школьников Ugra CTF Quals 2022

Критерии

Формат проведения этапа

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-2022-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 1425 и более баллов.
- Призёрами олимпиады были признаны участники, набравшие 200 и более баллов.

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

Задачи и решения

Термопринтер

Реверс-инжиниринг, 400 очков.

> Представьте, что вам подарили термопринтер Poooli L3, и вам нужно напечатать на нём вот эту картинку:
>
> 
>
> Вам даже подсказали, где скачать драйвера, и что драйвера для Мака как-то можно запустить под Линуксом.
>
> С помощью драйверов вам даже удалось распечатать слова `Test...` `test...` `test...`, но после этого всё почему-то перестало работать…
>
> *test-test-test.prn*
>
> *https://thermal.q.2022.ugractf.ru/9a18a6857bb051b1/*

Решение

Итак, нам надо сформировать .prn-файл с нужными данными, опираясь на уже существующий пример и драйвера.

Для начала поэкспериментируем с самим файлом. Если поискать в интернете начальные байты —1D 76 30, примерно все результаты будут про термопринтеры. Оттуда мы узнаем, что протокол называется ESC/POS, а эти три байта образуют заголовок одной из возможных команд печати растрового изображения. И действительно, эти же байты встречаются в файле ещё три раза.

Даже без изучения внутренней структуры команд можно попробовать переставлять их, поотправлять их несколько раз. Убеждаемся, что каждая команда отвечает за тонкую горизонтальную полоску из итоговой картинки. Также можем убедиться, что все данные за пределами команд ни на что не влияют, таким образом, больше ничего, кроме этих команд, в файле нет. При этом малейшее изменение данных внутри команды приводит к тому, что из «распечатанной» картинки пропадает целая полоска —по-видимому, некорректные команды «принтер» игнорирует.

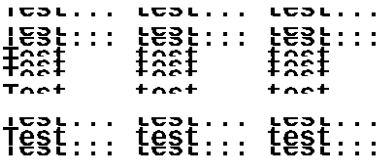

Рис. 1: Модифицированный пример

В первом результате нашего поиска приведено некое описание команды.

> Hex 1D 76 30 m xL xH yL yH d1⋯.dk
>
> <⋯>
>
> xL, xH, select the number of data bits ($xL + xH \times 256$) in the horizontal direction for the bit image.
>
> yL, yH, select the number of data bits ($yL + yH \times 256$) in the vertical direction for the bit image.
>
> <⋯>
>
> d indicates the bit-image data. Set time a bit to 1 prints a dot and setting it to 0 does not print a dot.

Смотрим в файл. В нашем случае m = 0 (режим «печати» без удвоения пикселей), xL xH образуют число 50, yL yH —число 10 (в последней команде —8).

Ширина 50 выглядит странно, потому что ширина области, «печатаемой» командой, больше, раз примерно в восемь. Это наводит на мысль, что ширина на самом деле почему-то задаётся в байтах. Опять же, можно поменять это число на, например, 100, и посмотреть, как поведёт себя «принтер».

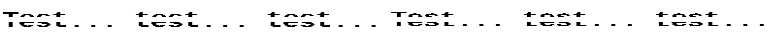

Рис. 2: Пример с модифицированной шириной

При числе 100 «распечатанные» данные занимают примерно 800 пикселей, то есть ширину из команды действительно нужно умножать на 8. Ну допустим.

Но дальше начинается удивительное: после ширины и высоты дальше должны идти непосредственно байты картинки, причём размер блока данных явно не указывается и, судя по всему, зависит от ширины и высоты. Но при одинаковой ширине и высоте команды явно разной длины, и байт в них явно меньше, чем было бы нужно, чтобы заполнить указанную область. А ещё, стоит поменять хоть один бит —и полоска пропадает целиком, хотя если бы это были биты картинки, должна была поменяться лишь небольшая область.

Наверное, куски картинки как-то сжаты.

Пришло время поковыряться в драйверах. В задании нам почему-то рекомендуют обратить внимание на драйвер для Мака. Открываем архив, находим там неинтересные .ppd.gz со всякими метаданными на английском и китайском языках, и два исполняемых файла.

```
$ file usr/libexec/cups/filter/*
usr/libexec/cups/filter/raster-mt800:  Mach-O 64-bit x86_64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>
usr/libexec/cups/filter/raster-poooli: Mach-O 64-bit x86_64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>
```

Судя по названию модели принтера, нас интересует второй из этих файлов. Разобраться в нём нам поможет Ghidra. Создадим новый проект и импортируем туда исполняемый файл, не трогая никаких настроек и игнорируя предупреждения.

Ищем (*Search* → *Memory*), где в программе встречаются байты 1D 76 30 (должна же она их использовать, формируя команду). Таких мест оказывается всего три, два из них приводят нас к функции с говорящим названием `_print_bitmap`.

Внутри декомпилированного кода видим вызов функции `_lzo1x_1_compress`. Наша догадка про сжатие подтвердилась, осталось лишь разобраться в деталях того, как формируются данные. Функция `_lzo1x_1_compress` принимает в качестве предпоследнего аргумента указатель на переменную `local_58`, куда записывает итоговый размер сжатых данных. Она затем преобразуется в тип `int` и записывается в файл. И действительно, если мы рассмотрим первые четыре байта блока данных как число, задающее длину, то ровно столько байт и будет после него и перед началом следующей команды.

Чтобы не разбираться в нюансах инициализации алгоритма —параметрах функции `___lzo_init_v2`, попробуем просто распаковать сжатые данные.

```
import lzo
prn = open("test-test-test.prn", "rb").read()
lzo.decompress(prn[12:12+35], header=False, buflen=1000000)
```

Заголовок LZO в данных отсутствует, потому что при `header=True` возникает ошибка распаковки, а при `header=False` мы не получаем ошибок, но получаем данные, похожие на те, которые в итоге «печатаются».

Попробуем теперь брать полоски из нашей целевой картинки и формировать команды. Сжатие `lzo.compress` нужно делать с параметром `level=1` (для других уровней сжатия драйвер вызывал бы не `_lzo1x_1_compress`, а другую функцию).

```
import lzo
import PIL.Image
import struct
import sys

HEIGHT_DELTA = 10
image = PIL.Image.open("thermal-printme.png")

out_file = open("out.prn", "wb")

for y in range(0, image.size[1], HEIGHT_DELTA):
    bits = ""
    for cy in range(y, min(image.size[1], y + HEIGHT_DELTA)):
        for cx in range(0, image.size[0]):
            if image.getpixel((cx, cy)) == 255:
                bits += "0"
            else:
                bits += "1"
    bits += "0" * ((8 - len(bits) % 8) % 8)

    data = int(bits, 2).to_bytes(len(bits) // 8, byteorder="big")
    z_data = lzo.compress(data, 1, False)

    out_file.write(b"\x1Dv0O")
    out_file.write(struct.pack("<hh", image.size[0] // 8, min(image.size[1] - y, HEIGHT_DELTA)))
    out_file.write(struct.pack("<L", len(z_data)))
    out_file.write(z_data)
```

Отправим этот файл на «печать». Рядом с «напечатанной» картинкой, если всё сделано правильно, на странице выводится флаг.

Флаг: **ugra_esc_pos_goes_lprrrrrrrr_5604ef89a7a6**

NFT Bezopasniy Enclave

Реверс-инжиниринг, 300 очков.

> ПАО «Агрокекстрой» планирует запустить производство микрочипов и электронных устройств.
>
> Производство секретное, но некоторые подробности просочились: судя по всему, они разрабатывают некое USB-устройство для безопасного хранения NFT «Безопасный Enclave»! Само устройство мы добыть не смогли (да и как бы это помогло делу —вас же много, а оно, видимо, существует в единственном экземпляре), зато добыли софт от него. Осторожно, внутри криптография.
>
> *anft_viewer_PRE-RELEASE*

Решение

Перед нами статический исполняемый файл —почти все библиотеки, от которых он зависит, встроены прямо в него, а не вызываются из системы. Это слегка мешает исследовать программу, поскольку не всегда можно с лёгкостью сказать, что относится к решению задания, а что не очень.

Из условия задачи и и того, как ведёт себя программа, можно сделать вывод, что она содержит в себе некоторые зашифрованные данные и расшифровывает их, если к ней подключено устройство компании «Агрокекстрой». Давайте откроем его в каком-нибудь дизассемблере.

Видим довольно ветвистый граф:

Рис. 3: Граф в дизассемблере

Попробуем пойти по простому и надёжному пути: патчинг. Для этого последовательно заменим инструкции перед каждым ветвлением: либо на противоположные (например, `JE` (jump if equal) —на `JNE` (jump if not equal)), либо же на `NOP` в случаях, когда нужная ветвь расположена сразу за прыжком.

В какой-то момент дойдём до вызова `fopen` с вряд ли возможно корректным в принципе аргументом `"/dev/mmcblk13"`. Его тоже аккуратно запатчим. Так же поступим с циклом, следующим сразу за ним, в котором файл считывается функцией `getc` —в общем-то, стандартный способ чтения файлов в Си.

После этого программа начнёт печатать в терминале кракозябры. Посмотрим, что это такое:

```
$ ./program | file -
/dev/stdin: GIF image data, version 87a, 362 x 140
```

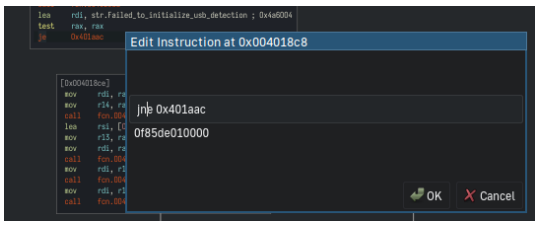

Рис. 4: Редактирование ассемблерного кода

Ого! Жаль, не не открывается —как будто бы и половины байтов не хватает.

Давайте немного повтыкаем в ассемблер —вдруг станет понятно. Вот блок, который печатает по символу расшифрованной картинки:

```
[0x00401a82]
movsx   edi, byte [rbx + r12]      ; int64_t arg1
inc     r12
call    printf                     ; int printf(const char *format)
cmp     r12, 0x536                 ; 1334
jne     0x401a82
```

Рис. 5: Вызов функции `printf` в цикле

> Можно и не втыкать в ассемблер, а просто поставить точку останова где-нибудь перед концом программы и снять образ памяти, в котором изображение легко найти по заголовку GIF-формата: `GIF87aW`.

Видно, что цикл выполняется `536h = 1334` раз. Сразу перед ним есть цикл, который похож на что-то, что расшифровывает картинку. Он выполняется `14D8h = 5336 = 1334*4` раз. Поправим эту оплошность. Теперь программа печатает картинку как надо:

Рис. 6: Расшифрованное изображение с флагом

Флаг: **ugra_kerckhoffs_saw_disassemblers_coming_ea107f7b**

На блюдечке···

Веб-программирование, 25 очков.

> ···с золотой каёмочкой.
>
> *https://onaplate.q.2022.ugractf.ru/8b39183ec7a2d89b/*

Решение

Открываем страницу. На странице действительно блюдечко. С золотой каёмочкой. А на блюдечке — адрес файла с флагом: `fl4g?is?h#r3`. Копируем, вставляем в конец адресной строки прямо в браузере. Проигрываем, поскольку имя флага содержит особые символы: `?` используется в GET-параметрах, `#` —в HTML-якорях.

Вспоминаем про URL-кодирование и «стерилизуем» имя файла: `fl4g%3Fis%3Fh%23r3`. Снова пробуем дописать его в конец адресной строки и на этот раз действительно начинается загрузка. Внутри видим флаг.

Флаг: **ugra_as_easy_as_it_gets_606fc4979a97**

Поорите

Бинарная эксплуатация, 75 очков.

> Расслабьтесь. Глубоко вдохните —и ···
>
> *shout*
>
> `nc shout.q.2022.ugractf.ru 4141`
>
> `Токен: 88aa87feae5a4877e859b3a9`

Решение

Перед нами задание на бинарную эксплуатацию. Посмотрим, что оно умеет, не открывая ни бинарный файл, ни дебаггер.

Нередко в таких заданиях бывает переполнение, и, чтобы забить нужное количество байт, обычно используют буквы `A`. Возможно, на это нам и намекают, когда просят поорать. Что ж, попробуем: введём 100 букв `A`:

```
$ ./shout
Shout here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Too weak, try again.
```

Попробуем ввести больше букв. Например, 2000:

```
Too weak, try again.
Segmentation fault (core dumped)
```

На этот раз приложение упало с ошибкой. Это означает, что уязвимость всё-таки есть —и эта уязвимость —переполнение стека. Уязвимость заключается в том, что приложение неправильно валидирует данные, хранящие на стеке приложения: мы можем ввести больше данных, чем есть места —в этом случае приложение их просто начнёт писать дальше, затирая какие-то важные данные.

Всё-таки дальше без изучения приложения не обойтись. Откроем его в IDA или Cutter и посмотрим, что внутри. Сразу замечаем какую-то непонятную секцию `.flags`, местоположение которой, скажем прямо, удивляет —она находится через 532 МБ от начала приложения.

В этой секции мы находим единственную функцию `print_flag` по адресу `0x21414141`, которая открывает файл `flag.txt` и печатает его. Казалось бы, вот и решение —но незадача: эта функция нигде в коде не вызывается.

Теперь нужно понять, как можно использовать найденную уязвимость для вызова функции. На самом деле, стек вызовов выглядит примерно так:

```
... | shout                                | main                                | ...
... | local vars | ebp | return addr | args | local vars | ebp | return addr | args | ...
```

Таким образом, если мы затрём данные после локальной переменной в функции `shout`, мы сможем переписать `ebp`, `return addr` и всё, что находится дальше. А что произойдёт потом? А потом программа успешно дойдёт до конца функции и попробует вернуться в «предыдущую функцию» по адресу, который лежит в return addr. И, если нам удастся подставить сюда адрес нашей функции `print_flag`, то мы победим.

Следующая проблема —формат ввода. Как известно, мы вводим в программу буквы, а адрес перехода состоит из цифр. Что же делать? Тут всё очень просто —мы вводим те символы, шестнадцатеричные коды которых соответствуют байтам нашего адреса. Есть одно маленькое «но» —в Linux порядок байт little endian, поэтому адрес нужно вводить «наоборот». Числу `41` соответствует символ `A`, а числу `21` —символ `!`. Получается, наш адрес мы будем вводить как `AAA!`.

Наконец, нужно понять, сколько байт нам нужно всего. Можно просто перебрать. Проще —локально. Создадим файл `flag.txt` и будем запускать наш файл с разным количеством `A`. Можно отреверсить этот бинарный файл и узнать, сколько байт выделяется на стеке.

В итоге мы получаем эксплойт —1039 букв `A` и восклицательный знак. Ровно так нужно проорать, чтобы получить флаг.

Флаг: **ugra_AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA_690868a2aeaff4c1**

Хохорейсинг

> Если вы уже наигрались в Разборчивую змейку, самое время переключиться на другую игру, попроще.
>
> *https://xoxoracing.q.2022.ugractf.ru/5b4e1c735f4f262c/*

Решение

Предлагается игра: нужно придумать какой-нибудь ключ шифрования, получить зашифрованный им текст и угадать, что же было зашифровано, пока не истекло время. После ввода своего варианта игра показывает, какой был правильный ответ.

Немного поиграв вхолостую, можно обратить внимание, что:

- Оригинальный текст —всегда несколько слов на английском языке
- Длина текста всегда одинаковая —40 символов; текстовые поля тоже предлагают ввести до 40 символов
- Непечатные символы в зашифрованном тексте заменяются изображением, восстановить их невозможно
- Шифрование побуквенное: одна и та же буква исходного текста при одной и той же букве ключа соответствует одной и той же букве зашифрованного текста

Последний факт (а также намёк в названии игры) приводит к выводу, что шифрование —не что иное, как операция XOR над текстом и ключом.

Если бы мы указывали в качестве ключа нулевые байты, текст в процессе шифрования не изменялся бы. Однако простого и удобного способа отправить такой ключ нет, поэтому можно подумать, как ограничиться символами, доступными с клавиатуры.

Операция XOR тем сильнее меняет текст, чем больше в ключе двоичных единиц. Меньше всего единиц из доступных нам символов в пробеле (код $32 = 00100000_2$) и собаке (код $64 = 01000000_2$).

По счастью, стандартные кодировки устроены так, что коды заглавных и строчных английских букв отличаются ровно на 32. Значит, если выбрать в качестве ключа строку из пробелов, то заглавные буквы превратятся в строчные, строчные —в заглавные, а пробелы —в непечатный символ с кодом 0. Это позволит по такому «зашифрованному» тексту без труда восстанавливать оригинал в уме.

Получается стратегия игры: в качестве ключа всегда вводить (или вставлять из буфера обмена) 40 пробелов, после чего аккуратно перепечатывать текст с противоположным регистром. После каждого успешно сыгранного раунда внизу открывается несколько букв флага. Сыграв 15–20 раз, можно получить флаг целиком.

Флаг: **ugra_go_go_go_come_on_yes_yes_a_bit_more_just_a_little_400f02858d7c656**

Well known and loved

> В интернете нашли страничку одного известного киберинженера. Узнайте, что он знает и любит.
>
> *https://well.q.2022.ugractf.ru*

Решение

В таске было несколько отсылок на то, что ответ может быть как-то связан с `.well-known` —это специальный путь для различных сервисов, которые хотят отдавать какую-то метаинформацию — например, специальные ссылки для iOS или Android, или коды подтверждений для автоматического получения TLS-сертификатов.

Список всего, что может находиться в этой директории по стандарту, можно найти на сайте IANA. Осталось просто перебрать все эти файлы.

По адресу `https://well.q.2022.ugractf.ru/.well-known/matrix` вместо ожидаемой ошибки 404 мы получаем 403 —доступ запрещён. В той же таблице находим стандарт Matrix, в котором говорится о существовании двух файлов —`matrix/server` и `matrix/client`. И, действительно, из этих файлов мы узнаём о существовании сервера для мессенджера Matrix на порту 5000.

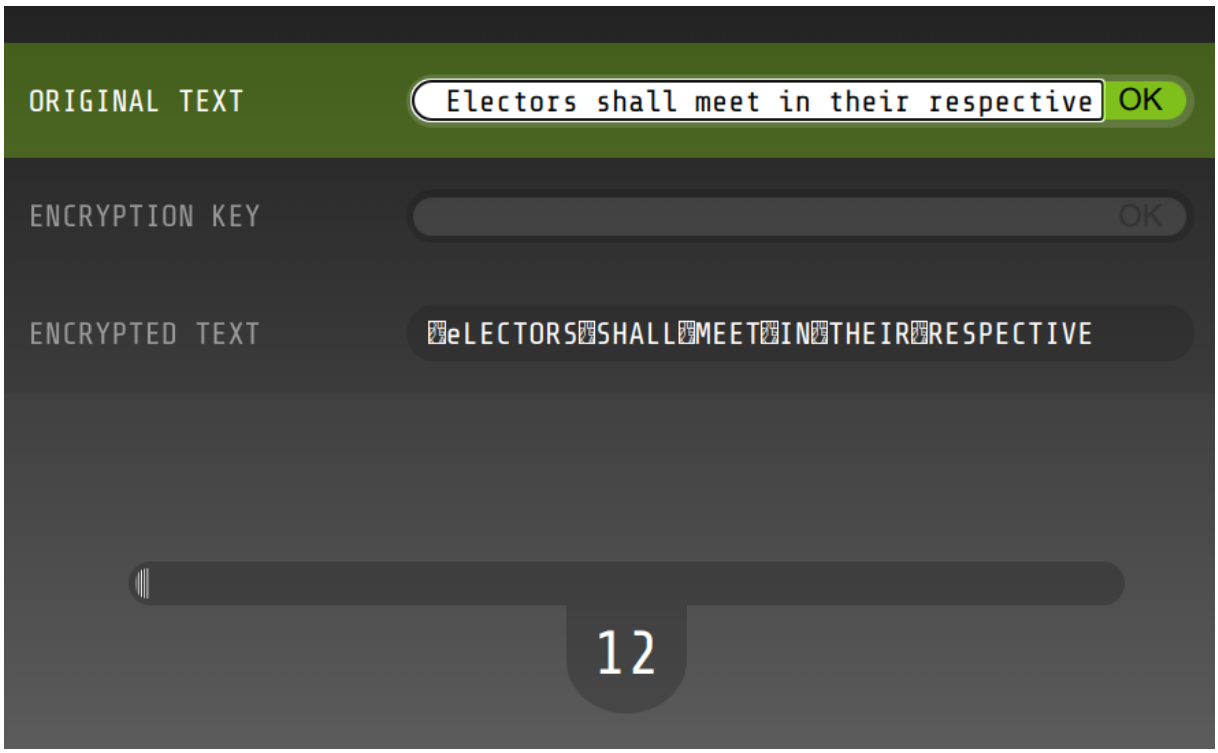

Рис. 7: Пример шифрования с пробельным ключом

Подключимся к нему любым клиентом —например, Element. Смотрим каналы —видим публичный, но закрытый, канал `kittens`, а в нём —флаг от одного из организаторов.

Флаг: **ugra_matrix_is_well_known_to_everyone_31b66f2022d2e62ed**

Разборчивая змейка

Программирование, 250 очков.

> Чем ближе рекорд, тем интереснее играть.
>
> *https://snekpeek.q.2022.ugractf.ru/cb8ed0507e65de8a/*

Решение

Классическая игра в змейку идёт на поле 42×42, за каждое приращение даётся 10 очков. Чтобы повторить указанный на странице рекорд в 17600 очков, придётся заполнить змейкой всё поле наглухо.

Делать это, конечно, лучше не вручную.

Для начала разберёмся, как устроена игра. Открываем инспектор браузера, в нём —вкладку Network, видим там открытый вебсокет и сообщения в нём. По-всякому играем, чтобы понять, что бывает.

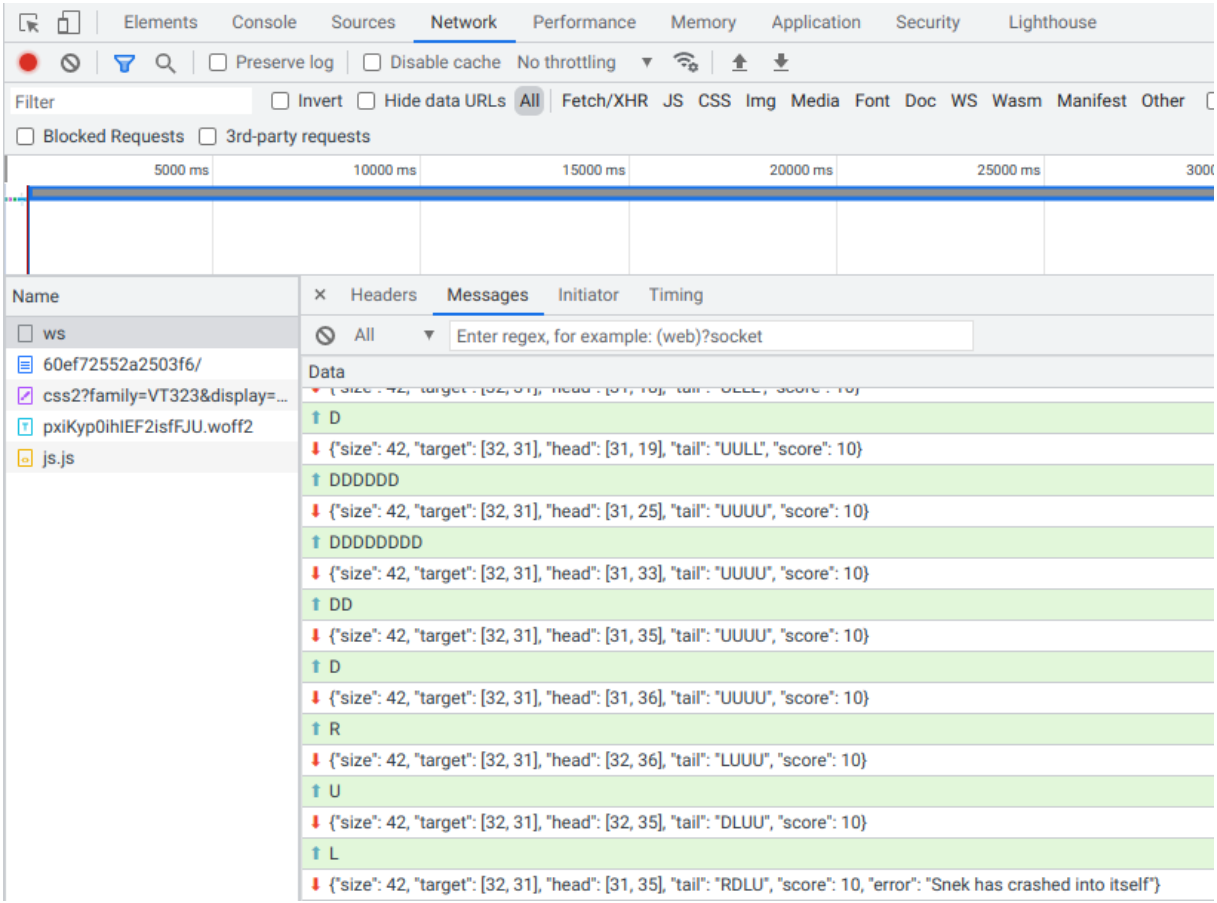

Рис. 8: Сообщения в инспекторе

Вроде всё понятно. Отправляем строчки с движениями (буквы U, L, D, R —up, left, down, right), получаем в ответ JSON с данными про положение целевой точки на поле, расположение самой змейки и счёт; если проиграть, там же будет сообщение об ошибке.

Важно заметить: * Можно посылать несколько движений в одном сообщении * Сервер лишь реагирует на посланные нами движения, но сам змейку не двигает * Тикающее время на странице не участвует в обмене данными с сервером, а значит, ни на что не влияет

Дальше можно было бы реализовать какой-нибудь умный и оптимальный алгоритм игры, но поскольку ходов можно делать сколько угодно, поступим проще. Выведем змейку в угол поля (например, левый верхний) и будем бесконечно гонять её замкнутым зигзагом, покрывая все точки поля. Где бы ни находилась цель, змейка до неё рано или поздно дойдёт без столкновений. Если повезёт, за один круг можно будет встретить цель даже несколько раз.

Этой траектории соответствует строка D D⋯D R U⋯U R D⋯D R U⋯U R ⋯D⋯D R U⋯U U L⋯L.

Наберёмся наглости и будем тупо посылать одну и ту же зацикленную траекторию бесконечно, пока игра не закончится.

```
TRAJECTORY = "D" + (("D"*40 + "R" + "U"*40 + "R") * 21)[:-1] + "U" + "L"*41

async with aiohttp.ClientSession() as ss:
    ws = await ss.ws_connect("wss://snekpeek.q.2022.ugractf.ru/cb8ed0507e65de8a/ws")
    await ws.receive()
    await ws.send_str("U"*28 + "L"*28)
    while True:
        msg = await ws.receive()
        print({**msg.json(), "tail": "..."})
        await ws.send_str(TRAJECTORY)
```

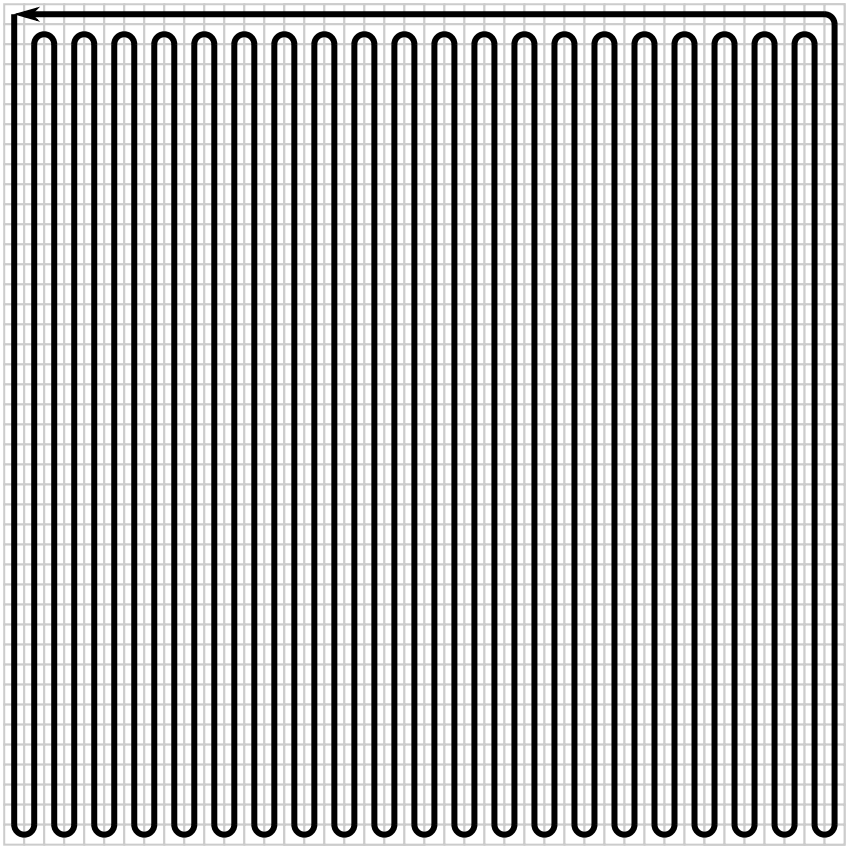

Рис. 9: Траектория змейки

Но игра всё не заканчивается и не заканчивается. Вернее, заканчивается, но почему-то всегда раньше, чем нам хотелось бы, и с вот такой ошибкой:


```
{"size": 42, "target": null, "head": [36, 36], "tail": ..., "score": 13410,
 "error": "Unable to place new target because all permitted target locations are already occupied by snek"}
```


Получается, есть какие-то области, в которых цель появляться не может. Попробуем запоминать в ходе игры и потом отметить на поле все места, где мы когда-либо видели цель —вдруг что-то станет понятно.

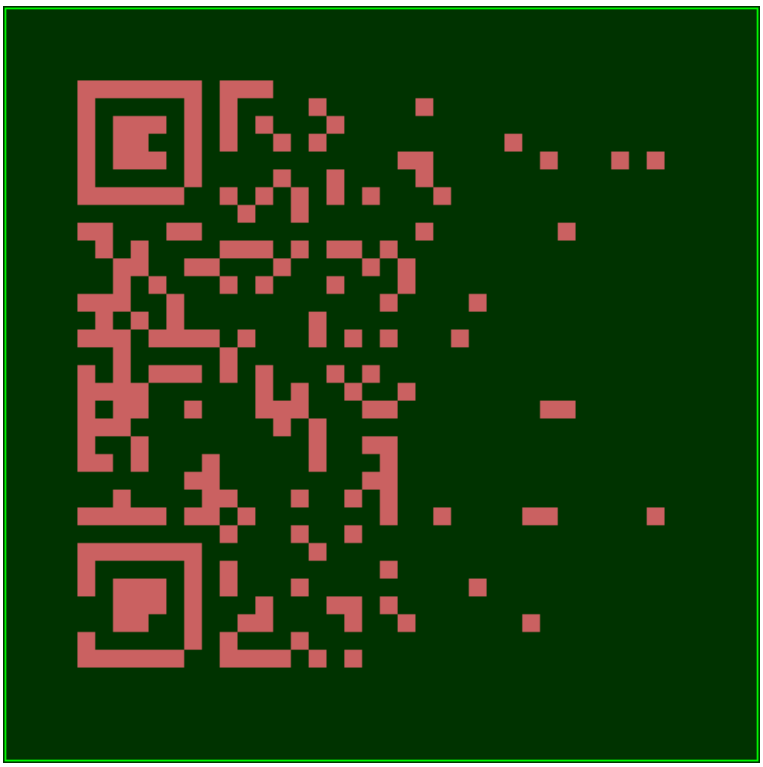

Рис. 10: Положения цели в течение игры

Явно угадывается QR-код, но какой-то неполноценный и уж точно не распознающийся. Точки в его правой части собрались хуже, чем в левой. Набрать недостающие точки можно разными путями — например, сыграть второй раз, перевернув траекторию на 180°. Или отправлять траекторию небольшими кусочками, чтобы успеть заметить побольше возможных расположений цели. Или начинать игру много раз и доигрывать до небольшого счёта. В ряде случаев может потребоваться дорисовать служебные области QR-кода.

Флаг: **ugra_i_specifically_requested_the_opposite_of_this_9e866bcf53ac**

Постмортем В первый день соревнований в сокет можно было послать только одну команду на перемещение за раз. Это делало решение хоть и не невозможным, но существенно более долгим (порядка нескольких часов на один прогон до упора), при этом разрыв соединения приравнивается к проигрышу и требует начинать сначала. После изменений один прогон стал занимать порядка минуты.

Прикольная находка

Форензика, 100 очков.

> Во дворе одной крупной корпорации была обнаружена связка ключей с брелком и флешкой. Пока мы изучаем ключи, предлагаем вам изучить содержимое флешки и брелка.
>
> *neatfind.raw.tar.gz*
>
> `Надпись на брелке: +79467073279`

Решение

Скачиваем файл и начинаем его изучать:

```
$ file neatfind.raw
neatfind.raw: DOS/MBR boot sector; GRand Unified Bootloader, stage1 version 0x3, 1st sector stage2 0x165838
```

Оказывается, флешка-то загрузочная! Пробуем загрузиться с неё в виртуальной машине. Например, в QEMU:

```
$ qemu-system-x86_64 -enable-kvm -hda neatfind.raw
```

Видим экран блокировки:

Рис. 11: Экран блокировки

Надеемся на лучшее (или худшее —тут как посмотреть) и ищем стандартный пароль пользовательской учётной записи в дистрибутиве TazPuppy. Находим их: оказывается, пароля нет! Два раза жмём «Ввод» и видим рабочий стол. Говорят, мы попали в какую-то *Корпоративную среду.*

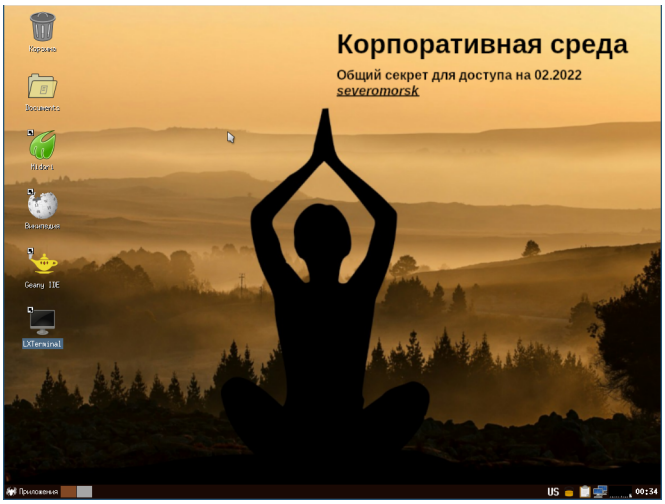

Рис. 12: Рабочий стол

Секрет с обоев переписываем в блокнотик —на будущее. В попытках зацепиться за что-то перебираем значки на рабочем столе: в браузере пусто, в текстовом редакторе тоже. Зато терминал радует богатой историей команд.

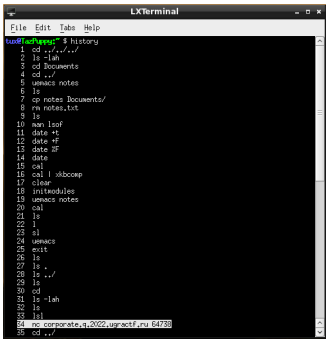

Рис. 13: История команд

Курсором на скриншоте выше выделена важная строка о подключении к серверу: `corporate.q.2022.ugractf:64738`. Пробуем подключиться. Спрашивают номер телефона (брелок-то пригодился) и секрет. Дальше не пускают, но флаг дают:

```
ОБНАРУЖЕНА ПОПЫТКА НЕСАНКЦИОНИРОВАННОГО ДОСТУПА К КОРПОРАТИВНОЙ ИНФРАСТРУКТУРЕ.
НАРУШИТЕЛЬ ЗАФИКСИРОВАН. ИНЦИДЕНТ ЗАФИКСИРОВАН.
```

```
IP-АДРЕС МЕСТА НАРУШЕНИЯ: [...]
ЛОКАЦИЯ НАРУШИТЕЛЯ: Russia, Yugorsk
ВНУТРЕННИЙ ИДЕНТИФИКАТОР НСД: ugra_every_big_thing_consists_of_many_little_ones_dc399161a8aa
```

> Во время соревнований нам часто поступали письма от участников —мол, флаг неверный. Дело было в том, что система на удалённом сервере требовала в явном виде вводить номер телефона без плюсика, в то время как в условии задачи плюсик-таки был. Система никак его не валидировала, но инциденты всё равно фиксировала. Вот такая притча без морали о человеческой внимательности.

Флаг: **ugra_every_big_thing_consists_of_many_little_ones_dc399161a8aa**

Поля слишком узки

Прочее, 100 очков.

> Многие математики пытались доказать Великую теорему Ферма, и один вроде даже смог, но так сложно и непонятно, что ему всё равно многие не поверили. Один из таких недоверчивых математиков надеется найти контрпримеры, да такие, чтобы всем сразу стало всё очевидно. Помогите ему и опровергните теорему.
>
> *https://margin.q.2022.ugractf.ru/d0b7b4755a0d67b0/*

Решение

Что ж, нам действительно нужно найти такие положительные целые числа x, y, z, что $x^3 + y^3 = z^3$. А потом ещё и другие, такие, что $x^4 + y^4 = z^4$.

> Невозможность существованиях таких наборов чисел для степеней 3 и 4 была показана давно, но окончательно доказать теорему для всех степеней удалось лишь в 1995 году математику Эндрю Уайлсу.

Вернее, не так. Нам достаточно *убедить валидатор* в том, что числа, которые мы предъявим, обладают этим свойством. Исходный код валидатора нам любезно предоставлен. По сути, если отбросить всякие неинтересные действия по преобразованию типов и проверке корректности самих чисел, остаются вот такие условия:

```
POWER_CUBIC = 3.0
POWER_QUARTIC = 4.0

...

if x_cubic ** POWER_CUBIC + y_cubic ** POWER_CUBIC != z_cubic ** POWER_CUBIC:
    ...

if x_quartic ** POWER_QUARTIC + y_quartic ** POWER_QUARTIC != z_quartic ** POWER_QUARTIC:
    ...
```

Наши числа возводятся в степень `3.0` и `4.0`. Это отличается от возведения в степень `3` и `4`, которое являлось бы точным вычислением благодаря встроенной в Python поддержке *длинной арифметики*. Здесь показатели степени являются значениями типа float, значит, и типом результата всей операции тоже является float, а он способнен хранить числа лишь с ограниченной точностью.

Таким образом, валидатор можно обмануть немного неточным решением, если погрешность будет достаточно мала. Ищем в интернете fermat last theorem near misses.

Первый результат —обсуждение на StackExchange, где показан способ генерировать неточные решения третьей степени со сколь угодно малой погрешностью. Например, можно взять $m = 244$, тогда получится, что $x = 179097$, $y = 43759448$, $z = 43759449$.

Второй результат —страничка одного гарвардского математика, где он приводит некоторые неточные решения для более высоких степеней, включая нужную нам четвёртую. Берём оттуда пример с самым большим по модулю r, чтобы наверняка: $x = 419904$, $y = 1257767$, $z = 1261655$.

Вводим значения, получаем флаг.

Флаг: **ugra_i_wish_fermat_had_css_63829b4f1f8c**

cmap I

Программирование, 150 очков.

> На объекте X, даже одно название которого совершенно секретно, введены чрезвычайные меры информационной безопасности —все компьютеры отключили от сети. Теперь сотрудники X передают данные только на бумажных носителях —с визой руководителя отдела.
>
> Однако, быстро выяснилось, что передавать сверхточные чертежи таким образом без потерь не получится. Поэтому в стенах X был разработан алгоритм потоковой отправки данных с использованием лазерной печати без потерь. Алгоритм настолько эффективен, что позволяет передавать даже цветные изображения с использованием лишь чёрно-белого принтера.
>
> Нам всё же удалось достать один из секретных файлов. Только почему-то ничего не копируется. Помогите разобраться :(
>
> *Примечание.* В этом задании два флага.
>
> *Примечание 2 (добавлено 28 февраля в 17:32).* не все программы для просмотра PDF показывают содержимое файла корректно. Вы должны увидеть много букв на каждой странице. Если вы их не видите —например, в Chromium, попробуйте другую программу.
>
> *flag.pdf*

Решение

В файле видим длинный текст из английских заглавных букв. Однако, при копировании мы получаем некую последовательность нечитаемых символов.

Название таска нас отсылает к библиотеке для языка вёрстки LaTeX, которая делала получаемые PDF-файлы «искабельными и копируемыми». Нам это, правда, никак не помогает.

Что ж, в файле всего 32 различных символа. Можно заметить, что одинаковые буквы в тексте превращаются в одинаковые символы при копировании. Найдём руками соответствие букв и символов и заменим.

Получаем файл в кодировке base32, после декодирования видим PNG-картинку с флагом.

Флаг: **ugra_you_do_not_need_usepackage_cmap_4228eae1c90b**

cmap II

Программирование, 150 очков.

> На объекте X, даже одно название которого совершенно секретно, введены чрезвычайные меры информационной безопасности —все компьютеры отключили от сети. Теперь сотрудники X передают данные только на бумажных носителях —с визой руководителя отдела.
>
> Однако, быстро выяснилось, что передавать сверхточные чертежи таким образом без потерь не получится. Поэтому в стенах X был разработан алгоритм потоковой отправки данных с использованием лазерной печати без потерь. Алгоритм настолько эффективен, что позволяет передавать даже цветные изображения с использованием лишь чёрно-белого принтера.
>
> Нам всё же удалось достать один из секретных файлов. Только почему-то ничего не копируется. Помогите разобраться :(
>
> *Примечание.* В этом задании два флага.
>
> *Примечание 2 (добавлено 28 февраля в 17:32).* не все программы для просмотра PDF показывают содержимое файла корректно. Вы должны увидеть много букв на каждой странице. Если вы их не видите —например, в Chromium, попробуйте другую программу.
>
> *flag.pdf*

Решение

Цель второго таска не ясна. Но нам явно говорят, что где-то в файле есть скрытая пометка. Можно начать исследование от первого таска —в нём была проблема со шрифтом, в нём и попробуем разобраться.

Воспользуемся любой утилитой для извлечения шрифта из PDF. Исследуем все символы в полученном шрифте. На случайных позициях видим заглавные буквы и цифры —ровно эти позиции и нужно было найти в первой части задания (возможно, так даже проще их находить).

Но наше внимание в этот раз привлекают символы с кодами больше 65536 —зачем они нужны? Присматриваемся к глифам (рисункам символов) и видим, что они последовательно составляют флаг.

Это и есть та самая метка.

Флаг: **ugra_oh_what_a_weird_font_here_017272589fec**

noteasy03

Криптография, 350 очков.

> *Цезарь скривился,*
>
> *Замкнулся в себе.*
>
> *Преумножение.*
>
> [рисунок] *ciphertext.txt*

Решение

Задание noteasy03 —продолжение давней югорской традиции: давать флаг, зашифрованный каким-нибудь шифром простой замены, и говорить загадками.

В этот раз нам дают график какой-то довольно сложной кривой, некоторым точкам которой сопоставлены символы алфавита. Если некоторое время поизучать, что вообще бывает в криптографии, то рано или поздно вы наткнётесь на эллиптические кривые. Или, возможно, они уже встречались вам. Как бы там ни было, на графике изображена именно такая кривая:

В криптографии, как правило, работают с конечными полями —всегда «берут» все числа по модулю, проще говоря. Поэтому кривая в данном случае задаётся уравнением вида:

$$y^2 = x^3 - ax + b \bmod r$$

Определяем параметры Как видно, у такой кривой есть три параметра: a, b и r.

Начнём с простого. Из школьного курса алгебры известно, что параметр b отвечает за смещение графика по оси Oy. Наш же график симметричен, следовательно, кривая никуда не смещена —то есть параметр b равен нулю.

Теперь r, порядок конечного поля, то есть число элементов, из которых оно состоит. Логично предположить, что порядок поля, на котором задана кривая, совпадает с мощностью алфавита шифра. Посчитаем точки на графике: их 31 штука —это число к тому же простое, а криптографы, как известно, просто обожают брать модули по простым числам. Берём!

Остался лишь один параметр, и это a. Его вполне реально подобрать. Для этого можно поступить вот как.

Сперва выпишем все точки, которые у нас есть:

Символ	Точка	Символ	Точка
A	(0, 0)	Q	(18, -13)
B	(1, 6)	R	(20, 12)
C	(1, -6)	S	(20, -12)
D	(2, 4)	T	(21, 13)
E	(2, -4)	U	(21, -13)

Символ	Точка	Символ	Точка
F	(3, 15)	V	(22, 14)
G	(3, -15)	W	(22, -14)
H	(4, 7)	X	(23, 13)
I	(4, -7)	Y	(23, -13)
J	(12, 3)	Z	(24, 1)
K	(12, -3)	[	(24, -1)
L	(14, 14)	\	(25, 15)
M	(14, -14)	]	(25, -15)
N	(15, 5)	^	(26, 14)
O	(15, -5)	_	(26, -14)
P	(18, 13)		

Затем возьмём какое-нибудь a, а также координаты x и y каждой точки и будем проверять, выполняется ли в них во всех равенство, которым задаётся кривая. Переберём, например, на Питоне:

```
POINTS = [(0, 0), (1, 6), ..., (26, -14)]
B = 0
R = 31

for a in range(-100, 101):
    for point in points:
        x, y = point
        left  = pow(y, 2) % r # левая часть уравнения
        right = (pow(x, 3) + a*x + b) % r # ...правая
        if left ≠ right:
            break
    else:
        print(f"All points are on curve for a = {a}")
        break
```

В этом случае a равно четырём, но вообще, у каждой команды этот параметр был разным.

Ура, с кривой разобрались.

Криптографический анализ Давайте ненадолго отвлечёмся на хаику, приложенное к заданию:

> _Цезарь скривился, _Замкнулся в себе. _Преумножение.

Ёмкость произведений в этом уникальном поэтическом жанре порой захватывает дух. Этот конкретный стишок, конечно, ни на что художественное не претендует, но всё же намёков из него можно извлечь достаточно:

- «Цезарь» —шифр простой замены? Возможно, присутствует число 3 (классический шифр Цезаря подразумевает сдвиг именно на столько позиций).
- «Замкнулся» —модульная арифметика?
- «В себе» —н/д.
- «Преумножение» —основа шифра —операция умножения?

Цезарь —это когда букву превращают в порядковый номер, прибавляют три, и превращают порядковый номер обратно в букву: `chr(ord('A') + 3) # 'C'`. Сложение точек на эллиптических кривых действительно определено, но только оно не совсем тут подходит: что прибавлять-то? (3, 0)? (3, 3)? Какая-то угадайка —точно не то. Умножение точек тоже определено, и там как раз можно умножать на целые числа —повторно складывать точку столько раз, какой величины число. Да и намёк на умножение присутствует.

Предположим, что все точки действительно взяли и умножили на одно и то же число (на 3 или нет —не так важно, поле конечное, а, следовательно, пространство перебора крайне небольшое). Получается, надо просто взять и разделить всё на какое-то число. Делается это путём умножения точки на число, обратное делителю ($a \div b = a \times b^{-1}$).

Давайте попробуем.

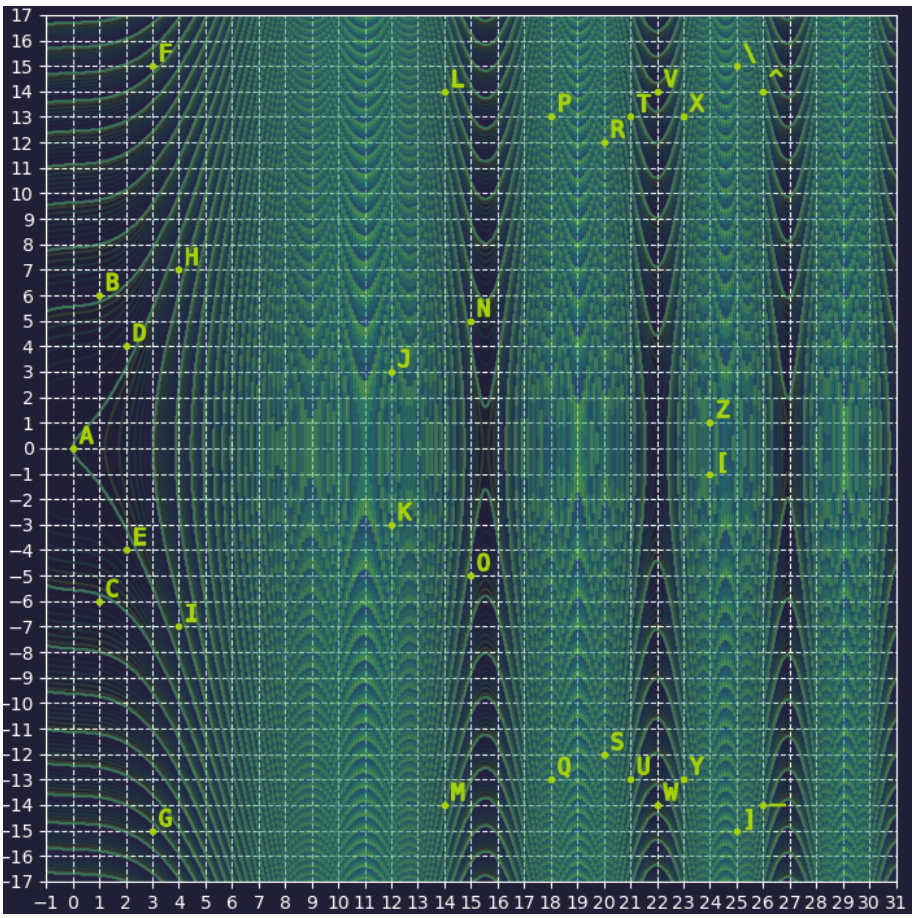

Рис. 14: Пример рисунка из задания

Практическая часть

> Эту часть можно читать по диагонали. Если вы не хотите вникать в нюансы, воспользуйтесь готовой библиотекой для работы с эллиптическими кривыми. Например, PyECCArithmetic.

Адаптируем формулу сложения из «Википедии». Почему для сложения используется несколько случаев лучше понять, если разобраться с геометрическим смыслом сложения таких точек. Но райтап и без того затянулся, поэтому покажу сразу код:

```
POINTS = [(0, 0), (1, 6), ..., (26, -14)]
B = 0
R = 31
A = 4


def ec_add(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    if x1 == x2: # точки на одной прямой линии...
        if y1 != y2 or y1 == 0: # ...но это разные точки...
            return (None, None) # получается точка (∞, ∞)
        else: # ...и это одна и та же точка
            l = (3 * pow(x1, 2) + A) * pow(2 * y1, -1, R) % R
    else: # ...и это вообще разные точки
        l = (y2 - y1) * pow((x2 - x1), -1, R) % R
    x3 = (pow(l, 2) - x1 - x2) % R
    y3 = (l * (x1 - x3) - y1) % R
    return (x3, y3)
```

Теперь умножение и «деление»:

```
def ec_mul(p, n):
    result = (0, 0)
    for i in range(n):
        result = ec_add(result, p)
    return result


def ec_div(p, n):
    return ec_mul(p, pow(n, -1, R))
```

Проверка всех догадок Дело за малым. Сконвертируем шифртекст из символов в соответствующие точки. Разделим каждую точку на три. Сконвертируем получившиеся точки обратно. Может ничего не получиться.

Дело в том, что координаты точек, которые вычислит скрипт выше, будут неотрицательными, в то время как на графике точки бывают всякие. Вспоминаем про модульность арифметики и каждую точку после деления преобразуем по правилу:

```
if y > (R // 2):
    y -= R
```

Флаг: **ugra_in_case_of_losing_your_sanity_dial_oh_three_bgkdahjoffifoljflcnkg**

ASCII-RSA

Криптография, 75 очков.

> Очередная реализация известного криптографического алгоритма впечатляет своей производительностью.
>
> *rsa.enc*
>
> *asciirsa.py*

Решение

Перед нами ещё одна странная реализация RSA. Странно в ней абсолютно всё —побуквенное шифрование, модуль 256 (очевидно не состоящий из двух простых делителей), непонятное умножение символов.

Пойдём самым простым путём: давайте просто посмотрим на все символы с кодами от 32 до 126 (это все «печатаемые символы» —пробел, цифры, буквы, подчёркивание и много чего ещё) и узнаем, во что зашифруется каждый символ.

Для этого можно, например, в текстовый файл `flag.txt` записать строку

```
 !"#$%&'()*+,-./0123456789:;≤>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

и зашифровать её. Теперь осталось найти в полученном файле символы из нашего зашифрованного флага и сопоставить.

Флаг: **ugra_dont_roll_asciirsa_crypto_4b72d642e66ced42**

Играет как умеет

> Один композитор при зарубежном военном оркестре, состоящем из людей на букву Y, написал симфонию.
>
> Неподготовленному слушателю может быть трудно расслышать в ней всё, что нужно, поэтому автор любезно предлагает начать с разминки.
>
> *symphony.pdf*
>
> *warmup.pdf*

Решение

Нам дана сумасшедшая нотная запись, и даже не одна, а целых две. Явно хочется послушать, как же это звучит.

Немного о том, как устроены ноты. Они идут подряд по строкам, как текст. Значок 𝄽 означает паузу, значок 𝅗𝅥 —звук. Высота головки ноты относительно линеек соответствует высоте звука; висящие на одной палке гроздья головок обозначают звуки, звучащие одновременно. Длительность всех звуков и всех пауз в обоих файлах одинаковая, отличия выражались бы формой используемых знаков. Всё остальное, что мы видим в записи, можно смело игнорировать (заметим лишь, что манера исполнения *raccapricciante* приблизительно переводится с итальянского как *криповo*).

Зная это, преобразуем запись во что-то, с чем будет работать удобнее. Будем искать головки нот и значки пауз и пытаться понять, к чему они относятся.

Исследовав первую страницу и посчитав, сколько какого размера объектов на них есть, выясним, что можно узнавать объекты по размерам:

- Пауза: 4.7223 36.4832
- Головка ноты: 6.4956 36.4832
- Ключ: 12.7721 36.4832

```
from pdfminer.layout import LAParams
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import PDFPageAggregator
from pdfminer.pdfpage import PDFPage
from pdfminer.layout import LTTextBoxHorizontal, LTTextLineHorizontal, LTChar

symbols = []

document = open("warmup.pdf", "rb")
rsrcmgr = PDFResourceManager()
laparams = LAParams()
device = PDFPageAggregator(rsrcmgr, laparams=laparams)
interpreter = PDFPageInterpreter(rsrcmgr, device)
for n, page in enumerate(PDFPage.get_pages(document)):
    interpreter.process_page(page)
    layout = device.get_result()
    for element in layout:
        if isinstance(element, LTTextBoxHorizontal):
            for e1 in element:
                if isinstance(e1, LTTextLineHorizontal):
                    for e2 in e1:
                        if isinstance(e2, LTChar):
                            w, h = round(e2.bbox[2] - e2.bbox[0], 4), round(e2.bbox[3] - e2.bbox[1], 4)
                            t = None
                            if (w, h) == (4.7223, 36.4832):
                                t = "pause"
                            elif (w, h) == (6.4956, 36.4832):
                                t = "notehead"
                            elif (w, h) == (12.7721, 36.4832):
                                t = "clef"
                            if t:
                                symbols.append((t, round(e2.bbox[0], 2), round(n * 1000 + e2.bbox[1], 2)))
```

Дальше будем интерпретировать их. По горизонтальной координате каждой головки ноты понимаем, к какому сгустку она относится, по вертикальной —на какой строке стоит. Дальше осталось для каждого распознанного нами сгустка определить позицию, в которой находилась бы (или находится) какая-нибудь конкретная нота из тех, что размещаются непосредственно на линейках, и отсортировать по этой позиции —получим ноты и паузы в хронологическом порядке.

Чтобы получить возможность всё это послушать, запишем данные в наиболее естественный для таких вещей формат —MIDI.

Делаем ноты не очень громкими, чтобы спокойно накладывались друг на друга, и перебираем

разные варианты темпа (его придётся в итоге ускорить примерно в 4–5 раз по сравнению с тем, что предписывают ноты).

Вот как это звучит: symphony (midi) и warmup (midi).

С удивлением обнаруживаем, что в нотах слышится искажённая, с трудом понятная, но речь. И правда *raccapricciante*. Говорящие пианино известны уже достаточно давно. Говорящих устройств на дующих бутылках пока никто не делал, но, видимо, однажды появятся и они. На самом деле, речь слышна на очень многих из доступных в MIDI инструментов, просто с этими бутылками как-то получается лучше всего (послушайте бутылочную версию symphony или её же в midi).

Какие-то слова удаётся расслышать очень чётко, с другими сложнее. Пробуем искать их в интернете и находить закономерности. Помочь нам может и формулировка задания: кем бы могли быть зарубежные военные на букву Y? Рано или поздно понимаем, что всё это —фонетический алфавит NATO (а букве Y как раз соответствует слово Yankee). Слушаем записи ещё раз, имея перед глазами шпаргалку-алфавит. Понимаем, что в разминке нам просто по порядку начитывают алфавит, подчёркивание (underscore) и цифры, а в симфонии —флаг по буквам: uniform —golf —romeo — alfa —underscore —···

Вероятно, флаг можно расслышать и так, немного сломав по дороге уши и мозг. Но лучше разобрать разминку на эталонный алфавит и сопоставлять с ним группы нот из симфонии. Например, сопоставляя длительности звучания каждой буквы, можно сразу исключить большинство неоднозначностей. Можно было бы пойти дальше и сопоставлять сами ноты, правда, в данном случае нужно учитывать возможность неточных совпадений, лишних или исчезнувших нот.

Код, делающий сопоставления, появится тоже чуть позже.

Флаг: **ugra_wavy_obnoxious_squirrels_7e180373**

ГОСТ 34.13-2022

Криптография, 100 очков.

> Настала эпоха постквантовой криптографии —завершить этот процесс предлагается внедрением революционного алгоритма шифрования ГОСТ 34.13-2022. Пока он в бета-версии —самое время изучить его криптостойкость.
>
> *flag.enc*
>
> *encrypt.py*
>
> *https://betagost.q.2022.ugractf.ru*

Решение

Итак, в этом задании флаг просто ксорится с некой ключевой последовательностью, которую выдаёт класс `Stream`.

Известно, что функция `XOR` обладает простым свойством —`a xor b xor b = a`. Это означает, что таск решается предельно просто —запустим тот же скрипт, но для зашифрованного файла —и файл расшифруется. Но не тут-то было.

Что ж, воспользуемся предложенным сайтом. Попробовав зашифровать одно и то же сообщение несколько раз, видим, как шифротекст меняется. Дело в том, что переменная `value` глобальная, и она меняется после каждого шифрования.

Алгоритм, по которому генерируется ключевая последовательность, довольно стандартный и известный. Называется он *LFSR*, или регистр сдвига с линейной обратной связью. Значение `value` в нём и обозначает этот самый регистр. Для решения задания даже необязательно разбираться, как работает алгоритм. Достаточно заметить две вещи:

- `value` никогда не превысит 2^{18}
- следующий бит и новое значение `value` зависит только от текущего `value`

Следовательно, если значение `value` совпадет с каким-то из предыдущих, вся дальнейшая последовательность `value` будет циклическим повтором. При этом `value` гарантированно хотя бы раз за 2^{18} шагов повторится —потому что всего различных возможных чисел у нас 2^{18}.

> Интересно то, что для данного секрета длина такого цикла —262143 —максимально возможная. Это связано с тем, что под неприметным числом 174807 скрывается примитивный многочлен степени 17.

Осталось перебрать все эти 262143 возможных значений `value` и для каждого узнать, не начинается ли флаг с `ugra`.

Флаг: **ugra_lfsr_is_so_cyclic_dbf6deae053a5be4**

Веб-департамент

Веб-программирование, 300 очков.

> Одна небезызвестная социальная сеть запустила веб-департамент. Ведущие специалисты будут искать неправдивую информацию в интернете —ссылки на неё в социальной сети разместить больше не выйдет. Теперь пользователи в безопасности!
>
> Однако, нам кажется, руководитель веб-департамента что-то скрывает. Мы нашли их служебный ресурс и даже QR-код одного из сотрудников для одноразовых кодов. Поможете немного освоиться в сети?
>
> *Примечание.* Чтобы решить это задание, вам нужно авторизоваться как пользователь `admin`.
>
> *otp-for-andy77.png*
>
> *https://webdept.q.2022.ugractf.ru/760ad278b2aefd1ffece972d6068129a/*

Сначала зайдём на сайт с помощью любого приложения для двухфакторной аутентификации — например, Google Authenticator или Authy. На сайте есть две страницы —одна позволяет запросить содержимое любого сайта, а вторая —отправить сообщение.

Обращаем внимание, что все ответы от сервера сопровождаются заголовком `X-Backend: 10.13.37.159`. Это намекает нам на атаку SSRF (Server-side request forgery) —мы можем получить доступ к внутренним ресурсам.

Начнём с сети `/24` и популярных веб-портов —например, с `:80`. В локальной сети обнаруживаем хост `http://10.13.37.62` —по этому адресу отдаётся некий JSON.

Изучая его, приходим к выводу, что это схема API —доступных эндпоинтов на этом сервере и формата запросов. Всего видим два эндпоинта:

- `GET /request-otp?username=...`
- `POST /check-otp username=...&code=...`

POST-запросы мы делать не можем, попробуем GET с именем пользователя `admin`: `http://10.13.37.62/request-otp?use`
Получаем непонятный текст, но видим в нём `PNG`. Вероятно, это такая же картинка с кодом, что была у `andy77`.

`\x89` выглядит как байт с хекс-кодом 89. Очень похоже на то, как Python кодирует бинарные строки. И действительно, даже куки намекают на использование aiohttp —это веб-фреймворк для Python.

Самый простой способ получить картинку —обернуть это всё в `b'...'` —это байтовая строка, и вывести полученную переменную в файл. Код, кстати, можно скопировать из какого-нибудь таска на криптографию:

```
image = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR...\x00IEND\xaeB`\x82'
with open("image.png", "wb") as f:
    f.write(image)
```

Открываем картинку, видим QR-код администратора. Входим, получаем флаг.

Флаг: **ugra_isolate_your_public_hosts_d611010a81dd**

Крипто-департамент

Веб-программирование, 300 очков.

Одна небезызвестная социальная сеть запустила веб-департамент. Ведущие специалисты будут искать неправдивую информацию в интернете —ссылки на неё в социальной сети разместить больше не выйдет. Теперь пользователи в безопасности!

Мы нашли архивы жалоб сотрудников департамента. Но, как и следовало ожидать, в крипто-департаменте их зашифровали. Раскройте, чем же так недовольны сотрудники веб-департамента.

Примечание. После решения задания *Веб-департамент* вам станет понятнее, о чём речь в условии.

otp-for-andy77.png https://webdept.q.2022.ugractf.ru/760ad278b2aefd1ffece972d6068129a/

Решение

После решения задания «Веб-департамент» мы авторизованы в системе под администратором. Кроме флага, мы также узнаём, что есть и вторая сеть —10.13.38.0/24. Просканировав все порты 80 и в ней, находим странную веб-страницу для печати сообщений —10.13.38.116.

Но какие же сообщения можно расшифровать? Вспоминаем о второй части сайта —сервисе с жалобами. Возвращаясь к ней уже с повышенными привилегиями, мы можем увидеть список последних жалоб. Но в зашифрованном виде.

Попробуем напечатать любую жалобу —и, действительно, сайт нам сообщает о её печати на принтере. Если же мы отправим абракадабру, то ничего не напечатается: нам скажут, что произошла ошибка.

> Обратите внимание: сама форма на загруженной странице работать и не будет —надо вручную дописывать `enc` именно в URL запроса.

Вспоминаем про AES-CBC, упомянутый ранее. Атака, которая подойдёт нам в этом случае — Padding Oracle —она позволяет всего лишь по сообщению о корректном или некорректном шифротексте получить флаг.

Посмотрим на формат зашифрованных сообщений. Независимо от длины сообщения, часть до | —base64 от 16 байт данных, а справа —base64 данных длины чуть большей того текста, что мы отправляем —если быть точным, длина округляется до 16 байт вверх.

Предполагаем, что слева от | находится IV, а справа —шифротекст, заполненный паддингом. Самый распространённый формат паддингов —PKCS7; предположим, что он и использовался.

Код эксплойта

Флаг: **ugra_do_not_ever_use_cbc_420d096df213**